Universität Ulm
Institut für Wirtschaftswissenschaften

Machine Learning in Economics

Kumulative Dissertation

zur Erlangung des Doktorgrades Dr. rer. pol.
der Fakultät für Mathematik und Wirtschaftswissenschaften
der Universität Ulm

vorgelegt von

Martin Kies

aus Berlin

Amtierender Dekan:
Prof. Dr. Martin Müller

Erster Gutachter:
Prof. Dr. Sebastian Kranz (Universität Ulm)
Zweiter Gutachter:
Prof. Dr. Georg Gebhardt (Universität Ulm)

Tag der Promotion:
07. Oktober 2020

Acknowledgements

I am deeply thankful to all the people who supported me directly or indirectly to write this doctoral thesis.

Most importantly, I thank my academic supervisor, Professor Sebastian Kranz, for accepting me as a doctoral student and employing me at both the Institute of Economics and the Institute of Sustainable Corporate Management. I am very grateful for him leaving me considerable intellectual freedom to pursue my research interests, while at the same time giving a lot of helpful advise. I can not overstate how thankful I am for his great and elaborate feedback and him always having an open ear and time for discussions. I further thank Professor Georg Gebhardt who accepted the invitation to become my second supervisor.
I am very grateful for the fruitful cooperation with my colleague Frederik Collin, who not only co-authored one of my articles, but also gave very insightful input to my other projects.
I thank the targens GmbH for the provision of the data set needed for my third article.
Furthermore, I thank all persons which provided me with valuable ideas, feedback and discussions, but most notably, in alphabetical order, Maria Baier, Dr. Patrick Biermann, Daniel Blochinger, Dr. Kay Jahnke, Dr. Enno Giese, Maxim Ott and Dr. Thomas Roscher. I also thank Julius Düker and Clara Ulmer for their support regarding my officelessness situation.
I thank my former colleagues at the Institute of Economics for the friendly atmosphere and the always interesting conversations. It was a pleasure working with you!

Finally, I thank my family and friends for all the invaluable moral support. First of all, I thank my parents Simone and René Kies for their love and unwavering support, not only during my doctoral studies but also throughout my whole life. I also thank my sister, Annika Kies, for being an important pillar of my life and always having my back.
I thank my friends who were always there for me, gave me much needed support and put up with me even in bad times. A very big thank you goes to Boon Choo, Sarah Menzel and Surya Wöhrle. A special thank you goes to Richard Verbeet, who believed in me, even though he suffered financial loss due to it.

Contents

Overview of Research Papers

Research papers included in this dissertation

1. Martin Kies (2020). “Finding Best Answers for the Iterated Prisoner’s Dilemma Using Improved Q-Learning”. In: *Available at SSRN.* DOI: `10.2139/ssrn.3556714`. URL: `https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3556714`
2. Martin Kies (2017). “Impacts of Sponsored Data on Infrastructure Investments and Welfare”. In: *Available at SSRN.* DOI: `10.2139/ssrn.3042563`. URL: `https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3042563`
3. Frederik Collin and Martin Kies (2020). “Impact of Near-Time Information for Prediction on Microeconomic Balanced Time Series Data using Different Machine Learning Methods”. In: *Available at SSRN.* DOI: `10.2139/ssrn.3559645`. URL: `https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3559645`

Copyright note

All articles have been published on SSRN. While this gives SSRN the non-exclusive rights to post and distribute my articles, this did explicitly not transfer my copyright. For this reason I am free to publish the articles again as part of this dissertation.

Introduction

The ever increasing possibilities to generate, distribute and analyze data have a profound impact on consumers, companies and research. With this doctoral thesis my goal is to not only further magnify these possibilities, but also to deepen the understanding of them. I hereby present three articles, which each touch on one of these facets:

Generate

Recent efforts in digitalization allow organizations to collect and store data more easily than ever. This type of data has an important drawback: Only events which have actually happened can be observed. As these depend on particulars of past actions and realized chances, it is difficult to develop an understanding about what could have been.
In certain situations, however, it is possible to generate a model of the system of interest. However, even if the rules of the environment can be described sufficiently, it might still be non-obvious which actions lead to the optimization of a given key performance indicator. Within this simulated environment one can simulate actions to observe their outcomes and effects on the indicator. Doing so might be considerably cheaper and faster than actually performing these actions in reality. However, it might nevertheless not be sufficient to blindly chose them by chance. This holds in particular, if the system is very complex, different situations require different actions or the sequence of actions is important.
Reinforcement Learning provides techniques and methods on how to chose which action in which situation. Choosing the correct actions optimizes how valuable the generated data is. The article *Finding Best Answers for the Iterated Prisoner's Dilemma Using Improved Q-Learning* (Kies, 2020) presents several improvements to the known reinforcement learning algorithm Q-Learning. These improvements are tested and analyzed on the use case of the Iterated Prisoner's Dilemma game with imperfect public monitoring. In Kies (2020), I developed an algorithm which takes a black box strategy as input and returns good, and often times near optimal counter-strategy. The performance of this counter-strategy allows researchers to have an easy numerical measurement of the exploitability of the given strategy. Additionally Kies (2020) gives a detailed introduction about the main ideas of Q-Learning and reinforcement learning in general aimed at economists.
The contents of Kies (2020) are presented in Chapter 1.

Distribute

Not all data of interest can be found in tabular data bases. In particular for consumers data often times means something completely different: When agreeing on a data plan with their internet service provider, the most data-intensive usages can lie in the field of entertainment. Examples include streaming music, movies and TV-serials as well as playing online games. The increasing demand for these services might require the internet service provider to invest into network infrastructure to guarantee sufficient distribution of data.
In my article *Impacts of Sponsored Data on Infrastructure Investments and Welfare* (Kies, 2017) I developed a theoretical model to analyze *Sponsored Data*, an alternative revenue stream for the internet service provider. Instead of having the consumers pay a flat price allowing them to access a given amount of data, a content provider might pay the internet service provider to take its specific content out of the data cap of the consumer. The model shows, that allowing Sponsored Content may indeed increase investments into the network infrastructure given that the costs to do so are very high. If the profitability of the content provider however is comparably high, i.e. he is willing to pay a lot for Sponsored Content, this can lead to bad incentives for the internet service provider. When considering the revenue from the content provider it can be more profitable for the internet service provider to invest less into network infrastructure than otherwise. A decrease in net welfare can be the result. The contents of Kies (2017) are presented in Chapter 2.

Analyze

Having generated or received data begs the question on how to best utilize it. Especially regarding industry-applications, but also in research, one is often interested in predicting responses based on input variables or in being able to continue a time series. A multitude of Machine Learning methods emerged over the years allowing to do so. The article *Impact of Near-Time Information for Prediction on Microeconomic Balanced Time Series Data using Different Machine Learning Methods* (Collin and Kies, 2020) analyzes a variety of Machine Learning methods and compares their performance on a time series data set. The analyzed methods are Linear Regression, Elastic Nets, Partial Least Squares, Generalized Additive Models, Random Forests, Gradient Boosting and Neural Networks. The data set consists of daily deposits of a multitude of stores. This allowed us to analyze whether it is possible to use same-day data of other stores to improve the prediction performance of a given store.
We find that in our case all methods perform at approximately the same level when being restricted to data from the store itself. Using same-day information of other stores improved prediction quality considerably. This held in particular for the method Random Forest which reduced the root mean squared error on the test data set by 24% compared to the best performing method using store-only data. The out-of-sample performance of Random Forests was considerably more robust when increasing the number of input parameters compared to Partial Least Squares.

To have a fair comparison between the different methods, we also developed a novel hyperparameter-optimization technique which uses a Regression Tree to find optimal settings for each Machine Learning method.
In contrast to the other two articles, this article was written in cooperation with Frederik Collin, with equal contributions.
The contents of Collin and Kies (2020) are presented in Chapter 3.

1 Finding Best Answers for the Iterated Prisoner's Dilemma Using Improved Q-Learning

Source:

Martin Kies (2020). "Finding Best Answers for the Iterated Prisoner's Dilemma Using Improved Q-Learning". In: *Available at SSRN*. DOI: `10.2139/ssrn.3556714`. URL: `https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3556714`

Finding Best Answers for the Iterated Prisoner's Dilemma Using Improved Q-Learning[*]

Martin Kies[§]

Wednesday 18th March, 2020

Given an arbitrary black-box strategy for the Iterated Prisoner's Dilemma game, it is often difficult to gauge to which extent it can be exploited by other strategies. In the presence of imperfect public monitoring and resulting observation errors, deriving a theoretical solution is even more time-consuming. However, for any strategy the reinforcement learning algorithm Q-Learning can construct a best response in the limit case. In this article I present and discuss several improvements to the Q-Learning algorithm, allowing for an easy numerical measure of the exploitability of a given strategy. Additionally, I give a detailed introduction to reinforcement learning aimed at economists.

Keywords: Iterated Prisoner's Dilemma, Repeated Prisoner's Dilemma, Imperfect Public Monitoring, Reinforcement Learning, Q-Learning, Neural Networks, Gradient Boosting, Machine Learning

JEL Classification: C61, C63, C72, C73

[*]I thank Sebastian Kranz for excellent feedback and support. I am grateful to Frederik Collin, Maxim Ott, Kay Jahnke, Enno Giese and Patrick Biermann for helpful discussions and comments.

[§]Department of Mathematics and Economics, Ulm University.
E-mail: martin.kies@uni-ulm.de

Contents

List of Figures

1. Introduction

Repeated games have been used extensively to study reoccurring interactions between different factions that allow for cooperation and conflict. Consider for example the long-term working relationship between two companies and assume further that they want to cooperate on a wide area of different projects. Due to the heterogeneity of their projects it is necessary to craft a specialized contract for each project separately. Both aim to streamline contract negotiations as much as possible. This results in most proposed wordings to be accepted by the other party without detailed proof-reading. If one of the companies sneaks in clauses which are favorable for it and the other company trustingly signs this contract, the profits of the project are distributed more uneven as per the general agreement. This might anger the other company after the fact and so they might try to recoup these calculatory losses by sneaking in favorable clauses as well. As each project is somewhat idiosyncratic however, an unfavorable clause might not be easily distinguishable between an honest mistake due to wrongfully perceived costs or a malevolent attempt to make the cooperation more one-sided.
The classic game theoretic approach is to study equilibria of repeated games. In a symmetric Nash equilibrium, the equilibrium strategy is a best response against itself. This means that given that the other players follow this strategy it is also optimal for oneself to follow it. Most game theoretic equilibrium concepts share this requirement of the strategies being best responses while adding additional constraints. In the case of the Subgame Perfect Equilibrium, for example, in every subgame the continuation equilibria must be mutual best responses.
The well-known folk theorems show for a large class of repeated games that, given the limit case of not discounting future payoffs, every feasible payoff that grants each player at least her Min-Max payoff of the stage game can be implemented as the expected equilibrium payoff (see e.g. J. W. Friedman (1971), Fudenberg et al. (1994) or Abreu, Dutta, et al. (1994)). Given a fixed discount factor reducing the impacts of future payoffs, there may exist a large, infinite set of equilibrium strategies and a large set of equilibrium payoffs. Algorithms to compute these equilibrium payoff sets for fixed discount factors have been developed conceptually for example by Abreu, Pearce, et al. (1990) and with a concrete numerical implementation by Judd et al. (2003). For the case of repeated games with transfers Goldlücke and Kranz (2012) provide a fast algorithm. Both numerical approaches are limited to perfect monitoring.
A crucial assumption in game theoretic equilibrium analysis is that the played equilibrium strategies are commonly known by all players. This might not necessarily be the case. If a player follows a slightly different strategy, the resulting outcome is in some cases much worse than the equilibrium outcome. For example, the grim trigger Nash reversion strategies proposed by J. W. Friedman (1971) can very efficiently implement mutual cooperation in the Iterated Prisoner's Dilemma (IPD) game if they are exactly followed by both players.

Should one player however slightly deviate and for some reason defect once on the equilibrium path, the other player retaliates. Following the strategy, the first first player is triggered to strike back in turn, which results in a defection spiral for both players. This problem is enhanced by the fact that typically a large set of equilibrium strategies exists. Coordinating on a specific game theoretic equilibrium can be a very daunting task in reality, in particular, if parties do not extensively communicate with each other.

An alternative approach to study repeated games has been popularized by Axelrod (1984). He invited experts to submit computer coded strategies for the IPD game. Afterwards he pitted all strategies against each other and declared the winner as the strategy with the highest average payoff against the pool of submitted strategies. The underlying idea of this approach is to evaluate a strategy based on their performance within a tournament setting. Interestingly, strategies which possess desired game-theoretic properties do not necessarily perform well according to this metric. The grim trigger Nash reversion strategy that allows to implement cooperation for the lowest discount factor in a subgame perfect equilibrium for example ranked place 52 of 63 in the second Axelrod tournament (Axelrod, 1984, p. 196). The strategy Tit-for-Tat on the other hand is not a subgame perfect equilibrium, but won both tournaments (Axelrod, 1984, p. 193ff.).

An advantage of using a tournament is that it is possible to use this metric for arbitrary complex strategies. In contrast to a possibly time-consuming theoretical analysis, it is sufficient to translate the strategy of interest into computer code and set it against a fixed pool of pre-defined strategies.

However, while a strategy might perform well against this pool, it might nevertheless have substantial weaknesses. If no strategy exists in the tournament pool which exploits these weaknesses, then they will not influence the performance measurement. To have a more comprehensive picture of the strategy it is thus necessary to construct a best response against the strategy. If the investigated strategy possesses considerable weaknesses, then the best response will achieve a substantially higher payoff against the strategy than the strategy against itself. If on the other hand the payoff of the best response is identical to the strategy, we know, that the strategy is an equilibrium and can not be exploited.

Theoretically deducing a best response can be very time-consuming if the submitted strategy is sufficiently complex. To evaluate an arbitrary invented strategy it would thus be useful to have a numerical tool that can construct best responses.

This article presents such a tool, using and extending modern machine learning approaches. It does so not only in the comparatively easy case of perfect information games, but also given imperfect public monitoring, where observations about the actions of the investigated strategy are non-reliable. The underlying algorithm has been built and tested on strategies playing an IPD game with discount factor and imperfect public monitoring, but can be extended to other economic games, as long as they fulfill the criteria as specified by Section 2.2.

While to my knowledge nobody developed a machine learning algorithm with the goal in mind to find weaknesses in arbitrary strategies, several other articles have been published which use machine learning methods in the context of the Prisoner's Dilemma game and other economic games.
One topic of interest is the emergence of cooperation, where reinforcement learning algorithms are used to simulate adapting or evolving behavior. J. Zhang et al. (2011), using particle swarm optimization, and Xue et al. (2017), using TD(λ)-Learning, study the emergence of cooperation in an evolutionary setting given a spatial Prisoner's Dilemma game, where players interact with their immediate neighbors. To construct adaptive strategies in an evolutionary setting W. Wang et al. (2018) use a variety of modern machine learning algorithms to study the evolution of cooperation in the IPD game by manipulating the degree of cooperation of different agents. Leibo et al. (2017) investigate different types of games, including the IPD game, to study the emergence of cooperating behavior which may be motivated by *greed* and/or *fear*. They use the IPD game as their example for a payoff matrix where players are motivated by both. Foerster et al. (2018) use their developed approach *LOLA* to not only model the mind of the opponent but also to actively consider the learning of the opponent and are thus able to modify their behavior to train the opponent with the goal of achieving mutually beneficial outcomes.
The explicit goal to develop strategies with a high performance in a tournament setting pursue Harper et al. (2017), who use particle swarm algorithms and evolutionary reinforcement strategies to produce strategies which win against the corpus of strategies archived in The Axelrod project developers (2016a). Similarly, Brunauer et al. (2007) use genetic algorithms with Lookup-Tables of various lengths to develop a strategy which shows a good performance against a subsection of the strategies of the original Axelrod tournament. A similar approach is pursued by Ashlock et al. (2014) who explicitly admix strategies which have an *exploitation key* which is a played pattern after which the strategy is exploitable. The master thesis Thomas (2018) combines the *Soar* cognitive architecture and Q-Learning to train against a selected number of deterministic strategies in a deterministic setting. As a tournament environment from the view of a reinforcement agent is very similar to training against a single strategy, all of the aforementioned approaches are also able to be trained against a single strategy.
A stronger focus on single strategies can be found with Sandholm and Crites (1996) who use basic Q-Learning with Boltzman exploration on an IPD game to train agents with Lookup-Tables and an Elman Recurrent Neural Network. As a basic proof that their algorithm converges against a given strategy they check against Tit-For-Tat. This has been extended by the course project K. Wang (2017) to LSTM networks and several other stationary strategies.

This article contributes to the existing literature by

1. Helping to bridge the gap between game theory and machine learning by developing an algorithm which uses various techniques to explicitly try to find and exploit weaknesses given an arbitrary strategy. With the exception of Harper et al. (2017), all mentioned articles trying to optimize against other strategies limit themselves to simple or classical strategies. I use an unbiased pool of strategies which vary greatly in complexity.

2. Allowing for observation errors to occur, thus significantly increasing the complexity of the optimization task.

3. Evaluating and discussing various known improvements of the Q-Learning algorithm in the context of the IPD game, which have only been discussed in other domains so far.

4. Developing and evaluating novel improvements of the Q-Learning algorithm, which improve performance and learning behavior in the dimensions of final result, necessary training data and necessary computational resources.

Section 2 lays the theoretical groundwork for this article regarding both, the game theoretic aspects and machine learning. It gives a fast-track introduction to develop an understanding of the concepts relevant to understand the resulting algorithm. Afterwards in Section 3, the developed algorithm is described and the different features of it are evaluated in detail.
Readers with a background in machine learning which are mainly interested in the developed improvements of the Q-Learning algorithm are encouraged to jump to Section 3.1 for an overview of the proposed improvements to Q-Learning. Section 3.2 compares the performance to Q-Learning and Section 3.12 summarizes the effects of the individual improvements.
Readers with a background in economics which are mostly interested in using the accompanying R-package Kies (2019) to have a numerical tool to find the exploitation potential in their strategies are encouraged to read Section 2.1.1 for the economic reasoning and the rules of the game where the algorithm has been tested on. Section 2.2 defines the basic requirements a game has to meet, if one wants to use the algorithm in another context.

2. Theoretical Foundation

This section gives an introduction to the foundational works upon which the algorithm presented in this article has been developed. The algorithm itself is presented in Section 3.
The main motivation behind the development of the algorithm has been to approximate best replies for the IPD with imperfect public monitoring. Section 2.1 gives context regarding this game in general. Section 2.1.1 defines the rules of the used variant of the IPD and introduces necessary nomenclature. Afterwards an assortment of well-known strategies is discussed in Section 2.1.2. These are used to explain basic concepts which help to understand the effects of some of the algorithmic improvements and pose as a benchmark.
Section 2.2 introduces the framework and the notation necessary to discuss the task of finding counter-strategies. The algorithm can be used to find optimal behavior in other games and environments as well, as long as these games meet the requirements of this presented framework.
Afterwards, in Section 2.3 an introduction to the machine learning field of Reinforcement learning is given. First, in Section 2.3.1 we present classic approaches to find optimal policies given that we have a complete model of the environment. The ideas herein are extended in Section 2.3.2 to algorithms which have to learn the environment as well. This allows us to find counter-strategies even if only the actions of a strategy can be observed and the strategy itself is not given explicitly. Section 2.3.3 presents the Q-Learning algorithm, which builds the basis to our improvements. As the classic Q-Learning algorithm assumes the environment to be sufficiently simple to try every combination of actions and save their results, Section 2.3.4 extends this algorithm by introducing function approximators. These are used to estimate the value of proposed actions even if a specific situation has never occurred before.
The machine learning field of Supervised Learning studies function approximators. We will focus in Section 2.4 on three of them: Gradient Boosting (Section 2.4.1), Neural Networks (Section 2.4.2) and Recurrent Neural Networks (Section 2.4.3).

2.1. The Iterated Prisoner's Dilemma game with Noise

2.1.1. Rules of the Game and Nomenclature

The Prisoner's Dilemma game was coined by Albert Tucker based on puzzles created by Merril Flood and Melvin Dresher in 1950 and has since been analyzed and referenced in thousands of scientific articles (Kuhn, 2019). A myriad of variants and extensions to the idea behind this economic game emerged over time. Within this article we concentrate on the following version:
Two players play an infinitely often repeated decision problem, the *stage game*, where each player has two actions: *Cooperate* (C) and *Defect* (D). Both players decide on their action without knowing the action of the other player. We will use the term *period* to refer to this single decision made by both players.
After each period, each player observes both, the previous action of their opponent and their own previous action. This observation may or may not correctly represent the taken action. With probability err_D a cooperation is erroneously displayed as a defection and with probability err_C a defection is erroneously displayed as a cooperation. We call the incorrect display of a previous action *noise*. Both opponents make the same observations, which makes this an imperfect public monitoring game. As a consequence each player always knows whether or not noise occurred regarding his own action.
We call the sequence of periods belonging to an Iterated Prisoner's Dilemma (IPD) game an *episode*. The players are allowed to remember past observations within an episode. This capacitates them to play complex patterns to react to the past observed behavior of the opponent.
We define a *strategy* as a full set of instructions for a player what to do in which situation given the perceived history within the episode. These instructions can be stochastic - we allow mixed strategies which randomize between actions in the same situation.
Each period $t \in \mathbb{N}$ has an associated payoff r_t for the given player according to the (undiscounted) payoff matrix of the stage game shown in Table 1.

		Player 2	
		Cooperate	**Defect**
Player 1	**Cooperate**	(R, R)	(S, T)
	Defect	(T, S)	(P, P)

Table 1: Undiscounted payoff matrix of the stage game of the IPD game. If player 1 cooperates and the other player defects, player 1 receives a payoff of S and his opponent a payoff of T.

Table 1 defines a Prisoner's Dilemma game if $T > R > P > S$ and $2R > S + T$. This way, the dilemma unfolds: For each player it is individually beneficial to defect, irrespective of the choice of his opponent. The highest mutual payoff, however, is generated when both players cooperate.
To keep the nomenclature of the machine learning literature, the payoff of a period will be termed *reward*. The rewards are calculated based on the actually taken actions of the players regardless of possible observation errors. While noise may influence the behavior of the players, it does not directly affect the rewards. As it would be possible to infer observation errors based on received rewards, the rewards are not available to the players within an episode.
The performance of a given strategy in a specific episode is measured by its average reward $\bar{r}$ over periods, where each new reward is cumulatively discounted using the factor $\gamma \in (0, 1)$. The average reward $\bar{r}$ is therefore calculated by

$$\bar{r} := (1 - \gamma) \cdot \sum_{t=1}^{\infty} \gamma^{t-1} r_t$$

It can be interpreted as the representative stage game reward of the strategy for the complete episode as it holds that

$$\bar{r} = (1 - \gamma) \cdot \sum_{t=1}^{\infty} \gamma^{t-1} \bar{r}$$

Due to noise and possibly stochastic elements of the strategies, r_t may differ between different episodes, even given a fixed period t and a fixed strategy pair $(\mathbf{s}, \mathbf{s'})$.
Assume that the strategy of the opponent $\mathbf{s'}$ is fixed. We define the expected reward of strategy $\mathbf{s}$ in period t as

$$R_t(\mathbf{s}) := \mathbb{E}[r_t|\mathbf{s}, \mathbf{s'}]$$

The expected average reward $\bar{R}(\mathbf{s})$ is calculated analogously to $\bar{r}$:

$$\bar{R}(\mathbf{s}) := (1 - \gamma) \cdot \sum_{t=1}^{\infty} \gamma^{t-1} R_t(\mathbf{s}) \tag{1}$$

A *best response* against some strategy $\mathbf{s'}$ is defined as a strategy $\mathbf{s^*}$ for which it holds that

$$\bar{R}(\mathbf{s^*}) \geq \bar{R}(\mathbf{s}) \quad \forall\, \mathbf{s} \in \mathcal{S}$$

with $\mathcal{S}$ being the set of all possible strategies. Of particular interest for us is the difference

$$\delta_R := \bar{R}(\mathbf{s^*}) - \bar{R}(\mathbf{s'}) \geq 0 \tag{2}$$

i.e. the difference in the average expected reward between the optimal strategy against $\mathbf{s'}$ and the average expected reward of the strategy against itself. In the case of $\delta_R = 0$ the strategy $\mathbf{s'}$ establishes a Nash equilibrium, i.e. is a best response against itself.
The purpose of the algorithm presented in Section 3 is to find a strategy $\mathbf{s}$, where $\bar{R}(\mathbf{s})$ comes as close to $\bar{R}(\mathbf{s^*})$ as possible. This allows us to estimate δ_R and thus the exploitability of strategy $\mathbf{s'}$.

As it is not possible to simulate an infinite number of periods to calculate the performance of a strategy $\mathbf{s}$, we use the following approach, as introduced by Roth and Murnighan (1978):
Instead of a discount factor one can interpret γ as the probability that an exogenous force allows our game to continue. In other words, with a probability of $1-\gamma$ the episode ends after the played period. Implementing the game this way, the non-discounted averaged value over all played periods converges towards the desired value $\bar{R}(\mathbf{s})$ with increasing number of episodes. This approach is implemented in the R-package *StratTourn* (Kranz and Kies, 2019) which is used for the performance measurements of Section 3.2.
As is detailed in Section 3.3, the testing pool for the algorithm is based on a tournament held by Ulm university. Consequently we use the same parameter settings as in the tournament within this article:

- The values $R=1$, $S=-1$, $T=2$ and $P=0$ for the payoff matrix of the stage game.
- A discount factor/continuation probability $\gamma=0.95$, leading to an average episode length of 20 periods.
- An observation error probability $\text{err}_D=0.15$. Given that the opponent cooperates, there is thus a 15% probability of erroneously detecting this action as a defection.
- An observation error probability $\text{err}_C=0$. A defection is always correctly perceived as one.

The presented rules are similar, but not identical to the implementation of the second tournament of Axelrod (see Wu and Axelrod (1995)). Our setting differs in the following aspects:

1. Axelrod used the values $R=3$, $S=0$, $T=5$ and $P=1$ for the payoff matrix of the stage game (Axelrod, 1984, p. 8).
   One way to compare the difference in payoff matrices is to assume that the default case is mutual cooperation (C,C) and to ask how many non-discounted periods of successful unilateral defections (D,C) the strategy needs to offset the negative effects of one mutual defection (D,D). With Axelrod exactly one period is needed, so player 1 is indifferent between the patterns {(D,D), (D,C)} and {(C,C),(C,C)}. In our setting the same holds as well.
   If by contrast one asks how many non-discounted periods of successful unilateral defections (D,C) are needed to offset the negative effects of the opponent unilaterally defecting (C,D), the effects differ. With Axelrod's values one needs 1.5 periods, but within our setting 2 periods are needed. In our setting it is thus more damaging to be caught in the situation of having cooperated when the opponent defected. This might reflect on the relative performance of strategies, with our setting favoring more cautious or aggressive strategies.

2. Axelrod used 5 episodes averaging to 151 periods in his second tournament (Axelrod, 1984, p. 193). While our performance tests in Section 3.2 using 1000 episodes are designed to average the effects of stochastic strategies and observation errors, the approach of Axelrod allows strategies to better adapt to the behavior of their opponent within an episode. Interestingly, complicated strategies did not fare better or worse than simple strategies in Axelrod's tournament (Axelrod, 1984, p. 43), so it is not obvious to which extent this is relevant.

3. The noise in Axelrod's tournament was symmetric, i.e. $\mathrm{err}_D = \mathrm{err}_C$. This has strong implications on which strategies perform well. Take for example a strategy which defects after a certain number of observed defections of its opponent until sufficiently many cooperations of its opponent can be observed. If this strategy is set against itself, it performs comparatively worse in our setting as measured by $\bar{R}(\mathbf{s})$. While in the setting of Axelrod there exists a certain probability, that bilateral observation errors allow the strategies to cooperate again, this is not possible in our setting, where a defection is always correctly observed as one.

4. The noise in Axelrod's tournament was implemented by a random shock which forced the strategy to implement the opposite action. This leads to a different reward of the strategies in this period. With our implementation there might be a wrongful observation, but the actually played actions are used to determine the reward. A strategy which always cooperates will therefore have an expected average reward $\bar{R}(\mathbf{s}) = R = 1$ against itself despite observation errors. Using the actual actions as basis for the rewards allows us to use less simulations when estimating $\bar{R}(\mathbf{s})$.

2.1.2. Strategies

Over time various strategies in the IPD game with noise emerged. In this section we will have a look at some strategies which are either useful as archetypes to explain some key features or have been developed with the goal in mind to create a good strategy. What constitutes as good depends on the specific intentions of the developer. In most cases the goal was to develop a strategy which achieves a high ranking in a tournament against unknown strategies.
We start with strategies which are interesting due to their simplicity:

1. Always Cooperate (*always.cooperate*)
   This strategy, sometimes also called ALLC, always cooperates independent of period or observed actions. If *always.cooperate* plays against itself it always receives an average reward of 1 in our setting, no matter the amount of actual noise. As no retaliatory measures are incorporated, this strategy has the maximum potential for exploitation.

2. Always Defect (*always.defect*)
   Also called ALLD, this strategy can be seen as the ultimate antagonist, as it always defects. It is by definition unexploitable and will always have at least the same average reward as the opposing strategy. Consequently, should the success of a strategy be measured by the number of matches where the strategy had a higher average reward than its opponent, *always.defect* is the best possible strategy. *always.defect* is a subgame perfect equilibrium even without noise and the best response against all non-retaliatory strategies, such as *always.cooperate*.
   Interestingly it is not obvious how good *always.defect* fares in a tournament, as this strongly depends on the fraction and harshness of the retaliatory strategies. If there is only a small percentage of retaliatory strategies, *always.defect* is able to generate high rewards from the non-retaliatory ones and might be able to ensure a good placing.

3. Random Action (*random.action*)
   As the name implies, this strategy plays a random action independent of period and observations. Within this article, we set the probability to play a cooperation $p_C = 0.5$, but other variants are possible. This is another example of a non-retaliatory strategy, against which *always.defect* is a best response.

4. Grim Trigger (*grim.trigger*)
   grim.trigger is the main example for a maximally harsh retaliator:

```
If First Period: Cooperate
If At least one defection of opponent in previous periods observed
    Defect
Else
    Cooperate
```

   The strategy cooperates until the very first defection is observed from the opponent and defects from then on. Thus, a single defection of the opponent is punished severely. In the absence of noise both, *always.cooperate* and *grim.trigger* itself, achieve an average expected reward of $\bar{R}(\mathbf{s}) = R = 1$ in our setting by always cooperating. In fact, given the payoff matrix of our setting (*grim.trigger*, *grim.trigger*) is a subgame perfect equilibrium in the absence of noise and sufficiently high discount factor γ (see e.g. McGillivray and Smith (2000)). However, if observation errors are possible, it is just a matter of time until *grim.trigger* changes into its punishing mode, thus decreasing the average reward against itself considerably.
   As measured by tournament performance, *grim.trigger* is a sub-optimal. The strategy received place 52 of 63 in the second Axelrod Tournament (Axelrod, 1984, p. 196). Note, that within this tournament *grim.trigger* was called FRIEDMAN, as it was the entry of Prof. James W. Friedman[1].

5. Tit-for-Tat (*tit.for.tat*)

```
If First Period: Cooperate
If Observe defection of opponent in previous period
    Defect
Else
    Cooperate
```

   This strategy starts with a cooperation and copies the move of the opponent in the following periods. This starting move allows the strategy to cooperate against itself and similar strategies until an observation error occurs.
   While being a very simple strategy, it was made famous by winning the first two rounds of Axelrods Prisoner's Dilemma tournament (Axelrod and Hamilton, 1981). Both times, *tit.for.tat* was the entry of Prof. Anatol Rapoport, a mathematical psychologist who researched the Prisoner's Dilemma intensively (see e.g. Rapoport et al. (1965)). The success of *tit.for.tat* triggered intensive research leading to a deep understanding of this strategy, including its limitations.

[1]Economics Department, The University of North Carolina at Chapel Hill, *1936-†2016

> One of the main problems of *tit.for.tat* in our setting is the limited robustness when confronted with noise, especially if it is asymmetrical as in our case. When *tit.for.tat* plays against itself or a sufficiently similar strategy an observation error leads to never ending defections as defections are always correctly observed. In a symmetric setting of observation errors, such as in the second Axelrod tournament, *tit.for.tat* resembles a *random.action* strategy. (Molander, 1985). Several variants of *tit.for.tat* exist, which show a higher performance in the presence of noise, both in tournament settings and against themselves. These will be presented later in this Section as examples for good strategies.

Using the methods of Section 2.2, it is possible to explicitly calculate a best response for the presented strategies. The basic idea is to reduce the problem to a number of so called *states*. These states can be interpreted as all situations in which a strategy might be, as seen solely from the point of view of the strategy itself. The number of states depends on the complexity of the strategy one wants to optimize against. In the case of *always.cooperate*, *always.defect* and *random.action* a single state is sufficient. All those strategies do not react to their observations and are therefore always in the same situation. Solving for the optimal action in this state, we see that it is always optimal to defect. In the case of *grim.trigger* two states are sufficient. Either *grim.trigger* has been triggered or not. Against *grim.trigger*, a strategy which defects if *grim.trigger* has been triggered and cooperates otherwise is a best response. A similar situation occurs with *tit.for.tat*. Here the two states depend on whether a defection has been observed in the previous period or not. The best response against *tit.for.tat* is *always.cooperate*.
In contrast to the situation with those simple strategies, calculating a best response for an arbitrary strategy which takes the complete observation history into account might become very complex. With each period the possible state space up to a given period increases by a factor of 8 (given that noise affects both actions) or 6 (given that noise affects only one action, as in our specific case) per period:
Taking the perspective of one of the two players, imperfect public monitoring implies, that this player is able to observe whether his own action of the last period has been observed correctly, leading to

$$2 \text{ (no. of actions)} \cdot 2 \text{ (own action correct?)} \cdot 2 \text{ (observed action of opponent)} = 8$$

states in total per period if both of his possible action choices might be influenced by noise. With increasing number of periods the state space might thus become quite large even concentrating on pure strategies. With the possibility of having mixed strategies, where actions may be chosen based on underlying probabilities, the number of possible strategies becomes infinite and thus difficult to handle.
A strategy which aims to have a high tournament standing has to be able to perform well in a wide array of situations. Consequently, there was a strong focus in the literature on comparably easy and robust strategies. A general discussion on how to construct strategies which have a good chance to perform well, even in the presence of noise, is presented in Appendix A.

Notable strategies which have shown good performance in tournaments which include noise are:

1. Variants of *tit.for.tat*, in particular

   a) Generous Tit-for-Tat (*generous.tit.for.tat*)

```
If First Period: Cooperate
With probability $0 < c < 1$: Cooperate; Else:
If Observe defection of opponent in previous period
  Defect
Else
  Cooperate
```

   This strategy cooperates with a fixed probability and follows the strategy *tit.for.tat* otherwise.
   Generous.tit.for.tat has been suggested by Axelrod (1984) and has first been formally examined by Molander (1985). Wu and Axelrod (1995) showed that *generous.tit.for.tat* fares well with noise, especially if the noise is comparatively small.
   The big advantage of this strategy is, that in contrast to *tit.for.tat* a never ending defection loop due to observation errors can be avoided. Increasing the value of c decreases the number of expected periods with mutual defections. The unconditional cooperation probability c can and in most cases should be chosen in a way that the expected value of a defection is smaller than the expected value of a cooperation which makes this strategy unexploitable and forces an all-knowing opponent to always cooperate.
   Two obvious caveats of this strategy come to mind: The probability c stays constant even when faced with an obviously mean spirited opponent. The higher the ratio of often-defecting strategies, the more the average reward is decreased due to unconditional generosity. Additionally, c does not incorporate all available information. The strategy *generous.tit.for.tat* does not take into account whether its own actions have been observed as a defection. Doing so might help to deduce whether or not the observed defection of the opponent was retaliatory or predatory in nature. This problem is solved by *contrite.tit.for.tat*.

b) Contrite Tit-for-Tat (*contrite.tit.for.tat*)

```
If First Period:
    Status ← "content"
    Cooperate
NewStatus ← Status
If Unilateral observed defection of the opponent in previous period AND
Status is "content" : NewStatus ← "provoked"
If Observed cooperation of opponent in previous period AND Status is
"provoked" : NewStatus ← "content"
If Unilateral observed defection of myself in previous period AND Status
is "content" : NewStatus ← "contrite"
If Observed cooperation of myself in previous period AND Status is
"contrite" : NewStatus ← "content"
If NewStatus is "content" or "contrite"
    Cooperate; Status of next period is current NewStatus
Else
    Defect; Status of next period is current NewStatus
```

This strategy assumes that a retaliatory action against an observation error is excusable and thus does not retaliate. Even if the apology attempt after the observation error is hit by an observation error an additional time, the strategy keeps trying to cooperate until a successful apology has been made by making a visible cooperation. In the absence of noise it behaves like *tit.for.tat*.

The strategy *contrite.tit.for.tat* has been proposed by Sugden (1986). Its name was coined by Boyd (1989). Wu and Axelrod (1995) compare this strategy to *generous.tit.for.tat* and find a similar performance given a noise probability of around 1%. Increasing the probability for noise, they find an increasing edge for *contrite.tit.for.tat*. Unfortunately no pseudo-code is given by the original sources, which leads to different interpretations and versions of this particular strategy. The strategy as displayed here has been extracted based on the explanations given by Wu and Axelrod (1995). An alternative interpretation *ContriteTitForTat* is implemented by The Axelrod project developers (2016a). Here, a *bilateral* defection after being "content" also results in the strategy to become "provoked". In our specific setting this is a disadvantage, as playing against itself and getting hit by a bilateral observation error results in never-ending defections.

The main advantage of *contrite.tit.for.tat* is to have a high expected average reward $\bar{R}(\mathbf{s})$ against itself. While each observation error results in a small loss, it has no lasting effects on the willingness to cooperate in the long run.

c) *tft.2forgive*

```
If First Period: Cooperate
If Observed cooperation of opponent in the last period
  Cooperate
Else
  If Observed coop. of opponent in the two periods before this one.
    Cooperate
  Else
    Defect
```

Structurally *tft.2forgive* behaves similar to *tit.for.tat*, but declares the other strategy to be sufficiently nice after the successful observation of two consecutive cooperating actions to refrain from retaliation.
It should be intuitively understandable, that the best response against this strategy has to be to cooperate until *tft.2forgive* observes two consecutive cooperations and defect immediately afterwards, assuming a sufficiently high discount factor γ. That this intuition is correct is shown in Section 2.2.

Several other notable variants of *tit.for.tat* exist, with a rich body of literature behind them. An extensive list of *tit.for.tat*-like strategies can be found with The Axelrod project developers (2016b).

2. Pavlov (*pavlov*)

```
If First Period: Cooperate
If Observed, that both cooperated in the last period: Cooperate
If Observed, that opponent cooperated and I defected: Defect
If Observed, that opponent defected and I cooperated: Defect
If Observed, that observation, both defected in the last period: Cooperate
```

This strategy is the embodiment of the idea of "Win-Stay, Lose-Shift" and is thus sometimes called by this name. Originally developed by Rapoport et al. (1965) (here named "simpleton"), this strategy was made famous by Nowak and Sigmund (1993) and named by them based on the ideas behind D. Kraines and V. Kraines (1989) due to the reflex-like behavior to the last period.
Effectively this strategy tries to switch back from situations, where it is either exploited (C,D) or the result is bad for both parties (D,D) and stays with its last action otherwise. The strategy is vulnerable to *always.defect*, where it reduces its average reward considerably by making a peace offering every second period. Despite that, Nowak and Sigmund (1993) show that *pavlov* is able to handle noisy environments. Their analysis shows, that *pavlov* performs better

than the compared *tit.for.tat*-variants in certain settings by being able to extort *always.cooperate* and otherwise non-retaliatory strategies. Wu and Axelrod (1995) compare various *tit.for.tat* strategies against each other and against *pavlov* and find that *pavlov* lacks robustness compared to *contrite.tit.for.tat* and *generous.tit.for.tat*.

3. The *net.nice* family, as for example “Moderate Envy”Lahno (2000) (*net.nice0*):

```
def.count.me ← “No. of my defections as displayed by history”
def.count.other ← “No. of opponent defections as displayed by history”
diff ← def.count.me - def.count.other
If diff ≥ 0:
    Cooperate
Else
    Defect
```

This strategy has been published by Lahno (2000) and allows to punish the opponent through a mechanism which takes the complete history into account. A defection of the opponent, assuming a sufficiently high discount factor γ, is not profitable, as similar to *tit.for.tat* a direct retaliation is the answer. On the other hand, *net.nice0* is forgiving, as it resumes with the cooperation after evening out the rewards between itself and its opponent.
Depending on the number of noise it is not obvious, that specifically *net.nice0*, i.e. setting the threshold for *diff* to 0, is the best choice within the *net.nice* family. Indeed, empirically within the tournament at Ulm university, *net.nice1* achieved a higher average reward even though it tolerated one additional defection of the opponent, making it obviously exploitable. When playing against itself, or sufficiently similar opponents however, one can see, that allowing this additional defection may be beneficial as more mutual cooperations can be played in total due to more observation errors being absorbed without retaliation.

2.2. Basic Framework

This section describes the basic framework and establishes necessary notations. They are based upon the notations and the general approach of Silver (2015) and Sutton and Barto (2018), but we deviate when useful to focus on necessary aspects regarding the algorithm and the IPD game.
The following framework is described through the lenses of an IPD game, where we take the role of one of the players, the so called *agent*. The strategy of the other player, the *opponent*, is fixed. The goal is to find a best response to the strategy of the opponent or at least a strategy which is performance-wise at about the same level.
Despite framing the examples with the help of the IPD game, it should be kept in mind, that the framework is a lot more flexible. It can be used for other games as well, as long as the basic requirements which are presented in this section are met. Without changes to the algorithm itself it is for example possible to use this framework for the optimization against an environment without a dedicated opponent (e.g. a multi-armed bandit problem) or for a game with multiple opponents who all have fixed strategies (e.g. an IPD game tournament). In the spirit of the IPD game of section 2.1.1 however, we will use notations which imply a single opponent which always plays as player 2.
Assume a progression of *state spaces* $\mathcal{S}_t,\ t = 1, ..., \infty$ with $\mathcal{S}_t$ being finite for each fixed *period* t. In each *state* $S_t \in \mathcal{S}_t$ the agent has to choose an *action* $A_t \in \mathcal{A}$ out of a finite *action space* $\mathcal{A}$. Afterwards the agent transitions to a new state $S_{t+1} \in \mathcal{S}_{t+1}$ into the next period. Which state S_{t+1} is realized depends on the realization of state S_t, the chosen action A_t and the rules of the underlying game, which might allow non-deterministic state changes.
The union $\bigcup_{t=1}^{\infty} \mathcal{S}_t$ is the set of all possible states. In the case of an IPD game as defined in Section 2.1.1 this union can be infinite in the case of some strategies which consider the complete history of the game, e.g. *net.nice0*. Given our goal, this is of no relevant concern:
We are interested in the average expected reward of a given strategy $\bar{R}(\mathbf{s})$ (see Equation 1) which is calculated using discount factor $\gamma < 1$. The game of interest has a bounded payoff matrix (see Table 1) which determines the reward of each period. Due to the cumulative discounting of the rewards, the relevance of future rewards vanishes with increasing number of periods. We can thus always find some $T_{\max} \in \mathbb{N}$ where the maximum impact of the possible rewards r_t for all $t > T_{\max}$, even in aggregate, falls below some given ε-threshold. For all practical purposes we can therefore ignore sufficiently high t.
It should be kept in mind, that the strategy of the opponent is fixed. Having a $T_{\max}$ does thus not imply, that backwards induction leads to never ending defections. This would be the case, if we allowed both players to iteratively optimize against each other.

We define

$$\mathcal{S} := \bigcup_{t=1}^{T_{\max}} \mathcal{S}_t$$

To simplify notations, we will use this finite $\mathcal{S}$ while still considering games with infinitely many periods. To achieve this formally, one slightly changes the game to have a new special state S^*. If the rules of the original game stipulate a transition to a state which is not part of $\mathcal{S}$, the resulting state is set to S^*. The special state S^* thus plays the role of a *game end state*. Once the game end state is reached, the game never leaves this state and no further rewards are acquired. The same idea can be used to model games with a finite, but undetermined number of periods. If one for example considers an IPD game which ends with a certain probability after a played period, the ending of the game is defined by a transition to the game end state. All definitions within this section can easily be extended to incorporate S^*, but we will refrain from doing so to increase readability and to keep the focus on core concepts.
Within the context of the IPD game it holds that

$$\mathcal{A} = \{\mathit{Cooperate}\ ,\ \mathit{Defect}\} = \{C, D\}$$

Regarding the states and state spaces of the IPD game one can take one of two stances:

1. If the strategy of the opponent is known and this strategy only acts upon publicly available information, it is possible to construct a state space based on its internal evaluations.
   Given that the opponent does not take into account the history of the game at all and acts consistently across all periods (e.g. *always.cooperate*, *always.defect* or *random.action*), it is only necessary to have one default state. One of the actions in $\mathcal{A}$ is always optimal and should be repeated. To find a best response it is thus sufficient to find this optimal action.
   If playing against an consistent opponent which only relies on observations of the single last period, as e.g. *tit.for.tat*, the state space can be reduced to the following five states: "No information available", (C,C), (C,D), (D,C), (D,D).[2]
   In the case of *tit.for.tat*, which does not take into account the own action of the last period, a further reduction is possible, as (D,C) and (D,D) may be combined to a "D" state. Here it should be noted that the opponent, in our case player 2, determines the state space, but all states are writing from the perspective of player 1.

[2]That the player with the shortest memory dictates the "rules of the game" has been used by Press and Dyson (2012) to develop their famous Zero-Determinant strategies. Here one may also find the proof of this proposition in a more general setting.

In the case of consistent multi-period memory strategies it is sufficient to accommodate only the combinations of the respective number of periods. When reducing the state-space even further, it is not sufficient to only consider states which are explicitly used by the opponent strategy for its choice of action. A good example for this effect may be found with strategy *tft.2forgive* (see Section 2.1.2), where our state space $\mathcal{S}$ may be reduced to the following states:

"no.info" - *tft.2forgive* has no observations to rely upon, i.e. the very first period.

"C" - *tft.2forgive* observes a cooperation of its opponent in the previous period, but not in the period before the previous one. Here it doesn't matter, whether this period does not exist (i.e. we are in the second period) or a defection has been observed.

"D" - *tft.2forgive* observes a defection of its opponent in the previous period and not only cooperations in the two periods before that. Here it doesn't matter, whether these periods do not exist (i.e. we are in the second or third period) or another defection has been observed.

"CC" - *tft.2forgive* observes two consecutive cooperations of its opponent in the two previous periods. This state is also used in the case of more than two consecutively observed cooperations in the previous periods.

"CCD" - *tft.2forgive* observes the actions "CCD" by its opponent in this particular order in the three previous periods.

Here, *tft.2forgive* does not explicitly check for a "CC" state. From its point of view the situation "CC" is identical to the situation "C". However, the state is still necessary to act as a bridge to the state "CCD". Using just states, we have to be able to exhaustively graph the structure of the game and capture all relevant information.

2. Alternatively one can take the black box approach and define the state space $\mathcal{S}_t$ based on all available information to us at time t, i.e the publicly available history of the episode based on the observations of the actions up to time $t-1$ and the privately known information of the own actions up to time $t-1$. With this approach the size of $\mathcal{S}_t$ necessarily explodes with t approaching infinity and the end game state S^* is needed.
   The main advantage of this approach is, that it is not necessary to analyze the opponent strategy by hand. Consequently, we use this interpretation of states for the algorithm of Section 3. The obvious problem however is, that significantly more data is needed to find a best response, as one has to find the optimal action for each of those states.

There exist methods which explicitly deal with the fact, that the opponent might react to private information which is not inferable by us. In this case it is possible to extend the state space by separating between a *true states* and *observed states*. The algorithm presented in Section 3 does not do that. Instead, it treats the *observed state*

as the *true state* within a stochastic environment. As long as all available information is used, this approach still infers the optimal actions given that information. A more detailed discussion of the non-observability of states can be found in Appendix B.
The transition between states is modelled based on the state transition function $\mathcal{P}$ with

$$\mathcal{P}(s, a, s') := \mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a]$$

denoting the probability to transition to state s' if the agent is in state s and takes action a. This function is defined by the rules of the game, e.g. the observation error probabilities and the strategy of the opponent.
By design all states S_t are *Markov*, i.e.

$$\begin{aligned}&\mathbb{P}[S_{t+1} = s' | S_t = s, A_t = a] \\ &\quad = \mathbb{P}[S_{t+1} = s' | S_t = s, S_{t-1} = s_{t-1}, ..., S_1 = s_1, A_t = a] \\ &\qquad \forall s_{t-1}, ..., s_1 \in \mathcal{S}_{t-1}, ..., \mathcal{S}_1\end{aligned}$$

In other words the transition probability does not depend on former states but only on the current state s.
Given the black box approach of states which model the complete history of the episode, the following effects have to be considered in contrast to having a well-defined small state space for a strategy:

1. Methods which rely strongly on the Markov Property (i.e. Q-Learning, Section 2.3.3) do not necessarily work very well. For any feasible number of simulations most states will not be visited at all. Even in the case of deterministic strategies observation errors result in different observed paths and therefore different states.

2. Approximating states by using only part of the history (e.g. the last three periods as done by Harper et al. (2017)) can make the underlying math much more complicated and inconsistent to the algorithms discussed here due to losing the Markov property. In fact, one can show, that the resulting strategy does not have to converge to the optimal one (see Appendix B for an example in a similar situation).

While switching the state the agent receives a to be discounted and potentially stochastic reward $R_t \in \mathbb{R}$. Given the IPD game as defined by section 2.1.1 the reward function $\mathcal{R}(s, a)$ is defined as

$$\mathcal{R}(s, a) := \mathbb{E}[R_t | S_t = s, A_t = a] = \sum_{a_2} P_2(a_2|s) g(a, a_2) \; \forall \; s \in \mathcal{S}$$

$\mathcal{R}(s, a)$ is the expected reward given action a and starting state s. $P_2(a_2|s)$ in this case is the probability that the opponent plays action a_2 given state s and $g(a, a_2)$ is the reward of the combination of action a by the agent and action a_2 by the opponent according to the payoff matrix depicted in Table 1.

To reflect that future rewards might be not as important or that they might not happen due to the game ending, a discount factor $\gamma \in [0, 1]$ can be used. In the case of the IPD game, this discount factor is identical to the γ introduced in Section 2.1.1. Generally speaking, the main motivation to use a discount factor γ is to find meaningful good policies in non-ending games (Sutton and Barto, 2018, p. 55).
A policy π is defined as the distribution

$$\pi(a|s) = \mathbb{P}[A_t = a|S_t = s]$$

giving the probability that action a is chosen given state s. Following, the wording *policy* is used to describe the strategy we want to generate using machine learning methods, while the term *strategy* stays for the, possibly handcrafted, strategy of the opponent. Despite this differentiation to increase readability, both strategy and policy are interchangeable concepts.
The policy is time-independent per design, i.e.

$$A_t \sim \pi(\cdot|S_t), \quad \forall t > 0$$

As the opponent strategy might react time-dependent, it might be necessary to use time-specific information such as the number of the current period when deciding on an action to find a best response. In this case the time-specific information has to be already incorporated into the state space.
The tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ describes a so called *Markov Decision Process* (MDP). A key feature of an MDP is, that the transition probability and the reward only depend on a fixed probability distribution based on the state and the chosen action. It is thus sufficient for any optimizing agent to know its current state to determine an optimal action.
We define $a_\pi(s)$ as the action which is played in state s given policy π, which may or may not be stochastic. To simplify notations the definition of the reward function $\mathcal{R}$ and the transition probability $\mathcal{P}$ are extended and redefined for stochastic $a_\pi(s)$ to

$$\begin{aligned} \mathcal{R}(s, a_\pi(s)) &:= \sum_{a \in \mathcal{A}} \pi(a|s)\mathbb{E}[R_t|S_t = s, A_t = a] \\ \mathcal{P}(s, a_\pi(s), s') &:= \sum_{a \in \mathcal{A}} \pi(a|s)\mathbb{P}[S_{t+1} = s'|S_t = s, A_t = a] \end{aligned}$$

In the case of π being a deterministic policy, this definition is consistent to the previous one.
To directly incorporate the policy, we write

$$\begin{aligned} \mathcal{R}_\pi(s) &:= \mathcal{R}(s, a_\pi(s)) \\ \mathcal{P}_\pi(s, s') &:= \mathcal{P}(s, a_\pi(s), s') \end{aligned}$$

The term $\mathcal{R}_\pi(s)$ is the expected reward if starting from state s and acting according to policy π. Correspondingly, $\mathcal{P}_\pi(s, s')$ is the transition probability from state s to state s' if acting according to policy π

The tuple $\langle \mathcal{S}, \mathcal{P}_\pi, \mathcal{R}_\pi, \gamma \rangle$ describes a so called *Markov Reward Process* (MRP). In contrast to a Markov Decision Process the actions will be taken by a specific policy are already incorporated. An MRP can therefore be used to evaluate and categorize the results and the quality of a given policy.
The goal is to find a policy which maximizes the sum of discounted future rewards. An intuitive idea to try to maximize those rewards is to use the action which maximizes $\mathcal{R}(s,a)$ in each state, i.e.

$$\pi_{\max}(a|s) := \begin{cases} 1 & \text{if } a = \underset{a \in \mathcal{A}}{\operatorname{argmax}} \, \mathcal{R}(s,a) \\ 0 & \text{otherwise} \end{cases}$$

or in other words

$$a_{\pi\text{-max}}(s) = \underset{a}{\operatorname{argmax}} \sum_{a_2} P_2(a_2|s) g(a, a_2)$$

This method results in a defection in the IPD game for all periods. While being the rational choice in the non-iterated Prisoner's Dilemma (i.e. $\gamma = 0$), this might be sub-optimal given a sufficiently high importance of later periods (i.e. $\gamma \gg 0$).
Take for example the strategy *grim.trigger* (see Section 2.1.2) in an IPD game without observation errors but according to the payoff matrix of Table 1. We start in the first period. Here, *grim.trigger* cooperates. If $\gamma = 0$ a defection from the agent results in a reward of 2, i.e. one point higher than with a cooperation. If $\gamma > 0$, each period after this will at best (i.e. the agent continues to defect) result in an average reward of zero. Should the agent instead opt for a cooperation in the first period and each one afterwards, it receives a to be discounted reward of unity for all eternity. Comparing those two policies it holds that

$$\begin{aligned} & \bar{R}(\mathit{always.defect}) = (1-\gamma) \cdot 2 \geq (1-\gamma) \sum_{t=1}^{\infty} \gamma^{t-1} \cdot 1 = \bar{R}(\mathit{always.cooperate}) \\ \Leftrightarrow \quad & (1-\gamma) \cdot 2 \geq 1 \end{aligned}$$

In this case we see, that with $\gamma > \frac{1}{2}$ the myopic option *always.defect* of choosing the best reward of the very next period is the worse option.
While more immediate periods are more important, it is still necessary consider the long lasting effects of decisions. In particular it might be prudent for the policy of the agent to maneuver the opponent into a situation where it is possible to extract higher rewards without being punished. To maximize the rewards it is thus necessary to keep in mind the state changes which may be influenced by the policy.
The *state-value function* $v_\pi : \mathcal{S} \to \mathbb{R}$ is defined as

$$v_\pi(s) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k} | S_t = s \right]$$

The state-value function measures the expected sum of discounted rewards given a specific policy starting from state s and starting the discounting at this point.

It holds that

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k}|S_t = s\right] \\ &= \mathbb{E}_\pi\left[R_t + \gamma\sum_{k=0}^{\infty}\gamma^k R_{(t+1)+k}|S_t = s\right] \\ &= \mathbb{E}_\pi\left[R_t + \gamma v_\pi(S_{t+1})|S_t = s\right] \\ &= \mathcal{R}_\pi(s) + \gamma\sum_{s'\in\mathcal{S}}\mathcal{P}_\pi(s,s')v_\pi(s') \end{aligned}$$

or short

$$v_\pi = \mathcal{R}_\pi + \gamma\mathcal{P}_\pi v_\pi \tag{3}$$

as $\mathcal{R}_\pi(s)$ and $\mathcal{P}_\pi(s,s')$ can be written as a vector/a matrix due to the finite structure of $\mathcal{S}$. This so called *Bellman equation for* v_π allows for an explicit solution.
By allowing to deviate from the policy in the very first step the *action-value function* is given by

$$q_\pi(s,a) := \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+k}|S_t = s, A_t = a\right]$$

The function $q_\pi(s,a)$ measures the value of a state, given that the agent will chose action a at the present decision and continue afterwards according to policy π.
The relationship between state-value function v_π and action-value function q_π is given by

$$v_\pi(s) = \sum_{a\in\mathcal{A}}\pi(a|s)q_\pi(s,a)$$

This means, that we can calculate the value of a state s under policy π by calculating all action values of this state and weighting them by the probability, that the specific action will be chosen under policy π. Analogously to the definitions above we can use the notation

$$v_\pi(s) = q_\pi(s, a_\pi(s))$$

We can now write the *Bellman equation for* q_π:

$$\begin{aligned} q_\pi(s,a) &= \mathcal{R}(s,a) + \gamma\sum_{s'\in\mathcal{S}}\mathcal{P}(s,a,s')v_\pi(s') \\ &= \mathcal{R}(s,a) + \gamma\sum_{s'\in\mathcal{S}}\mathcal{P}(s,a,s')q_\pi(s', a_\pi(s')) \end{aligned} \tag{4}$$

Transferring these concepts to the example of the IPD game, we write the problem in a more concise fashion. By definition, as no previous history is known our policy always starts with the same "no.info" state s_1. It holds for each policy π, that

$$\bar{R}(\pi) = (1-\gamma)v_\pi(s_1) = (1-\gamma)q_\pi(s_1, a_\pi(s_1))$$

Having calculated $v_\pi(s_1)$ for some policy π it is therefore straightforward to calculate the average expected reward of a given policy in the IPD game. As a direct consequence we have an easy metric to compare different policies to each other by comparing their $v_\pi(s_1)$-values.
More generally speaking, two different problems are of interest when confronted with a MDP:

1. Find values q_π for each state such that all those values are consistent with each other according to Equation 4. [Evaluation]
2. Find the optimal policy π^* which maximizes those values for each state. [Optimization]

To further specify and later on find a solution to the second problem, the *optimal action-value function* $q_*(s,a)$ is defined as

$$q_*(s,a) = \max_\pi q_\pi(s,a) \quad \forall s \in \mathcal{S},\, a \in \mathcal{A}$$

The relation

$$\pi \geq \pi'$$

is defined to hold if it holds that

$$q_\pi(s, a_\pi(s)) \geq q_{\pi'}(s, a_{\pi'}(s)) \quad \forall s \in \mathcal{S}$$

In this spirit policy π is said to be better than policy π' if for every state at least the value of policy π' could be achieved and there exists at least one state s where it holds that $q_\pi(s, a_\pi(s)) > q_{\pi'}(s, a_{\pi'}(s))$.
It holds that (Silver, 2015, chapter 2, p. 43):

Theorem 1 For any Markov Decision Process

- There exists an optimal policy π_* that is better than or equal to all other policies, $\pi_* \geq \pi, \forall \pi$
- All optimal policies achieve the optimal action-value function, $q_{\pi_*}(s,a) = q_*(s,a) \quad \forall s \in \mathcal{S},\, a \in \mathcal{A}$

Of course, depending on the problem, π^* does not have to be unique and there might actually be vastly different optimal policies achieving the same result. For simplicity sake we will nevertheless only talk about *the* optimal policy π^* as we are mainly interested in the value $\bar{R}(\pi)$ and not in finding all possible optimal policies. The machine learning concept of π^* and the game theoretic concept of a best response can be used interchangeably in the context and the scope of this article.
Given that $q_{\pi_*}(s,a)$ is known it is straightforward to construct a deterministic optimal policy by

$$\pi_*(a|s) = \begin{cases} 1 & a = \underset{a' \in \mathcal{A}}{\operatorname{argmax}}\, q_*(s,a') \\ 0 & \text{otherwise} \end{cases}$$

The basic idea here is that the best possible action in a given state is to chose the action which maximizes the reward under the assumption that we continue to act optimal. Should there be several optimal actions one may chose any of those.
Given that the agent fully observes in which state it is, it is always possible to find a deterministic optimal policy. One can nevertheless design a stochastic optimal policy, by giving all optimal actions a positive probability and all non-optimal actions a probability of zero (Sutton and Barto, 2018, p. 64).

2.3. Reinforcement Learning

2.3.1. Dynamic Programming

Dynamic programming is an umbrella term summarizing a class of algorithms which are used to find "optimal policies given a perfect model of the environment as a Markov decision process" (Sutton and Barto, 2018, p. 59) and was initially developed by Richard E. Bellman in the 1950er years [3]. The basic idea is to compute intermediate steps, using the different Bellman equations (see Section 2.2) to find the value of each state or state-action pair given a specific policy and use those to iteratively determine the best possible policy. How optimal policies can be found without a perfect model of the environment is presented in Section 2.3.2.
For small problems one can solve Equation 3 directly:

$$\begin{aligned} v_\pi &= \mathcal{R}_\pi + \gamma \mathcal{P}_\pi v_\pi \\ v_\pi - \gamma \mathcal{P}_\pi v_\pi &= \mathcal{R}_\pi \\ (I - \gamma \mathcal{P}_\pi) v_\pi &= \mathcal{R}_\pi \\ v_\pi &= (I - \gamma \mathcal{P}_\pi)^{-1} \mathcal{R}_\pi \end{aligned} \tag{5}$$

Here a $\gamma < 1$ ensures that $(I - \gamma \mathcal{P}_\pi)^{-1}$ exists and that every state is valued with a finite number.
There exist two main directions to achieve optimal play:

1. *Policy iteration* - One tries different policies in a way, that they converge towards the best possible policy. To do so it is helpful to be able to evaluate a given policy, e.g. through the direct method of Equation 5.
2. *Value iteration* - One knows the solution to some sub-problems $v_*(s')$ and tries to make them compatible and to find solutions $v_*(s)$ for states s which are able to reach s'. Afterwards an optimal policy can be constructed by always playing the best possible action.

While the first direction corresponds to a more intuitive way a human might approach a problem ('Lets try something and if it works do something similar'), the second direction corresponds to a more structured approach not unlike the basic strategy of backwards induction in game theory.

[3]See Bellman (2003) for the updated version of his book about dynamic programming

Policy Iteration Given that the Markov Decision Problem $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ is known, one algorithm to iterate the policy is (adapted from Silver (2015, chapter 3, p. 12)):

```
Generate some policy $\pi$ by generating a distribution $\pi(a|s)$ for each state $s \in \mathcal{S}$.
Repeat until stop condition $\pi$-good:
    Calculate $q_\pi(s, a_\pi(s))$ for each $s \in \mathcal{S}$ # Evaluation step
    $q_\pi(s,a) \leftarrow \mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s,a,s') q_\pi(s', a_\pi(s'))$
    Redefine $\pi$ by setting $\pi(a|s) = \begin{cases} 1 & a = \operatorname{argmax}_{a' \in \mathcal{A}} q_\pi(s,a') \\ 0 & \text{otherwise} \end{cases}$
End Repeat
```

The condition π-good can be set to be a certain number of iterations or be based on an ε-criterion, i.e. stopping as soon as the policy improvements start to become sufficiently small. It can be shown, that by increasing the number of repetitions to infinity the policy converges to the optimal policy π^* (Silver, 2015, chapter 3, p. 12). This holds independent from the choice of the initial strategy π.
There are several reasons, why calculating $q_\pi(s, a_\pi(s))$ can prove troublesome in a practical setting even though $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$ is known. For small state spaces Equation 5 using identity $q_\pi(s, a_\pi(s)) = v_\pi(s)$ can be used. If the state space is too large however, the more direct approach

$$q_\pi(s, a_\pi(s)) = \mathbb{E}_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k} | S_t = s \right]$$

has to be used. Even given the MDP is completely known, calculating R_i for big i can be very complicated and numerically unstable due to the game branching out. To circumvent this, it is possible to use Bellman Equation 4 for an iterative calculation:

```
Initialize values $q_0(s, a_\pi(s)) \equiv 0 \quad \forall s$
$k \leftarrow 0$
Repeat until stop condition $q$-good:
    $q_{k+1}(s, a_\pi(s)) = \mathcal{R}(s, a_\pi(s)) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s, a_\pi(s), s') q_k(s', a_\pi(s')) \quad \forall s \in \mathcal{S}$
End Repeat
```

For k to infinity q_k converges to q_π (Sutton and Barto, 2018, p. 65). The condition *q-good* might be set again to a certain number of steps or with an ε-criterion.
To summarize, with policy iteration one starts with some arbitrary policy and calculates, possibly using an iterative approach, the value of each state given that we continue with this policy. Based on this the policy is improved. This is repeated until a satisfying result is achieved.

Value Iteration Bellmans Principle of Optimality (Bellman, 1957, paraphrased) tells us:

Theorem 2 A policy $\pi(a|s)$ achieves the optimal value from state s, $v_\pi(s) = v_*(s)$, if and only if for any state s' reachable from s π achieves the optimal value from state s', $v_\pi(s') = v_*(s')$

Analogously to backwards induction it is therefore possible to deduce the optimal state-value $v_*(s)$ and a corresponding policy if the optimal state-values of all reachable states are known. If the game has terminal states, i.e. game end states, than these are already known and can be used as a basis for subsequent iterative calculations

$$v_*(s) = \max_{a \in \mathcal{A}} \left\{ \mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s,a,s') v_*(s') \right\}$$

If no terminal states exist, for example due to loops between the state in an infinite game, this principle can't directly be used. However, similar to the policy evaluation step of the policy iteration it is possible to iterate through the values using

$$v_{k+1}(s) = \max_{a \in \mathcal{A}} \left\{ \mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s,a,s') v_k(s') \right\} \quad \forall s \in \mathcal{S}$$

Instead of trying to evaluate a specific policy π, calculating q_π values and acting greedy afterwards, we assume that our policy is greedy all the time. This way it is possible to ignore the actual policy iteration and generate the policy based on converged state-values v_k directly. The algorithm is (Sutton and Barto, 2018, p. 67):

```
Initialize values $v_0(s) \equiv 0 \quad \forall s$
$k \leftarrow 0$
**Repeat until** stop condition $v$-*good*:
    $v_{k+1}(s) = \max_{a \in \mathcal{A}} \left\{ \mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s,a,s') v_k(s') \right\} \quad \forall s \in \mathcal{S}$
**End Repeat**
# $k_n$ is the index of the most recent value iteration.
$\pi(s|a) = \begin{cases} 1 & a = \operatorname{argmax}_{a' \in \mathcal{A}} \left\{ \mathcal{R}(s,a') + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s,a',s') v_{k_n}(s') \right\} \\ 0 & \text{otherwise} \end{cases} \quad \forall s \in \mathcal{S}$
```

Example given a strategy To illustrate dynamic programming in the IPD game setting of Section 2.1.1 we use policy iteration on the strategy *tft.2forgive*. We show that the intuitive candidate for a best response from Section 2.1.2 is indeed correct. It is to show, that $\pi =$ "Cooperate unless the opponent observed me playing two cooperations, then defect" is the best possible policy. To be more precise, as policy iteration converges toward the optimal policy π^*, it is sufficient to show that the resulting policy $\pi' = \pi$, that i.e. the policy is already fully converged and an additional step does not change policy π.
Recall from section 2.2, that *tft.2forgive* may be reduced to the states "no.info", "C", "D", "CC" and "CCD". As a first step it is necessary to calculate $q_\pi(s, a_\pi(s)) = v_\pi(s)$ for all these states. Equation 5 states that

$$v_\pi = (I - \gamma \mathcal{P}^\pi)^{-1} \mathcal{R}^\pi$$

Due to the structure of the game and given strategy π it holds for transition matrix $\mathcal{P}_\pi$ that

$$\mathcal{P}_\pi = \begin{array}{c} \\ \text{no.info} \\ \text{C} \\ \text{D} \\ \text{CC} \\ \text{CCD} \end{array} \begin{array}{c} \begin{array}{ccccc} \text{no.info} & \text{C} & \text{D} & \text{CC} & \text{CCD} \end{array} \\ \begin{pmatrix} 0 & 1-\text{err}_D & \text{err}_D & 0 & 0 \\ 0 & 0 & \text{err}_D & 1-\text{err}_D & 0 \\ 0 & 1-\text{err}_D & \text{err}_D & 0 & 0 \\ 0 & 0 & 0 & \text{err}_C & 1-\text{err}_C \\ 0 & 1-\text{err}_D & \text{err}_D & 0 & 0 \end{pmatrix} \end{array}$$

and

$$\mathcal{R}_\pi = \begin{array}{c} \text{no.info} \\ \text{C} \\ \text{D} \\ \text{CC} \\ \text{CCD} \end{array} \begin{pmatrix} 1 \\ 1 \\ -1 \\ 2 \\ 1 \end{pmatrix}$$

Using $\text{err}_D = 0.15$, $\text{err}_C = 0$ and $\gamma = 0.95$ it holds that

$$v_\pi = (I - \gamma \mathcal{P}_\pi)^{-1} \mathcal{R}_\pi \approx \begin{array}{c} \text{no.info} \\ \text{C} \\ \text{D} \\ \text{CC} \\ \text{CCD} \end{array} \begin{pmatrix} 21.11 \\ 21.54 \\ 19.11 \\ 22.06 \\ 21.11 \end{pmatrix}$$

Now, using the identity $q_\pi(s', a_\pi(s')) = v_\pi(s')$, we calculate $q_\pi(s, a)$

$$q_\pi(s, a) \leftarrow \mathcal{R}(s, a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s, a, s') q_\pi(s', a_\pi(s'))$$

It holds that

$$q_\pi(s, \text{Cooperate}) \approx \begin{array}{c} \text{no.info} \\ \text{C} \\ \text{D} \\ \text{CC} \\ \text{CCD} \end{array} \begin{pmatrix} \mathbf{21.11} \\ \mathbf{21.54} \\ \mathbf{19.11} \\ 21.82 \\ \mathbf{21.11} \end{pmatrix}; \; q_\pi(s, \text{Defect}) = \begin{array}{c} \text{no.info} \\ \text{C} \\ \text{D} \\ \text{CC} \\ \text{CCD} \end{array} \begin{pmatrix} 20.16 \\ 20.16 \\ 18.16 \\ \mathbf{22.06} \\ 20.16 \end{pmatrix}$$

Updating the strategy so that the maximum value $q_\pi(s, a)$ is chosen gives us

$$\pi' \begin{pmatrix} \text{no.info} \\ \text{C} \\ \text{D} \\ \text{CC} \\ \text{CCD} \end{pmatrix} = \underset{a \in \mathcal{A}}{\text{argmax}} \; q_\pi(s, a) = \begin{pmatrix} \text{Cooperate} \\ \text{Cooperate} \\ \text{Cooperate} \\ \text{Defect} \\ \text{Cooperate} \end{pmatrix}$$

which is identical to our starting policy π: The policy iteration has already converged and we have shown that π is already a best response. It holds that $\pi = \mathbf{s}^*$, using the notation from section 2.1.1.
With these results it is straightforward to calculate the expected average reward, given the best response to the strategy *tft.2forgive*:

$$\begin{aligned} \bar{R}(\mathbf{s}^*) &= (1 - \gamma) q_\pi(s_1, a_\pi(s_1)) \\ &= (1 - \gamma) q_\pi(\text{no.info}, a_\pi(\text{no.info})) \\ &= (1 - \gamma) q_\pi(\text{no.info}, \text{“C”}) = 1.055 \end{aligned}$$

2.3.2. Model-Free Reinforcement Learning

Both, policy iteration and value iteration, as presented in Section 2.3.1 use the full knowledge of the MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle$. This holds true even when using the iterative approach to find state or state-action values instead of the direct calculation method of Equation 5. In most practical situations and indeed in our case however neither transition matrix $\mathcal{P}$ nor reward function $\mathcal{R}$ are known. Given our specific example of the IPD game the rules of the game are known, but the actual rewards and transition probabilities depend on the opponent. A calculation as seen in Section 2.3.1 for *tft.2forgive* can only be made for the most simple strategies with reasonable effort. Even if the strategy of the opponent is explicitly given, it is therefore often unfeasible to calculate $\mathcal{P}$ and $\mathcal{R}$ explicitly.
Model-free reinforcement learning does not depend on knowing $\mathcal{P}$ and $\mathcal{R}$ but rather approximates them based on collected data. We will concentrate here on two different methods: *Monte Carlo learning* and *One-step temporal difference learning*. Monte Carlo learning provides an important aspect of the algorithm of Section 3. One-step

temporal difference learning on the other hand is an important predecessor of the ideas behind Q-Learning, which is presented in Section 2.3.3.
First we discuss how these methods can be used to estimate the value function of such an unknown MDP and afterwards we show the optimization and construction of a resulting policy.

Model-Free Evaluation The goal of this subsection is to find a way to accurately approximate $q(s, a_\pi(s))$ even in the absence of knowledge about $\mathcal{P}_\pi$ and $\mathcal{R}_\pi$. This allows us to compare two given strategies, but does not allow us to construct a new, improved strategy. To improve a strategy, e.g. analogously to policy iteration, it is necessary to evaluate alternative actions. Here, we calculate $q(s, a_\pi(s))$ for fixed policies π and not $q(s, a)$ for arbitrary actions a. It is therefore not possible to directly apply the same methods as before to find a best policy, even if $q(s, a_\pi(s))$ is known.
Both, Monte Carlo learning (MC) and Temporal Difference learning (TD), generate data by playing the game and estimate state values $q(s, a_\pi(s)) = v_\pi$. The difference is that MC learning uses the data of a complete game directly. TD learning on the other hand uses possibly limited number of state transitions. In particular, the last transition does not have to be to an end-state. One-step TD learning for example uses the one step from one state to its very next successor to estimate v_π.

Monte-Carlo Learning Monte-Carlo methods are known since the 1940s and exist in a wide variety of versions (Sutton and Barto, 2018, p. 76). We concentrate on an archetype to generate an understanding of the core principles. For more details the reader is referred to Sutton and Barto (2018).
The basic idea behind Monte-Carlo reinforcement learning is that a game is played from the beginning to an end period T and the actual historic rewards r_t are used to estimate the state values $q(s, a_\pi(s))$. Such a complete game run is called an *episode*, in line with the definition of the term in Section 2.1.1. One can use multiple episodes with the same starting state s to get more accurate results. In this case the rewards are aggregated by averaging them to approximate an expected value:

$$q(s_t, a_\pi(s_t)) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{k=0}^{T-t} \gamma^k r_{t+k}^{(i)} \tag{6}$$

Here, $r_{t+k}^{(i)}$ is the historically observed reward in period $t+k$ and episode i with observed state s_t in period t and acting according to the to be evaluated policy. With N approaching infinity, i.e. infinitely many episodes, this construct converges to the true value of $q(s_t, a_\pi(s_t))$ (Sutton and Barto, 2018, p. 76).
With games which do not have a natural end, e.g. the IPD game as introduced in Section 2.1.1, MC learning out of the box is not possible. We mitigate this problem by using an artificial maximal period $T_{\max}$ after which the game is aborted, as discussed in Section 2.2. It should be kept in mind that we are mainly interested

in $q(s_1, a_\pi(s_1))$, i.e. correctly assessing the starting state, to calculate the expected average reward $\bar{R}(\pi)$ for a specific strategy. $T_{\max}$ can thus be chosen by considering the discounting effect of γ, the payoff matrix and the wanted precision of $\bar{R}(\pi)$. Having artificially shortened the game, we use $T_{\max}$ as T.
MC can be implemented the following way (see Silver (2015, chapter 4, p. 11)):

```
Initialize $V(s) \equiv 0 \quad \forall s$ # $V(s)$ shall converge to the value $v_\pi(s) = q(s, a_\pi(s))$
Initialize $N(s) \equiv 0 \quad \forall s$ # $N(s)$ is the number of visits of state $s$
**Repeat until** stop condition $V$-*good*:
    Run an episode of the MDP and generate $s_1, a_1, r_1, ..., s_T$
    **For each** $s_t$, $t = 1, ..., T$:
        Calculate $G_t \leftarrow \sum_{k=0}^{T-t} \gamma^k r_{t+k}$
        $N(s_t) \leftarrow N(s_t) + 1$
        $V(s_t) \leftarrow (1 - \alpha(N(s_t)) \cdot V(s_t) + \alpha(N(s_t)) \cdot G_t$
    **End For each**
**End Repeat**
```

Given that $\mathcal{P}_\pi$ and $\mathcal{R}_\pi$ stay constant, a natural way to choose the function α is as $\alpha(N(s_t)) = \frac{1}{N(s_t)}$. This way the correct mean is generated incrementally (compare Silver (2015, chapter 4, p. 10)) and the idea behind Equation 6 is achieved.
V-good can be set via a fixed, sufficiently large number of iterations or with an ε-criterion, where the algorithm stops when the change in $V(s_t)$ is only miniscule.
The problem becomes more complex, as soon as $\mathcal{P}_\pi$ and $\mathcal{R}_\pi$ do not stay constant. They might for example change if we want to optimize, and therefore change, π as an additional step within this algorithm. In this case it is necessary to forget old episodes - ideally in a way, that no complex calculations are necessary. Assume that strategy π is gradually changed to strategy π' and that there is a reasonably smooth underlying problem. In this case one should expect that more recent rewards sequences are more useful to estimate the desired $v_{\pi'}(s)$ values. Older episodes however might still hold some information. It is thus prudent to save computation resources and iterations by using old rewards, but one does not want them to dominate more recent data.
One way to achieve a continuous update in the face of changing $\mathcal{P}_\pi$ and $\mathcal{R}_\pi$ is to use $\alpha(N(s_t)) \equiv \alpha$ for some reasonable value of $0 < \alpha \leq 1$. With this fixed α effectively running mean is tracked, even if this opens up the MC algorithm to some of the critiques discussed in the one-step difference learning section below.
With a higher value of α a greater importance is put on more recent episodes. Choosing an optimal value for α strongly depends on the change of pace of $\mathcal{P}_\pi$ and $\mathcal{R}_\pi$ as well as the underlying rate of randomness of the problem. With a higher rate of randomness one should chose a lower value for α as it is necessary to build enough data to accurately estimate the expected value. Otherwise one risks to overwrite the generated estimations with a particularly (un-)lucky reward sequence.

One-Step Temporal Difference Learning The early beginnings of temporal difference learning can be attributed to Samuel (1959), but a rich variety of different variants emerged since then. Similar to the approach with MC learning, we will concentrate on one specific form, one-step temporal difference learning, to explain the main features.
In contrast to MC, TD uses a bootstrapping like approach to calculate the state values $q(s, a_\pi(s))$ instead of solely relying on historic rewards. This allows TD to be used on incomplete episodes. It is therefore applicable for never-ending MDPs as well. Given a terminating game, some variants of TD incorporate information about the environment at run time without having to wait for the MDP to terminate.
As we do not necessarily wait for the game to terminate, it is not possible to draw on the rewards up to the end-state. It is however possible to use previously learned estimations $V(s_t)$. In the simplest variant of TD-Learning, so called TD(0) or one-step TD learning, only the very next state is considered. The update rule is

$$V(s_t) \leftarrow (1-\alpha)V(s_t) + \alpha\left(r_t + \gamma V(s_{t+1})\right)$$

Here $r_t + \gamma V(s_{t+1})$ replaces G_t of the MC algorithm and is called the *TD target*.
Despite these advantages, for simplicity sake we will now concentrate on an algorithm, where the learning takes place offline, i.e. not at run time. This is done for two reasons:

1. Having the game terminate similar to the MC algorithm allows for a better comparison and contrasting of both algorithms.
2. In the IPD game with the black box encoding there are by design no loops within our states, i.e. each state is only visited once each episode. It is thus preferable to run a lot of short episodes to running few (or one) very long one.

One possible implementation of an (offline) TD(0)-algorithm is (see Silver (2015, chapter 4, p. 13):

```
Initialize values $V(s) \equiv 0 \quad \forall s$ # $V(s)$ is the appr. of the value $v_\pi(s) = q(s, a_\pi(s))$
**Repeat until** stop condition $V$-*good*:
    Generate $s_1, a_1, r_1, ..., s_T$ by running the MDP until period $T$ or until the MDP terminates.
    **For each** $s_t$, $t = 1, ..., T$:
        $V(s_t) \leftarrow (1-\alpha)V(s_t) + \alpha\left(r_t + \gamma V(s_{t+1})\right))$
    **End For each**
**End Repeat**
```

In contrast to the MC algorithm there exist several possibilities why this TD algorithm in a general setting might not stop or fail to converge towards the correct values $q(s, a_\pi(s))$:

1. As TD learning is designed to work with non-terminating MDPs it is necessary to chose T exogeneously as it might not be inferred from experience. Choosing T too small might lead to the situation that certain states s are never experienced as they are only visited after a certain number of steps. It is thus necessary to choose T very carefully and reasonably high.
2. If the MDP is stochastic in nature, the update using the factor α leads to a change in $V(s_t)$ even if one assumes, that it already holds the correct value $V(s) = q(s, a_\pi(s))\ \forall s$ as the value of the specific reward r_t depends on a random draw. If *V-good* is thus an ε-criterion which has been set to narrow, the algorithm might never stop. This effect can be mitigated by choosing a smaller α or by depending *V-good* on a sufficiently large running mean of the $V(s_t)$ which negates the random fluctuations.
   Dayan and Sejnowski (1994) show for the more general TD(λ) algorithm, that using an decreasing α over the episodes with
   $$\sum_{i=1}^{\infty} \alpha_i = \infty \text{ and } \sum_{i=1}^{\infty} \alpha_i^2 < \infty$$
   solves this problem given a stationary MDP and achieves convergence of the TD algorithm. However, using a diminishing step update rate might prove problematic in the case of a changing, non-stationary MDP - which we have, as soon as we want to want to improve, i.e. change our policy π.

Silver (2015, chapter 4, p. 18) notes, that in contrast to the unbiased estimate G_t of the MC algorithm, the TD-algorithm uses with the TD target $r_t + \gamma V(s_{t+1})$ a biased estimator. The bias is introduced when the V-Values are initialized with some arbitrary values. While this bias vanishes with an increasing number of update steps given the here discussed tabular approach, the same does not necessarily hold true when using function approximators. Here some of those biases may persist or even lead to numerical instabilities.
Sutton and Barto (2018, p. 169) name as an advantage of TD learning over MC learning that a specific TD target shows a lower variance if a lower number of steps (e.g. only one, as discussed here) is considered. This stabilizes the learning, at least if there are loops across state transitions. Silver (2015, chapter 4, p. 18) summarizes that MC learning shows good convergence properties, works well with function approximation and is not as sensitive to the initial values, while TD is usually more efficient and makes much better use of the Markov property.
The classic Q-Learning algorithm draws strongly from the TD(0) algorithm. In the use case of the IPD game we effectively do not use the Markov property and depend on function approximators. Extending Q-Learning with an Monte Carlo element, as done in Section 3.4, consequently improves upon the classic Q-Learning approach in our setting.

Model-Free Optimization The evaluation step of policy iteration can be replaced by the algorithms discussed above. To improve the policy however, policy iteration uses $\mathcal{P}_\pi$ and $\mathcal{R}_\pi$ which are not necessarily known. Similarly, value iteration is not possible if $\mathcal{P}_\pi$ and $\mathcal{R}_\pi$ are unknown, as those are used in the calculation of the next iteration.
Both, MC learning and TD(0) learning, as far as discussed, are only able to evaluate a given, fixed policy. One could in theory generate all possible policies, evaluating all of them either with MC or TD(0) and choose the best one. This however is only a viable option in very limited environments. We want therefore to limit the policies which have to be evaluated or find an alternative approach to construct a good policy based on experienced episodes.
Based on the idea behind policy iteration one might have the idea to replace the policy improvement step from policy π to policy π'

$$\pi'(a|s) = \begin{cases} 1 & a = \underset{a' \in \mathcal{A}}{\operatorname{argmax}} \left\{ \mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{S}} \mathcal{P}(s,a,s') q_\pi(s', a_\pi(s')) \right\} \\ 0 & \text{otherwise} \end{cases}$$

with

$$\pi'(a|s) = \begin{cases} 1 & a = \underset{a' \in \mathcal{A}}{\operatorname{argmax}}\, Q_\pi(s,a') \\ 0 & \text{otherwise} \end{cases}$$

where $Q_\pi(s,a)$ are the empirically calculated values of the state-action values $q_\pi(s,a)$ based on an empirical valuation with MC or TD(0).
This proves troublesome as with this approach all generated policies π' are deterministic. This approach however assumes, that it is possible to calculate different $Q_\pi(s,a')$-Values to determine their maximum. After the first update only the same action will be taken given the same state s and alternative actions are never explored.
To complete these thoughts, it is necessary to define a policy which given enough episodes takes each action with non-zero probability, for example

$$\pi'(a|s) = \begin{cases} \frac{\varepsilon}{m} + 1 - \varepsilon & a = \underset{a' \in \mathcal{A}}{\operatorname{argmax}}\, Q_\pi(s,a') \\ \frac{\varepsilon}{m} & \text{otherwise} \end{cases}$$

These types of policies are called *ε-greedy.* Here, $\varepsilon \in (0,1]$ controls the probability to take the (assumed) optimal action. A value of $\varepsilon = 1$ implies complete randomness between all actions. Choosing $\varepsilon = 0$ on the other hand would imply that only the greedy action is taken and results in the problem discussed above. The value m is the number of actions which can be taken in this state. With a slight extension to the discussed framework it is possible to allow for games where the available actions differ between states. In this case it can hold that $m < |\mathcal{A}|$. In the case of the IPD game the rules of the game always allow both actions. It therefore holds that $m = |\mathcal{A}| = 2$.

Incorporating these elements, policy iteration is able to find the best possible ε-greedy strategy with fixed ε. Lowering ε slowly and gradually allows us to find the best deterministic strategy in the limit case (Silver, 2015, chapter 5, p. 16).
This leads us to the following Monte Carlo algorithm (Silver, 2015, chapter 5, p. 16):

```
Initialize values $Q(s,a) \equiv 0 \quad \forall s,a$; $N(s,a) \equiv 0 \quad \forall s,a$; $k = 0$
**Repeat until** stop condition $Q$-*good*:
    $k \leftarrow k+1$
    Generate $s_1, a_1, r_1, ..., s_T$ by running the MDP until period $T$ or until the
    MDP terminates.
    **For each** $s_t$ and $a_t$, $t = 1, ..., T$ in this episode:
        $N(s_t,a_t) \leftarrow N(s_t,a_t) + 1$
        $Q(s_t,a_t) \leftarrow \left(1 - \frac{1}{N(s_t,a_t)}\right) Q(s_t,a_t) + \frac{1}{N(s_t,a_t)} \sum_{k=0}^{T-t} \gamma^k r_{t+k}$
    **End For each**
    $\varepsilon \leftarrow \frac{1}{k}$
    $\pi'(s|a) = \begin{cases} \frac{\varepsilon}{m} + 1 - \varepsilon & a = \underset{a' \in \mathcal{A}}{\operatorname{argmax}}\, Q(s,a') \\ \frac{\varepsilon}{m} & \text{otherwise} \end{cases}$
**End Repeat**
```

This algorithm converges towards the optimal policy π^* (Silver, 2015, chapter 5, p. 16) despite not waiting until the policy iteration with fixed ε has converged.
Analogously one can generate an algorithm based on TD(0)-Learning to make use of the advantages of temporal difference learning. With the following algorithm (based on Sutton and Barto (2018, p. 106) with some small changes to keep consistency) it is for example not necessary to wait until the episode ends before updates are possible:

```
Initialize values $Q(s,a)$ $\forall s,a$.
**Repeat until** stop condition $Q$-*good*:
    Chose starting state $s$ of this episode # This is always the same state
    with the IPD game
    Choose action $a$ using a policy derived from Q (e.g. $\varepsilon$-greedy)
    **Repeat until final state is reached**:
        Take action $a$, observe $r$ and resulting state $s'$
        Choose action $a'$ from $s'$ using policy derived from Q (e.g. $\varepsilon$-greedy)
        $Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha\,(r + \gamma Q(s',a'))$
        $s \leftarrow s'$; $a \leftarrow a'$
**End Repeat**
```

As with TD(0) learning the factor γ is the discounting rate and α measures the learning rate. Using diminishing ε and α, analogously to TD(0) learning, this algorithm converges toward the optimal state-action values $q^*(s, a)$ (Silver, 2015, chapter 5, p. 26). Given these values, an optimal policy can be created.
This algorithm is known under the name *SARSA* (**S**tate-**A**ction-**R**eward-**S**tate-**A**ction) and has been developed by Rummery and Niranjan (1994). Despite the structure in this document, SARSA has been developed after the development of Q-Learning, which is discussed in the next section. The main advantage of SARSA over Q-Learning is that SARSA by design is more accurate in estimating the values of the current (ε-greedy) strategy instead of trying to calculate the state-action values of the optimal strategy directly. As an effect SARSA tends to be more robust in finding a good answer, but is not as suited to find the *optimal* answer.
Note, that the description of this algorithm may easily be changed to define a new policy with each step explicitly, as has been done with the Monte Carlo algorithm. We have refrained from doing so, as SARSA is not restricted to ε-greedy policies.

2.3.3. Q-Learning

Q-Learning is a reinforcement learning algorithm developed as a Ph.D. thesis by John Watkins (Watkins, 1989) whose convergence was proven by Watkins and Dayan (1992). To keep consistency with more recent literature and the rest of the article, we do not use the original definitions and notations, but the notations of Sutton and Barto (2018, p. 107) with some small changes:

```
Initialize values $Q(s,a) \quad \forall s, a$.
Repeat until stop condition $Q$-good:
    Chose starting state $s$ of this episode # This is always the same state with the IPD game
    Repeat until final state is reached:
        Choose action $a$ using a policy derived from Q (e.g. $\varepsilon$-greedy)
        Take action $a$, observe $r$ and resulting state $s'$
        $Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha \left(r + \gamma \max_{\hat{a}} Q(s', \hat{a})\right)$
        $s \leftarrow s'$
End Repeat
```

The main difference to SARSA is, that Q-Learning does not use the action which is used according to the current policy to update the Q-Values. Instead it assumes, that we will continue optimal from outgoing from the current state. While the Q-Values of SARSA approximate q_π according to the current policy, Q-Learning aims to approximate the optimal q^*-values directly. In fact, given enough data, this Q-Learning algorithm will converge for each $Q(s,a)$ to the optimal state-action values $q^*(s,a)$ (Sutton and Barto, 2018, p. 108). Having obtained them, it is possible to construct the best possible strategy π^*.
As the update rule does not depend on the current exploration but on the assumed optimal choice, Q-Learning does not require the current policy to converge towards the optimal policy. Using ε-greedy policies, all discussed algorithms up to this point required a vanishing ε. This is not the case with Q-Learning. Even using only uniformly random actions Q-Learning generates an optimal policy given sufficient iterations. In the words of Peng and Williams (1994): "Q-learning is not *experimentation-sensitive*".
Nevertheless it is sub-optimal to use uniformly random exploration when the state space is very large, like in our case with the IPD game. If it is not feasible to visit and process all state spaces, it is a prudent idea to start with a broad assessment of the state space and concentrate on the most promising actions. This way less time is spent on assessing low-value states and an optimum can be found faster. Depending on chance and the speed of the reduction of the exploration (e.g. the ε parameter) this might however only be a local optimum. Further discussions regarding optimal exploration can be found in Section 3.5 and Section 3.6.

2.3.4. Approximative Solutions

Up to this point all discussed algorithms have been so called *tabular solution methods*. Here the assumption is that it is possible to hold all information, e.g. all possible and visited states, the taken actions, subsequent rewards and successor states, in one single table. These methods provably produce the optimal policy π^*.
In a lot of practical problems, and indeed given the IPD game as discussed in Section 2.1, this tabular approach is not feasible due to two reasons:

1. With increasing state space it might be difficult to hold all states or state-action pairs in the memory. Chess, which can be modeled with the presented framework, for example has 10^{43} board states without considering game history (Shannon, 1950). The IPD game has infinitely many states in its untruncated form. Given the simplification from Section 3.3 with a maximum period limit of $T_{\text{max}} = 60$ there are still around 10^{46} states and thus more than chess.

2. It is necessary to visit each state sufficiently often to ensure a correct Q-Value. Parallel to the necessity to save each possible, or at least each theoretically visitable, state this implies the necessity for a lot of computational resources to calculate this tabular data by playing the game sufficiently often.

Given the IPD game, an obvious approach to solve these problems might be to simplify the state space:

1. One could analyze the opponent strategy to find the true state space as discussed in Section 2.2. If one wants to optimize against a simple strategy and this is easily doable this is certainly a viable approach. However, if the strategy does not allow for an easy state space or one wants a simple algorithm where one does not have to analyze the strategy by hand, this approach is insufficient.

2. One could limit the state space to the last X periods, as has been done for example by Harper et al. (2017) or Sandholm and Crites (1996). This works well against some simple strategies which only act upon the last X periods. However, there exist strategies where the loss of information by doing so is too grave. In fact, one might not even get a good answer, for example if the opponent strategy tries to find patterns to exploit on run time and the discount factor γ is sufficiently high.

Both solutions have in common, that they want to reduce the state space through some mechanism - either through theoretical analysis or through the assumption, that the very last periods are the most important.
The most elegant solution however would be to find a function f_* which is defined for all states and actions and accurately approximates the state-action values, i.e.

$$f_*(s, a) \approx q_*(s, a) \quad \forall s \in \mathcal{S},\, a \in \mathcal{A} \tag{7}$$

As the state-action values are not known, one might construct a series of function approximators $f_1, ..., f_N$, with N being the most recent episode, which approximate

the Q-Values of the tabular method. The construction of f_i is chosen well, if $f_i \to f_*$ $(i \to \infty)$ not only for encountered combinations of state-action pairs (s, a), but for all of them. This necessitates that the function approximators f_i are able to extract all relevant information out of historic episodes regarding to be evaluated state-action pairs (s, a). This could for example be achieved if the function approximators are able to identify and recognize states which are near each other in the sense, that these states inhibit the same optimal course of action.
A selection of function approximation methods is presented in Section 2.4. The effects of using them in our setting are shown in Section 3.9.
A possible implementation of Q-Learning with function approximation is:

```
Initialize function approximator $f : (s, a) \mapsto f(s, a) \quad \forall s, a$ in some way.
**Repeat until** stop condition *Q-good*:
    Chose starting state $s$ of this episode
    **Repeat until final state is reached**:
        Choose action $a$ using a policy derived from $f(s, a_i)$, $a_i \in \mathcal{A}$ (e.g. $\varepsilon$-greedy)
        Take action $a$, observe $r$ and resulting state $s'$
        Update $f(s, a)$ in line with the Q-Learning update rule
        $s \leftarrow s'$
**End Repeat**
```

This formulation of the algorithm enables on-line learning, i.e. to improve the function approximator while an episode is playing. This is not necessarily optimal due to the following considerations:

1. It is non obvious whether on-line learning is necessary if there are no repeating states as with a complete encoding of the state space of the IPD game.
2. Depending on the technical implementation of the function approximator, the update step might be accompanied by fixed time costs. Loading the updating routine after each single step might thus severely impact the speed of the algorithm.
3. Updating each step might introduce oscillations and other fluctuations in the learning process of the function approximator and thus have a negative impact on convergence stability.

In fact Mnih et al. (2015), who developed the so called *deep Q-network* (DQN), used *mini-batch processing*, where the update is not done after each step but rather on uniformly random drawn samples (s, a, r, s') out of the memory. This allows a more stable learning, as old memories can stabilize the function approximator and is a special form of so called *experience replay* which was introduced by Lin (1992).
The effects of experience replay given the IPD game and our implementation are further discussed in Section 3.8.

2.4. Supervised Learning

In Section 2.3.4 we introduced the concept of function approximators, which are necessary to reduce the complexity of the state-action space. Generally speaking with the approximation of state-action values one tries to find a function f such that $f(s, a)$ is sufficiently close to the to be learned Q/Q_π-Values as given by the experienced state-action pair (s, a) and the learning algorithm. Furthermore f should be able to generalize in a way, that given a never before seen state-action pair (s, a) the difference between $f(s, a)$ and hypothetically encountered corresponding Q/Q_π-Values is minimal by some specified metric. The machine learning field *supervised learning* provides a wide array of possible methods and approaches on how to construct a good function approximator.
No clear rule exists which function approximation method is the best. Sutton and Barto (2018) describe in great detail linear function approximators, where - possibly handcrafted - features are given weights. The sum product of those features can then be used to estimate Q-Values. Learning takes place by gradually changing the weights of these features.
If one wants to allow arbitrary complex functions, linear function approximators may not suffice. We focus on three different non-linear approaches in this section which allow for complex generalizations:

1. **Gradient Boosting** - Build several simple decision trees in sequence which capture not yet captured features of the to be predicted variable.
2. **Neural Networks** - Interconnected layers of so called neurons allowing arbitrarily complex calculations.
3. **Recurrent Neural Networks** - A special case of neural network, where the output depends on a sequence of inputs and an internal mechanism allows the recurrent neural network to extract valuable information out of the current input and save it in some kind of short-term and/or long-term memory.

2.4.1. Gradient Boosting

Gradient boosting is a supervised learning technique which was developed by J. Friedman et al. (2000) based on Adaboost (Freund, Schapire, et al., 1996). We will focus here on developing a basic intuition. For good introductions the reader is referred to Grover (2017), Gorman (2017) and for a more rigorous in-depth introduction to C. Li (2016). An interactive graphical tool to develop an intuitive understanding can be found with Rogozhnikov (2016).
Gradient boosting is a special type of a so called *boosting* technique. The idea behind boosting is to use one or several, possibly very simple, function approximators in sequence. Here each of the simple function approximators aims to predict the residuals of the function approximators before it. Gradient boosting as a concept does not dictate which kind of simple function approximator should be used. So

called *decision trees* are one of the most common choices and are also chosen within this article.
A decision tree is a simple function approximator which separates the data set in different groups, so called *leafs* by asking, usually binary, questions. A graphical representation of a decision tree is depicted in Figure 1.

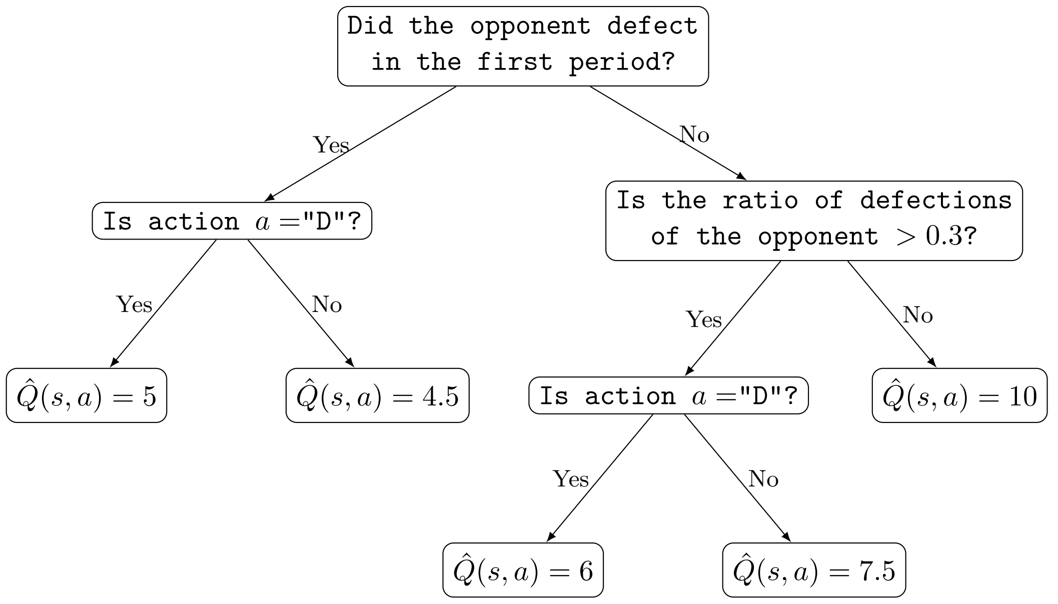

Figure 1: An illustrating example of a decision tree, estimating a $Q(s,a)$-Value. Assume (s,a) to be given.

Figure 1 shows, that these questions can be asked in regards to categories ("Is action $a =$"D"?") or in regards to numerical values ("Is the ratio of defections of the opponent > 0.3?"). Note, that calculations like the ratio or the sum of features have to be given by the encoding (see Section 3.7 for a more detailed discussion).
An example on how a sequence of decision trees can generate a good fit in a scenario with one single numerical explanatory variable is given by Figure 2. Here the maximum depth of the trees is 2 levels. In other words, each tree can only generate $2^2 = 4$ leafs. Even these simple trees are able to approximate a complicated and non-continuous function to a very high degree.

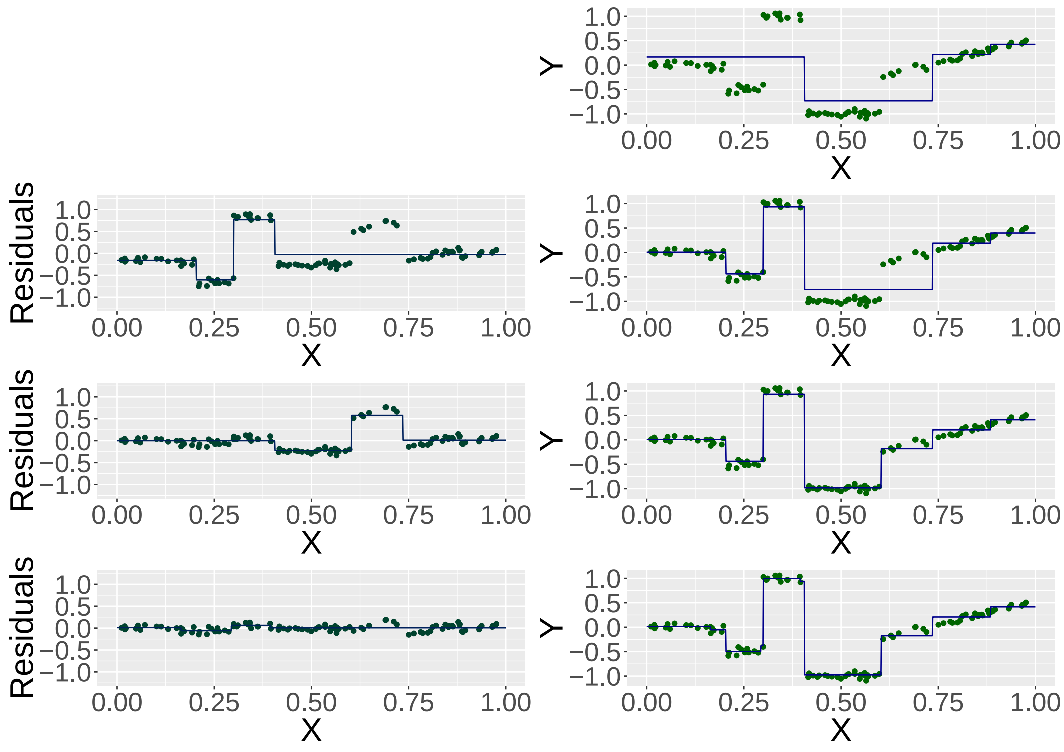

Figure 2: An illustrative example of gradient boosting. The goal is to find a good function (blue) which approximates the data points (green). Each row represents one boosting step. In the first column the current residuals are seen after applying all previous decisions trees in sequence. These residuals are then approximated by a new decision tree. The results of aggregating these simple decision trees with a maximum depth of 4 (so a maximum of four build categories) up to the step of the respective row can be seen in the second column.

Gradient boosting as used within this article is more complicated due to several improvements made over the last years. The first improvement is that the decision trees can be *pruned*. Pruning a tree is defined as discarding sub-trees to minimize complexity and reduce overfitting (see e.g. Almuallim (1996) for pruning of trees). Additionally, gradient boosting is normally not applied to the full extend as has been done in Figure 2. Instead each tree only contributes with a certain factor $\nu \in (0, 1)$ to the result of the tree sequence. This makes the result more conservative and thus less prone to overfitting (Hastie et al., 2009, p. 364).
While gradient boosting shows good results in classical supervised learning tasks and competitions, it is seldom used in reinforcement learning. The most important reason not to use gradient boosting is that it does not allow for incremental improvements based on new data. In fact, to improve our model with gradient boosting we build it from scratch each time. As gradient boosting is much faster than the alternatives however, it is still a viable in our case, as is shown in Section 3.9.

2.4.2. Neural Networks

Neural networks are a very powerful supervised learning technique which have found great interest in the recent years. They provide a key component to impressive recent developments in machine learning in a wide area of applications. Examples include image classification tasks (Krizhevsky et al., 2012), adding color to black and white images (Iizuka et al., 2016) or predicting mortality of hospital patients (Rajkomar et al., 2018).
Of special interest for us are their usefulness in the field of reinforcement learning, as they were a corner stone for several new algorithms. These were able to achieve feats such as beating grand masters in the ancient game of Go (Silver, Schrittwieser, et al., 2017) or achieving super human scores in ATARI games (Van Hasselt et al., 2016).
Modern neural networks have developed significantly from their historical roots, which were motivated by McCulloch and Pitts (1943) and Hebb (2002), first published in 1949. Still used, the *perceptron* of Rosenblatt (1958) is a simple part of a neural network and allows to develop an intuition about this supervised learning method. For a more in depth introduction the reader is referred to Nielsen (2019).
Assume, within our context of the IPD game, we receive observations of the last three periods of the opponent. Assume further more, that we want to defect if the opponent defects often and/or recently. One way to model this would be using the *perceptron* of Rosenblatt (1958):

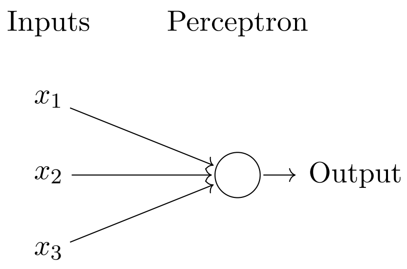

Figure 3: A schematic perceptron as a building block for a neural network.

If we encode a defection with 1 and a cooperation with 0 and define x_1 to be the observation three periods ago, x_2 two periods ago and x_3 the very last period, we have encapsulated all available information in the vector x.
Each of the three edges of Figure 3 is weighted with a weight w_i, defining the vector w. Additionally, a perceptron has an internal threshold b. The output of the perceptron is defined the following way:

$$\text{Output} = \begin{cases} 0 & \text{if } w \cdot x + b \leq 0 \\ 1 & \text{if } w \cdot x + b > 0 \end{cases} \tag{8}$$

Using the weights $w_1 = 1$, $w_2 = 1$, $w_3 = 3$ and $b = 2.5$ achieves our goal. The perceptron recommends a defection if the very last period has been a defection, the

two periods before that or all three of them and a cooperation otherwise.
Using layers of perceptrons as seen in Figure 4 allows for more complicated calculations. Here the output of a perceptron is used as the input for another one. This way is possible to build arbitrarily complex logic gates (Nielsen, 2019) and thereby solve arbitrarily complex problems.

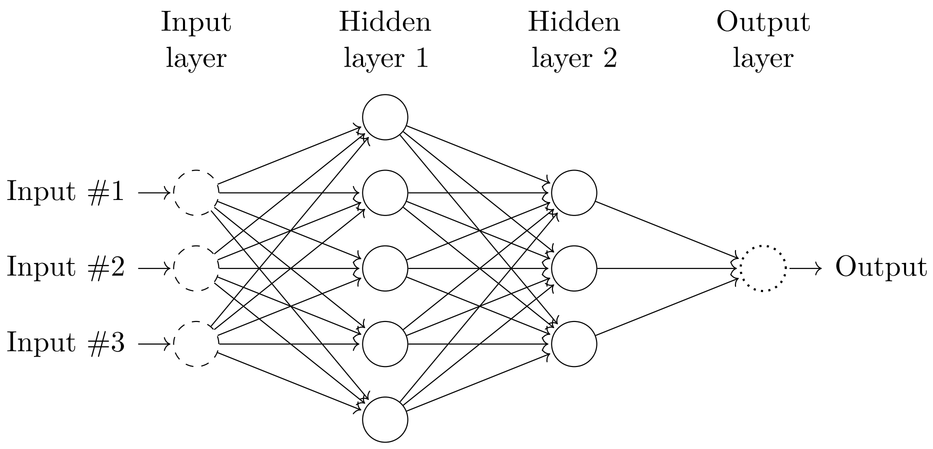

Figure 4: A schematic Neural Network outputting a single value with two hidden layers of 5 and 3 hidden neurons. Given the IPD game, Input 1 might be 1 if the opponent defected in the last period and 0 otherwise. Input 2 might be equivalently coded for the action of the agent itself from the last period. Input 3 might similarly be the to be evaluated action a.

Two obvious challenges remain with perceptions:

1. Given the discussed algorithms, we want to approximate numeric Q-Values which are not binary.
2. While finding good values w and b was straightforward in the example above, this is not the case in a complex system of hundreds of neurons and thousands of edges.

To show how these problems are solved, we will first generalize the inner workings of the perceptrons. General neurons can have different ways of handling the term $wx + b$ in contrast to the step-wise function of the perceptron of Equation 8.
The output layer might for example have a linear activation function, i.e. we simply output $wx+b$. As w and b can be changed, the resulting output can have an arbitrary value, even if all inputs x are constrained to the interval $[0, 1]$ from previous layers. To make it easier to find fitting weights w and b while at the same time keeping the feature of being able to generate arbitrary logic gates one can smooth the step-wise function of the perceptron.

A common way to smooth the step-wise function of the perceptron is to use the sigmoid function

$$\text{Output} = \frac{1}{1 + e^{-(wx+b)}}$$

which is illustrated in Figure 5.

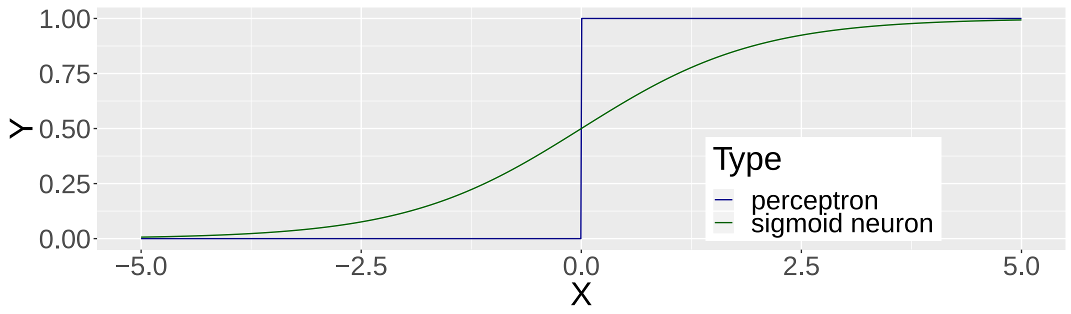

Figure 5: Difference between the perceptron activation function and the sigmoid activation function. X is the weighted input $wx + b$.

The advantage of using such a smooth function is that an incremental change of one of the weights w_i leads to an incremental change in the output. The reverse holds true as well: If the output is non-optimal, i.e. the neuron outputs a value which is marginally to high, one can change the weights to increase the fit, as it holds that

$$\Delta\text{Output} \approx \sum_i \frac{\partial \text{Output}}{\partial w_i} \Delta w_i + \frac{\partial \text{Output}}{\partial b} \Delta b \tag{9}$$

We use *gradient descent* to decide which of the weights should be changed how strongly. The basic idea is to choose the direction of the biggest change.
While gradient descent is an important idea to update a single neuron, explicit gradient descent on the whole network is too costly to do when confronted with several hidden layers as for example in Figure 6. Rumelhart et al. (1986) proposed using the so called *backpropagation* algorithm to solve this problem. Here, the effects of different layers on the output are solved iteratively starting from the output layer - the back - by using fast to compute matrix operations. This allows to efficiently change exactly those neurons which have the most impact on the difference between current and desired output.
Using a neural network to approximate structurally the same function as with the example for gradient boosting results in Figure 6. In contrast to gradient boosting the neural network is better in the approximation of the linear parts of the function. However, it has significantly more difficulties with the step-wise parts. Given a sufficiently complex structure, sufficient training data and sufficient time to converge however, both techniques are able to approximate the example function to an arbitrary degree.

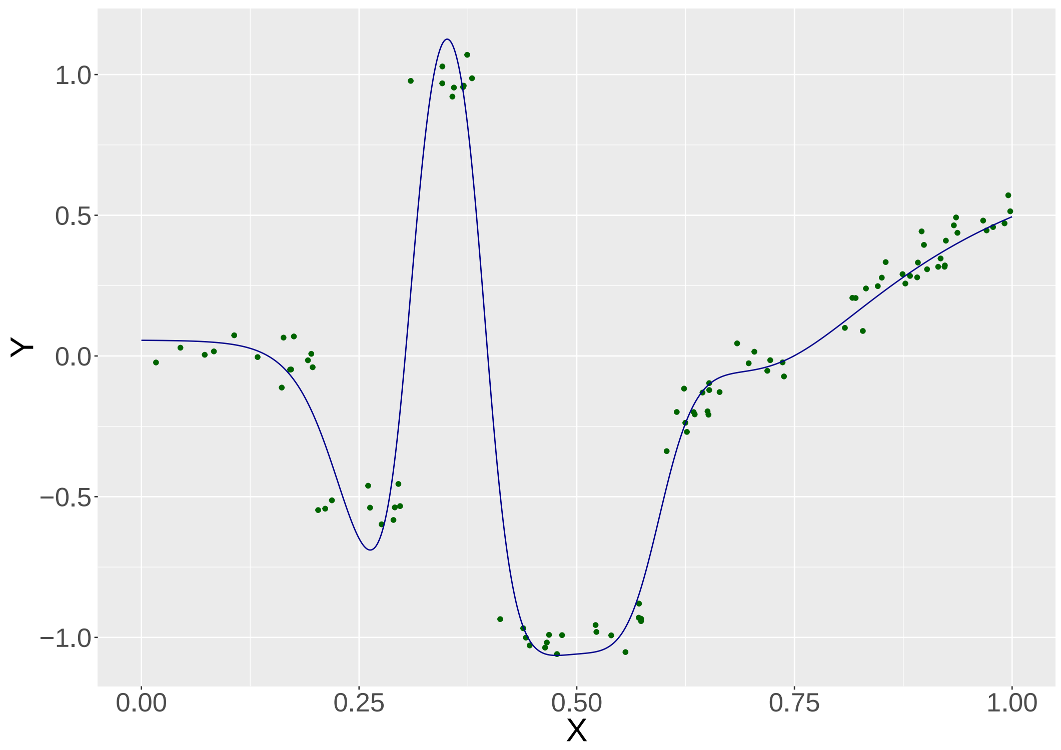

Figure 6: An illustrative example of a function approximated by a neural network.

2.4.3. Recurrent Neural Networks

Standard neural networks are *feedforward*. This means, that the information and calculation flow starts with the input neurons and propagates through the different layers with the last layer producing the output. By contrast, *recurrent neural networks* go through a sequence of inputs, calculate intermediate results, so called *hidden states*, and use the same network multiple times to evaluate the sequence. This effect is illustrated with Figure 7. If one wants to calculate the respective output, an additional output layer to transform the relevant hidden state is used.
Their sequential design makes them ideal candidates for all data which are naturally sequential, like text-to-speech transformation, translations, interpreting videos or generating texts. They are also used in reinforcement learning, most notably with The AlphaStar Team (2019), where a program was able to play Starcraft II on an very high level, challenging and beating professional players. Their advantage relative to normal neural networks is that there is no necessity to provide the complete history of a sequence. Instead, one can provide only recent events and have the network work out itself which kind of information should be transferred to the next iteration.

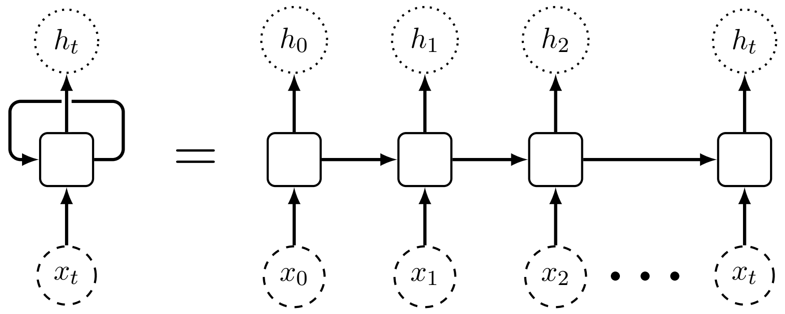

Figure 7: A schematic view of a generic recurrent neural network. Given is a sequence of inputs x_t. The recurrent network processes the input and information based on previous cells, outputs the hidden states h_t and gives some information to the next part of the sequence.

Historically, Recurrent Neural Networks are based on the works of Rumelhart et al. (1986) but they received their big takeoff with the development of *Long short-term memory* (LSTM) cells by Hochreiter and Schmidhuber (1997). A schematic view of a standard LSTM cell is given by Figure 8. A good explanation on the inner workings of such a cell is given by Olah (2015).

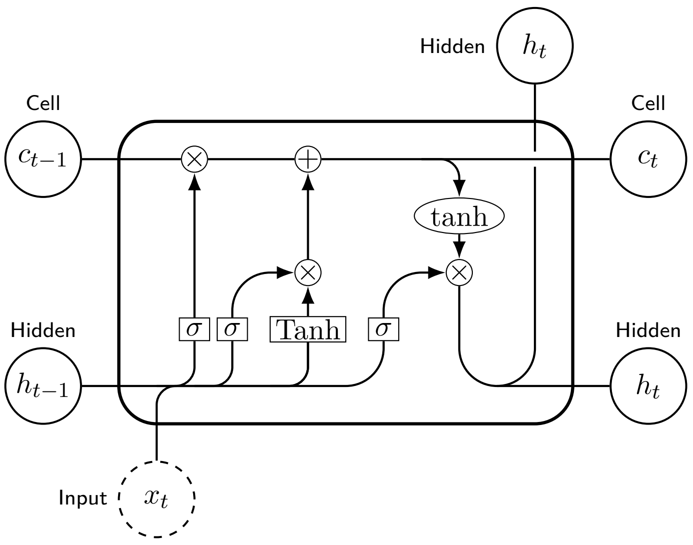

Figure 8: A standard LSTM Cell based on the schematics of Olah (2015). Two conjoining lines are a vector concatenation, $\times$ is an element-wise multiplication, $+$ an element-wise addition. The single *tanh* within an elliptic box is the element-wise application of the tanh-function. A σ within a tetragon stands for a transformation of the vector according to a sigmoid layer, i.e. between 0 and 1. The single *Tanh* within a tetragon transforms the vector according to a tanh-layer, i.e. between -1 and 1.

One of the main problems of basic recurrent neural networks is that they tend to forget information along the way as they have no good mechanism to decide which information to keep. This problem is solved by the LSTM cell, utilizing three sigmoid gates. Each of those gates takes as input the concatenation of the so called *hidden state* and the input x_t. The output is a vector with some size n which is both the dimension of the *hidden state* and the so called *cell state*. The cell state is used to hold long-term memories without strong changes. One such memory could be the information "I am playing against a strategy of type X".
In more detail, the intuition behind the three sigmoid gates is as follows:

1. The first sigmoid gate decides which parts of the cell state should be forgotten. Those parts of the cell state which should be forgotten are then multiplied with 0 and those which should be retained are multiplied with 1.
2. The second sigmoid gate is the so called *input gate*. The tanh-layer extracts the necessary information out of the input and the current hidden state and after the filtration through the second sigmoid layer adds them to the cell state. The cell state has thus been updated with the new information and may be used in the next LSTM cell.
3. The third sigmoid gate is the so called *output gate*, as here the new hidden state is generated. One could say, that the new hidden state is the new cell state - normalized by a element-wise tanh operation - which is filtered by the output gate.

When using the recurrent neural network, we are interested in the very last h_t, which is an array of the length of the number of used hidden neurons. Analogously to the neural network we filter this result through an output layer which scales and aggregates the calculated values to a single usable Q-Value. Similarly to the multi-layered neural network one can use additional intermediary layers and filters to increase the power of the network and capture more complicated patterns. The actual schematic of the LSTM as used in Section 3 is shown in Figure 9.

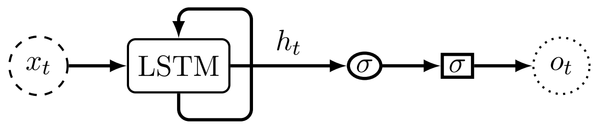

Figure 9: A schematic view of the Recurrent Neural Network as implemented in Section 3. There is a sequence of inputs up to x_t. The recurrent network processes this sequential input using LSTM cells, and filters the result through a sigmoid activation function, a dense hidden layer with sigmoid activation and finally the output layer with linear activation function and a single neuron to produce the output.

In the specific case of the IPD game and using one of the simplest encodings, x_t might be the observed actions of both players regarding period t concatenated with the to be evaluated action. To receive the Q-Value regarding the action of the third period, the calculation would thus go as following given the specifics of the setup:

1. Vectors h_0 and c_0 are initialized with a default value.
2. h_0, c_0 and x_1 are given to the (trained) LSTM cell, which outputs c_1 and h_1. The vector x_1 could for example be $(-1, 1, 0, 1)$ denoting that the agent observed a defection of itself [first element] and a cooperation of the opponent in the first period [second element] as well as that that the agent has taken the second of two possible actions [third and fourth element][4].
3. c_1, h_1 and x_2 are given to the (trained) LSTM cell, which outputs c_2 and h_2. Here, x_2 could be $(1, 1, 1, 0)$ describing the observations of period 2.
4. c_2, h_2 and x_3 are given to the (trained) LSTM cell, which outputs c_3 and h_3. An example of x_3 could be $(0, 0, 1, 0)$ indicating, that no info is available regarding the observations of period 3 and that the agent is currently evaluating the first of two actions.
5. h_3 is filtered through a sigmoid function to keep comparability to the setup of the neural network.
6. $\sigma(h_3)$ is connected to a dense hidden layer with sigmoid activation, producing h'_3
7. h'_3 is connected to a dense linear output layer with a single neuron, resulting in the wanted Q-Value.

[4]Actions are encoded differently within the R-package Kies (2019) because the encoding of the state has been chosen with the specifics of the IPD game in mind, but the action space is managed game-independently. The IPD game only has two actions, so the encoding -1 for a defection, 0 for having no information and 1 for a cooperation is very space minimizing. In the more general approach of the action space, all actions are encoded in binary bins for each action and a 1 is given to the action which is considered/has been played.

3. Finding Best Answers

3.1. Overview

This section presents and discusses an algorithm, whose development was guided by the following objective:

> Find an algorithm which converges fast to the maximum achievable expected average reward against a black-box strategy in the Iterated Prisoner's Dilemma (IPD) game with noise.

In the sense of the objective a black-box strategy is defined as a strategy which outputs an action a when presented with a game state s but where we are not able or willing to analyze its explicit plan of action.
We distinguish between the concept of a best response and the so called *best answer*. A best response is the game-theoretic optimal counter-strategy $\mathbf{s}^*$ as defined by the rules of the IPD game in Section 2.1.1. The maximum achievable expected average reward $\bar{R}(\mathbf{s}^*)$ by definition is given by the performance of a best response. By design the machine learning framework of Section 2.2 allows us to use the machine learning concept of a best policy π^* and the best response $\mathbf{s}^*$ interchangeably. In the sense of the objective it is therefore sufficient to find an algorithm which converges against the best policy.
The best answer by contrast is the counter-strategy $\mathbf{s}$ which shows the best known performance $\bar{R}(\mathbf{s})$. Hereby it is of no relevance whether this strategy is given by an explicit plan of action or has been derived as a machine learning policy. If a best response is known, than it is automatically also the best answer. However, deriving a provable best response might be unfeasible as the game state space might be large and the strategy stochastic. If no exogeneously given counter strategies are known a sequence of increasingly better performing policies π define a new best answer with each new best policy. To find a best answer therefore means to find a policy π which is better than all known alternatives.
There exist several algorithms which are guaranteed to find a best policy given an infinite amount of time. We focus on automatically generating a good counter-policy given a finite time frame. This way, it is possible to evaluate the susceptibility of a given strategy to dedicated counter-strategies without having to manually analyze it. In Section 3.2 it can be seen, that using the performance of the found counter-policies is a viable alternative to hand-crafting best answers, even if the source code of the strategy is known.
The detailed algorithm can be found within the R-Package Kies (2019) on GitHub, including some working samples. The foundation of the algorithm is the Q-Learning algorithm with function approximation and ε-exploration (see Sections 2.3.3 and 2.3.4). The proposed modifications should therefore be interpreted as modifications to this baseline.

The following modifications to the Q-Learning algorithm are analyzed:

1. **Q-Switching** - In the initial phase of the learning process the state-action values generated by Q-Learning might be unreliable. Q-Switching uses the Monte Carlo Values of the experienced episodes to kick-start the learning. During the learning process a seamless transition to Q-Learning takes place.

2. **Multi-Exploration** - Instead of a single exploration variant, up to four different ways to explore the policy-space can be combined: *ε-exploration* (see Section 2.3.3), *noisy actions* (similar to Boltzmann exploration), *maximizing surprise* and *minimizing familiarity.*

3. **Exploration Pathing** - The standard method to use ε-exploration is to lower the exploration parameter ε in a predefined way. *Exploration pathing* automatically changes the exploration parameter based on the impact of the parameter on the performance of the current policy. This enables us to prespecify a certain impact of the exploration parameter at a given time in the learning process and allows the usage of *Multi-Exploration* in a controlled fashion by balancing their exploration parameters to each other.

4. **Feature Selection** - The power of the algorithm is strongly determined by how the current state is presented to it. Simply providing raw data (or only the last X periods) makes it generally harder for a function approximator to find exploitable patterns. The used state-space representation adds calculated features to the raw history of the game, as for example the sum of cooperations of the opponent.

5. **Choice of Function Approximator** - The main focus in the literature of reinforcement learning is on neural networks. Using a different function approximator as for example gradient boosting can significantly improve the quality of the result given time constraints while at the same time providing a more robust result.

6. **Experience Replay** - Instead of updating the function approximator iteratively after each action or episode, the data is stored and batches are used to train the function approximator after a certain number of episodes. This allows the use of non-updating function approximators like gradient boosting.

7. **Model Persistence** - Instead of solely using the most recent model for future actions, both the currently best model and the newest model are used to generate data. If the best model is preceded by the most current one, it is replaced. *Model Persistence* helps to increase the stability of learning and to avoid local optima.

8. **Memory Initialization** - Assume that a strategy plays a key action sequence to identify and subsequently cooperate with other variants of itself. *Memory Initialization* sets the strategy against itself and adds these experiences to the memory pool. This allows the algorithm to copy these action sequences without having to discover them itself.

The novelty of these modifications ranges from "proposed by this article" (*Q-Switching*, *Multi-Exploration*, *Exploration Pathing*, *Model Persistence*) over "novel in its application to the IPD game" (*Choice of Function Approximator*, *Experience Replay*, *Memory Initialization*), sometimes with adaptations to reflect the specific challenges of the game, to "extension of existing approaches" (*Feature Selection*). Each modification has its own section, discussing it in more detail. Here, we discuss the extent to which the modifications are new ideas and compare them to similar approaches pursued in the literature.

In the these sections we also explore to which extent the modification improves the general performance within our setting (see Section 3.3) or only given special circumstances. The concept behind *noisy actions* for exploration for example is a well-known alternative to ε-exploration and can improve the performance of an Q-Learning algorithm (see e.g. Tijsma et al. (2016) for a comparison of exploration strategies given mazes). However, given the specifics of our game, exchanging ε-exploration with *noisy actions* does not add any significant benefits. In this article we not only present novel extensions to the Q-Learning algorithm in general, but also shed light on the usefulness of known extensions given the challenges of the IPD game.

Within this article we concentrate on the performance of the algorithm given an IPD game with noise. Nevertheless, with the exception of the game-specific *Feature Selection*, all modifications are game-independent and can be applied to learning processes in the context of other games. To do so, these games have to match the general framework laid out in Section 2.2.

A pseudo-code-like overview of the general structure and basic composition of the algorithm is presented in Appendix C.

The remainder of this section is organized as follows: Section 3.2 discusses the performance of the algorithm as a whole as well as the potential for possible improvements. The subsequent sections dive into the specific modifications. Section 3.3 describes in detail how the modification-specific studies have been conducted. Sections 3.4 to 3.11 focus on the individual modifications. Section 3.12 gives a summary of the most important results and motivates the exact configurations which were used for the final performance tests in Section 3.2.

3.2. Performance

The performance of the algorithm making use of the full extent of its capabilities in regards to the classical strategies (Section 2.1.2) playing the IPD game (Section 2.1.1) are shown in Figure 10. Which modifications were used to which extent is summarized in Section 3.12.

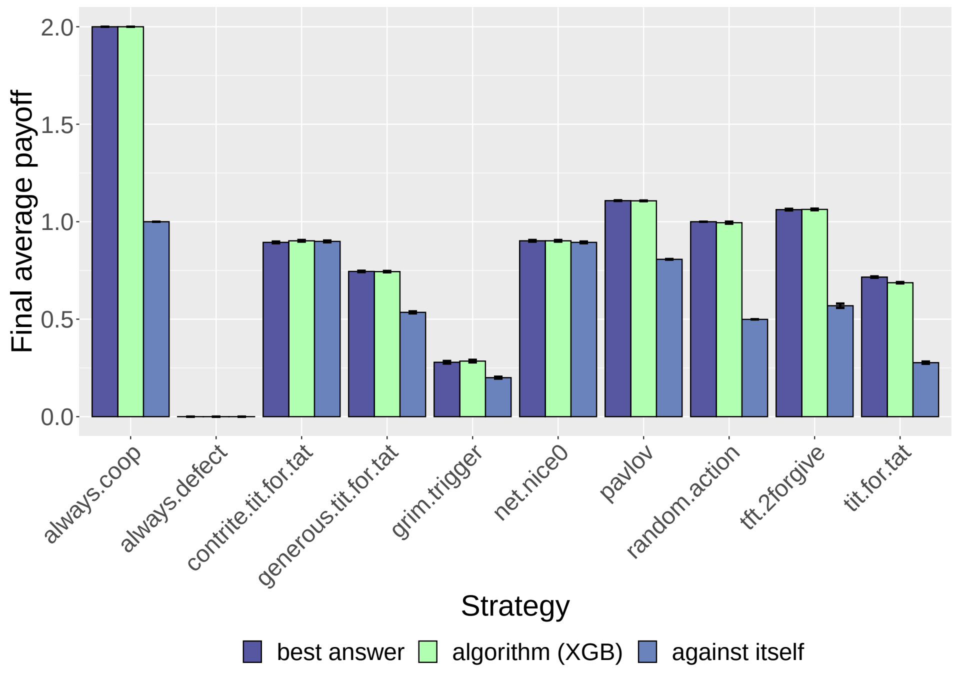

Figure 10: Final results of the developed algorithm against the classical strategies discussed in Section 2.1.2 given an IPD game played with the rules as outlined by Section 2.1.1. Shown is the estimated expected average reward after a single run of the algorithm on an arbitrary chosen seed with a limitation to 60 periods per game. "Best answer" is the performance of the corresponding best answer strategy or the theoretical expected average reward of the best response, when available. "Algorithm (XGB)" used gradient boosting and 150 updates of the function approximator. Between each update of the function approximator the game was played for 4 episodes. "Against itself" displays the performance of the strategy against itself. The error bars represent the standard errors of these results based on the methods of Cochran (1977) after 1000 simulated games using the R-package *StratTourn* (Kranz and Kies, 2019) with a maximum period limit of 60 periods. If the result could be calculated explicitly, this result is shown.
The detailed settings can be found in Appendix H and the R-package Kies (2019).

In all cases except against *tit.for.tat* the algorithm performs at the same level as the best answer despite the very noisy environment. As the classical strategies are not black-box strategies and comparatively trivial to optimize against, the shown best answers are all best responses. Even in the case of *tit.for.tat* the algorithm only performed marginally worse than optimal.

An illustration on how these results have been derived is shown in Figure 11.

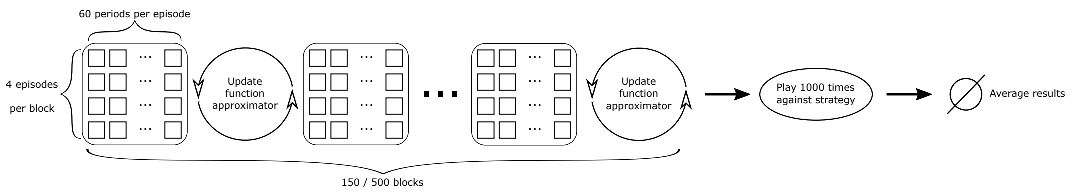
60 periods per episode
4 episodes per block
Update function approximator
Update function approximator
Play 1000 times against strategy
Average results
150 / 500 blocks

Figure 11: Schematic representation of the generation of the results of Figure 10 and Figure 12. A block consists of 4 episodes à 60 periods where the algorithm plays against the strategy. After each block the function approximator is updated. This is repeated 150 or 500 times depending on the analysis, as specified in the respective figure. The final policy is set 1000 times against the strategy using the R-package *StratTourn* (Kranz and Kies, 2019), taking into account the period cap of 60 periods. The average result of this analysis is the displayed final average reward.

While the performance against the classical strategies poses a good test of robustness, it does not allow for an adequate evaluation on how the algorithm would fare against more complicated strategies. The classical strategies tend to be simple and easy to optimize against. This does not necessarily hold true for arbitrary strategies humans might find fitting for tournament participation. Generally speaking, evaluating the performance of machine learning algorithms is difficult, as the effectiveness of a particular change in an algorithm strongly depends on the underlying problem. Even given a specified environment (e.g. the Iterated Prisoner's Dilemma game) deviations (e.g. changing the complexity of the strategy of the opponent) can and will significantly influence the performance of a learning algorithm. This change not only impacts the absolute performance level, but can also change the relative performance of different variants of learning algorithms to each other.
A standard approach to evaluate a new learning algorithm is to run it, as well as proposed counter candidates, on a set of specified environments. A popular choice for such environments are for example ATARI games (see e.g. Kaiser et al. (2019), Hessel et al. (2018)). If the new algorithm scores higher in an aggregated metric, one has some indication that it provides value. Examples for such metrics are the sum of games an algorithm outperforms its opponents or the mean over standardized scores which can be calculated in each environment.
We are interested in the performance of our algorithm given economic games with a strong focus on the IPD game. Of special interest to us are strategies which have been developed by humans expecting that their strategies should fare well in a tournament. These strategies inhibit two important features: On the one hand they are special in the sense, that they are not arbitrary candidates of the strategy space but actually expected to be good, and thus relevant, strategies. On the other hand they might be much more complex than the classical strategies and possess hard to find flaws which can be exploited. Having such a pool of strategies would therefore be a natural environment to test our algorithmic modifications.

The Institute of Economics at Ulm university hosted a tournament where participating students developed strategies for the IPD game. The students were tasked to set up teams and to

1. Develop a strategy which will perform well in a tournament setting and is stable against attackers as a first stage of their task.
2. Build individual counter-strategies against the strategies of the other teams as a second stage.

This tournament provided the input to build our testing pool. In contrast to the strategies of the Axelrod tournament (Axelrod, 1984) it also provides corresponding best answers as a benchmark and normalization factor of our performance tests.
The strategies of our testing pool have been developed to behave identical to the strategies of the student teams of the first stage of the tournament. Based on the ideas of the counter-strategies of the second stage I developed a set of corresponding best answers.
Each used best answer matches at least the performance of all corresponding counter-strategies provided by the second stage of the tournament. These best answers are not necessarily the actual game-theoretic best responses. In fact in one case, with strategy *strat.i*, our algorithm has been able to generate a policy which achieves a higher average reward than the hand-crafted best answer. Despite this result, one should have high confidence that improving upon the best answers is at least non-trivial. For most strategies the very same counter-strategy has been submitted in the second stage, giving some indication, that a best response might have been found. The results of the algorithm against the testing pool are shown in Figure 12.
To mirror the situation against the classical strategies, the algorithm variant "alg. (XGB)" was deployed with the same configuration for Figures 10 and 12.
In contrast to Figure 10, Figure 12 shows two additional variants of learning algorithms against the testing pool strategies:

1. While the standard configuration "alg. (XGB)" used gradient boosting for a fast development of the policy, the configuration "alg. (RNN)" used a recurrent neural network with LSTM-cells. Additionally, this alternative configuration ran longer both in terms of computation-time as well as the number of played episodes. Furthermore, it was allowed to update its function approximator more often with 500 updates compared to the "alg. (XGB)" variant with 150 updates. This configuration allows to estimate the upper limits of the algorithm lacking time constraints.
2. The configuration "Q-Learning" is included as a benchmark. Its parameters were optimized based on the studies of the following sections but it did not use the modifications which are proposed by this article. To make for a fair comparison, it uses the same number of played episodes and updates of the function approximator as "alg. (XGB)".

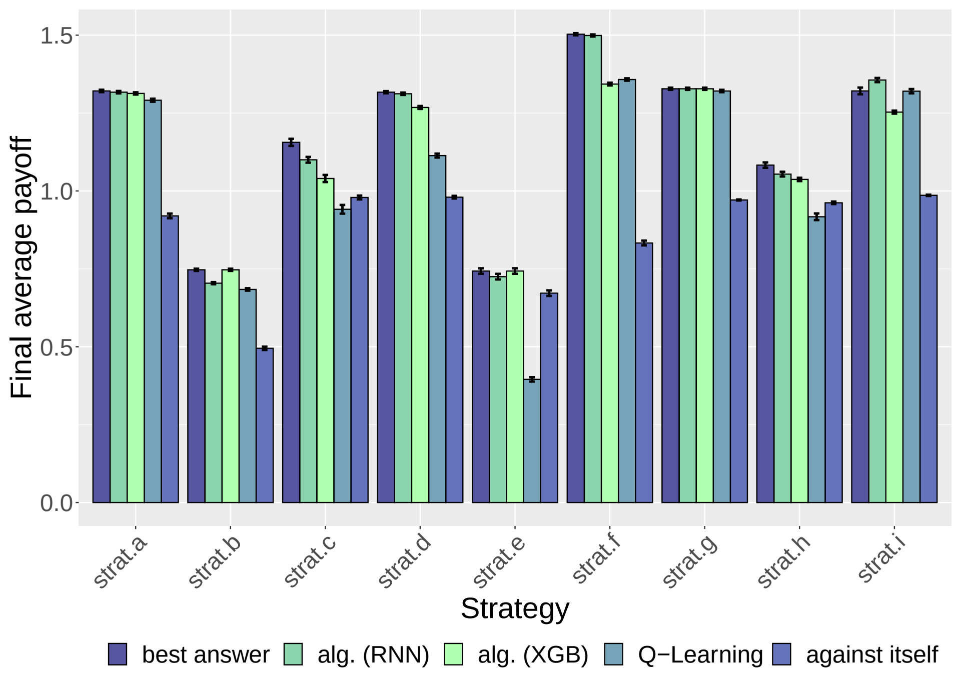

Figure 12: Final estimated expected average reward of the developed algorithm against the training pool after a single run on an arbitrary chosen seed with a limitation to 60 periods per game. "Best answer" is the performance of the corresponding hand-crafted best answer strategy. "Alg. (RNN)" used a Recurrent Neural Network with LSTM-cells and 500 updates of the function approximator. "Alg. (XGB)" used Gradient Boosting and 150 updates. "Q-Learning" used basic Q-Learning without the modifications proposed by this article but otherwise optimized parameters. It used ε-exploration, a neural network, the feature selection of Harper et al. (2017) and 150 updates of the function approximator. Between each update of the function approximator the game was played for 4 episodes in all configurations. "Against itself" displays the performance of the strategy against itself. The error bars represent the standard errors of these results based on the methods of Cochran (1977) after 1000 simulated games using the R-Package *StratTourn* (Kranz and Kies, 2019) limited to 60 periods.
The detailed settings can be found in Appendix H and the R-Package Kies (2019).

On average, we achieved a final difference in average rewards to the best answer of -0.013 [-0.019 without *strat.i*] with the more powerful recurrent neural network run and -0.049 [-0.047 without *strat.i*] with the gradient boosting run given the testing pool. The Q-Learning run achieved -0.13 [-0.147 without *strat.i*]. In other words, using the proposed modifications to Q-Learning reduced the distance to the best answer by 62% [68% without *strat.i*] in the aggregated view and using the same number of episodes and updates of the function approximator.

Both, the gradient boosting run and the normal Q-Learning run, took about 40 minutes run time per strategy. The recurrent neural network run lasted about 8 hours per strategy. The basis for these times is the implementation of the algorithm by the R-package Kies (2019) and using a notebook as specified by Appendix G.7. The presented run times correspond to the time difference between starting the algorithm and having calculated the final model, without considering the performance measurement phase with the *StratTourn* package. Note, that the figures of the following sections show a final performance difference of around -0.1. These studies were conducted on a limited function approximator and used fewer updates and episodes than the final performance tests. To achieve this result, which lies between the performance of the gradient boosting run and the Q-Learning run, the algorithm ran approximately 10 minutes per strategy. In other words, even if one is content with a performance level as displayed by the Q-Learning run, one can use the modifications of this article to quarter the necessary run time.
Assessing the performance in regard to the individual strategies, we were always able to improve upon the strategy against itself. In most cases we effectively matched the best answer. The worst result was against the comparatively complex strategy *strat.c* which uses a Bayesian updating rule to detect exploitation. The recurrent neural network run achieved the best results on average but not for each strategy individually. Gradient boosting performed better with strategies which are based on simple rules. These strategies are more easily captured by the specific encoding of the *Feature Selection* which favors gradient boosting. If one has sufficient resources, it might thus be beneficial to run both approaches and then use the best resulting policy. This procedure exploits the different strengths of the different function approximators.
The recurrent neural network run beat classical Q-Learning in every single instance. The same does not hold true with the more comparable gradient boosting run. Here, the algorithm beat Q-Learning 7 out of 9 times, with 5 substantial improvements. Q-Learning beat the gradient boosting 2 out of 9 times with only one meaningful difference. This difference could be achieved with the peculiar case of strategy *strat.i*. Notably, the standard Q-Learning algorithm failed to beat the performance of the strategy against itself in 3 out of 9 cases.
Strat.i shows deviating results from the other strategies and as such merits a deeper analysis. The source code of both the strategy and its proposed best answer can be found in Appendix F.9. The strategy is quite complex, changing thresholds when to defect throughout an episode. The proposed best answer is to always cooperate when the opponent observed a defection of the strategy and defect otherwise. This counter-strategy can be played by directly mapping the simpler *Feature Selection* of Harper et al. (2017) to the to be taken action. This *Feature Selection* of Harper et al. (2017) has been used for the classical Q-Learning run. As a consequence it was able to capture this counter-strategy perfectly. The configuration of the gradient boosting run is able to produce policies of same complexity. This can be observed in Figure 10 against the classical strategies. The best response against *strat.i* however is not this specific counter-strategy, as can be seen by the performance of the recurrent

neural network run. While the standard Q-Learning is trapped in the local optimum of the proposed best answer, the gradient boosting run is not. In this case this leads to a worse result, as the gradient boosting run correctly identifies the proposed best answer as non-optimal, but fails to produce a better policy.
Figures 10 and 12 have both been generated based on a single run on a single parameter set on an arbitrary chosen seed. Using multiple runs this result may easily be improved upon, as the resulting model is always subject to stochastic draws within the learning process. For that, one simply runs the algorithm multiple times using different stochastic seeds and takes the best generated policy.
Using hyperparameter tuning - especially on a per strategy level - will also improve the results. As this procedure increases training time immensely, I have refrained from doing so. Another meta-approach which increases training time is to run the algorithm repeatedly using a changing complexity of the underlying feature set. The results regarding *strat.i* highlight how a local optimum might still be better than chasing the global optimum.
All results shown in Figures 10 and 12 are generated using a limit of $T_{\max} = 60$ periods. The results of the recurrent neural network configuration tested in an infinite period setting are comparable and can be found for each strategy in Appendix F. While the training took place on 60 periods, the recurrent neural network was able to interpolate the policy exceeding the period limit.
A variety of other improvements of Q-learning exist, which are notably combined in the state-of-the-art algorithm *Rainbow* by Hessel et al. (2018). Further research might reveal whether the improvements by *Rainbow*, which were tested on ATARI-games, are helpful for our case and which of my modifications may be worked into *Rainbow* to further enhance performance.
While the performance results of this section highlight that the algorithm is able to develop good counter-policies, they do not provide insights in respect to the relative importance of the different proposed modifications. A detailed analysis is given in the following sections.

3.3. Approach of Analysis

In most cases the proposed improvements to the basic Q-Learning algorithm fall in one or both of two categories:

1. Improvements which increase the final performance of the generated policy.
2. Improvements which increase the speed of learning measured by the number of played episodes - mostly due to improved data processing or by improving exploration.

Improvements of the first category can be shown using a performance test analogously to the studies of Section 3.2. However, to gauge the effects of the second type of improvement one would ideally perform such a test after each policy change. Indeed, the package *RLR - Reinforcement Learning with R* (Kies, 2019) does allow for a dedicated evaluation of the performance of the current policy after each update of the function approximator.
Since the game itself, the update of the function approximator, and the exploration methods all inhibit stochastic elements, it is necessary to repeat all runs sufficiently often to show whether changes to the algorithm are significant. To draw general conclusions which are not limited to specific features of individual strategies, one has to repeat the analyzes for every strategy within the testing pool. A performance measure analogously to Section 3.2 after each update of the function approximator is therefore very time-consuming. Instead, we exploited an inherent feature of the algorithm, the so called *best effort* episode. In this case after each update of the function approximator an episode is played according to the most recently developed policy without any type of exploration. Since an update of the function approximator is a necessary requirement for a change in the played policy, this approach guarantees that at least one episode is played for each developed policy.
We call a *block* the set of played episodes between two updates of the function approximator. A standard block uses a single best effort episode in addition to a single exploration episode. Unless noted otherwise, these standard blocks are used in the subsequent sections.
A graphical representation of the used approach is shown in Figure 13. It shows, how periods, episodes, blocks, and runs are nested into each other. The runs are all independent and used to eliminate stochastic effects with respect to the learning behavior of the algorithm as well as to aggregate over all strategies of the testing pool. Unless noted otherwise, each strategy always provided the same number of runs in each analysis. Comparisons take place across different sets of such runs, were within each set all parameters remained unchanged, except for replacing the strategies. The different sets of runs differ in their use of the to be analyzed modification of the algorithm.

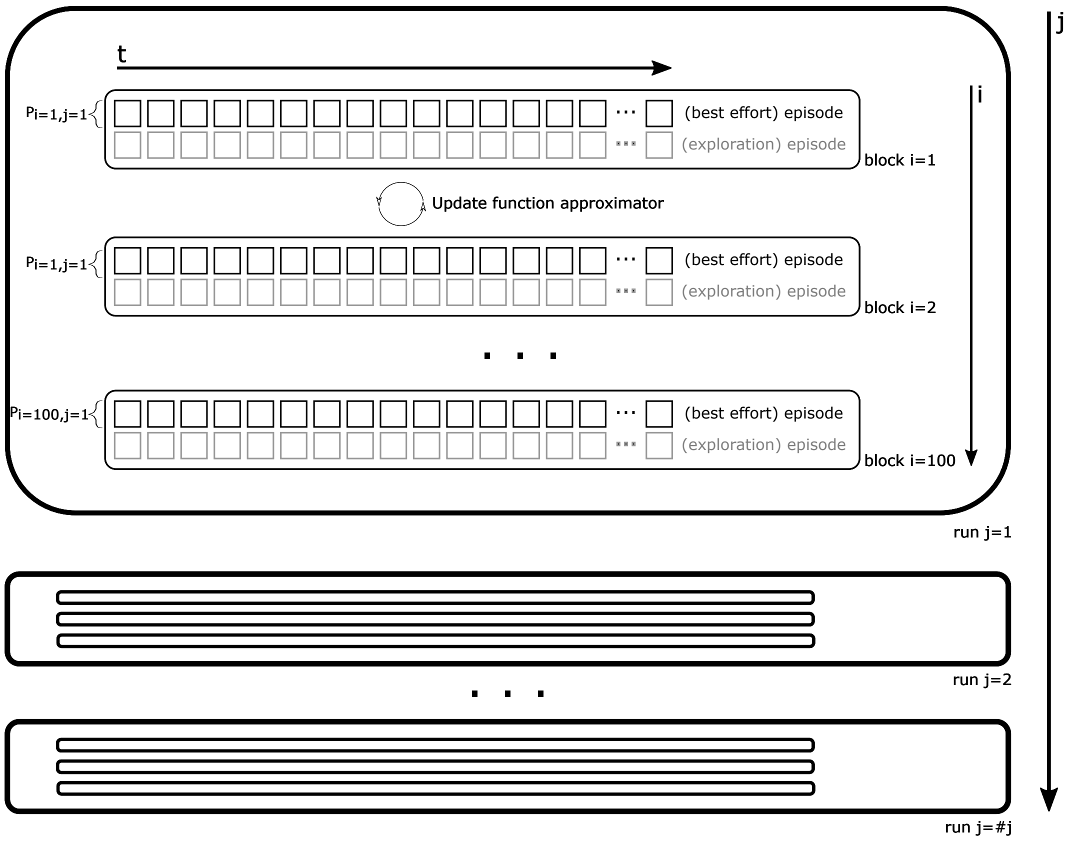

Figure 13: A schematic of how the data of the figures of Sections 3.4 to 3.11 have been generated, unless noted otherwise. Each *run* consists of 100 *blocks* of two *episodes* each. In each block a *best effort* episode and an *exploration* episode is played. The *best effort* episode is played optimally according to the most recently generated policy, while the *exploration* episode takes suboptimal choices to generate information about the environment. After each block the function approximator is updated, which most likely leads to a change in the policy. Each episode is an IPD game, consisting of 60 periods. The performance indicator $P_{i,j}$ is calculated based on the best effort episode of block i in run j.

As the testing pool strategies differ in their average expected reward up to a factor of 2, we introduce the standardized performance measure for block i of run j

$$P_{i,j} = R_T^{\text{best}} - \frac{1-\gamma}{1-\gamma^T} \cdot \sum_{t=1}^{T} \gamma^{t-1} r_{t,i,j}$$

where $r_{t,i,j}$ is the reward for the policy in period t, block i and run j of the best effort episode. R_T^{best} is the numerically generated expected average reward of the proposed best answer of the considered strategy limited to T periods. The R_T^{best} for each strategy can be found in Appendix F and have been generated using 1000 episodes.

The correction factor $(1-\gamma)/(1-\gamma^T)$ accounts for the limited number of periods.
The performance indicator $P_{i,j}$ is a numerically generated variant of δ_R from Equation 2 in Section 2.1.1. Each generated $P_{i,j}$ is the realization of a random variable due to observation errors inherent to our version of the IPD game and might therefore take different values even given the same generating policy. In fact, some episodes generated a performance measure $P_{i,j}$ which is larger than zero even were the proposed best answer was indeed a best response.
Optimally, the performance of the algorithm is evaluated in a setting of indefinitely often repeated periods. For practical reasons however we chose a cut-off $T = 60$ after which we assumed all further rewards to be negligible, in analogy to Section 3.2. One might argue, that the algorithm was able to learn an end-of-episode effect, as it is always optimal to defect in the last period. Generously assuming that this last period would have been a cooperation otherwise, this does indeed bias our result - assuming the algorithm achieves to learn this effect - with a value of approximately 0.002. The end-of-episode effect is thus negligible compared to the effect sizes of the analyzed modifications.
There are two main approaches to reduce noise that allow for a comparison of parameter sets and their speed of learning:

1. Aggregate all runs of the same parameter set over $P_{i,j}$ with fixed block i to some measure
$$\tilde{P}_i = \sum_j P_{i,j}$$
The indicator $\tilde{P}_i$ shows the average performance of the given parameter set at block i. Comparing $\tilde{P}_i$ given different blocks i does allow to observe the speed of learning.
2. Use a smoothing function within each run j, where each data point $\tilde{P}_{i,j}$ is calculated by combing neighboring blocks i, e.g. using kernel smoothing.

The first approach allows for a fine view on the actual convergence of the algorithm over blocks. Solely relying on this approach is however time consuming because of the large number of runs necessary to obtain a sufficiently smooth, interpretable curve. The second approach might obfuscate the exact block number where a paradigm shift takes place, but since the performance of the algorithm tends to shift smoothly on aggregate, using neighboring data points allows for a better exploitation of our data. To limit necessary run time, we combined both approaches. Using a localized triangle kernel with 12 blocks to either side, the results within a run were smoothed and aggregated afterwards across several runs of the same parameter set. This defines an aggregated measure P_i for each block.
An example of plotting this smoothed indicator P_i is presented in Figure 14, here for the modification *Q-Switching.* The difference to zero can be interpreted as the remaining learning potential, given that the best answer R_T^{best} is indeed optimal.
Across Sections 3.4 to 3.11 we will use analogous figures to highlight the learning properties of the algorithm.

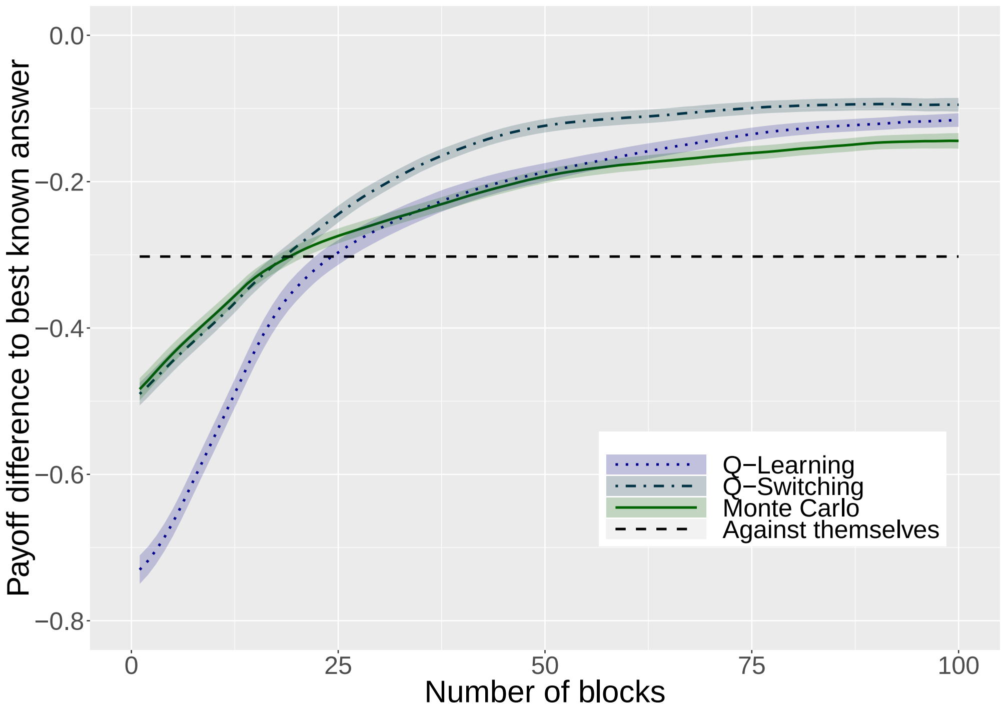

Figure 14: Exemplary convergence using the improvement *Q-Switching*.

The figures always show the performance of different parameter sets given a baseline algorithm. These changes are for example variations in the specifics of a modification or the existence the modification itself. The baseline algorithm used the modifications *Q-Switching*, a *Multi-Exploration* with a combination of *ε-exploration* and *noisy actions*, *Exploration Pathing*, the main encoding as defined by Section 3.7, the function approximator gradient boosting and *Experience Replay* but neither *Model Persistence* nor *Memory Initialization*.
The shaded areas behind the lines are 95%-confidence intervals as calculated according to the methodology of Morey (2008). Here, the differences between strategies are normalized to reflect a within-subject design. Additionally, a dashed black line denotes the average performance of each strategy against itself as a basic benchmark.
In those cases, where we want to compare a sizeable number of variants and are mainly interested in the final performance after training, we use the type of chart presented in Figure 15. Here, only the vary last block P_{100} with corresponding confidence intervals is shown. As with the figures varying across blocks, the dashed line shows the average performance of each strategy against itself.
In expectation, using more blocks leads to a better final performance. However, with an increasing number of blocks the marginal rate of improvement shrinks. Since each block costs computational resources, the training was limited to 100 blocks as a compromise between resources and final performance.

A more powerful function approximator (e.g. more trees for gradient boosting or more neurons in the case of a neural network) would improve the results of the algorithm as well[5]. This however would also increase necessary computational resources. In contrast to Section 3.2, a less powerful and more resource efficient function approximator has been used. This allowed us to run a sufficient number of runs to show significant effects. The specific parameters of the used function approximator are referenced in the corresponding sections. These less powerful function approximators explain the worse overall performance displayed in Sections 3.4 to 3.11 compared to the performance test of Section 3.2.

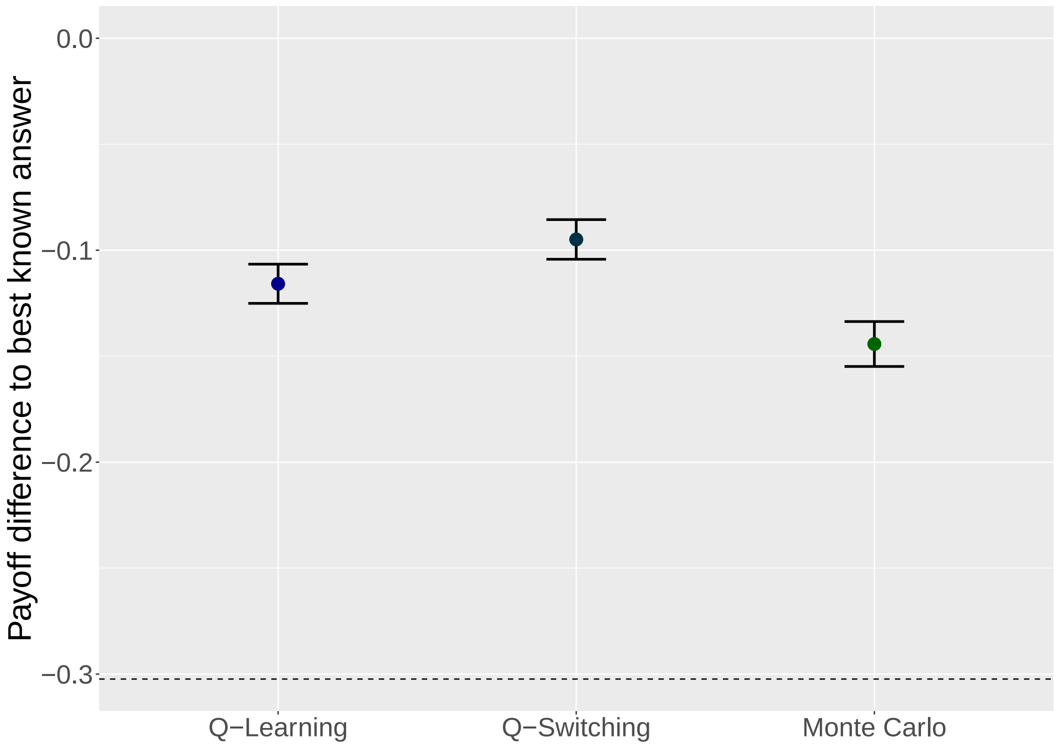

Figure 15: Exemplary performance at block $i = 100$ to show final performance. Here, the same data as in Figure 14 is presented. The dashed line shows the average performance of each strategy against itself.

[5]While increasing the number of blocks practically always improves or at least does not reduce performance, a similar effect does not necessarily hold true in the case of increasing the power of the function approximator. A very powerful function approximator in combination with a low number of blocks leads to overfitting of the training data. If one wants to increase the power of the function approximator, one has to increase the number of blocks in turn as well.

3.4. Q-Switching

Recall (see Section 2.3.3) that the update rule of Q-Learning is

$$Q(s,a) \leftarrow (1-\alpha)Q(s,a) + \alpha\left[r + \gamma \max_{\hat{a}} Q(s',\hat{a})\right]$$

where (s,a,r,s') are observed values of the tuple (S_t, A_t, R_t, S_{t+1}) from one of the played episodes.
On-policy Monte Carlo learning for a fixed policy and non-approximated state-action pairs under the assumption that each state can only be visited at a specific time t uses

$$Q(s,a) = \frac{1}{N}\sum_{i=1}^{N}\sum_{k=0}^{T-t}\gamma^k r_{t+k}^{(i)}$$

Here, N is the number of visitations of the given state-action pair (s,a) and $r_{t+k}^{(i)}$ the historic reward in period $t+k$ after its i-th visitation. The outer sum calculates the mean reward of all discounted rewards which have historically been experienced based on state s when taking action a.
The notation becomes more complex when encoding the states in a way that information is lost. Encoding for example each state with only its last X periods might group multiple distinct states. This grouped state representation might be visited several times within the same episode and at different times t across several episodes. Additionally, a function approximator might collapse distinct states onto grouped state representations as well.
As is detailed in Section 3.7, we used an encoding which includes a period counter. Consequently, the encoding does not collapse states of different periods by itself. Unless noted otherwise, an encoded state by construction can thus only be experienced once and at a specific point t within an episode. Using this encoding also implies, that the distinction between every-visit Monte Carlo and first-visit Monte Carlo is obsolete in our case. For the sake of simplicity, we will keep the notation despite the usage of a possibly collapsing function approximator to focus on the key elements of *Q-Switching*.
Q-Switching combines Monte Carlo learning and Q-Learning by changing the update rule to

$$\begin{aligned}\tilde{Q}(s,a) &= (1-\alpha)Q(s,a) + \alpha\cdot\left[r + \gamma \max_{\hat{a}} Q(s',\hat{a})\right]\\ Q_{\mathrm{MC}}(s,a) &= (1-\alpha_{\mathrm{MC}})Q(s,a) + \alpha_{\mathrm{MC}}\cdot Q_{\mathrm{hist}}^{(i)}(s,a)\\ Q(s,a) &\leftarrow (1-w)\tilde{Q}(s,a) + w\cdot Q_{\mathrm{MC}}(s,a)\end{aligned}$$

with

$$Q_{\mathrm{hist}}^{(i)}(s,a) = \sum_{k=0}^{T-t}\gamma^k r_{t+k}^{(i)}$$

The learning rate $\alpha_{\text{MC}} \in (0, 1)$ measures how strong the updating element $Q^{(i)}_{\text{hist}}$ is weighted and is conceptually identical to the learning rate α. The weighting factor $w \in [0, 1]$ balances $\tilde{Q}$ and Q_{MC} and therefore the weight which should be put on the Monte Carlo part of the update.
The value $\tilde{Q}$ is built according to Q-Learning while $Q^{(i)}_{\text{hist}}$ is the historic Monte Carlo value of some episode i. Without *Experience Replay* (see Section 3.8) and without using several episodes per block for the update of the function approximator, episode i is by definition the most recent episode. Otherwise $Q(s, a)$ is updated according to the specific rewards experienced in the former episode(s) currently used in the replay. To allow learning from single state-action pairs instead of complete episodes, it is necessary to save the tuple $(s, a, r, s', Q^{(i)}_{\text{hist}})$ into the memory instead of (s, a, r, s'). The value $Q^{(i)}_{\text{hist}}$ is only calculated once for each experienced state-action pair after completing the episode and is not updated with information of new episodes in contrast to normal Monte Carlo Learning. This prevents not only costly searches and recalculations but also allows us to forego possibly induced variance of the function approximator.
The weighting factor w is not constant, but rather calculated separately for each update of the function approximator. For early updates, the parameter w is set to high values and is reduced afterwards. For the purposes of our analyzes we started with $w = 1$ and decreased w by 10% with each function update. Alternative rules to decrease w lead to similar results. To reduce computational resources, one can for example refrain from calculating Q_{MC} if w is small enough to have no discernible impact. If w converges towards zero, *Q-Switching* converges towards Q-Learning and consequently inherits the same properties. If one for example uses *ε-exploration*, *Q-Switching* converges towards the optimal policy given that the rate of convergence of w towards zero is higher than the one of ε.
The basic intuition to use Monte Carlo learning is that it stabilizes and jump-starts the learning process. As mentioned in Section 2.3.2, Monte Carlo is a viable choice if the underlying problem does not rely too much on the Markov property. This might for example be the case if the to be countered strategy actively uses the current period number or a long history of observed actions to determine its next action. Additionally, Monte Carlo learning shows a better performance, if the first few periods determine actions of much later periods. Here, Q-Learning has to use a lot of update cycles to propagate the Q-Values through the periods. This effect becomes more relevant given a very high discount factor γ and a high maximum period limit T.
Another advantage of Monte Carlo learning is, that in the early stages of the learning process the value of $\max_{\hat{a}} Q(s', \hat{a})$ is inaccurate. A strong dependence on the function approximator to correctly extrapolate from past experiences is the result. The value $Q^{(i)}_{\text{hist}}$ on the other hand is known and well defined by former episodes and can thus be used more reliably.

Monte Carlo learning has the major disadvantage that one does not receive accurate Q-Values of the best policy. Instead, the calculated values represent the specific policy used when generating $(s, a, r, s', Q_{\text{hist}}^{(i)})$. This leads to two major effects:
With an increasing number of played periods, the algorithm should be able to improve its actions and receive higher rewards than historically observed. For this reason, the value $Q_{\text{hist}}^{(i)}$ becomes inherently more untrustworthy compared to $\max_{\hat{a}} Q(s', \hat{a})$ the more the understanding of the environment increases. When using *Experience Replay*, i.e. updating the function approximator based on historic observations, older episodes might thus misrepresent the actual values which can be achieved by taking a certain action. Section 3.8 shows, that it is beneficial to use these older episodes to boost the rate of learning. Simply using only the very last episode is therefore not a viable alternative.
The other effect is, that even if one limits oneself to the very last episode, $Q_{\text{hist}}^{(i)}$ is still influenced by the occurring exploration and is therefore not a correct representation of the optimal policy. Imagine for example a situation where we must chose between two general ways of playing: On the one hand, we can play a very specific string of actions to receive a high reward. This comes at the cost of receiving a very low reward if we deviate slightly. On the other hand we can play a safe policy which always generates a medium reward. Using Monte Carlo learning based on exploration episodes, which sometimes play suboptimal actions, lets us severely underestimate the potential of the first path.
One can mitigate the effect of learning from suboptimal policies by using so-called *importance sampling*. Here, data from a suboptimal exploratory policy is used to improve the understanding of the optimal policy by weighting the rewards according to the similarity of the policies. This, however, is computationally expensive and might introduce a lot of variance into the results (Silver, 2015, lecture 5, p. 37).
Q-Learning, by contrast, does not need importance sampling (Silver, 2015, lecture 5, p. 39) and works well with *Experience Replay*. Here an accurate model of the environment can be build by the function approximator. This makes Q-Learning a good choice to find the optimal strategy. However, the major drawback is, that it takes several updates of the function approximator to transport the information of a good early decision back to the state where the decision was made. In fact, in early tries using a neural network as a function approximator and only the average expected reward of the episode as learning input all useful information got lost in the variance of the function approximator. Consequently, the algorithm did not converge against a high performing policy. Using a Monte Carlo approach fixed this problem, which suggests, that the advantages of *Q-Switching* are even more pronounced in a setting with sparse rewards or in games with a very long duration where the opponent uses the complete history to make decisions.

By using *Q-Switching*, we are able to benefit from the advantages of Q-Learning and the Monte Carlo approach: Early on the Monte Carlo approach sets the state from which a more refined policy can be developed. Interestingly, using *Q-Switching* also works even if the exploration strategy is completely random at this point of the learning process and one might thus expect $Q^{(i)}_{\text{hist}}$ to be quite inaccurate. Still, a sufficiently powerful function approximator is able to infer that on average certain combinations of actions lead to good or bad outcomes and estimates the corresponding actions accordingly. While the estimated values $Q(s,a)$ might be inaccurate, the order of actions in terms of added value might still be correct, leading to a good policy. With increasing number of episodes and updates of the function approximator the understanding of the environment increases, allowing the algorithm to face more complex combinations, which makes it beneficial to increasingly switch to Q-Learning. The difference of the convergence between pure Monte Carlo learning as specified by Q_{MC}, Q-Learning and *Q-Switching* is shown in Figure 16.

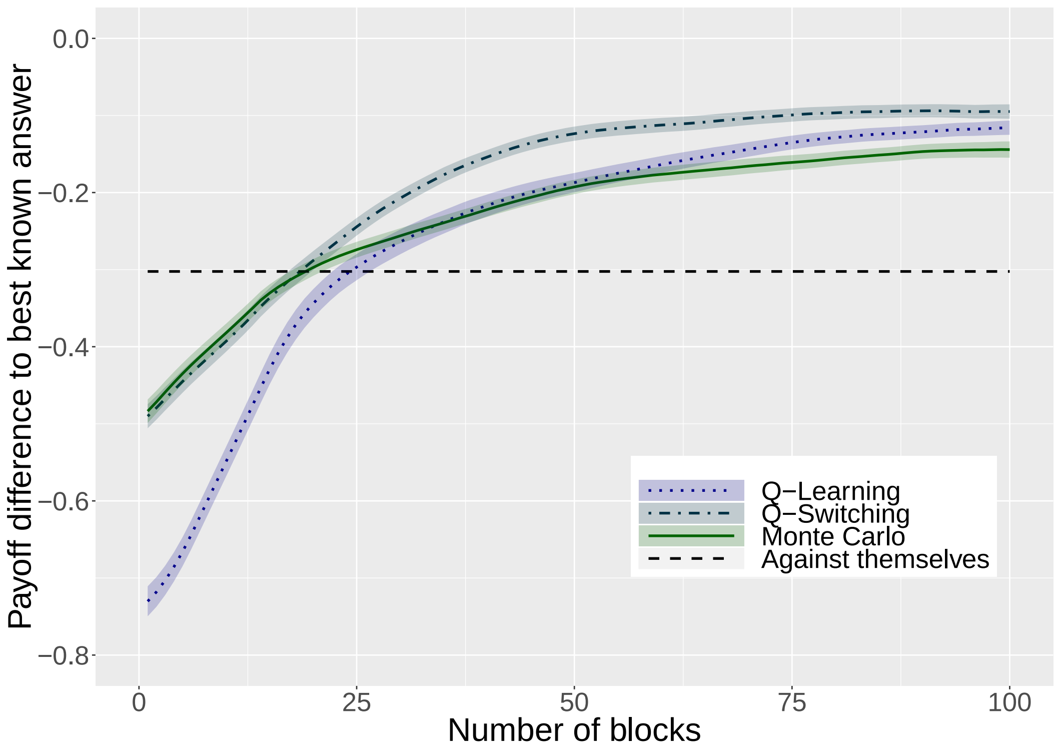

Figure 16: Comparison of Q-Learning (blue, dotted), *Q-Switching* (dark turquoise, dot-dashed) and Monte Carlo learning based on Q_{MC} (green) according to the methodology described in Section 3.3.
The *Q-Switching* weight w starts at unity and decreases by 10% each block. We chose $\alpha = \alpha_{\text{MC}} = 0.25$ for all variants. Each calculation is based on 50 runs of each of the strategies. For the specific parameters we refer to Appendix G.2.

We see that *Q-Switching* develops a suitable policy much faster in terms of number of necessary blocks compared to both pure variants. In the beginning, *Q-Switching* and Monte Carlo learning are identical, so they show a very similar performance. By allowing Q-Learning to take over, *Q-Switching* is able to generalize the necessary patterns to improve the performance over both pure approaches. While all three approaches seem to converge towards the same value[6], even after 100 blocks Q-Learning was not able to catch up to *Q-Switching*.

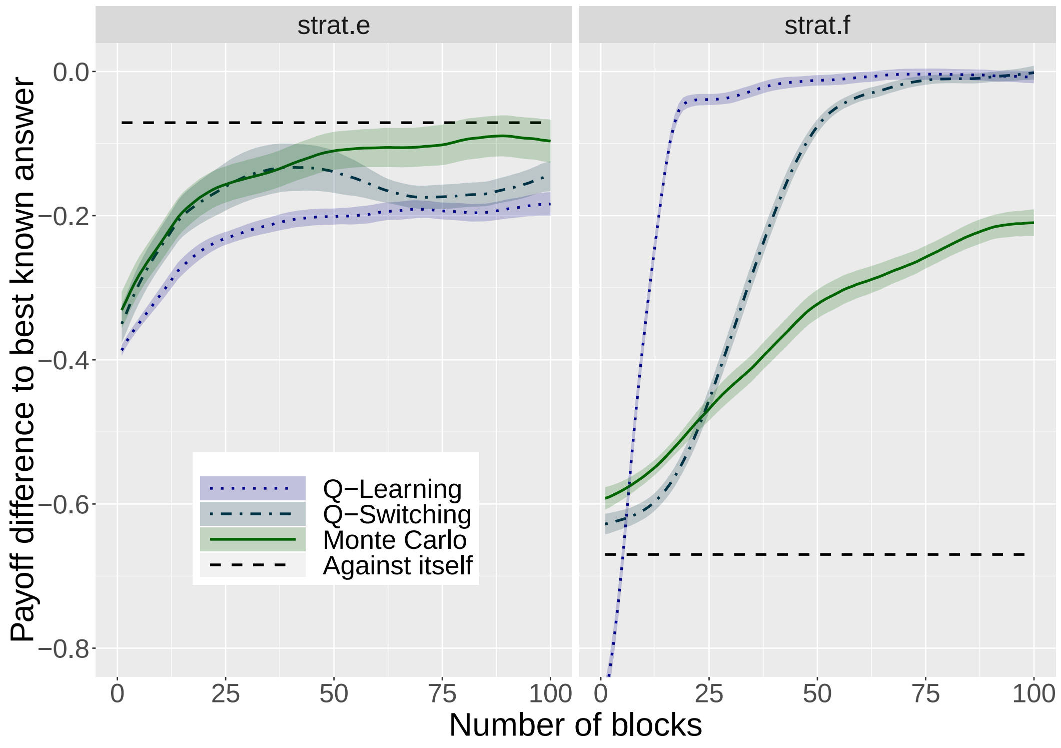

Figure 17: Comparison of Q-Learning (blue, dotted), *Q-Switching* (dark turquoise, dot-dashed) and Monte Carlo learning based on Q_{MC} (green) according to the methodology described in Section 3.3 for the strategies *strat.e* and *strat.f*.
The *Q-Switching* weight w starts at unity and decreases by 10% each block. We chose $\alpha = \alpha_{\text{MC}} = 0.25$ for all variants. For the specific parameters see Appendix G.2.

The optimal approach depends strongly on the examined strategy. From Figure 17 we see, that neither approach is always the best. Here, we chose the most extreme strategies based on their final performance in regards to Q-Learning and Monte Carlo Learning.

[6]All figures within this section have been generated with the same *Exploration Path* with a final target exploration value of 99% at block 100. As Monte Carlo learning in our implementation is strictly on-policy learning, it is expected that the final value of Monte Carlo learning is slightly lower due to the exploration effects. Given our specific setting, this effect lowers the expected average reward of the final block by approximately 0.005.

Both strategies are described in detail in Appendix F, but the key difference can be summarized as follows:
strat.e is a *grim.trigger*-like strategy which defects with some random component. The best answer cooperates until it detects that observation errors have triggered *strat.e* and defects afterwards. Finding a good policy is thus a question of identifying whether the strategy is triggered or not. The strategy uses the complete observed history to check whether it should be triggered, so it is not sufficient to check the last few observations. Optimally, the algorithm is able to detect the trigger condition at the same time as *strat.e* itself. As a second best, the algorithm might recognize whether the strategy has been triggered after the fact and change to an always-defect mode. In both cases one is not able to mitigate the negative effects by playing specified patterns. Once triggered, all subsequent Q-Values are considerably worse than before the triggering. Given the situation that the next action might trigger *strat.e* and both actions have been experienced in the past, Monte Carlo learning shows a clear difference in the evaluation of those actions, even if a pure random policy is played in the subsequent periods. Q-Learning on the other hand has to use several updates to pinpoint the exact situation where the triggering happened.
This in mind, one can influence the performance of *Q-Switching*: If one suspects a path dependence, the performance of *Q-Switching* can be considerably increased by modifying the changing-rate of the switching parameter w. The higher the path dependence, the longer the algorithm should stay with Monte Carlo learning and vice versa. This effect is further highlighted by comparing the performances in regard to strategy *strat.f*.
Strategy *strat.f* does not take into account a long history. The best answer against this strategy needs the last 4 periods of data to determine the best course of action. Here, Monte Carlo learning loses all its advantages, as it is not possible to make a lasting mistake and there is nearly no path dependence. Q-Learning is quick in determining the optimal pattern and stays there, as there is no necessity to attribute a bad reward to a far-away action. *Q-Switching* takes much longer to converge because it has to unlearn all path dependencies which where implied by its Monte Carlo aspect.
Having the extreme examples of *strat.e* and *strat.f* in mind, it might be possible to improve upon the mechanic decrease of the weight w by determining a way to automatically detect whether Q-Learning or Monte Carlo learning is more relevant. Future research might find cheap heuristic ways to set w, which could considerably improve performance.
Additionally to the assumed path dependence, the complexity of the strategy can also be used to set weighting factor w. A more complex strategy which takes a lot of information into account might require a more nuanced policy and more input for the function approximator. In this case one might want w to decrease slower as this relates to less dependency on the accuracy of the function approximator. One way to determine the complexity of the strategy at run time might be to use a second, more simple, function approximator to predict the next action of the strategy. If the accuracy of this estimation is low, this might imply a complex strategy.

Another advantage of *Q-Switching* is shown in Figure 18.

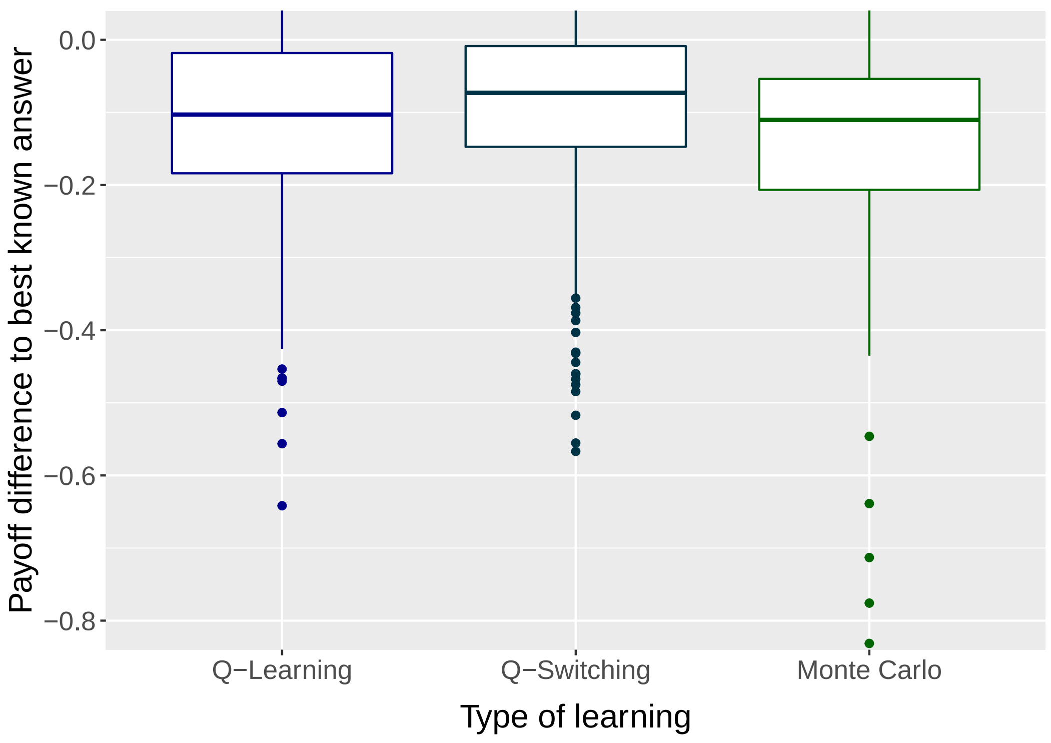

Figure 18: Boxplot of performance in the final block of Q-Learning (blue), *Q-Switching* (dark turquoise) and Monte Carlo learning based on Q_{MC} (green) using the testing pool strategies.
All other parameters have been held constant. The *Q-Switching* weight w starts at unity and decreases by 10% each block. For the specific parameters we refer to Appendix G.2.

Figure 18 shows that not only the final performance is higher with *Q-Switching* than with the pure alternatives, but that it is also more stable. While one should not draw strong conclusions about the outliers, the difference between the 25% percentil and the 75% percentil is lower as indicated by the size of the box. In a practical setting one probably wants to run the algorithm only once or at at least very few times, so using *Q-Switching* provides a more stable approach to assess the stability of a strategy.
While *Q-Switching* to my knowledge is a novel approach to combine the advantages of Q-Learning and Monte Carlo learning, there have been other approaches to combine long-term and short-term views on the rewards:
Both, one-step-temporal difference learning and Monte Carlo learning, as described in Section 2.3.2 may be generalized to so-called TD(λ) learning, developed by Sutton (1988). A parameter λ is defined, which determines how strong the effects of future rewards have to be weighted. A λ of unity implies an equal weighting of all discounted

rewards and thus Monte Carlo learning. On the other hand, vanishing λ implies that only the very next step is weighted, i.e. one-step temporal difference learning, TD(0). The parameter λ controls the importance of future rewards by discounting them exponentially[7]. In contrast to *Q-Switching* in the limit case, TD(λ) is on-policy, i.e. does not explicitly optimize over actions but rather explains the current policy. As we want to find the optimal policy, we have to let the current policy converge towards the optimal one when using TD(λ). This necessitates a difficult balance in the speed of changing the exploration parameter, as decreasing the exploration too fast increases the risk of getting stuck in a local optimum. Tuning the parameter w is a lot less problematic, as it only changes the speed of convergence towards the global optimum, as long as w does indeed go to zero in the limit, given that we always have some amount of exploration.

This strictly on-policy concept of TD(λ) has been extended to Q(λ)-Learning by Peng and Williams (1994). Even though here a Q-Learning part allows for better generalizations, the values generated this way are still policy-dependent. Consequently, Q(λ) learning "cannot be expected to converge to the correct [theoretical Q-Values] under an arbitrary policy that tries every action in every state" (Peng and Williams, 1994, p. 6). They note, that gradually reducing the λ Parameter would mitigate this. Indeed, while being calculated differently, this is very comparable to reducing the parameter w in *Q-Switching*. Using a constant factor w would result in the same problem. The drawback of using Q(λ)-Learning however, is that the values for the state-action pairs are recalculated at each step, thus being a lot more computational expensive than our approach.

Another on-policy variant of combing TD(λ)-Learning with Q-Learning which shows better performance than Q(λ)-Learning can be found in Wiering (2005). They use the state-value function (i.e. the $Q(s, a_\pi(s))$ values) to calculate their Q-Values. The same drawbacks as with TD(λ) apply due to the on-policy nature of the algorithm.

The state-of-the-art Q-Learning classed algorithm *Rainbow* (Hessel et al., 2018), as measured by the performance on ATARI games, uses so called *multi-step learning* to address the challenge Q-Learning faces with its myopic update rule. With multi-step learning not only the very next step but a forwards-view n-step reward is used. The integer n is generally of medium to short length - in fact, they find within their setting $n = 3$ to be optimal. This improvement, while certainly helpful compared to basic Q-Learning, is not sufficient in our case due to the strong path dependence one can expect from strategies of the IPD game. Classificator strategies as for example discussed in J. Li et al. (2011) use the first few periods to determine the rest of their play. The optimal actions against those strategies strongly depend in the those first periods and it is thus necessary to take a very long term view as we do with Monte Carlo learning.

[7]This is a direct consequence of not being able to visit the same state twice within an episode. As there are no loops, the concept of *eligibility traces* simplifies.

3.5. Multi-Exploration

In addition to the basic *ε-exploration* several other types of exploration strategies exist. Thrun (1992) gives a review of the most common ones. We focus on the following exploration strategies:

1. *ε-exploration*
   The most basic exploration strategy: With a certain probability a random action is taken.
2. *noisy actions*
   A variant of so called *Boltzmann exploration*. To each calculated Q-Value a normally distributed random variable is added and the action with the highest value is taken.
3. *maximizing surprise*
   Actions which resulted in a stronger update of Q-Values of comparable situations in the past are preferred.
4. *minimizing familiarity*
   An additional function approximator tries to keep track which action has been taken in the past in a similar situation. Less common actions are preferred.

No single exploration strategy is dominant, which is why the algorithm may use all of them at the same time in unison. On the specific balancing of the exploration strategies the reader is referred to Section 3.6, the section about *Exploration Pathing*. This section is organized as follows: First every exploration strategy is explained in detail. Subsequently we will have a look at the performance of using solely these single exploration types. Afterwards we discuss why it can be a good idea to combine them and whether such a combination is indeed beneficial in our case.

3.5.1. ε-exploration

One of the most basic exploration strategies is *ε-exploration* with decreasing ε:

```
With P($\varepsilon$):
    Choose random action based on uniform distribution over all actions
Else:
    Choose greedy action, i.e. $a = \operatorname{argmax}_{\hat{a} \in \mathcal{A}} Q(s, \hat{a})$
```

Given infinitly many episodes and no time or memory limitations, this exploration strategy convergences against the optimal policy π^* given a sufficiently slow descent of ε and an on-policy learning algorithm such as SARSA (see Section 2.3.2). This descent is sufficiently slow if all states are visited infinitely often, despite vanishing ε.

Q-Learning and in the limit case *Q-Switching*, however, are off-policy algorithms. It is thus not necessary that ε actually converges towards zero. In an extreme case one might have a constant ε of unity (*random exploration*) and due to the converging property of Q-Learning itself the optimal policy π^* will still be found (Sutton and Barto, 2018, p. 108).
However, such an approach is inefficient. As discussed in Section 3.6, it can be beneficial to focus on learning to play the best action given optimal and near-optimal play. Learning to play optimal actions in situations which only arise after playing suboptimally is not an efficient use of exploration episodes if one wants to estimate the potential of the optimal policy. Decreasing ε is therefore beneficial even given an off-policy algorithm.
One of the main drawbacks of *ε-exploration* is that the environment might react with differing harshness to a suboptimal action. Some strategies only search for patterns within the last few periods. Missing one of such patterns leads to a suboptimal performance, but the severeness of the error is limited. With strategies which behave *grim.trigger*-like, we might find ourselves in the situation, that playing the wrong action in a key situation has severe consequences for all following rewards of this episode. As is shown in Section 3.5.5, making the right choice in this situation can help to increase the usefulness of the currently played episode. A common exploration strategy which aims to minimize very unfavorable actions is soft-max learning, usually in the form of Boltzmann exploration, which will be discussed in the following section.

3.5.2. Noisy Actions

The main idea behind the exploration strategy *noisy actions* is to play more promising actions with a higher probability. This allows to vary uniformly between actions which have about the same Q-Value and to refrain from playing an action which has a high probability to waste the usefulness of the episode. The most discussed variant is *Boltzmann exploration* (see e.g. Kaelbling et al. (1996)).
Here the action to be played is chosen according to the Boltzmann distribution

$$P(A_t = a_i|S_t) = \frac{e^{-Q(S_t,a_i)/\theta}}{\sum\limits_{j=1}^{M} e^{-Q(S_t,a_j)/\theta}}$$

with M being the number of actions, and $\theta > 0$ controlling the exploration strength. For θ approaching infinity a random action is chosen and for θ approaching zero only the best action is used. For fixed θ, actions with higher Q-Values are chosen more often. In the case of two identical Q-Values a random selection is the result. For all values of θ every action is chosen with non-zero probability. As a result, with infinitely many repetitions and sufficiently slow convergence of θ approaching zero, all combinations of the environment will be explored and the optimal policy will be achieved.

Given only two actions to chose from, this distribution is identical to a logistic function (Sutton and Barto, 2018, p. 29). In fact, the well-known multinomial logit model as used in economics has the same form as the Boltzmann distribution.
The Boltzmann distribution has remarkable mathematical properties, but conceptually easier and with a factor of 4 faster to compute is the following approach to choose an action, which is used by the algorithm:

```
Calculate $Q(s, a_j)$ for all $a_j \in \mathcal{A}$ for a given state $s$
Draw iid variables $N_j \sim N(0, \sigma^2)$, $j = 1, .., M$ # M equals number of actions
For  $j = 1$ to $M$:
    $Q'(s, a_j) \leftarrow Q(s, a_j) + N_j$
$a \leftarrow \underset{a_j}{\text{argmax}}\, Q'(s, a_j)$
```

The basic idea is to add to each calculated Q-Value a random shock N_j with some variance $\sigma^2 > 0$ and to take the highest resulting value. For all intents and purposes this is fundamentally the same approach as Boltzmann exploration, except that a slightly different distribution for the shock is used. In fact, using an extreme value distribution for N_j instead of a normal distribution would imply that the *noisy action* approach is mathematically identical to Boltzmann exploration. See Appendix D for a numerical comparison with respect to similarity and speed.
Consequently all relevant features can be found analogously:
For σ approaching infinity the calculated Q-Values lose their impact and only the random draw of the shock is relevant. A uniformly random choice between the actions is the result. For vanishing σ the algorithm approaches a greedy selection. For all fixed values of σ all actions will be played with non-vanishing probability, thus ensuring to find the optimal policy. If the calculated Q-Values have a higher distance to each other, for fixed σ it is less likely to chose the lower estimated option.
Both the approach presented above and Boltzmann exploration suffer from the same drawback I encountered frequently in the development of the algorithm: While those methods outperform *ε-exploration* in finding a better and more precise policy, there was a noticeable probability that the policy got stuck in a local optimum. Due to the theoretical possibility of choosing a series of actions to escape the local optimum with a sufficient amount of tries, this effect can be mitigated by having a slower decrease of σ and training longer. Nevertheless this effect is at odds with our goal to find a best answer in a reasonable time frame.
An interesting alternative to making the actions noisy has been explored by Plappert et al. (2017), who add noise to the parameters of a neural network, i.e. change the values of the neurons slightly, instead of adding noise to the actions. As we analyze and use different types of function approximators (see Section 3.9), I refrained from implementing this method.

3.5.3. Maximizing Surprise

Inspired by the general ideas of Martin et al. (2017), Burda, Edwards, Storkey, et al. (2018) and Burda, Edwards, Pathak, et al. (2018), the third variant of *Multi-Exploration* is based on surprise: Ideally, the algorithm notices which action led to a considerably different outcome as expected in a previous block and tries to recreate the situation in an effort to further understand what has happened. Especially in combination with other exploration variants doing so could lead to a higher probability to explore situations which provide richer information about the environment.
Incorporating surprise as the only exploration variant results in the following way to chose an action:

```
Calculate $Q(s, a_j)$ for all $a_j \in \mathcal{A}$ for a given state $s$
Calculate $\tilde{S}(s, a_j)$ for all $a_j \in \mathcal{A}$ for a given state $s$ based on a function approximator.
**For** $j = 1$ to $M$:
    $Q'(s, a_j) \leftarrow Q(s, a_j) + s_{\text{surp}} \cdot \tilde{S}(s, a_j)$
$a \leftarrow \underset{a_j}{\text{argmax}}\, Q'(s, a_j)$
```

Here, a second function approximator is trained the following way within the general updating routine:

$$\tilde{S}(s, a) \leftarrow (Q_{\text{old}}(s, a) - Q_{\text{new}}(s, a))^2 \quad \forall (s, a) \text{ within the training batch}$$

With *maximizing surprise* the algorithm tries to estimate the quadratic difference between the Q-Value of a given state-action pair before the update of the function approximator, $Q_{\text{old}}(s, a)$, and the Q-Value of the same state-action pair after the update, $Q_{\text{new}}(s, a)$. Actions which let to a higher difference receive a preferential treatment in the exploration episodes. The strength of this treatment is calibrated though the factor $s_{\text{surp}} > 0$ which is changed each block according to the exploration path, analogously to ε of *ε-exploration* and σ of *noisy actions*.
The calculation of $\tilde{S}(s, a)$ has been inspired by the supervised learning method *ordinary least squares*. Using the quadratic distance has the effect, that it does not matter, whether the Q-Value increased or decreased. Additionally, more pronounced changes in Q-Values are over-weighted compared to a linear approach. Further research might show whether this particular formulation is superior (and in which cases) to maximizing positive surprises or to an equal treatment of all deviations.
Using *maximizing surprise* has the disadvantage, that by design the algorithm is only driven by already experienced changes in Q-Values. Additionally, using this concept of surprises can lead to procrastination-like behavior, where the algorithm optimizes to reach environments which have a high rate of natural noise. Confusing Q-Value

updates which occur due to increased environmental randomness with updates which reflect actual learning progress may severely hinder learning performance (Burda, Edwards, Pathak, et al., 2018).

3.5.4. Minimizing Familiarity

Minimizing familiarity aims to avoid the backward-looking behavior of *maximizing surprise* while keeping an explicit drive of the agent to explore the environment. Similarly to *noisy actions* and *maximizing surprise*, *minimizing familiarity* is implemented the following way:

```
Calculate $Q(s, a_j)$ for all $a_j \in \mathcal{A}$ for a given state $s$
Calculate $\tilde{F}(s, a_j)$ for all $a_j \in \mathcal{A}$ for a given state $s$ based on a function approximator.
**For** $j = 1$ to $M$:
    $Q'(s, a_j) \leftarrow Q(s, a_j) + f_{\text{fam}} \cdot (1 - \tilde{F}(s, a_j))$
$a \leftarrow \underset{a_j}{\text{argmax}}\, Q'(s, a_j)$
```

The factor $f_{\text{fam}} > 0$ is calibrated each block to keep the exploration episodes on the exploration path. The value $\tilde{F}(s, a_j)$ aims to measure the familiarity of choosing action a given state s.
Let $\bar{\pi}$ be a notional policy, which represents the past behavior of the algorithm. The approximation $\tilde{F}(s, a_j)$ is calculated using a second function approximator and aims to predict the probability with which the policy $\bar{\pi}$ would chose the given action. In other words, given that the algorithm has always chosen action a_j given state s, $\tilde{F}(s, a_j)$ should be predicted to be unity. Analogously to the estimation of Q-Values, the function approximator has to extrapolate $\bar{\pi}$ to states which have never been previously encountered. For the purpose of representing $\bar{\pi}$ no distinction is made between exploration and best effort episodes.
$\tilde{F}$ is updated the following way within the general updating routine:

```
For each $(s, a_j)_i$ within the memory batch underlying the update:
    Define $F_i(s, a_j) \leftarrow 1$
    Define $F_i(s, a_k) \leftarrow 0 \; \forall k \neq j$
Train/Update the function approximator $\tilde{F}$ on those $F_i$.
```

To have a sufficient prediction power of function approximator $\tilde{F}$, it is necessary to use a sufficiently large set of experiences and/or a function approximator, which is updateable and does not forget past trainings. However, even using only the last block and a non-updating function approximator (e.g. gradient boosting which is only calculated based on this last block) results an interesting effect: The algorithm is encouraged to do something different compared to the last block, but only to a certain degree. As $f_{\text{fam}} \cdot (1 - \tilde{F}(s, a_j))$ still has to be greater than the difference between the best action and its alternative this approach aims to avoid very suboptimal plays.

3.5.5. Effects of Different Exploration Types

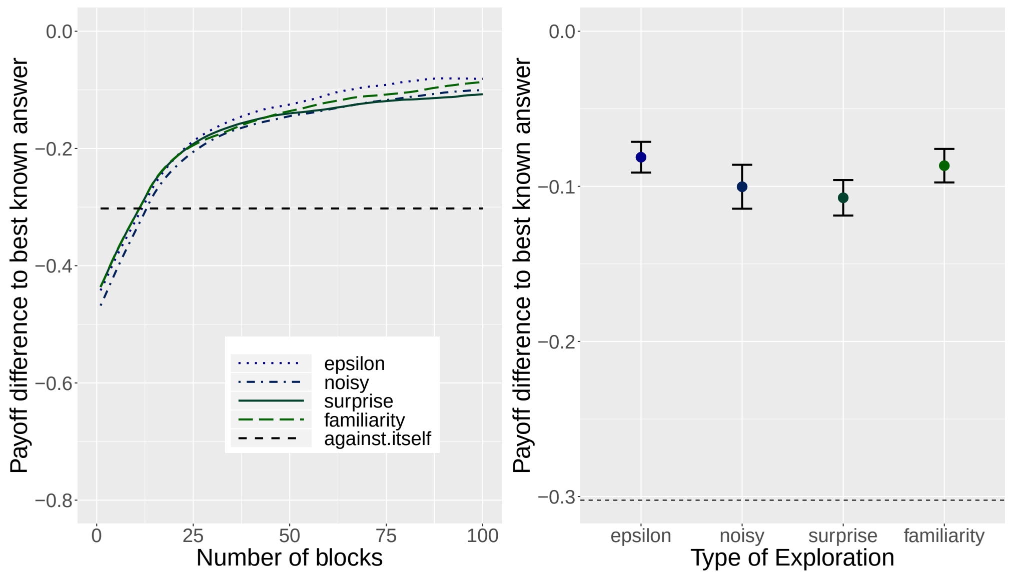

Figure 19: Comparison of *ε-exploration*, *noisy actions*, *maximizing surprise* and *minimizing familiarity* over all strategies according to the methodology described in Section 3.3. No confidence bands are shown to improve readability.
All parameter sets used the same exploration path. For each result 35 runs of each of the 9 testing pool strategies were used. For the specific parameters and more detailed information see Appendix G.3

Both, theoretical considerations and empirical evidence, suggest that using more sophisticated exploration methods can be benefical over *ε-exploration* (see e.g. Thrun (1992)). In our specific case this does not hold true however. All methods show approximately the same learning behavior as can be seen in Figure 19. In fact, after 100 blocks *ε-exploration* shows the best performance. *Minimizing familiarity* performs second best, but uses a second function approximator. Even if one limits the potential of this function approximator, this necessarily implies much more computational power. *Maximizing surprise* does worst, but not by a great margin. I also implemented count-based exploration in the feature space as developed by Martin et al. (2017) as a baseline alternative to *maximizing surprise*, but while working fine on toy models, it often ran into numerical instabilities within our setting.

Based on the results discussed above, one might be inclined to think, that the best exploration method given the IPD game is *ε-exploration*. This is not necessarily the case and might depend on our specific setting, e.g. the high probability of observation errors. An example where *noisy actions* is superior to *ε-exploration* is shown in Figure 20. The data for Figure 20 has been generated using an IPD game without observation errors and against the strategy *compare.exploration*.
I developed the strategy *compare.exploration* (see Appendix E for the pseudo code) to highlight the difference between *ε-exploration* and *noisy actions* and to explain why it can be beneficial to combine several exploration strategies. When detecting a defection within the first nine periods, *compare.exploration* reacts *grim.trigger*-like and defects from here on out. After the first nine periods the strategy categorizes its opponent as "generally friendly" and repeats the opponents move, similar to *tit.for.tat* (see Section 2.1.2). To allow the policy to learn something more complex as having to play *always.cooperate*, a random (but fixed) selection of 15 specific periods after the first ten ones have been sampled, where the strategy will cooperate against any "generally friendly" strategy independent on the most current observation.
A good procedure to find a best answer from a human perspective is to quickly understand that in the beginning there is a *grim.trigger*-like time span and to try to pinpoint the exact period where this mode changes. Afterwards one should focus all exploration efforts on later periods to identify those where a defection goes unpunished. Indeed, *noisy actions* focuses its exploration efforts most times on the later periods after having experienced very unfavorable rewards when defecting in the early periods. There is however a considerable drawback:
With non-vanishing probability the first episodes are played in such a way, that the function approximator draws wrong conclusions. In the most extreme case it may assume every deviation from *always.defect* to be expensive. From its point of view, the Q-Values of both actions are then so different from each other that the random shock is insufficient to bridge their gap with a relevant probability. As a result the algorithm is stuck in an unfavorable local optimum. Consequently all played episodes, exploration episodes and best effort episodes alike, are played according to *always.defect*. This behavior is only temporary due to counter-acting effects of the modification *Exploration Pathing* (see Section 3.6). The modification increases σ until a difference between exploration episodes and best effort episodes can be observed. Nevertheless this is suboptimal behavior as several blocks are needed to adjust and then re-adjust the level of σ, thereby wasting blocks where no profitable exploration takes place.
As this example shows, the main motivation for combining different exploration methods is to decrease the chance of getting stuck in a local optimum while still focusing on the most promising actions even given the possibility that the function approximator might completely misjudge the environment.

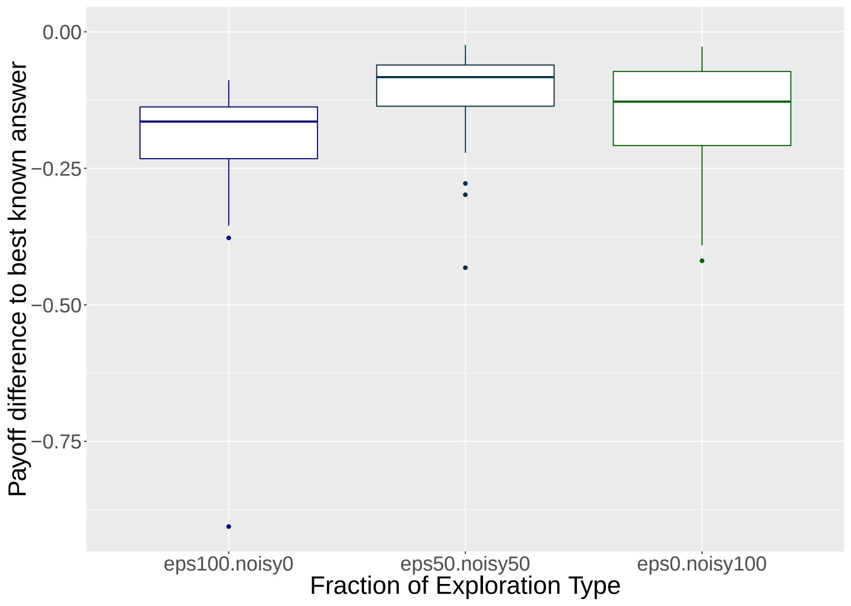

Figure 20: Boxplot of the final performance of *ε-exploration* (eps100.noisy0), *noisy actions* (eps0.noisy100) and a combination of both of them (eps50.noisy50) against the strategy *compare.exploration*. Each Boxplot has been generated using 50 runs. In contrast to other figures this IPD game was played without observation errors. The specific parameters are listed in Appendix G.3.

3.5.6. Effects of Multi-Exploration

The next step is to investigate the effects of *Multi-Exploration* based on the exploration methods *ε-exploration* and *noisy actions*. Given only those two methods, *Multi-Exploration* is identical to the following algorithm:

```
Calculate $Q(s, a_j)$ for all $a_j \in \mathcal{A}$ for a given state $s$
With P($\varepsilon$):
    Choose random action based on uniform distribution over all actions
Else:
    Draw $M$ iid variables $N_j \sim N(0, \sigma^2)$ # M equals number of actions
    For $j = 1$ to $M$:
        $Q'(s, a_j) \leftarrow Q(s, a_j) + N_j$
    $a \leftarrow \underset{a_j}{\text{argmax}}\, Q'(s, a_j)$
```

The balancing between the exploration values σ of the *noisy actions* part and ε of the *ε-exploration* part is done by the mechanism of *Exploration Pathing.*

A comparison of the final performance between the pure methods and *Multi-Exploration* against the strategy *compare.exploration* is depicted in Figure 20. Here, *Exploration Pathing* aimed to keep an equal balance between both exploration methods in the *Multi-Exploration* approach. One can see, that *Multi-Exploration* not only achieves a better performance in most cases, but also shows considerably less variance in its results.
Using only *ε-exploration* implies that each action has the same probability to deviate. Within the first nine periods a deviation is very costly due to the *grim.trigger*-like behavior of *compare.exploration*. Given that the algorithms has already learned that the first nine periods should never be defected, a defection within this time frame has no potential to generate an improved policy. It is thus necessary to chose ε sufficiently small to have a realistic chance of playing nine cooperations in sequence. Such a choice, however, has the disadvantage that very few exploratory actions occur within the periods of interest. Using *noisy actions* allows us to approach this problem by concentrating our deviations on those actions which are more promising. However, only using *noisy actions*, there is a non-vanishing chance of getting stuck in a local optimum. Combining both approaches increases the probability of playing actions outside of the local optimum and does therefore allow a change in the policy. This increases the performance of the algorithm, even at the cost of wasting some of the episodes by defecting within the first few periods.

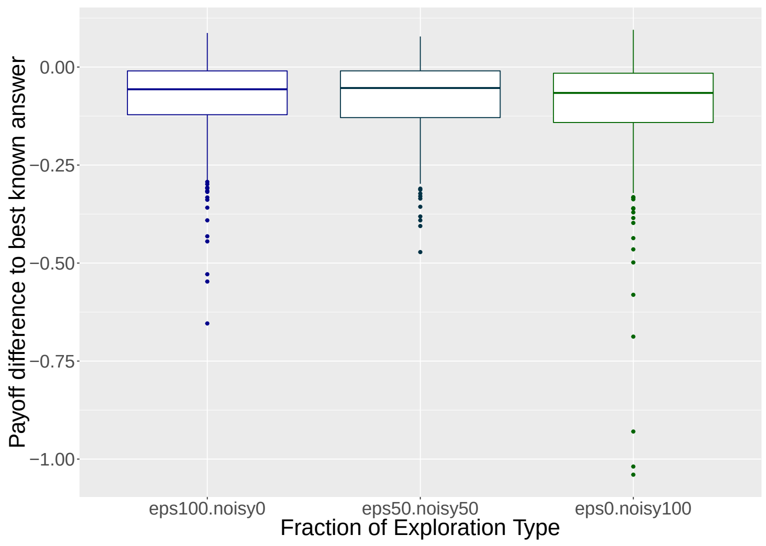

Figure 21: Boxplot of the block 100 performance of pure *ε-exploration* (left), equal mixing of *ε-exploration* and *noisy actions* through an exploration path (middle) and pure *noisy actions* (right) over all testing pool strategies. Each Boxplot has been generated using 35 runs on each of the 9 testing pool strategies.
Note that a performance of larger than zero does not imply that we have found a policy which consistently beats the best answer, due to the noisiness of the game. The specific parameters are listed in Appendix G.3

Figure 21 shows the analogous situation for our setting with observation errors using all strategies of the testing pool. The positive effect is not nearly as pronounced compared to the proof-of-concept example. While the combination of both exploration methods is slightly better in terms of median performance, this difference is barely noticeable and not significant. Given the complete testing pool one can see, that *noisy actions* generated outlier performances which are considerably worse compared to the other approaches. This effect can be mitigated by setting a more exploration-oriented *exploration path*, i.e. higher values of σ and ε throughout the run, in combination with a higher number of blocks. Increasing the number of blocks also increases computational power, so using *Multi-Exploration* might be a better alternative.

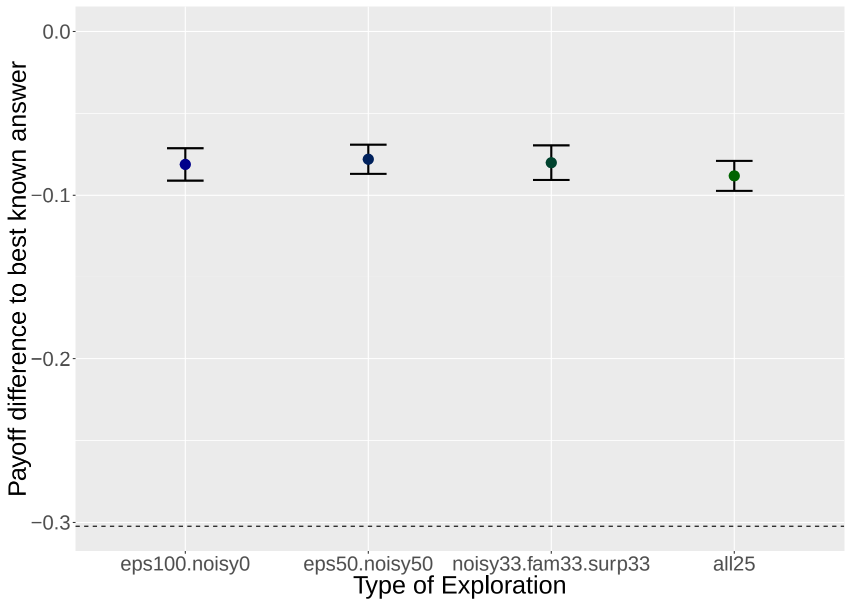

Figure 22: Comparison of pure ε-*exploration*, an equal mixing of ε-*exploration*, an equal mixing of ε-*exploration*, *noisy actions* and *minimizing familiarity* and an equal mixing of all four exploration strategies (from left to right) on the complete testing pool according to the methodology described in Section 3.3.
For each result 35 runs of each of the 9 testing pool strategies were used. The specific parameters are listed in Appendix G.3.

Combining more than two types of exploration by adding their respective shocks $\tilde{S}$ and $\tilde{F}$ to the Q-Values leads to the results presented in Figure 22. Here, four different exploration approaches are show: Only using ε-*exploration*, using an equal balance between ε-*exploration* and *noisy actions*, combining all exploration variants given constant $\varepsilon = 0$ and using an equal balance between all four presented exploration methods. All of them show a comparable performance and do not differ significantly. Interestingly, even though pure ε-*exploration* performed best given single-type exploration, refraining from using it and only using the three others still produces a comparable result.

Even though Burda, Edwards, Storkey, et al. (2018) suggest that curiosity-driven learning might improve performance, our results imply that using it in our setting is not beneficial. Neither as single-exploration methods nor within *Multi-Exploration* the methods *maximizing surprise* and *minimizing familiarity* where able to significantly improve performance. As both those methods need a second function approximator, they are considerably more computationally expensive. Consequently, we refrain from using them within our algorithm.
As the combination of *ε-exploration* and *noisy actions* showed a slightly, albeit non-significantly, higher average performance and performed better in other settings, we use this combination for our algorithm.
In addition to the explored variants of equal mixing of different exploration strategies, the R-package Kies (2019) allows for arbitrary mixing ratios. Another feature provided by the package is to vary the mixing ratios during the learning process by setting different ratios for the first and last block. The mixing ratios for the intermediate blocks are then interpolated. This allows the user for example to start with *ε-exploration* to generally map the environment and then circle in using more precise exploration types. Within our setting however, this approach and also other tested variants performed approximately in line with the shown results.
While most examined modifications can be used independently, *Multi-Exploration* is strongly interconnected with *Exploration Pathing* as this mechanism balances the values of the exploration parameters. At the start of a run, the exploration parameter of each method has to be set to a predefined value. I chose these starting values to approximately produce the desired balances between the exploration methods. As the *Exploration Path* mechanism needs several blocks to achieve the specified balance, the choice of these starting values still influences the results. Similarly, if the *Exploration Pathing* mechanism fails to balance the different exploration methods properly, the presented results are also influenced.
All discussed exploration methods are generalizable to other games. In fact one should expect *Multi-Exploration* to perform better in a setting with more than two actions. If specific patterns out of several actions have to be played, the more sophisticated approaches can help to navigate the player to the territory which has to be mapped, while *ε-exploration* can help to avoid over-optimization of suboptimal policies.

3.6. Exploration Pathing

The standard approach to balance learning is to start with completely random actions and lower the exploration parameter gradually in a pre-determined fashion. In the case of off-policy learning, e.g. Q-Learning, there is no necessity for the learning rate to got to actual zero. Consequently, one often introduces a minimal learning rate. State of the art Q-Learning algorithm as measured by performance against ATARI games *Rainbow* (Hessel et al., 2018) for example does so with ε*-exploration.* It starts with $\varepsilon = 1$ and after staying there for 4 million frames gradually lowers ε to 0.01. This approach is not necessarily optimal, as one can easily image a situation where the basics of the game have been figured out quickly and only further refinement is necessary. Recall the example situation from Section 3.5 with *compare.exploration*. Gradually decreasing the exploration parameter in a pre-defined way can easily lead to the situation, that most episodes of the run are spent suboptimally. Episodes where the strategy gets triggered can not be used to gather information about optimal behavior in later periods. More generally speaking, this problem always exists if the policy might act in a way which is non-recoverable, e.g. against any kind of *grim.trigger* like strategy.
Exploration Pathing solves this problem by calibrating the exploration parameter on the current best effort episode. Every block the current exploration episode is compared to an episode (or an aggregate of previous episodes) without exploration. *Exploration Pathing* now calibrates the exploration parameter(s) in a way that the average reward of the exploration episode is a specified fraction of the best effort episode. This fraction can depend on the current block number, i.e. the progress of the run. Changing this fraction during the run, for example by letting it converge towards unity, signals the algorithm that it has to reduce the exploration parameter. Effectively this means, that in uncharted environments exploration stays high. In an environment where there is no information about good actions, we expect that the performance of the best effort episode is comparable to taking random actions. The fraction between the performance of these episodes is therefore high. Assuming that the desired fraction between both types of episodes is lower this means that the exploration parameter is increased.
When the algorithm finds a way to exploit the environment (i.e. the best effort episode is a lot better than random), the exploration parameter is quickly adapted. The result is the automatic focus on interesting states within the exploration episodes. As has been discussed in Section 3.5, *Exploration Pathing* can also be beneficial to avoid local optima. If the best effort episode and the exploration episode show the same performance this can be a sign that not enough exploration takes place. In this case the exploration parameter is gradually increased until a sufficient amount of exploration takes place again.
Despite these deliberations, *Exploration Pathing* does not guarantee that it is impossible to get stuck in a local optimum. Using only *noisy actions* and an ill-calibrated function approximator for example can lead to the situation that the exploration episode is stuck playing the same kind of suboptimal actions while still maintaining

the correct fraction to the best effort episode. It is therefore advised to at least partially use ε-*exploration* within *Multi-Exploration*.
The results of using *Exploration Pathing* can be seen with Figure 23. Here, four different kinds of exponentially decreasing ε-*exploration* are shown, ranging from a depreciation with a factor of 0.85 (ε below 0.01 after 29 blocks) to 0.99 (ε still above 0.35 at the end of the 100 blocks). All of them perform significantly worse than the two displayed variants of *Exploration Pathing*. To show general robustness of *Exploration Pathing*, one of of the two paths is held constant at a fraction of 85%, while the other one starts at 90% and increases to 99%.
It is possible to construct arbitrary bad *exploration paths*, for example by setting the fraction to constant zero. They do however perform stable and consistently well within our setting, as long as the fractions are chosen in a reasonable vicinity to unity with at least some room for exploration. To avoid being influenced by different exploration types and to have a fair comparison, all data points were generated using only ε-*exploration*.

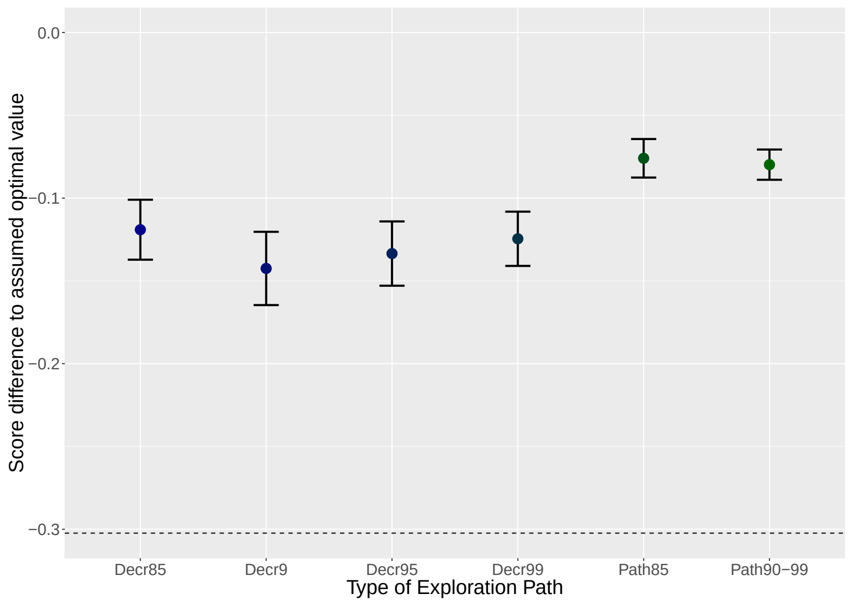

Figure 23: Comparison of different parameter sets without (four on the left) and with (two on the right) *Exploration Pathing* over all strategies according to the methodology described in Section 3.3.
All sets are ε-*exploration* only. With *DecrXY* exponential decreasing of the exploration parameter took place: Starting with $\varepsilon = 1$, the parameter was multiplied by XY% each block. *Path85* used a constant pathing fraction of 85%, *Path90-99* used a pathing fraction which starts with 90% and linearly increases up to 99%.
For each result 30 runs of each of the 9 testing pool strategies were used. The specific parameters can be found in Appendix G.4.

One of the main reasons to develop *Exploration Pathing* was the necessity to have an objective measure to balance *Multi-Exploration*. If multiple different types of exploration are used, it is non-obvious how the different exploration parameters should be calibrated to each other. A decrease in one parameter might non-linearly effect the ratio of the exploration types to each other depending on the environment. Pre-defined rules as used for a single exploration parameter are therefore not feasible. *Exploration Pathing* solves the balance problem between the different exploration parameters. It is therefore possible to demand an equal share of exploration between *noisy actions* and *ε-exploration* despite their different ways to induce exploration.
Technically speaking this is achieved through the use of modified PID controllers, which are a well known tool within the field of electrical engineering and have its origins in Minorsky (1922). The abbreviation PID stands for *Proportional-Integral-Derivative* and describes the three components which are analyzed to determine the force and direction of the change of a specific exploration parameter.
In the following paragraphs is described how *Exploration Pathing* is achieved. To be able to understand its functioning in Sections 3.10 (*Model Persistence*) and 3.2 (Performance) the explanations are more general than necessary for the largest part of the analysis. In particular it is shown how *Exploration Pathing* deals with multiple different episodes of the same type per block as well as different types of best effort episodes.
To reduce complexity we first present the procedure given a single exploration type:

1. Restrict underlying data points to the last $N_{\text{db.best}}$ blocks.
   As within the learning process of the algorithm several paradigm changes might take place, it is necessary to forget old results. If not as many blocks have been generated, $N_{\text{db.best}}$ is the number of available blocks.

2. Calculate ceiling of the best effort episodes C^{best}
   Within the restricted data points generate a vector

   $$C^{\text{best}} = \left(C_1^{\text{best}}, ..., C_{N_{\text{db.best}}}^{\text{best}}\right)$$

   using

   $$C_k^{\text{best}} = \sum_{i=1}^{k} w_i^{\text{best}} B_k^{\text{best}}, \quad w_i^{\text{best}} = \frac{\gamma_{\text{best}}^{k-i}}{\sum_{j=1}^{k} \gamma_{\text{best}}^{k-j}}$$

   with B_k^{best} being the mean average reward over all runs which are of type *best* (historically best) or *current* (most recent model) of the respective block (see Section 3.10 for more details). Discount factor $\gamma_{\text{best}} \in (0, 1]$ determines how strong older blocks should be weighted.
   Effectively this allows us to calculate a smoothed ceiling of the actual rewards of the best effort episodes over time. Due to the restriction of the blocks in the first step, C_1^{best} corresponds to the earliest available block, i.e. the one $N_{\text{db.best}}$ blocks before, and $C_{N_{\text{db.best}}}^{\text{best}}$ to the most recent one.

3. Calculate goal vector G
   Based on the wanted starting fraction of the path and the final one, the fractions $frac_k$ are interpolated given their relative position in the run. It holds that for $G = (G_1, ..., G_{N_{\text{db.best}}})$
   $$G_k = C_k^{\text{best}} \cdot frac_k$$
   The values G_k define which values we would have wanted for the exploration episodes.
   In rare cases the best effort episode might have a negative performance given the rules of the IPD game of Section 2.1.1. This results in negative C_k^{best}. The standard calculation of G_k in this case results $G_k > C_k^{\text{best}}$ as $frac_k < 1$. This would lead to a correction of the exploration parameter into the wrong direction. In this exceptional case G_k is therefore calculated by dividing C_k^{best} by $frac_k$.

4. Calculate error vector $err = (err_1, ..., err_{N_{\text{db.best}}})$
   It holds that
   $$err_k = G_k - B_k^{\text{expl}}$$
   with B_k^{expl} being the mean average reward over all exploration episodes of the respective block.
   A positive err_k implies, that the performance of exploration episode was too low and the exploration parameter should be decreased. A negative err_k implies a too high performance of the exploration episode and therefore a necessary increase in the exploration parameter.

5. Use a modified PID-controller to determine the change of the exploration parameter to get the next err_k to zero.
   It holds that
   $$\delta_{\text{PID}} = K_P \cdot P_{\text{PID}} + K_I \cdot I_{\text{PID}} + K_D \cdot D_{\text{PID}}$$
   with $K_P > 0$, $K_I > 0$ and $K_I > 0$ being weights which balance the different components.
   In the basic PID-theory one uses the very last data point to calculate P_{PID}, all data points to calculate I_{PID} and the last two data points to calculate D_{PID}. The extreme randomness of the used setting makes this classic approach unstable. Instead, we use smoothed averages as this stabilizes our controls. We use the last N_P, N_I and N_D data points of err_k to calculate the respective components. To simplify notations, we now assume the vector to be already filtered for each component separately and the indizes of err_k to be shifted accordingly.

P_{PID} is the so called proportional response. It manages the direct response of increasing the exploration parameter if more recent err_k are negative and decrease it otherwise. P_{PID} is calculated by

$$P_{\mathrm{PID}} = \sum_{i=1}^{N_P} w_i^{\mathrm{P}} \cdot err_i, \quad w_i^{\mathrm{P}} = \frac{\gamma_{\mathrm{P}}^{N_P - i}}{\sum_{j=1}^{N_P} \gamma_{\mathrm{P}}^{N_P - j}}$$

I_{PID} is the integral component. If the err_k are constantly positive (e.g. due to there being a constant upwards shift of the best effort episodes), this component increases the force on the exploration parameter:

$$I_{\mathrm{PID}} = \sum_{i=1}^{N_I} \gamma_{\mathrm{I}}^{(N_I - i)} \cdot err_i$$

In the standard version of PID-controlling it holds that $\gamma_{\mathrm{I}} = 1$. We used a number close to unity, as this achieves a similar effect, but smooths out the effect of using only the last N_I data points instead of all of them. In practice balancing N_I and γ_{I} results in similar behavior. It is possible to chose a comparatively low N_I with an $\gamma_{\mathrm{I}} = 1$ on the one hand or $N_I = N_{\mathrm{db.best}} = \infty$, resulting in the usage of all available data points, and a lower γ_{I} on the other hand.
D_{PID} is the derivative component. It aims to avoid an overcorrection of the PID controller: If more recent err_k are moving fast in the correct direction, then one might want to dampen the correction to avoid overshooting the target. Failing to do so might result in oscillatory behavior of err_k.

$$D_{\mathrm{PID}} = \sum_{i=2}^{N_D} w_i^{\mathrm{D}} \cdot (err_i - err_{i-1}), \quad w_i^{\mathrm{D}} = \frac{\gamma_{\mathrm{D}}^{N_D - i}}{\sum_{j=2}^{N_D} \gamma_{\mathrm{D}}^{N_D - j}}$$

Having calculated δ_{PID} it is added to the exploration parameter. As δ_{PID} might be negative, the exploration parameter is capped at zero should the addition result in a negative exploration parameter otherwise.

If there are multiple exploration types at the same time, the algorithm stays similar. The calculation of err_k differs, as will be detailed below. Additionally, the factors K_P, K_I and K_D have to be chosen on a per exploration type basis. In fact, within these analyzes I chose K_P, K_I and K_D to be each by factor 10 smaller for *ε-exploration* than for *noisy actions*. The performance of exploration episodes is considerably more sensitive to directly choosing another action compared to a small nudge of the Q-Values. As a result, this has to be reflected in the magnitude of the PID factors which directly influence the strength of change of an exploration parameter.

The basic idea behind the calculation of err_k^{expl} on a per exploration type basis is to divide the occurred total err_k between the different exploration types. To do so, a so called *suboptimality score* $\xi_{t,j,k}$ is calculated for each single action at period t of exploration episode j of block k. Additionally, for each of the exploration types a type-specific suboptimality score is calculated. If there are multiple exploration episodes within one block, the arithmetic average is used. The suboptimality scores are calculated the following way:

1. If the action with the higher Q-value has been taken it holds that $\xi_{t,j,k} = 0$ for the total suboptimality score of this action and for each of the idiosyncratic exploration suboptimality scores.

2. If the ε shock has been realized, it holds

$$\xi_{t,j,k} = \xi_{t,j,k}^{\varepsilon} = Q(s,a_1) - Q(s,a_2); \quad \xi_{t,j,k}^{\text{noisy}} = \xi_{t,j,k}^{\text{surprise}} = \xi_{t,j,k}^{\text{fam}} = 0$$

   with $Q(s,a_1)$ being the Q-value of the best possible action as determined by the most recent function approximator and $Q(s,a_2)$ the Q-value of the actually chosen action.

3. If the ε shock has not been realised, it holds

$$\begin{aligned}
\xi_{t,j,k} &= (N_2^{\text{noisy}} - N_1^{\text{noisy}}) + (N_2^{\text{surprise}} - N_1^{\text{surprise}}) + (N_2^{\text{fam}} - N_1^{\text{fam}}) \\
\xi_{t,j,k}^{\varepsilon} &= 0 \\
\xi_{t,j,k}^{\text{noisy}} &= (N_2^{\text{noisy}} - N_1^{\text{noisy}}) \\
\xi_{t,j,k}^{\text{surprise}} &= (N_2^{\text{surprise}} - N_1^{\text{surprise}}) \\
\xi_{t,j,k}^{\text{fam}} &= (N_2^{\text{fam}} - N_1^{\text{fam}})
\end{aligned}$$

   Here, N_2^{noisy} the noise due to *noisy actions* of the chosen, suboptimal action. The difference between N_2 and N_1 is therefore the effective noise which influenced the decision of the respective exploration type. The other values are defined analogously.

4. These values are aggregated to block-wise suboptimality scores:

$$\begin{aligned}
\bar{\xi}_k &= \frac{1}{T \cdot N_{\text{ep}}} \sum_{t=1}^{T} \sum_{j=1}^{N_{\text{ep}}} \xi_{i,j,k} \\
\bar{\xi}_k^{\text{expl}} &= \frac{1}{T \cdot N_{\text{ep}}} \sum_{t=1}^{T} \sum_{j=1}^{N_{\text{ep}}} \xi_{t,j,k}^{\text{expl}}
\end{aligned}$$

   with T the number of periods/actions of an episode (i.e. 60 in our default case), N_{ep} being the number of exploration episodes within a block (i.e. 1 in our default case and 2 for Section 3.10 and Section 3.2) and "expl" being a stand-in parameter for each of the exploration types.

5. Calculate

$$
\begin{aligned}
w_k^{\text{expl}} &= \frac{\bar{\xi}_k^{\text{expl}}}{\bar{\xi}_k} \\
err_k^{\text{expl}} &= \left(C_k^{\text{best}} - B_k^{\text{expl}}\right) \cdot w_k^{\text{expl}} - \left(C_k^{\text{best}} - G_k\right) \cdot w_k^{\text{expl,goal}}
\end{aligned}
$$

   with $w_k^{\text{expl,goal}}$ the wanted fraction of block k given exploration type "expl". The value w_k^{expl} by contrast is the actually observed influence of the given exploration type. The first part of the subtraction is the effective distance generated by the given exploration type, while the second part measures the desired distance.

6. Use an PID-controller to set future err_k^{expl} to zero analogously to the single exploration case of above.

The current R-package Kies (2019) provides linear and exponential paths between start and end fractions. I found linear paths to work marginally better, but it is not obvious, that this is an optimal shape. In fact, *Rainbow* from Hessel et al. (2018) uses an exploration period of completely random actions to set the stage. This implies that a more s-shaped path might be suitable in some settings, even though we can easily replicate this aspect using *Memory Initialization* (see Section 3.11).
Exploration Pathing might be further improved by integrating automatic PID-tuning methods (see e.g. Cominos and Munro (2002)). This would allow a more accurate path compared to a coarse manual adjusting of different tuning parameters as has been done for these analyses.
As far as I am aware, this type of automatic setting of exploration parameters is a novel idea within the context of machine learning, both regarding the concept of an exploration path in general and regarding the usage of a PID controller to balance different parameters.

3.7. Feature Selection

Feature Selection is the decision on what parts of the raw data should be used in which way as input for the function approximator. If the selection is too narrow this has direct effects on the definition of the effective space state $\mathcal{S}$. Additionally, one can enrich raw data by precalculating and providing useful statistics.
Feature Selection plays an important role for nearly all machine learning methods. Generally speaking, function approximators can greatly profit from working on preprocessed data. Image recognition for example can be improved considerably by using convolutional neural networks instead of simply providing raw pixels of each image to a neural network (Krizhevsky et al., 2012).
Not only the final performance, but also also the speed of learning depends strongly on *Feature Selection*. Encoding heuristics as additional information can help the algorithm to guide it across the environment. In the edge case, one might explicitly provide the algorithm with the actual states of the opponent as discussed in Section 2.2.
Directly providing the actual states means that any additionally provided information is noise and hurts the learning rate of the algorithm. In this case it has to learn which parts of the provided input are irrelevant. This might be non-trivial due to spurious correlations and complex chains of effects. Given the strategy *grim.trigger* for example only providing whether or not it is triggered optimizes the speed of learning.
On the other hand it is important to provide sufficient information in the sense, that each state can be identified based on the input data. Failing to do so implies that the function approximator is unable to map input data to true states of the underlying MDP. In this case, the learning algorithms loses the theoretical property to converge against the best possible policy. This might for example be the case, if only the last two episodes are provided and the opponent strategy uses the last three episodes to make its decision. Compare Appendix B for a discussion regarding unobservable states.
To summarize, ideally exactly the necessary states are provided. This might not be possible. Our stated goal to find best answers for black-box strategies for example implies that these states are unknown. In this case one has to guarantee that the provided encoding at least contains all possible information which might be relevant for finding a best answer. Given a true black-box strategy this means that the encoding may not destroy any information. The performance can be enhanced by additionally providing useful heuristics. These might be used to deduce a possible state of the opponent, even if they in and for themselves do not represent the states directly.

As mapping the states directly gives an immense boost to the algorithm, I decided to use the following encoding without analyzing the pool of strategies beforehand:

1. The complete observed history of the actions of the agent itself (including observation errors) (1 if cooperated, -1 if defected, 0 if this period has not been observed yet)
2. The complete actual history of the actions of the agent itself (1 if cooperated, -1 if defected, 0 if this period has not been observed yet)
3. The complete observed history of the actions of the opponent (1 if cooperated, -1 if defected, 0 if this period has not been observed yet)
4. The complete observed reverse history of the actions of the agent itself outgoing from the current period (including observation errors) (1 if cooperated, -1 if defected, 0 if this period has not been observed yet)
5. The complete actual reverse history of the actions of the agent itself outgoing from the current period (1 if cooperated, -1 if defected, 0 if this period has not been observed yet)
6. The complete observed reverse history of the opponent outgoing from the current period (1 if cooperated, -1 if defected, 0 if this period has not been observed yet)
7. The current period
8. The average number of observed defections of the opponent
9. The sum of observed defections of the opponent
10. The average number of observed defections of the agent itself
11. The sum of observed defections of the agent itself
12. The difference of observed defections between the agent and the opponent
13. The ratio of observed defections between the agent and the opponent
14. The absolute value of the average reward the agent would receive, given that the reward would be calculated based on the observed actions.

This encoding is called the *main encoding.*
An advantage to include the reverse history is, that this makes it significantly easier for the function approximator to map common patterns. Such a pattern might be "something happened within the last X periods", which is for example used by *tit.for.tat*. With this encoding the most recent period is always at the same position within the input data. The alternative would be, that the function approximator has to cross reference the number of the current period with the actions or correctly handle abstract concepts such as the importance of "the last input which is not a zero in a certain part of the input array". This would pose a challenge for all discussed function approximators of Section 2.4.

Additionally, I added the history in the correct order as it seems probable that an opponent strategy might use actions at specific periods to change its behavior. A plausible case for this are the very first actions which might fundamentally change the behavior of the opponent.
This encoding is a strict superset of the encoding of Harper et al. (2017). They restrict themselves to the two first and last observed actions and lack relative data regarding the comparison between both strategies. Additionally, they do not incorporate the number of the current period.

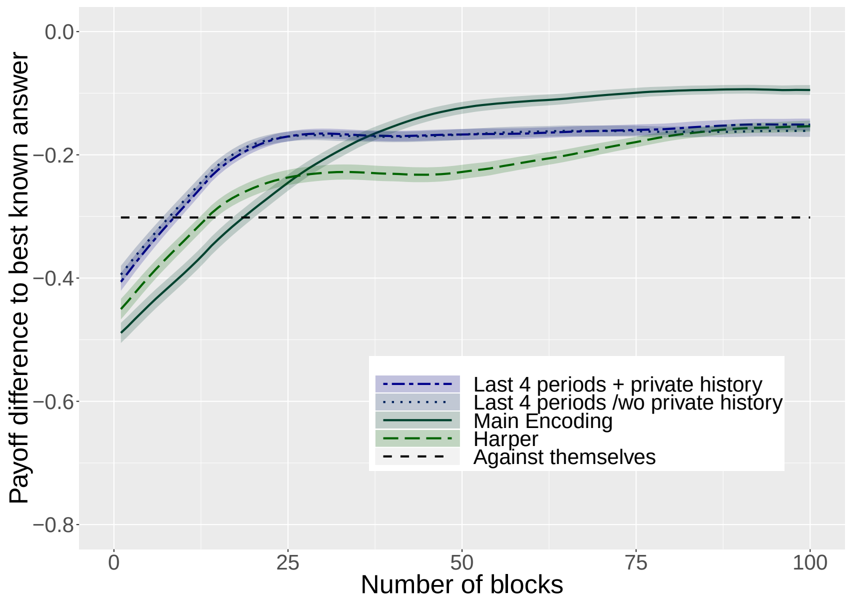

Figure 24: Comparison of learning behavior between the following encodings from top to bottom: Only using the last 4 periods, Only using the observable history of the last 4 periods, the main encoding used throughout this article and finally the encoding from Harper et al. (2017). The methodology described in Section 3.3 has been used.
All data points have been generated on 50 runs for each strategy. For the specific parameters and more detailed information see Appendix G.5.

A comparison of an encoding using only the very last four episodes with and without using private information, the main enconding and the encoding of Harper et al. (2017) is shown in Figure 24. Restricting the algorithm to the last four periods allows it to converge considerably faster. At approximately block 25 however, i.e. after having played 25 episodes with and 25 episodes without exploration, the algorithm stops to improve. Here, the algorithm has already found the (nearly) optimal policy given available information. The main encoding on the other hand reaches its potential at around block 75. At this point the learning is stopped by other factors. Possible causes include local optima, the complexity of the function approximator and a possibly to high learning weight a in the Q-Learning algorithm. Choosing a

high learning weight might lead to an overweight of recent noisy information. This might result in oscillatory learning behavior and a failure to capture expected values. The encodings which use the last 4 periods do not differ meaningfully. The effects when including private information are significant, but weak in effect size. As should be expected, providing the encoding with additional information leads to a minimally slower convergence in the beginning. After processing sufficient data, final performance is minimally higher when including private information. However, the effect size is negligible and has no practical relevance. This is not an aggregate effect, as both graphs follow approximately the same path for every single testing pool strategy. This should be expected, as the underlying states are defined by the opponent, which by design is not able to observe the actual actions of the agent. The optimal policy can therefore be constructed using only the observed history. Nevertheless, having private information as part of the input might allow the algorithm to infer hidden states when the complete history is not available.
The encoding of Harper et al. (2017) performs strictly worse than simply using the last four periods. Despite lacking input about actions 3 and 4 periods ago, their final performance convergences to the same level as the 4-period-encodings. This shows, that on aggregate over all strategies additional information can balance missing periods. However, both are not interchangable even given sufficient training time. Their relative performance depends on the analyzed strategy. With the quite complex strategy *strat.c* (see Appendix F) for example the encoding of Harper et al. (2017) shows significantly worse performance. The *grim.trigger* like strategy *strat.e* on the other hand explicitly uses the number of defections of its opponent to change its state. This information explicitly included in the encoding of Harper et al. (2017). Consequently, it performs considerably better than the 4-period-encodings.
The final performance of the main encoding is considerably higher than the tested alternatives. This is bought by a slower convergence in the beginning, where the function approximator has to learn which parts of the input are not relevant.
Brunauer et al. (2007) analyzed on their testing pool the relevance of the number of provided periods. They focused on “last X periods” encodings due to their simplicity. These types of encodings are of special interests, as they are comparatively common (see Harper et al. (2017)). Brunauer et al. (2007) find, that a history of 4 or 5 periods performs best within their setting. The results of an analogous analysis in our setting are shown in Figure 25.

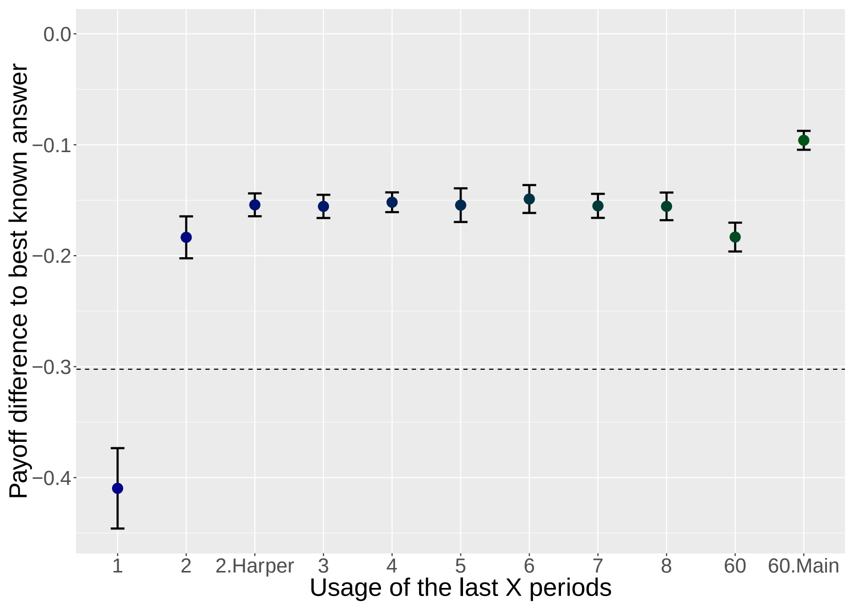

Figure 25: Comparison of performance if only the last X periods are given as input to the encoding according to the methodology described in Section 3.3. Additionally two more complex encodings are shown: "60.Main" is the main encoding, which uses all 60 periods. "2.Harper" is the encoding of Harper et al. (2017) which uses the last two periods and additionally provided aggregated data.
All data points have been generated on 20 runs for each strategy, except 3, 4, "2.Harper" and the main encoding, where 50 runs have been used. For the specific parameters and more detailed information see Appendix G.5.

In our setting 4 to 6 periods perform best, similarly to Brunauer et al. (2007). Indeed, using all 60 periods leads to a significantly lower performance. The main encoding therefore does profit from the additionally provided statistics, similarly to Harper et al. (2017).
A recurrent neural network, for example with LSTM cells, should be able perform well even when restricted to the very last observation. In theory it should be able to extract all relevant information and create an internal state representation (see Section 2.4.3 for an overview of RNNs). Furthermore, the LSTM function approximator should be able to calculate helpful intermediary information by itself. Statistics as provided by the main encoding should therefore be less relevant.
Figure 26 shows that this is indeed the case in our setting. Here, the comparison between two different LSTM encodings is shown. *Main LSTM Encoding* is identical to the normal main encoding, except that history-wise only the last 10 periods are explicitly used as input to save computation time. Additional statistics are generated on the complete history and provided analogously to the main encoding. *Minimal LSTM Encoding* receives only the very last period, including the actually played action of the last period, as input.

Both encodings achieve the same performance after 100 blocks and show similar learning rates. The minimal encoding is able to achieve a higher average reward around block 20, probably due to finding a good local optimum based on the less complex information. As before, using more complex input data results in a slower start of the learning process. Extracting complicating patterns however is comparatively easier when providing more complete information as well as additional statistics directly. The main LSTM encoding therefore catches up and dominates the majority of the run.

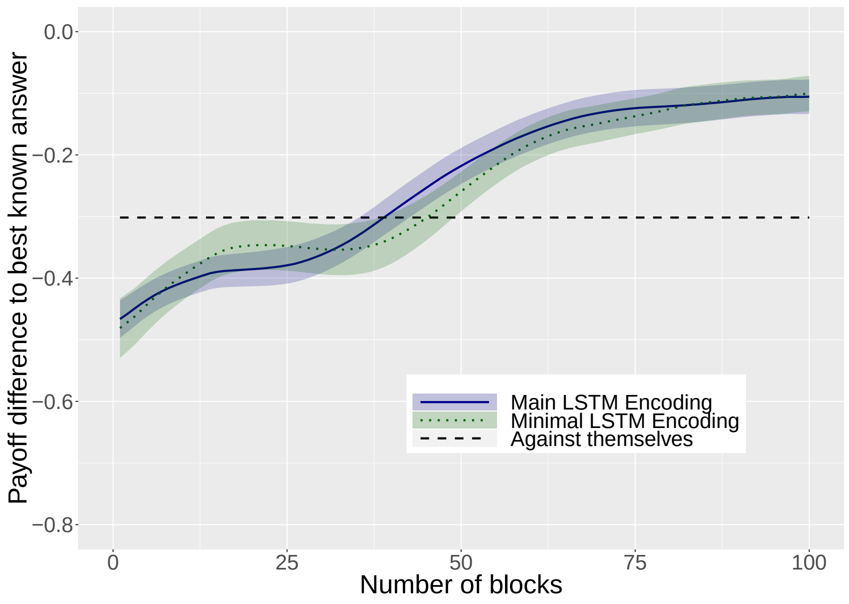

Figure 26: Comparison of performance of two different encodings (with additional information and history vs. only the last period, respectively) using the LSTM function approximator according to the methodology described in Section 3.3. For a detailed description of the different encodings see Section 3.7.
All data points have been generated on 20 runs for the "Main LSTM Encoding" and 10 runs for each strategy with the "Minimal LSTM Encoding". For the specific parameters and more detailed information see Appendix G.5.

3.8. Experience Replay

Instead of only using the very last period or episode to update the function approximator, as is suggested by classical Q-Learning, it is prudent to use non-recent experiences as well. Re-using data from older episodes when updating the function approximator is called *Experience Replay*. The basic idea behind this technique was developed by Lin (1992).

Within the update routine the function approximator is changed to reflect new data. Some function approximators are updated by re-building the underlying model from scratch. One such example is gradient boosting. In this case using *Experience Replay* is crucial as sufficient data is necessary to build an accurate model. So called *updating* function approximators provide an alternative. They use new data to reshape the existing model. The standard example are neural networks which can be retrained. As these function approximators already incorporate data from old episodes it might seem intuitive to only use recent experiences for their update. However, this can lead to overfitting and a less stable function approximator.

The *Deep Q-Network* (DQN) algorithm from Mnih et al. (2015) was for a short time the gold standard for the often used ATARI games benchmark. They used *Experience Replay*, within their article called *mini-batch processing*, to increase the robustness of their neural network. DQN generates a memory (s, a, r, s') of all state-action pairs, their rewards and their successor states and updates the function approximator based on a uniformly drawn sample of those experiences out of the memory.

Experience Replay is particularly effective combined with an off-policy learning algorithm like Q-Learning. An off-policy learning algorithm is able to generate an optimal policy out of suboptimal exploration data. This is important, as we expect older episodes to contain a larger number of suboptimal actions. When using an on-policy algorithm, as for example Monte Carlo learning, older data should therefore negatively impact performance. The proposed concept *Q-Switching* is on-policy at the start of the learning process and only asymptotically off-policy. Even though *Experience Replay* for Q-Learning is an established technique, albeit in other domains, it is therefore not obvious whether or not and to which extent it should be used in our case. Consequently, we study its effects on our algorithm and given the IPD game.

Analogously to the approach of Mnih et al. (2015), we build up a memory stack of all experiences made by the algorithm. For this memory we do not differentiate between best effort and exploration episodes. I added one additional feature to the baseline of Mnih et al. (2015): Instead of using uniformly random experiences, one can force the algorithm to use the most recent experiences up to a number of specified blocks when updating the function approximator. The intuition behind this change is that while there is a stabilizing effect in using older experiences we do not want to refrain from learning from the newest ones. Using random mini-batches to update the function approximator might lead to not evaluating the most recent episode at all. A possible paradigm shift of the function approximator might therefore go undetected. If this paradigm shift leads to a significantly worse performance, we produce episodes with this suboptimal policy until this knowledge is incorporated. This is an inefficient use of resources.

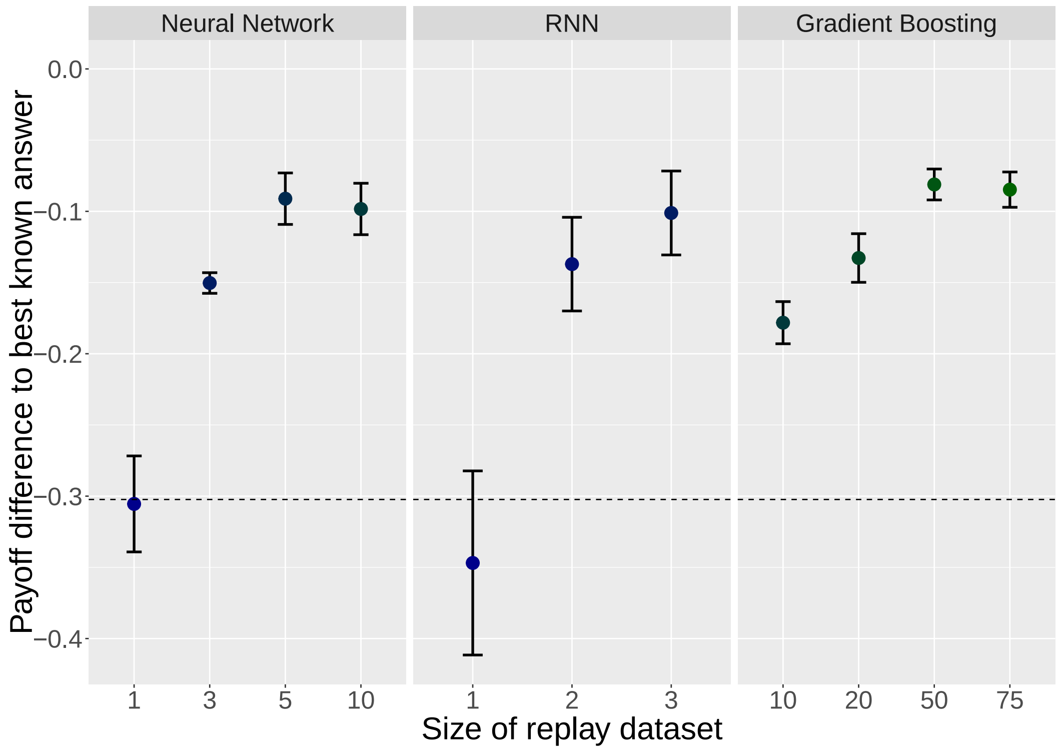

Figure 27: Comparison of various sizes of replay data sets, as measured by number of blocks, when updating the function approximator. Displayed are the function approximators neural network (left), recurrent neural network with LSTM Cells (middle) and gradient boosting (right) according to the methodology of Section 3.3. Each of the testing pool strategies has been run 15 times in the case of the neural network, except for replay size 3, where 50 runs have been used. In the case of the recurrent neural network 10 runs per strategy have been used. In the case of Gradient Boosting 25 runs per strategy have been used. For the specific parameters and more detailed information see Appendix G.6.

The effects of using *Experience Replay* with a fixed inclusion of the very last block are shown in Figure 27. All things equal, a bigger replay data set increases the performance of all function approximators. Doing so however shows a decreasing marginal effect size.
Using a higher number of episodes in the update cycle increases run time of the algorithm. In particular for neural networks and recurrent neural networks run time scales approximately linear with the size of the replay data set. While a certain size of the replay data set is recommended, a further increase might be inefficient.
The performance increase of the networks can not be attributed to longer training

time alone. Training them longer without using *Experience Replay* resulted in overfitting the training data, thereby decreasing performance.
Gradient Boosting shows the same general effects as the networks, but uses a considerably higher number of episodes in the replay data sets to achieve a comparable performance level. This should be expected, as gradient boosting is non-updating. While using a non-updating function approximator is generally not recommended in machine learning literature (Sutton and Barto, 2018, p. 162) this does not necessarily hold true given our setting. Section 3.9 compares the performance of different function approximators.

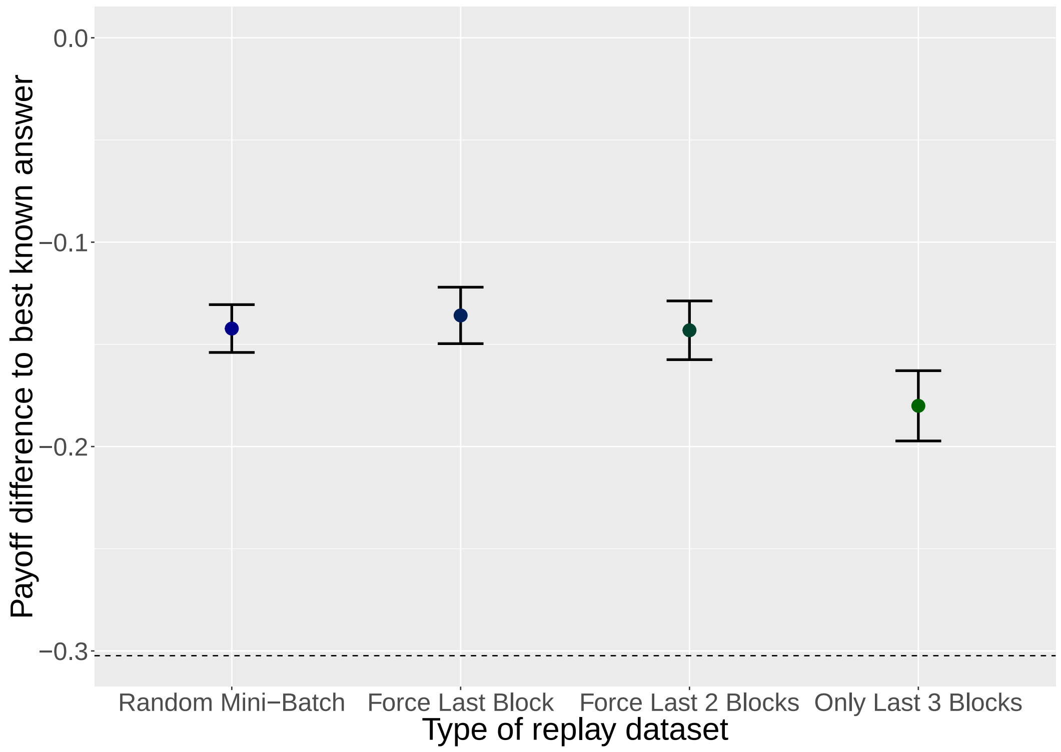

Figure 28: Comparison of the effects of forcing the last X blocks to be used in the replay data set according to the methodology of Section 3.3.
For this figure a neural Network with a replay size of 3 blocks has been used. All parameter sets use 50 runs per strategy. For the specific parameters and more detailed information see Appendix G.6

The effect of forcing the last X blocks is presented in Figure 28. Forcing the very last block increases performance. This performance increase however is negligible and non-significant. Further increasing the number of forced blocks decreased performance. In fact, only using the last three blocks showed the lowest performance. The stabilizing effect of using older episodes is therefore not only beneficial given the ATARI benchmark, but also in our setting of the IPD game.

One possible explanation of this lower performance when using only the last three blocks is that only complete episodes were used for the update. Using uniform sampling on the other hand implies that experience sets (s, a, r, s') are drawn out of the complete body of the memory independent of the generated episode. As experiences within one episode are highly correlated this richer diversity might drive the results. A possible "more recent information is more valuable" effect might therefore be counteracted. An interesting extension to this idea might thus be to use random sampling which is not uniform, but rather prefers more recent experiences to combine the advantages of both approaches. Alternatively, further improvements might be achieved using *prioritized replay*, which was proposed by Schaul et al. (2015). Here, experiences are prioritized which had the highest effect on changing associated Q-Values.

3.9. Choice of Function Approximator

Classic Q-Learning uses a tabular approach where all possible state-action pairs (s, a) are saved and evaluated explicitly. In our case this is not possible due to the large state space. A function approximator $f(s, a) \rightarrow Q$ generalizes from historical experiences to similar situations. This allows to construct a policy which is defined outside of already experienced state-action pairs and plays there profitably.
The theoretical foundation regarding function approximators is built in Section 2.3.4. Section 2.4 gives an overview about the function approximators discussed in this section. Here, we will concentrate on the effects of different function approximators within our specific setting as measured by the approach as defined by Section 3.3.
While the R-package Kies (2019) is customizable and supports different types of function approximators, our analysis focuses on the following three:

1. Gradient boosting shows very strong performance in supervised learning, is fast to train, but can not be updated.
2. Neural networks are the go to standard approach in reinforcement learning, can be updated and scale well.
3. Recurrent neural networks create policies for arbitrary large episode lengths T even if all training data is capped at a lower level.

The following points determine which function approximator to use:

1. In howfar is the function approximator able to capture the underlying best answer and generalize correctly?
2. How fast does the function approximator generate a good policy?
3. Is the function approximator able to generalize to games with a longer number of episodes?

Regarding the first point it holds that after optimizing hyperparameters by hand, all three discussed function approximators show practically the same performance after 100 blocks. All are therefore able to generate policies of the same quality. Figure 29 shows that all function approximators receive an average reward of the same level on the last 25% of blocks.
Regarding the second point, one has to differentiate between two definitions of speed. On the one hand, speed of learning can refer to the number of episodes necessary to achieve a certain quality in the generated policy. One the other hand it can refer to the actual time difference until the final policy has been generated.
Given a specific problem, the actual time is the more important factor. If one wants to generalize however, it is important to keep in mind the other concept as well. If one for examples wants to find a best answer against a strategy which needs considerable time for each action, the number of played episodes becomes relatively more important than the update time for the function approximator.

In the following we will focus on the challenges of our specific setting, aggregated over the testing pool. It should be kept in mind however, that a change in the environment might have profound effects on these results.
The performance of the different function approximators given the number of used blocks is presented in Figure 29.

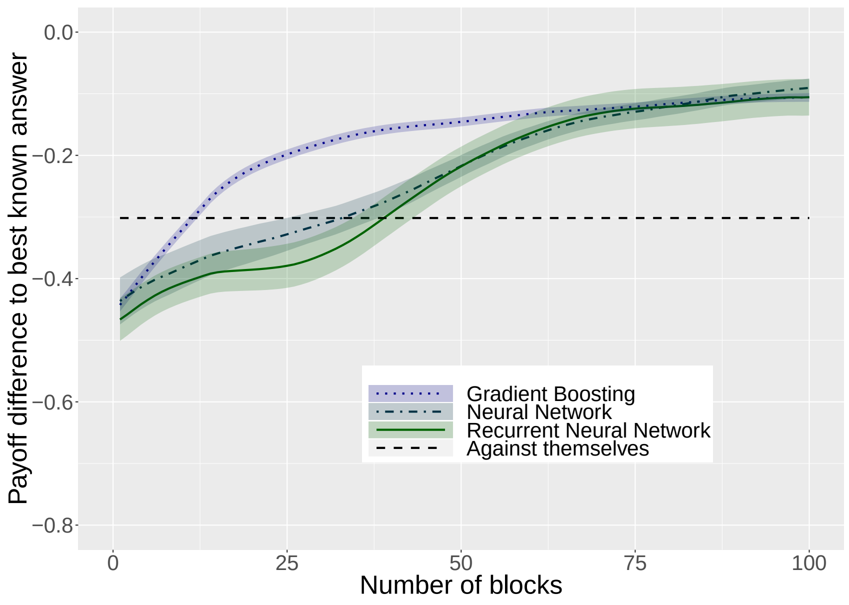

Figure 29: Comparison of performance of three different function approximators with optimized hyper parameters for each according to the methodology described in Section 3.3.
All data points have been generated on 100 runs for each strategy for gradient boosting and 15 runs for each strategy for neural network and recurrent neural network respectively. All use the same features of the algorithm, except for a difference in size of the replay dataset (see Section 3.8) which has a strong interaction with the chosen function approximator. Additionally, recurrent neural network uses its own encoding (see Section 3.7). For the specific parameters and more detailed information see Appendix G.7.

Figure 29 gives some indication, that gradient boosting is better able to generalize based on a smaller number of episodes. While being practically relevant, for example if one stops the evaluation after a specific block, this does not guarantee, that the displayed graph represents the upper potential of the networks given the data.
The networks use the same potential effort to update after each block. If new data of early blocks is especially informational, these update efforts might be chosen too small in the beginning. Using *Experience Replay*, this information is then potentially incorporated steadily when updating the networks in later blocks. The slow rise in performance might in this case be partially attributed to the networks having a longer set up time to incorporate new information.

Gradient Boosting on the other hand uses the majority of all experienced episodes in the update cycle. In fact, for the calculations of Figure 29 gradient boosting used all available data up to block 50. It is therefore able to update Q-Values based on a large proportion of the data simultaneously and make them consistent to each other. Having stops (i.e. every 10 blocks), where the networks try to incorporate all data would more accurately show the potential of the available data. This could be combined with using several replay cycles subsequently to increase the consistency of Q-Values. These ideas can be implemented using the R-package Kies (2019). Doing so, however, increased run time considerably. They are therefore not feasible for practical applications in our setting.
Alternatively, one could set the replay data set to the same sizes as for gradient boosting. This also allows the networks to show a higher performance relative to the number of played blocks. As has been discussed in Section 3.8 however, increasing the replay data set has declining marginal utility. As simultaneously the training time increases linearly, this approach is also not practicable under efficiency considerations.

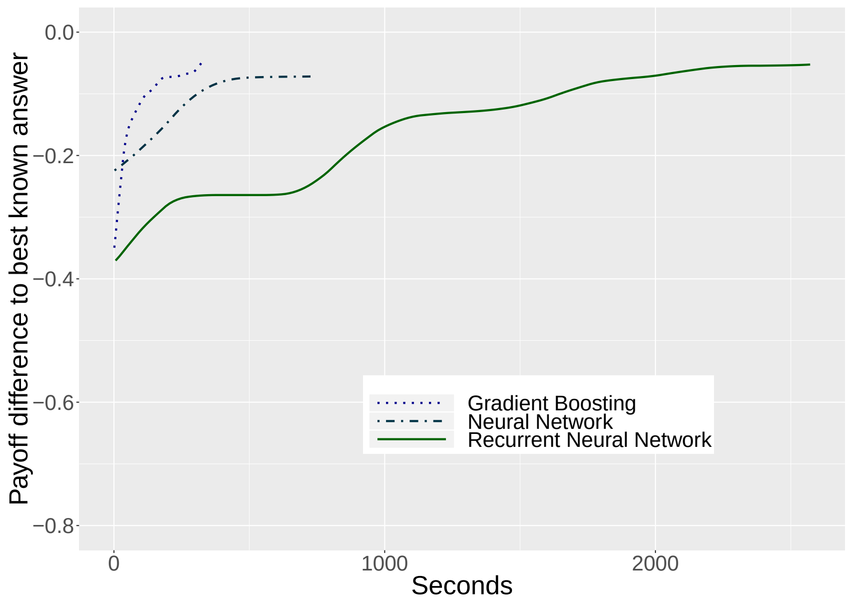

Figure 30: Comparison of performance of three different function approximator types according to the methodology described in Section 3.3 in relation to used seconds.
Displayed is the best aggregated performance up to the respective time over all hyper-parameter combinations given the respective function approximator type with at least 10 runs. The ticks have a distance of approximately 2.5 seconds. For the specific parameters, the specifications of the used hardware and more detailed information have a look at Appendix G.7.

Instead of focusing on the number of episodes, it is possible to ask what kind of final performance can be achieved if the hyperparameters are optimized with a specific time limit in mind. In the case of the networks one can for example increase the number of used neurons within a broad range to simultaneously increase performance and training time. Conversely, gradient boosting can change the number and depth of the used trees. The corresponding chart is depicted in Figure 30. Here, a simulation study over a wide variety of hyperparameters has been conducted. Depicted is the best result which could be achieved in the given time. To limit the impact of outliers, the depicted result had to be achieved by 10 runs. Nevertheless this method by design generates an upwards bias for each data point.
It can be seen, that in terms of time gradient boosting significantly outperforms the alternatives. If one is not interested in generating an extremely accurate final performance but rather wants to have a quick estimate about the strategy, gradient boosting is clearly the superior option.
This is not an artifact induced by the R-package Kies (2019). The overhead in regards to all function approximators is approximately the same. The bulk of the difference between the calculation times lies in external packages: Kies (2019) uses the R-package *xgboost* (xgboost.ai, 2019) for gradient boosting and *keras for R* (Arnold, 2019) for the networks.
Nevertheless, non-standard control of these external functions might influence the displayed times. Indeed, two major points should be discussed:
With the current implementation of the recurrent neural network each state-action pair is seen independently, just as with the other function approximators. This implies, that for each decision the complete history has to be recalculated and the hidden states of former calculations are not used as stepping stones. Fixing this issue should significantly speed up the recurrent network. Despite this consideration, the majority of time is lost on the actual training of the model and not the execution of the actions. The general point of the recurrent neural network being considerably slower therefore still stands.
On the other hand both networks are significantly speed up by an innovation I developed and implemented in Kies (2019). Classically, one generates a replay data set each replay and trains the network on it for a certain, fixed number of training epochs. An *epoch* of a neural network is a full update cycle of the network through the training data. Choosing the number of training epochs has a profound effect on convergence speed and the quality of the final result due to over/underfitting. I developed a simple method which changes the number of epochs at run time. Instead of choosing a single parameter for the number of epochs, two parameters are defined. Those are a base epoch number and the number of maximum repetitions. First, the network is trained on a number of epochs equal to the base epoch number. Afterwards, the algorithm checks the mean squared error on the training data. If this error is at least as small as in the previous replay the algorithm continues. If this is not the case, the network is trained again according to the base epoch number. The algorithm stops training the network if either the maximum number of repetitions is reached or the new training error is at least as small as the one of the previous replay.

This method allows the algorithm to concentrate training time on those updates which are actually relevant. In other words, more effort is used to update the network when the new data is surprising in the sense that a big correction in the network has to take place. If the to be trained Q-Values on the other hand are already expected, only a small amount of training time is used. As a result, the same performance can be achieved with significantly less training time. Indeed, always using the maximum number of epochs, i.e. base epochs multiplied with maximum number of repetitions, not only increases run time, but also lower performance due to overfitting.
Gradient Boosting could be greatly sped up, if one could update it incrementally. This however is currently not yet possible, even though incremental improvements for Gradient Boosting have been developed by C. Zhang et al. (2019) for classification problems.
We now focus on the third point on how to determine which function approximator should be used when arbitrary long games are of interest. In principle all function approximators can be used for arbitrary long games if the encoding (see Section 3.7) is chosen appropriately. One might for example only use the last X periods in combination with some additional helpful aggregate statistics over the complete history as input. This however can lead to forgetting important information from the beginning of the episode if it is not accurately reflected by the additional aggregate information. Only the recurrent neural network provides the built-in functionality to avoid this situation. If the goal is to evaluate the strategy without limiting the number of periods, the recurrent neural network is therefore the best choice.
To summarize, no function approximator is the optimal choice in all cases. In particular it is not obvious, that the standard approach of using a neural network is the recommended choice given the setting. If fast results are desired, gradient boosting is the best choice based on our studies. Given sufficient time and a high number of episodes, an updating function approximator might be the best choice. The more flexible and in the end more powerful version according to our studies is the recurrent neural network.
These studies can easily be extended to other function approximators, as for example *genetic algorithms* or *particle swarms*. The function approximator *random forest* is already implemented in the R-package Kies (2019). Here, the underlying implementation of the R-package *grf* (J. Tibshirani et al., 2019) has been used. Random forest produces results very similar to gradient boosting. It has however the strong disadvantage that the evaluation of actions takes considerably longer. It is necessary to evaluate a great number of state-action pairs when generating experiences. Using random forest therefore slows down the speed of learning significantly and makes it a weaker choice than gradient boosting.

3.10. Model Persistence

In theory Q-Learning, and by extension the algorithm, convergences toward the optimal policy given a sufficient function approximator. The learning process is not deterministic due to stochastic exploration and innate features of the setting. Consequently, convergence is not monotonous, even though the shown figures might suggest otherwise. All displayed figures are based on aggregating up to several hundred runs and display an approximation of the convergence path in expectation. Within a single run it can easily be the case, that an update of the function approximator results in a worse policy. This might for example happen when non-characteristic observation errors suggest Q-Values which are in line with a suboptimal policy.
The, to my knowledge, novel approach of *Model Persistence* introduces a simple concept: Do not throw away a model which has performed good in the past, even if it no longer accurately represents current experiences. A *model* in this case is the set of all approximated Q-Values and therefore directly defined by the used function approximator. Saving a model is consequently identical to saving the mathematical representation of the specific function approximator itself. The name has been chosen because *Model Persistence* can directly be applied to tabular Q-Learning. The existence of a function approximator is therefore not required.
Note, that a different model does not necessarily imply a different policy. To change the policy there has to be at least one state, where a different action is taken. A change in Q-Values is a necessary but not a sufficient condition for this.
Within the R-package Kies (2019) this is implemented by using at least one additional episode per block or alternatively replacing one of them. This new episode is played according to the so-called *best* model. The standard approach within this article uses one exploration episode and one episode according to the current model per block. Including *Model Persistence* we might therefore for example use one exploration episode, one with the best model and one with the current model per block.
The best model is chosen the following way:

```
Both models, current and best, are initialized.
For $i = 1$ to number of to be played blocks:
    Play the episodes of block $i$.
    If (average) performance of current model in block $i$ > (average) performance of best model in block $i$
        best model $\leftarrow$ current model
    Else
        # Nothing happens, the best model stays the same
    current model is updated.
```

The underlying used performance is the average historic reward of the episodes with the given class of the block in question. By design *Model Persistence* requires at least one episode according to the current model per block. As before, at least one exploration episode per block is necessary to have a learning process. Also analogously to before all experiences are indiscriminately saved into the memory.

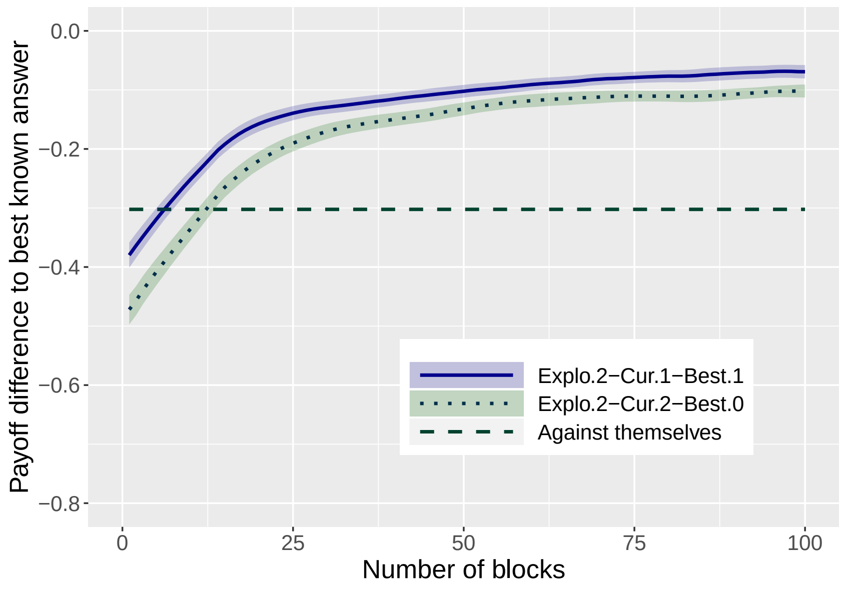

Figure 31: Comparison of performance of two different block compositions according to the methodology described in Section 3.3. “Explo.2-Cur.2-Best.0” uses two episodes with exploration and two episodes with the most recent, current model per block. “Explo.2-Cur.1-Best.1” uses two episodes of exploration, one episode of the best model and one episode of the most current model.
All data points have been generated on 20 runs for each strategy. For the specific parameters and more detailed information see Appendix G.8.

The effects of using *Model Persistence* are shown in Figure 31. It can be seen, that using *Model Persistence* has a significant and relevant effect on the performance across all stages of learning as well as final performance. Note, that to keep comparability only the average performance of the episodes with the current model are shown. Additionally, each block uses the same number of total episodes. Having the same number of total episodes allows to separate the effects of *Model Persistence* from having more training data.
To understand why *Model Persistence* is beneficial, consider the following:

1. While the best model does not reflect the most recently gained knowledge it is by design able to generate high rewards. Assume that there is a difference between the best and the current model and the best model scores higher. This implies, that the Q-Values estimated by the current function approximator do not represent the true values. Generating more data with the best model might thus improve the algorithm by providing more example data on how to play with a high performance.

2. Having two different best effort episodes allows for more variation in learning. This variation allows for a different form of exploration additionally to the exploration episode. Using data generated this way makes the function approximator more robust and thereby better able to generalize.

It might be reasonable to conduct further research regarding *Model Persistence* and variants thereof. The performance increase due to *Model Performance* might be dependent on the amount of randomness within the environment. With the current implementation the best model is often replaced because of a different draw in observations errors and not necessarily due to the newer model having a better expected performance. It is possible to counteract this effect by playing a higher number of episodes per block for each type of episode. This allows to get a more accurate representation of the expected value of the performance of the underlying policies. Doing so however comes with a cost. The algorithm learns considerably slower, as the function approximator is updated less often in relation to the number of episodes. A more sophisticated approach however might be to have a stack of best models and using the most probable best model with a higher probability, for example using Bayesian updating.
At first glance similar, but fundamentally different is the concept of *double Q-Learning* (see Van Hasselt (2010) for the origin and Van Hasselt et al. (2016) for the combination of double Q-Learning with the Deep Q-Network of Mnih et al. (2015)). Double Q-Learning addresses the problem, that there is an upwards bias in the Q-Learning algorithm which overestimates the value of the to be chosen action. A good intuitive explanation for this effect can be given if one assumes, that there is a stochastic environment and all actions lead to the same probability of next state and expected reward. Due to chance some of these actions will have higher rewards within the set of experiences. As the Q-Learning algorithm takes these results at face value, the calculated Q-Values for these actions are too high. Double-Q Learning trains two different function approximators, which each train on a random subset of experiences, thus differing in their assessment. When updating a function approximator one of them is used to determine which action $\hat{a}$ should be played, but the other one generates the $Q(s, \hat{a})$ value used for the update. This counteracts a significant portion of the upwards bias. Double Q-Learning is compatible with both *Q-Switching* as well as *Model Persistence*. Extending the algorithm of this article with double Q-Learning could increase final performance and speed of learning.

3.11. Memory Initialization

Basic Q-Learning uses a blank slate approach to learn about the environment. This means that all Q-Values are initialized independent on their specific state-action pairs. Several initialization variants exist for the tabular approach. One basic approach is to initialize each Q-Value with the same value.

Function approximators also have to be initialized. As a default case they are often initialized as zero. This approach is also used with gradient boosting throughout this article. The convergence of neural networks can strongly be increased by using various techniques to initialize the weights of the network to non-zero (Mishkin and Matas, 2015). The R-package Kies (2019) adds small random noise when setting up a network by default. This type of initialization has been used for the studies regarding the networks. From a reinforcement learning point of view this approach however still effectively amounts to an initialization at zero, as the effect on the generated Q-Values is small. While adding noise to the starting-weights of the network is beneficial for technical reasons, it therefore has no relevant influence on the starting policies.

If already some information about the environments is known, it can be incorporated into the initialization. If for example theoretical considerations imply certain actions to be more desirable, these actions can be initialized with higher starting values. The algorithm will then play these actions with higher probabilities, thus giving it a head start.

Given a game with sparse rewards this approach might not be sufficient. Take for example a racing game, where a policy is judged by the time it needs to complete a lap. Having random or zero initialization of Q-Values and always starting at the start of the lap, it might take a nearly infinite time to actually complete a lap at all. One possible solution is to introduce subgoals. The algorithm might for example get an additional reward based on the time the policy needs to complete the first percent of the lap. In this case, however, the incentives for the algorithms get distorted. As a result hard-coding additional rewards for subgoals can result in a suboptimal policy as measured by the initial performance measure.

Often times a more elegant way to nudge the algorithm in the right direction is *Memory Initialization*. This method works in unison with the concept of *Experience Replay*: The memory database is initialized with experiences which were not calculated by the algorithm itself, but through other means. The general approach is to provide the memory database with successful experiences. These allow the algorithm to correctly calibrate on the to be expected rewards given good play. In the example of the racing game this could for example be data based on several completed laps. This type of initialization is often done with experiences of human experts. Silver, Schrittwieser, et al. (2017) for example used a data set of 30 million game-state positions to build the groundwork for their policy network of AlphaGo.

The R-package Kies (2019) provides a variety of types of initialization. Within this article however we focus on a particularly important one, which we call *self initialization*. Self initialization in this context is the acquisition of experiences by having the strategy play against itself. Without noise a single run is sufficient, but in our setting it is favorable to observe how the strategy reacts to noise at different points in its play.
One advantage of self initialization is that no environment specific knowledge is needed. This form of initialization therefore does not require manual pre-analysis.
Given the goal to evaluate the stability of a strategy, self initialization is particularly interesting. If the strategy is a Nash equilibrium, no deviation from the strategy itself can be found. Having a memory database with self-play, it is probable, that the generated policy at least matches the general performance level of the strategy. This is beneficial if the strategy indeed already is a Nash equilibrium.
Even if the strategy is not a Nash equilibrium, there is a distinct chance that the optimal counter-policy will be found in some vicinity to it. Two examples for such strategies are *grim.trigger* and *tft.2forgive*. Here the best responses are slight deviations of the strategies themselves. That the optimal policy might be found within the vicinity of the strategy holds especially true, if the strategy uses some kind of key pattern to determine self-play and ensuing cooperation. A good candidate for optimal play against this kind of strategy is a policy which masks itself by playing the key pattern. Afterwards a switch to exploiting the opponent can take place. Finding such a key pattern might be difficult if it is sufficiently distinct from patterns which generate promising rewards otherwise. Given self initialization, however, exploration episodes will tend to explore policies which are similar to the strategy itself. This allows the algorithm to find these types of policies considerably faster.
Memory Initialization is not without risk, however. The algorithm might get stuck in a local optimum if the provided experiences strongly suggest a particular avenue of play but a slight deviation is costly. If the best policy follows a sufficiently different paradigm, it might never be found. This is one of the reasons why the DeepMind Team made the deliberate decision to forgo *Memory Initialization* when constructing AlphaZero (Silver, Hubert, et al., 2017). This allowed them to negate possible bias of human strategies.
The actual effect of using self initialization given our setting is presented in Figure 32. Using a higher number of self plays indeed significantly improves performance at the beginning of the learning phase. Even using a simple initialization with just a single playthrough of the strategy against itself notably speeds up learning. Interestingly, this does not matter in the long run, as all variants converge towards the same final performance.

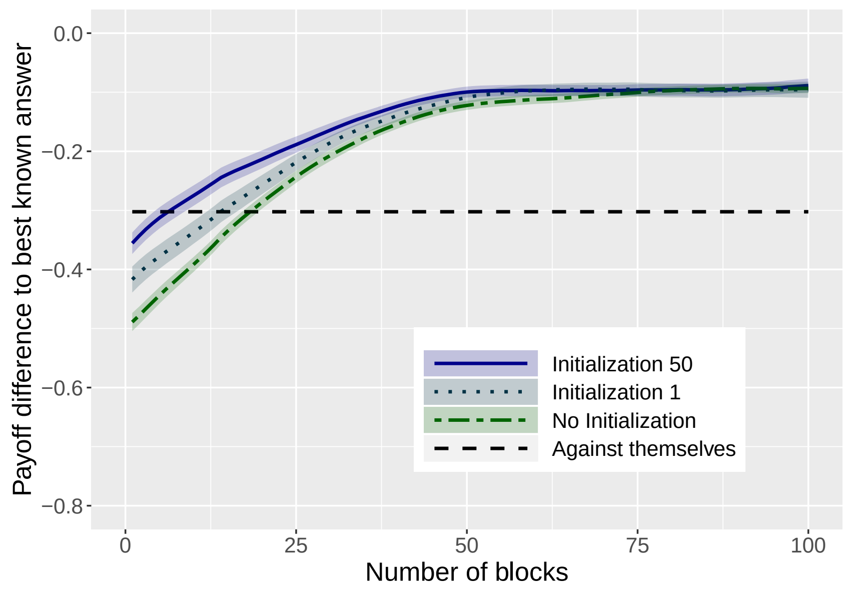

Figure 32: Comparison of different strengths of self initialization according to the methodology described in Section 3.3. "Initialization X" filled the memory database with X episodes of self-play of the opponent strategy beforehand.
The data for both self initialization types has been generated using 20 runs for each of the strategies. In the case of no initialization, 60 runs per strategy have been simulated. For the specific parameters and more detailed information see Appendix G.9.

Given the setting and approach of analysis, the following summary can be made:

1. *Memory Initialization* considerably improves the performance at the beginning of the learning process.

2. At least on average the algorithm is not trapped in a local optimum due to *Memory Initialization*. Branching out the search space outgoing from self initialization does not hurt final performance.

3. The testing pool strategies do not have patterns, which are not otherwise deducible by the *Exploration Path* with a *ε-exploration* / *noisy action* exploration. *Memory Initialization* is therefore not necessary if only final performance is considered.

3.12. Summary of Analysis

To test exploitability of given strategies in the Iterated Prisoner's Dilemma game with noise, I developed an algorithm which uses reinforcement learning to find near optimal policies as proxies for game-theoretic best responses. Outgoing from a basic Q-Learning algorithm with function approximation as described in Sections 2.3.3 and 2.3.4, I developed and/or tested the following modifications. Each of them is detailed in its respective section. The following list gives a short summary of the main learning points given the testing environment. It also discusses briefly which configurations were used for the final algorithm to generate Figures 10 and 12 (see Section 3.2).

1. **Q-Switching**
   Combining Monte Carlo learning and classic Q-Learning allows both pure approaches to contribute with their respective advantages. The learning process when using *Q-Switching* shows in the beginning approximately the same performance as Monte Carlo learning. However, with increasing number of episodes it improves faster. Across the complete training period, *Q-Switching* generates policies with a higher expected average reward than its pure alternatives. In particular, the average performance of the final policy was highest when using *Q-Switching*.
   Additionally, policies generated by *Q-Switching* show less variation in final performance compared to the analyzed alternatives. This modification therefore increases reliability of training.
   Strategies exist, where either Monte Carlo learning or Q-Learning is a much better choice to construct a counter-policy compared to the other one. In these cases *Q-Switching* performs between both pure approaches. This makes it a generalized approach which is useful in the absence of deeper knowledge about the to be analyzed strategy but not necessarily optimal in edge cases.
   Q-Switching was used in the final algorithm due to its clear advantages.

2. **Multi-Exploration**
   Situations and settings exist, where more sophisticated exploration methods improve learning behavior compared to *ε-exploration*. In the analyzed setting, however, this proved not to be the case. Pure *ε-exploration* performs best compared to using pure methods based on Boltzmann-exploration, trying to maximize surprise or encouraging exploration by letting the agent seek out unfamiliar territory.
   Given the IPD game, there exist parameter settings, where *ε-exploration* indeed performs worse. With these settings, we showed that combining several exploration methods can increase performance. However, this does not hold true for the standard setting. Here, the combination of exploration approaches does not achieve significant improvements. It is therefore questionable to use exploration methods which are comparatively resource intensive.

The combination of *ε-exploration* and *noisy actions* showed the best performance and is resource-efficient. Additionally, there exist theoretical and empirical arguments in favor of their combination. Consequently, this combination was used for the final configuration.

3. **Exploration Pathing**
   Instead of decreasing the exploration factor by a fixed rule, it is possible to let the degree of exploration depend on relative performance between episodes with exploration and those without. This allows for a more efficient use of exploration episodes. *Exploration Pathing* results in a significantly better performance of the average final policy compared to a fixed exponential decreasing rule.
   Furthermore, *Exploration Pathing* complements *Multi-Exploration* by balancing the size and therefore relative importance of different exploration parameters.
   For the final configuration a constant ratio between exploration episodes and best effort episodes of 85% was used. My intention was to balance the effects of possibly getting stuck in a local optimum on the one hand and staying sufficiently near the optimal policy to achieve efficient exploration on the other hand. On the recurrent neural network run with a much higher number of episodes the ratio increased linearly from 85% to 95% to reflect the goal of generating a near optimal policy.

4. **Feature Selection**:
   Using the right features for the input of the function approximator has a profound impact on the performance of the developed policy. I propose an encoding which works well on a wide selection of strategies and produced policies with a higher performance than the encoding of Harper et al. (2017).
   If the encoding is restricted to only use raw data of the last X periods, then using 4 to 6 periods resulted in the best performances.
   Using a recurrent neural network which is able to keep track of desired features mitigates the need for a handcrafted feature space.
   I used my developed feature selection for the gradient boosting run. Only the very last period as raw data was used for the recurrent neural network run, as this configuration showed comparable performance but was considerably more resource-efficient.

5. **Experience Replay**
   The improvement of Mnih et al. (2015) to use random batches of the memory to train the function approximator is beneficial in our setting as well. These benefits are not limited to neural networks and hold over a variety of different function approximators.
   Due to being a non-updating function approximator, gradient boosting has to use *Experience Replay.* As it also proved beneficial for the recurrent neural network, it was used here as well.

6. **Choice of Function Approximators**
   The standard choice of using a neural network as the function approximator is not a forgone conclusion. Gradient boosting achieves the same final performance in a faster time and earlier in the training process, i.e. using less training data. The recurrent neural network is slower in computation and speed of learning, but generates policies which do not have to be restricted to a maximum number of periods. In contrast to gradient boosting the recurrent neural network is updating, i.e. needs approximately a constant amount of time per update when converging towards the best policy. Gradient boosting on the other hand has to be rebuild on an increasing amount of training data if the same effect has to be achieved and needs therefore an increasing amount of time per update. The recurrent neural network is therefore the preferred choice if a very large amount of training data needs to be generated to achieve a high degree of precision.

7. **Model Persistence**
   Instead of deleting old models when updating the function approximator, it is beneficial to keep the historically best performing model and use it to generate more data. This significantly improves the speed of the learning process as well as the average performance of the final policy.
   Consequently, I used this feature for both variants of the final algorithm.

8. **Memory Initialization**
   Instead of using a green field approach, one can initialize and pre-train the function approximator on training episodes. We analyzed the effect of using episodes of the strategy playing against itself to set a starting point for exploration. Doing so significantly improves performance of the policies generated at the start of the learning process. However, this effect tethers out in our setting. Given a sufficient number of training episodes, final performance is not significantly changed.
   Even though no real benefits for the final policy are to be expected, I used this feature due to it being non-harmful and adding only a relatively small amount of computation time.

References

Abreu, Dilip, Prajit K. Dutta, and Lones Smith (1994). "The Folk Theorem for Repeated Games: A NEU Condition". In: *Econometrica: Journal of the Econometric Society*, pp. 939–948. DOI: 10.2307/2951739.

Abreu, Dilip, David Pearce, Ennio Stacchetti, et al. (1990). "Toward a Theory of Discounted Repeated Games with Imperfect Monitoring". In: *Econometrica* 58.5, pp. 1041–1063. DOI: 10.2307/2938299.

Almuallim, Hussein (1996). "An efficient algorithm for optimal pruning of decision trees". In: *Artificial Intelligence* 83.2, pp. 347–362. DOI: 10.1016/0004-3702(95)00060-7.

Arnold, Taylor (2019). *kerasR: R Interface to the Keras Deep Learning Library.* Version 0.6.1. URL: https://cran.r-project.org/package=kerasR.

Ashlock, Daniel, Joseph Alexander Brown, and Philip Hingston (2014). "Multiple Opponent Optimization of Prisoner's Dilemma Playing Agents". In: *IEEE Transactions on Computational Intelligence and AI in Games* 7.1, pp. 53–65. DOI: 10.1109/TCIAIG.2014.2326012.

Axelrod, Robert (1984). *The Evolution of Cooperation.* New York: Basic Books. ISBN: 9780465021222.

Axelrod, Robert and William D. Hamilton (1981). "The Evolution of Cooperation". In: *science* 211.4489, pp. 1390–1396. DOI: 10.1126/science.7466396.

Bellman, Richard E. (1957). *Dynamic Programming.* Princeton University Press. ISBN: 9780691146683.

— (2003). *Dynamic Programming.* Dover Books on Computer Science Series. Dover Publications. ISBN: 9780486428093.

Boyd, Robert (1989). "Mistakes Allow Evolutionary Stability in the Repeated Prisoner's Dilemma Game". In: *Journal of theoretical Biology* 136.1, pp. 47–56. DOI: 10.1016/S0022-5193(89)80188-2.

Brunauer, Richard et al. (2007). "Evolution of Iterated Prisoner's Dilemma Strategies with Different History Lengths in Static and Cultural Environments". In: *Proceedings of the 2007 ACM symposium on Applied computing.* ACM, pp. 720–727. DOI: 10.1145/1244002.1244163.

Burda, Yuri, Harrison Edwards, Deepak Pathak, et al. (2018). "Large-Scale Study of Curiosity-Driven Learning". In: *arXiv preprint arXiv:1808.04355.* URL: https://arxiv.org/abs/1808.04355.

Burda, Yuri, Harrison Edwards, Amos Storkey, et al. (2018). "Exploration by Random Network Distillation". In: *arXiv preprint arXiv:1810.12894.* URL: https://arxiv.org/abs/1810.12894.

Cochran, William G. (1977). *Sampling Techniques, 3rd Edition.* Wiley & Sons Ltd. ISBN: 9780471162407.

Cominos, P. and N. Munro (2002). "PID controllers: recent tuning methods and design to specification". In: *IEE Proceedings-Control Theory and Applications* 149.1, pp. 46–53. DOI: 10.1049/ip-cta:20020103.

Dayan, Peter and Terrence J. Sejnowski (1994). “TD (λ) Converges with Probability 1”. In: *Machine Learning* 14.3, pp. 295–301. DOI: 10.1007/BF00993978.

Foerster, Jakob et al. (2018). “Learning with Opponent-Learning Awareness”. In: *Proceedings of the 17th International Conference on Autonomous Agents and MultiAgent Systems*. International Foundation for Autonomous Agents and Multiagent Systems, pp. 122–130. URL: https://dl.acm.org/citation.cfm?id=3237408.

Freund, Yoav, Robert E. Schapire, et al. (1996). “Experiments with a New Boosting Algorithm”. In: *icml*. Vol. 96. Citeseer, pp. 148–156. ISBN: 9781558604193. URL: https://cseweb.ucsd.edu/~yfreund/papers/boostingexperiments.pdf.

Friedman, James W. (1971). “A Non-Cooperative Equilibrium for Supergames”. In: *The Review of Economic Studies* 38.1, pp. 1–12. DOI: 10.2307/2296617.

Friedman, Jerome, Trevor Hastie, Robert Tibshirani, et al. (2000). “Additive Logistic Regression: A Statistical View of Boosting (With Discussion and a Rejoinder by the Authors)”. In: *The Annals of Statistics* 28.2, pp. 337–407. URL: https://web.stanford.edu/~hastie/Papers/AdditiveLogisticRegression/alr.pdf.

Fudenberg, Drew, David Levine, and Eric Maskin (1994-09). “The Folk Theorem with Imperfect Public Information”. In: *Econometrica* 62.5, pp. 997–1039. DOI: 10.2307/2951505.

Goldlücke, Susanne and Sebastian Kranz (2012). “Infinitely repeated games with public monitoring and monetary transfers”. In: *Journal of Economic Theory* 147.3, pp. 1191–1221. DOI: 10.1016/j.jet.2012.01.008.

Gorman, Ben (2017). *A Kaggle Master Explains Gradient Boosting*. URL: http://blog.kaggle.com/2017/01/23/a-kaggle-master-explains-gradient-boosting/ (visited on 2019-10-07).

Grover, Prince (2017). *Gradient Boosting from scratch*. URL: https://medium.com/mlreview/gradient-boosting-from-scratch-1e317ae4587d (visited on 2019-10-07).

Harper, Marc et al. (2017). “Reinforcement learning produces dominant strategies for the Iterated Prisoner’s Dilemma”. In: *PloS one* 12.12, e0188046. DOI: 10.1371/journal.pone.0188046.

Hastie, Trevor, Robert Tibshirani, and Jerome Friedman (2009). *The Elements of Statistical Learning: Data mining, Inference, and Prediction*. Springer Science & Business Media. ISBN: 9780387848570.

Hebb, Donald Olding (2002). *The Organization of Behavior: A Neuropsychological Theory*. Psychology Press. ISBN: 9780805843002.

Hessel, Matteo et al. (2018). “Rainbow: Combining Improvements in Deep Reinforcement Learning”. In: *Thirty-Second AAAI Conference on Artificial Intelligence*. URL: https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/viewPaper/17204.

Hochreiter, Sepp and Jürgen Schmidhuber (1997). “Long Short-Term Memory”. In: *Neural computation* 9.8, pp. 1735–1780. DOI: 10.1162/neco.1997.9.8.1735.

Iizuka, Satoshi, Edgar Simo-Serra, and Hiroshi Ishikawa (2016). “Let there be Color!: Joint End-to-end Learning of Global and Local Image Priors for Automatic

Image Colorization with Simultaneous Classification". In: *ACM Transactions on Graphics (Proc. of SIGGRAPH 2016)* 35.4. DOI: 10.1145/2897824.2925974.

Judd, Kenneth L., Sevin Yeltekin, and James Conklin (2003). "Computing Supergame Equilibria". In: *Econometrica* 71.4, pp. 1239–1254. DOI: 10.1111/1468-0262.t01-1-00445.

Kaelbling, Leslie Pack, Michael L. Littman, and Andrew W. Moore (1996). "Reinforcement Learning: A Survey". In: *Journal of artificial intelligence research* 4, pp. 237–285. DOI: 10.1613/jair.301.

Kaiser, Lukasz et al. (2019). "Model-Based Reinforcement Learning for Atari". In: *arXiv preprint arXiv:1903.00374*. URL: https://arxiv.org/abs/1903.00374.

Kies, Martin (2019). *RLR*. https://github.com/MartinKies/RLR. URL: https://github.com/MartinKies/RLR/commit/3b8b1bfd4b4766b1e612de85a96c8d24e95d33e6.

Kraines, David and Vivian Kraines (1989). "Pavlov and the prisoner's dilemma". In: *Theory and decision* 26.1, pp. 47–79. DOI: 10.1007/BF00134056.

Kranz, Sebastian and Martin Kies (2019). *StratTourn*. https://github.com/skranz/StratTourn. URL: https://github.com/skranz/StratTourn/commit/7c942551f510084de5ab20c5874d62fbe47332d5.

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton (2012). "ImageNet classification with deep convolutional neural networks". In: *NIPS'12 Proceedings of the 25th International Conference on Neural Information Processing Systems*. Vol. 1, pp. 1097–1105. URL: https://dl.acm.org/citation.cfm?id=2999257.

Kuhn, Steven (2019). "Prisoner's Dilemma". In: *The Stanford Encyclopedia of Philosophy*. Ed. by Edward N. Zalta. Summer 2019. Metaphysics Research Lab, Stanford University. URL: https://plato.stanford.edu/archives/sum2019/entries/prisoner-dilemma/.

Lahno, Bernd (2000). "In Defense of Moderate Envy". In: *Analyse & Kritik* 22.1, pp. 98–113. DOI: 10.1515/auk-2000-0105.

Leibo, Joel Z. et al. (2017). "Multi-agent Reinforcement Learning in Sequential Social Dilemmas". In: *Proceedings of the 16th Conference on Autonomous Agents and MultiAgent Systems*. International Foundation for Autonomous Agents and Multiagent Systems, pp. 464–473. URL: http://www.ifaamas.org/Proceedings/aamas2017/pdfs/p464.pdf.

Li, Cheng (2016). *A Gentle Introduction to Gradient Boosting*. URL: http://www.ccs.neu.edu/home/vip/teach/MLcourse/4_boosting/slides/gradient_boosting.pdf (visited on 2019-10-07).

Li, Jiawei, Philip Hingston, and Graham Kendall (2011). "Engineering Design of Strategies for Winning Iterated Prisoner's Dilemma Competitions". In: *IEEE Transactions on Computational Intelligence and AI in Games* 3.4, pp. 348–360. DOI: 10.1109/TCIAIG.2011.2166268.

Lin, Long-Ji (1992). "Self-Improving Reactive Agents Based On Reinforcement Learning, Planning and Teaching". In: *Machine learning* 8.3-4, pp. 293–321. DOI: 10.1007/BF00992699.

Littman, Michael L. and Richard S. Sutton (2002). “Predictive Representations of State”. In: *Advances in Neural Information Processing Systems 14*. Ed. by T. G. Dietterich, S. Becker, and Z. Ghahramani. MIT Press, pp. 1555–1561. URL: http://papers.nips.cc/paper/1983-predictive-representations-of-state.pdf.

Martin, Jarryd et al. (2017). “Count-Based Exploration in Feature Space for Reinforcement Learning”. In: *arXiv preprint arXiv:1706.08090*. URL: https://arxiv.org/abs/1706.08090.

McCulloch, Warren S. and Walter Pitts (1943). “A logical calculus of the ideas immanent in nervous activity”. In: *The bulletin of mathematical biophysics* 5.4, pp. 115–133. DOI: 10.1007/BF02478259.

McGillivray, Fiona and Alastair Smith (2000). “Trust and Cooperation through Agent-specific Punishments”. In: *International Organization* 54.4, pp. 809–824. DOI: 10.1162/002081800551370.

Minorsky, Nicolas (1922). “Directional Stability of Automatically Steered Bodies”. In: *Journal of the American Society for Naval Engineers* 34.2, pp. 280–309. DOI: 10.1111/j.1559-3584.1922.tb04958.x.

Mishkin, Dmytro and Jiri Matas (2015). “All you need is a good init”. In: *arXiv preprint arXiv:1511.06422*. URL: https://arxiv.org/abs/1511.06422.

Mnih, Volodymyr et al. (2015). “Human-level control through deep reinforcement learning”. In: *Nature* 518.7540, pp. 529–533. DOI: 10.1038/nature14236.

Molander, Per (1985). “The Optimal Level of Generosity in a Selfish, Uncertain Environment”. In: *Journal of Conflict Resolution* 29.4, pp. 611–618. URL: https://www.jstor.org/stable/174244.

Monahan, George E. (1982). “State of the Art—A Survey of Partially Observable Markov Decision Processes: Theory, Models, and Algorithms”. In: *Management Science* 28.1, pp. 1–16. DOI: 10.1287/mnsc.28.1.1.

Morey, Richard (2008). “Confidence Intervals from Normalized Data: A correction to Cousineau (2005)”. In: *Tutorials in Quantitative Methods for Psychology* 4, pp. 61–64. DOI: 10.20982/tqmp.04.2.p061.

Nielsen, Michael (2019). *Neural Networks and Deep Learning*. URL: http://neuralnetworksanddeeplearning.com/index.html (visited on 2019-10-07).

Nowak, Martin and Karl Sigmund (1993). “A strategy of win-stay, lose-shift that outperforms tit-for-tat in the Prisoner’s Dilemma game”. In: *Nature* 364.6432, pp. 56–58. DOI: 10.1038/364056a0.

Olah, Christopher (2015). *Understanding LSTM Networks*. URL: https://colah.github.io/posts/2015-08-Understanding-LSTMs/ (visited on 2019-10-07).

Peng, Jing and Ronald J. Williams (1994). “Incremental Multi-Step Q-Learning”. In: *Machine Learning Proceedings 1994*. Elsevier, pp. 226–232. DOI: 10.1016/B978-1-55860-335-6.50035-0.

Plappert, Matthias et al. (2017). “Parameter Space Noise for Exploration”. In: *arXiv preprint arXiv:1706.01905*. URL: https://arxiv.org/abs/1706.01905.

Press, William H. and Freeman J. Dyson (2012). “Iterated Prisoner’s Dilemma contains strategies that dominate any evolutionary opponent”. In: *Proceedings* *of the National Academy of Sciences* 109.26, pp. 10409–10413. DOI: 10.1073/pnas.1206569109.

Rajkomar, Alvin et al. (2018). “Scalable and accurate deep learning with electronic health records”. In: *NPJ Digital Medicine* 1.1, p. 18. DOI: 10.1038/s41746-018-0029-1.

Rapoport, Anatol, Albert M. Chammah, and Carol J. Orwant (1965). *Prisoner’s Dilemma: A Study in Conflict and Cooperation.* Vol. 165. University of Michigan press. ISBN: 9780472061655.

Rogozhnikov, Alex (2016). *Gradient Boosting Interactive Playground.* URL: http://arogozhnikov.github.io/2016/07/05/gradient_boosting_playground.html (visited on 2019-10-07).

Rosenblatt, Frank (1958). “The perceptron: a probabilistic model for information storage and organization in the brain.” In: *Psychological review* 65.6, p. 386. DOI: 10.1037/h0042519.

Roth, Alvin E. and J. Keith Murnighan (1978). “Equilibrium Behavior and Repeated Play of the Prisoner’s Dilemma”. In: *Journal of Mathematical psychology* 17.2, pp. 189–198. DOI: 10.1016/0022-2496(78)90030-5.

Rumelhart, David E., Geoffrey E. Hinton, Ronald J. Williams, et al. (1986). “Learning representations by back-propagating errors”. In: *Nature* 323. DOI: 10.1038/323533a0.

Rummery, Gavin A. and Mahesan Niranjan (1994). *On-Line Q-Learning Using Connectionist Systems.* Vol. 37. University of Cambridge, Department of Engineering Cambridge, England. URL: https://www.researchgate.net/publication/2500611_On-Line_Q-Learning_Using_Connectionist_Systems.

Samuel, Arthur L. (1959). “Some Studies in Machine Learning Using the Game of Checkers”. In: *IBM Journal of Research and Development* 3 (3), pp. 210–229. DOI: 10.1147/rd.33.0210.

Sandholm, Tuomas W. and Robert H. Crites (1996). “Multiagent reinforcement learning in the Iterated Prisoner’s Dilemma”. In: *Biosystems* 37.1-2, pp. 147–166. DOI: 10.1016/0303-2647(95)01551-5.

Schaul, Tom et al. (2015). “Prioritized Experience Replay”. In: *arXiv preprint arXiv:1511.05952.* URL: https://arxiv.org/abs/1511.05952.

Shannon, Claude E. (1950). “Programming a computer for playing chess”. In: *The London, Edinburgh, and Dublin Philosophical Magazine and Journal of Science* 41.314, pp. 256–275. DOI: 10.1007/978-1-4757-1968-0_1.

Siebrasse, Norman (2006). “’The Prince’-A Robust Strategy in the Repeated Prisoner’s Dilemma with Noise”. In: *SSRN.* DOI: 10.2139/ssrn.952370.

Silver, David (2015). *UCL Course on RL.* University College London. URL: http://www0.cs.ucl.ac.uk/staff/d.silver/web/Teaching.html (visited on 2019-10-07).

Silver, David, Thomas Hubert, et al. (2017). “Mastering Chess and Shogi by Self-Play with a General Reinforcement Learning Algorithm”. In: *arXiv preprint arXiv:1712.01815.* URL: https://arxiv.org/abs/1712.01815.

Silver, David, Julian Schrittwieser, et al. (2017). “Mastering the game of Go without human knowledge”. In: *Nature* 550.7676, p. 354. DOI: 10.1038/nature24270.

Sugden, Robert (1986). *The Economics of Rights, Co-Operation, and Welfare.* Oxford: Blackwell Pub. ISBN: 9780333682395.

Sutton, Richard S. (1988). “Learning to Predict by the Methods of Temporal Differences”. In: *Machine learning* 3.1, pp. 9–44. DOI: 10.1007/BF00115009.

Sutton, Richard S. and Andrew G. Barto (2018). *Reinforcement Learning: An introduction.* MIT press. ISBN: 9780262039246. URL: http://incompleteideas.net/book/RLbook2018.pdf.

The AlphaStar Team (2019-01-24). *AlphaStar: Mastering the Real-Time Strategy Game StarCraft II.* URL: https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/ (visited on 2019-10-07).

The Axelrod project developers (2016a-04). *Axelrod: v4.6.0.* DOI: 10.5281/zenodo.3050770.

— (2016b). *Axelrod: Version v4.6.0, Source code for axelrod.strategies.titfortat.* URL: https://axelrod.readthedocs.io/en/stable/_modules/axelrod/strategies/titfortat.html (visited on 2019-10-07).

Thomas, Konstantinos (2018). “Building a Reinforcement Learning A.I. for the Iterated Prisoner’s Dilemma using Soar cognitive architecture”. Master Thesis. National and Kapodistrian University of Athens. URL: https://pergamos.lib.uoa.gr/uoa/dl/frontend/file/lib/default/data/2778307/theFile (visited on 2019-10-07).

Thrun, Sebastian B. (1992). *Efficient exploration in reinforcement learning.* Tech. rep. URL: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.44.4011.

Tibshirani, Julie et al. (2019). *grf: Generalized Random Forests.* Version 0.10.4. URL: https://cran.r-project.org/package=grf.

Tijsma, Arryon D, Madalina M Drugan, and Marco A Wiering (2016). “Comparing Exploration Strategies for Q-learning in Random Stochastic Mazes”. In: *2016 IEEE Symposium Series on Computational Intelligence (SSCI).* IEEE, pp. 1–8. DOI: 10.1109/SSCI.2016.7849366.

Van Hasselt, Hado (2010). “Double Q-learning”. In: *Advances in Neural Information Processing Systems 23.* Ed. by J. D. Lafferty et al. Curran Associates, Inc., pp. 2613–2621. URL: http://papers.nips.cc/paper/3964-double-q-learning.pdf.

Van Hasselt, Hado, Arthur Guez, and David Silver (2016). “Deep Reinforcement Learning with Double Q-Learning”. In: *Thirtieth AAAI conference on artificial intelligence.* URL: https://www.aaai.org/ocs/index.php/AAAI/AAAI16/paper/viewPaper/12389.

Wang, Keven (2017). “Iterated Prisoners Dilemma with Reinforcement Learning”. Course Project. Stanford University. URL: http://web.stanford.edu/class/psych209/Readings/2017ProjectExamples/wangkeven_17581_1628229_psych209_paper.pdf.

Wang, Weixun et al. (2018). “Towards Cooperation in Sequential Prisoner’s Dilemmas: a Deep Multiagent Reinforcement Learning Approach”. In: *arXiv preprint arXiv:1803.00162*. URL: https://arxiv.org/abs/1803.00162.

Watkins, Christopher John Cornish Hellaby (1989). “Learning from Delayed Rewards”. Ph.D. Thesis. King’s College. URL: https://www.researchgate.net/publication/33784417_Learning_From_Delayed_Rewards.

Watkins, Christopher John Cornish Hellaby and Peter Dayan (1992). “Q-learning”. In: *Machine Learning* 8.3, pp. 279–292. DOI: 10.1007/BF00992698.

Wiering, Marco A. (2005). “QV(λ)-learning: A New On-policy Reinforcement Learning Algorithm”. In: *Proceedings of the 7th European Workshop on Reinforcement Learning*, pp. 17–18. URL: https://dspace.library.uu.nl/handle/1874/20276.

Wu, Jianzhong and Robert Axelrod (1995). “How to Cope with Noise in the Iterated Prisoner’s Dilemma”. In: *Journal of Conflict resolution* 39.1, pp. 183–189. URL: https://www.jstor.org/stable/174327.

xgboost.ai (2019). *Scalable and Flexible Gradient Boosting.* Version 0.71.2. URL: https://xgboost.ai/ (visited on 2019-10-07).

Xue, Lei et al. (2017). “An Adaptive Strategy via Reinforcement Learning for the Prisoner’s Dilemma Game”. In: *IEEE/CAA Journal of Automatica Sinica* 5.1, pp. 301–310. DOI: 10.1109/JAS.2017.7510466.

Zhang, Chongsheng et al. (2019). “On Incremental Learning for Gradient Boosting Decision Trees”. In: *Neural Processing Letters*, pp. 1–31. DOI: 10.1007/s11063-019-09999-3.

Zhang, Jianlei et al. (2011). “Resolution of the Stochastic Strategy Spatial Prisoner’s Dilemma by Means of Particle Swarm Optimization”. In: *PloS one* 6.7, e21787. DOI: 10.1371/journal.pone.0021787.

Appendices

A. General Features of Successful Strategies

There exists no single perfect strategy, as the performance of a strategy strongly depends on its opponents. Against very simple strategies it can be feasible to construct a best answer, a candidate for a best response, by hand. Against complicated strategies one can use machine learning approaches to construct best answers, for example the algorithm presented in this article. These approaches against specific strategies are however not suited to construct a strategy which has to hold its own against a myriad of unknown opponents.
Fortunately, there exist some basic ideas which can be extracted from the most successful strategies. Here, the underlying assumption is, that the pool of opponents shows a wide array of different behaviors: Some opponents might be very erratic, some might be very friendly and exploitable, some might be overly aggressive and other might be build with the same ideas in mind as will be discussed in the following paragraphs.
Siebrasse (2006) describes his strategy *the.prince* the following way:

1. Win-Stay, Lose-Shift
    a) Never give a sucker an even break
    b) If you can't beat 'em, join 'em
2. Punishment proportionate to the crime
    a) Crime doesn't pay
    b) Forgive a wrongdoer who has paid his debts

While intended to highlight the specific features of *the.prince*, these basic ideas may be used to explain desired properties of good strategies in general:

1. *Win-Stay, Lose-Shift*
    a) *Never give a sucker an even break*
    A good strategy aims to maximize its own rewards and does not want to waste potential. Paired with a *sucker*, i.e. an exploitable strategy which does not (sufficiently) retaliate, it is beneficial to act aggressively and exploit as much as possible. When constructing a best answer by hand, this is the recommended way to go: Find a pattern which allows the beast answer to defection as much as possible without triggering retaliations.
    Confronted with an unknown pool it is not obvious how aggressive one might want a good strategy to act. Most historical good strategies err on the side of being too nice as it is not obvious how high the percentage of extremely harsh retaliators, as for example *grim.trigger*, might be. Probing whether the opponent is a sucker by defecting a few times or starting with a defection is thus not recommended (Axelrod, 1984, p. 202). Several empirically successful strategies work around this by not being

the first one to defect, but examining closely how the opponent reacts when confronted with a defection due to an observation error.

b) *If you can't beat 'em, join 'em*
If it is not possible to exploit the opponent it has to be guaranteed, that one is not exploited oneself, as this would result in a bad tournament standing. One way to achieve this is to try to match the rewards generated by the other strategy.
Two simple ways to do so are used by strategies of the *tit.for.tat* family and the *net.nice* family. *tit.for.tat* copies the move of the other strategy, thereby ensuring a certain kind of fairness and automatic retaliation. *net.nice* aims to keep the difference between observed defections between itself and the opponent within a small range and defects if the opponent defected (a certain number of times) more than itself. Given that the other strategy is reasonably sophisticated, it might deduce, that a strategy build with *If you can't beat 'em, join 'em* in mind can't be exploited and might thus prefer mutually beneficial cooperation, thus generating comparatively high average reward for both strategies.

In general when having a good setup - e.g. mutual cooperation or a pattern which is able to exploit the opponent - it is a good idea to keep this pattern. Conversely, if the current setup does not work, it might be sensible to change tactics.

2. *Punishment proportionate to the crime*

a) *Crime doesn't pay*
The crime in this case being a defection, a good strategy often times ensures that the expected value of a defection against it is not higher than a cooperation. In other words the optimal action of the opponent strategy against me should be to always cooperate. Combining this with *If you can't beat 'em, join 'em* implies, that the strategy should -ideally- be constructed in a way that it cooperates as often as possible if vetted against itself. Ideally this would mean, that the strategy plays the highest possible Nash equilibrium against itself, i.e. that any deviation from playing a cooperation all the time results in a worse outcome in expectation.
Given the absence of noise and sufficiently high discount factor γ, *grim.trigger* is ideal in the sense that *always.cooperate* is a best answer. As a single defection results in a never ending series of retaliatory defections, *grim.trigger* can be seen as the archetype of a *Crime doesn't pay* strategy.

b) *Forgive a wrongdoer who has paid his debts*
As soon as noise is introduced to the game, or the other strategies do not know that they are playing against *grim.trigger*, which might lead them to test the waters by defecting, *grim.trigger* answers too harshly. Given a reasonable high probability err_D of a cooperation wrongfully perceived as a

defection in combination with a sufficiently high continuation probability γ, *grim.trigger* shows a similar behavior as *always.defect*, as only the first few periods are played cooperative. Consequently a lot of potential cooperation is wasted and the strategy should place rather badly in a tournament with sufficiently many strategies willing to cooperate.
The idea behind *Forgive a wrongdoer who has paid his debts* is that while there is a need to retaliate when faced with an aggressive strategy, it is unwise to discard the potential for future cooperation. In this sense it is recommended to defect only the necessary amount.

Punishment proportionate to the crime aims to strike the right balance between having a strong enough retaliatory reaction against aggressive strategies so that defections are deterred while still projecting the image of a cooperation friendly strategy. The optimal way to do so depends on the opponent, as a retaliatory action has to be perceived as one which depends on the probability of an detection error and the expected length of the game.
Generally speaking an increasing possibility of wrongfully perceiving a cooperation as a defection favors more lenient strategies. This can be exemplified with the strategy *tit.for.tat*, which always copies the perceived move of the opponent and starts in the first period with a cooperation: Assume there are no detection errors and *tit.for.tat* plays against itself. Both strategies start with a cooperation, will not deviate and generate the maximum joined reward. However, introducing the possibility that a cooperation might be perceived as a defection changes this picture. As soon as one of the two strategies wrongfully detects a defection, it retaliates with a defection. The other strategy answers likewise and both strategies are thus stuck in a loop of never ending defections. With a higher probability of this defection error this effect happens at an earlier point in time and thus reduces the expected average reward. Building in some leniency mitigates this problem and prevents this defection loop. Given a higher probability for a wrongfully observed defection the number of observed defections increases and thus it is necessary to increase the leniency if one wants to maintain a steady cooperation.

Most strategies, which are presented as successful in tournaments in Section 2.1.2 incorporate one or all of those aphorisms. The optimal balance between those ideas however depends strongly on the pool of opponents and the given parameters of the game. In the absence of noise against a pool which consists of only *tit.for.tat*-like strategies and *always.coop*-strategies, it might be prudent to use a defection early on to separate those two groups and play accordingly given the gained information. If there are very harsh retaliators in the pool, it might be necessary to forgo the potential of exploiting the very nice strategies to limit retaliatory measures of the other ones. Given noise, the same situation holds still but the effects are less pronounced due to the stochastic nature of the game. The main difference is, that one can cooperate in the beginning and still utilize random observation errors to gauge the strongness of retailiatory measures.

B. Non-Observability of States

As a general assumption throughout this article it is assumed that the agent is perfectly able to observe the state as defined by Section 2.2. This is not necessarily the case in a game with private information, where some information is only known to the opponent. Instead, the agent has to rely on publicly available information and possible private information of its own. The set of all possible information available to the agent at this point in time is called the *observation*.

Practically speaking there might be a direct mapping between some or all of the observations and states. In an extreme case with an opponent who does not react to inputs (e.g. *random.action* or *always.cooperate*) and assuming that the environment does not incorporate the history as well (as per the rules of the IPD game of Section 2.1.1), only one state exists. On can therefore map all observations to this single state and only has to determine which action is optimal. Repeating this optimal action regardless of the history determines the best policy.

A similar argumentation can be made for the class of strategies, which only rely on public information. Here one can infer all relevant states based on the observations and develop an optimal policy. However, as soon as the opponent takes information into account which is not public, the best policy given full information might differ from the best policy based on public information. In fact, it is not possible to determine the correct state S_t.

Some ideas exist on how to deal with such missing information. One popular approach is using a so called *Partially Observable Markov Decision Process* (POMDP) (Sutton and Barto, 2018, p. 384). Here the basic idea is, that there exists an unobserved hidden Markov model with a true, but hidden state X_t. Which observations O_t are made each period depend on this hidden state and thus allow the agent to infer the distribution of possible states X_t. Depending on which information about the model is known, Bayesian updating can for example be used to calculate this distribution. A survey of some popular approaches on how to deal with POMDPs in particular can be found with Monahan (1982).

An alternative to POMDPs are so called *Predictive State Representations* (Littman and Sutton, 2002) with the basic idea to use tests to predict future states and actions and update a vector which tracks the probability distribution over those tests. This vector is then used as the Markov state.

Our approach was to use neither of those methods but instead use the observations O_t directly as states S_t and filter them through the function approximator. This was done for the following reasons:

1. Due to the complexity of the problem the theoretically optimal dynamic programming approaches of Section 2.3.1 are not viable and function approximators have to be used in any case. The usage of approximating functions is very similar to having partially observable states and is a viable way to circumvent some problems regarding non-observability (Sutton and Barto, 2018, p. 161). An approximation function per design flattens the dimension of the parameter

space and loses information. Even with perfect public monitoring the agent therefore does not see the full state of the state, but only the representation as shown by the function approximator. Some of the limitations of the partial observation can therefore be transferred to the considerations in regard to the function approximation which are further discussed in Section 3.7 and Section 3.9.

2. Adding another layer of complexity increases runtime due to computational complexity. At least in the case of POMDPs Sutton and Barto (2018, p. 385) write that they "scale poorly". Even using alternative versions of keeping track or building hypotheses are necessarily very memory intensive in our case. With other problems one might have access to a considerable amount of data to infer one of a few states, but with our specific problem we have to assume in every single period that there has been an observation error. Tracking all those different routes increases complexity exponentially. Nevertheless, there certainly exist situations, where it might be prudent to use more sophisticated models without increasing complexity too much:
   Assume that a strategy decides before its first action, whether it wants to play with policy A or policy B. Having a way to infer which of those two policies is played in an explicit way might greatly improve the algorithm to find a best answer. A fundamentally comparable situation takes place if the agents plays against a pool of known opponents. Using an explicit model to determine the specific opponent might improve tournament performance. As this was not the focus of this article and I wanted to present general solutions, I refrained from implementing this idea.

3. Some considerations have to be made to keep the Markov property. If the agent for example only considers the very last period, but the opponent uses the last two periods to calculate its action, the best possible achievable policy might be sub-optimal. In such a situation it no longer holds, that our best possible policy has to be deterministic. Limiting the agent to deterministic strategies can therefore result in it not even converging to the best possible one-period strategy.
   An example for this effect in a gridworld setting can be found with Silver (2015, Lecture 7, p. 7). A similar example can be constructed in our case. As a basic intuition one should think about an opponent strategy which wants to see one of two specific patterns and rewards the strategy strongly if it observes the pattern. Which pattern is rewarded is chosen by chance at the beginning of the game. The perfect deterministic one-period policy might be able to always play the pattern with the higher expected value, but loses out on all games with the other pattern, given that the patterns are reasonable different. A mixed policy on the other hand might be able to find a distribution which has a high chance to activate both patterns of the other strategy, thus reaping the reward in both cases.
   Thanks to correct representation and encoding (see Section 3.7) this challenge

is solvable. It has to be guaranteed, that it is not possible to loop through observed states, so that the same sub-optimal action is repeated.
In the case of the IPD game the right move has to be made in the right situation and each state might only be visited once per episode, for example if the strategy of the opponent depends on the period counter. Using all available information, i.e. the public history, the period counter and the private history of the game, the agent might not be able to necessarily play the best theoretically possible policy, but is able to play the best possible policy using all available information as the Markov property is maintained. Even though the opponent might act differently based on his private information, this becomes a feature of the environment from the view of the agent. Using sufficiently many episodes the actions of the opponent become effectively a stochastic part of the game. This stochastic element is eliminated by the function approximation which tries to estimate the expected utility.

To summarize: By using function approximation and correct encoding of states, e.g. a period counter, private information does not invalidate the idea of Markov states but rather changes the effective environment the agent experiences. As this is the environment we want to optimize, it is possible to use the same kinds of algorithms as with perfect information but a stronger stochastic component.

C. General Structure of Algorithm

The main algorithm follows the following structure:

```
Define game information and save it into game.object # Step 1
Define parameters of algorithm and save them into algo.par # Step 2
Define parameters of function approximator and save them into model.par # Step 3
Initialize and instantiate function approximators (evaluator) and all changeable variables (algo.var). # Step 4
If relevant: Generate memory according to memory initialization # Step 5
# At this point the Training function is called which executes the following steps
If relevant: Update function approximators # Step 5
For $i$=0 to # of blocks:
    Play a specified number of episodes with and without exploration. # Step 6
    Update the memory based on new episodes # Step 7
    Observe and save aggregated rewards and exploration influences of the different play modes. # Step 8
    Delete some part of the memory, if memory stack is full. # Step 9
    If relevant: Select new best model. # Step 10
    Replay-step: Train function approximator based on memory and experience replay. # Step 11
    Recalculate exploration variables. # Step 12
# Here the Training function ends
Evaluate model and save results # Step 13
```

In more detail, these steps solve the following objectives:

1. Define game information and save it into *game.object*
   The package *ReinforcementLearningwithR* (RLR) (Kies, 2019) can easily be extended to a wide array of different games, given that they are turn-based and have a finite number of executable actions. *game.object* provides all relevant information and functions regarding the game. Examples include the number of actions, the number of periods (which may be stochastic), a function which determines how a state is changed given an action and a mapping function from the internal state to the state as viewed from the agent. This view might

be obfuscated due to noise and/or enhanced with helpful calculated statistics. More information regarding the importance of the encoding can be found in Section 3.7.

2. Define parameters of algorithm and save them into *algo.par*
   Here, all parameters which do not change within a single execution of the algorithm are defined. Notably, this parameter list determines which of the developed features (in contrast to classic Q-Learning) are activated and which properties they possess. Examples include the exact nature on how to explore the environment, the parameters regarding *Q-Switching*, *Experience Replay* including memory usage and how many episodes are played.

3. Define parameters of function approximator and save them into *model.par*
   The R-package Kies (2019) allows for flexible usage of several function approximators and may easily be extended with additional ones. In this step the function approximator, for example a neural network or gradient boosting, as well as their specific parameters are specified. This includes but is not limited to the number of neurons and layers, type of activation function and number of training epochs for the neural network and number and complexity of trees in the case of gradient boosting. More information regarding the effects and relevance of the function approximator can be found in Section 3.9.

4. Initialize and instantiate function approximators (*evaluator*) and all changeable variables
   In this technical step instances of the function approximators are initialized so that for all encountered states a meaningful default value can be delivered. Additionally the (empty) memory is built and variables which change within a run, e.g. the ε in the case of ε-*exploration*, are set to their starting values.

5. Generate memory according to memory initialization and update function approximators
   In this optional step the feature *Memory Initialization* is executed. Here, a data base is initialized as a memory foundation. This data base is generated by playing the game according to specified rules. Examples for such rules are taking random actions or simply repeating the same action. As the exploitability of the to be analyzed strategy is of interest, the studied memory initialization method within this article is self-play. With self-play, the strategy is set against itself to built the memory data base. Why this might significantly improve performance is discussed in Section 3.11.

6. Play a specified number of episodes with and without exploration.
   The number of blocks the algorithm is cycling through defines the number of replays, i.e. the number of times the function approximator is updated. Depending on the complexity of the function approximator it might be time-consuming to update it relative to generating more data by playing more episodes. In this case it might be beneficial to increase the number of episodes played by the same function approximator before retraining. Increasing this

number corresponds to increasing the size of each block. Another benefit of increasing the block size is that within a stochastic environment the function approximator can more easily infer the actual expected values of a given policy if several episodes generated by the same policy exist. Further discussions on why it is beneficial to train the function approximator on several episodes can be found in Section 3.8.
In line with the function approximator, all other variable parameters of the algorithm are only updated on a per block basis. Examples include the exploration parameter ε given ε-exploration.
Within a block, episodes with the following features can be played:

a) Trying to play as well as possible using the **current** function approximator, i.e. the most recently trained.

b) Trying to play as well as possible using the **best** function approximator of the past. Why this might prove beneficial in addition to the most current one is discussed in Section 3.10.

c) Exploring, using only *ε-exploration*.

d) Exploring, using only *noisy actions*.

e) Exploring, using only *maximizing surprise*.

f) Exploring, using only *minimizing familiarity*.

g) Exploring, using a to be specified combination of *noisy actions* and *ε-exploration*.

h) Exploring, using a to be specified combination of all 4 exploration strategies, named *Multi-Exploration* (see Section 3.5). This combination might exclude certain exploration strategies, if one so wishes.

Multi-exploration is able to emulate all other exploration strategies listed here. They are listed separately however, as one can decide to run them separately within the same block. This way one might for example run a single episode using only *noisy actions*, another single episode using only *ε-exploration* and five episodes playing a combination of both exploration variations. The function approximator will then be retrained on this pool of experiences.
To use the feature of *Exploration-Pathing* (see Section 3.6), it is mandatory to have at least one *best effort* episode per block. In this context a best effort episode is defined as an episode which has either been generated by the current or the best function approximator. This best effort episode is used to generate a benchmark of the current ability of the function approximator to generate rewards. Based on this benchmark the exploration parameters are adjusted. Providing at least one episode for exploration is not technically mandatory but necessary to generate learning behavior. The ability of the algorithm to learn depends strongly on differently played actions and correspondingly different rewards. A certain minimum of exploration is therefore necessary.

Within this step various useful statistics are saved for later analysis. One such calculated statistic is the estimated aggregated sub-optimality of the various exploration strategies within *Multi-Exploration*. This allows the algorithm to adjust them in subsequent blocks in the direction of their respective targets.

7. Update the memory based on new episodes
   In this step the episodes which were generated in the step before are integrated into the memory storage. For each experienced period the values of the current state, the next state, the reward and whether or not it was a closing state are saved. If *Q-Switching* (see Section 3.4) is used additionally the Q_{hist}-Value is added. This value measures the discounted rewards of the periods until the end of the covered episode.
   If count-based exploration in the feature space according to Martin et al. (2017) is activated, the age of the experience is noted and R_ϕ is updated as well. See Martin et al. (2017) for more detail.

8. Observe and save aggregated rewards and exploration influences of the different play modes.
   In this technical step detailed information on each block are saved. This includes the aggregated reward of each episode, the aggregated reward of the mean of all episodes within the block corresponding to the same exploration family and measures regarding the sub-optimalities due to exploration.

9. Delete some part of the memory, if memory stack is full.
   The following considerations apply:

   a) With an increasing number of played episodes, one might exceed the limits of the hardware. Given the current implementation in Kies (2019) the most relevant ceiling is the working memory which limits the size of R-objects.

   b) Q-Learning, and in extension our algorithm, in theory converges against a correct understanding of the best action choice for every single state of the game. For practical purposes however, we are more interested in states which have a realistic chance to be encountered by high performing policies. With an increasing number of blocks the policies should converge towards the best policy. More recent strategies are thus more valuable.
   This does not imply, that very old experiences are useless. If we use a function approximator which is not incremental but has to be built from scratch (e.g. gradient boosting) excluding old experiences may lead to repeating the same mistakes periodically.

   Using old memories can increase the performance in the case of incremental function approximators (see Section 3.11). Given function approximators which are newly generated each block, using old information is essential. Due to the aforementioned reasons however, we limit the memory space by deleting a specified percentage out of the experienced state-transitions once the memory overflows. The algorithm uses a uniform sampling to decide which states have to be deleted. This procedure aims to mitigate the problem that very unfavorable states might be visited periodically.

10. Select new best model.
    In this optional step the modification *Model Persistence* (see Section 3.10) is executed. It is checked whether the model defined by the current function approximator is plausibly better than the assumed best model. If the current model shows better performance, the best model is replaced with the current model.

11. Replay-step: Train function approximator based on memory and experience replay.
    In this step first a part of the memory is selected. Several variants on how to select this part are provided by Kies (2019). One might for example choose to only select all data generated since the last update of the function approximator. Alternatively one can chose random state-transitions out of all episodes in the memory storage. It is also possible to combine both of those approaches. This selection choice is discussed in more detail in Section 3.8.
    Based on this selection, for each experiences state-action pair the Q-Value (either based on Q-Learning, *Q-Switching* or Monte Carlo learning) is calculated. This calculation assumes that Q-Values of the next states are correctly approximated by the function approximator. In the supervised learning problem used to update the function approximator those Q-Values are the response variable. The concatenation of the encoded state and the chosen action are used as the explanatory variables. The function approximator is then updated based on this data.

12. Recalculate exploration variables.
    Given only standard Q-Learning in this step the exploration variable ε is recalculated. Analogously the current block number and data from the most recent and older blocks is used to calculate the current set of exploration variables according to *Exploration Pathing*. More information regarding these calculations is presented in Section 3.6.

13. Evaluate model and save results.
    This last step takes place after the policy has been developed by the reinforcement algorithm. The most current function approximator defines the final model. This model is evaluated using the *StratTourn*-package. Optionally one can also additionally execute an evaluation for each single block. This generates more precise data about the learning process of the algorithm.
    Finally, the function approximator as well as the memory stack and various additional data useful to analyze the algorithm are saved.

D. Comparison of Boltzmann distribution to Noisy Actions

In Section 3.5.2 we discussed the similarities between the variants *noisy actions* and the more traditionally used *Boltzmann exploration*.

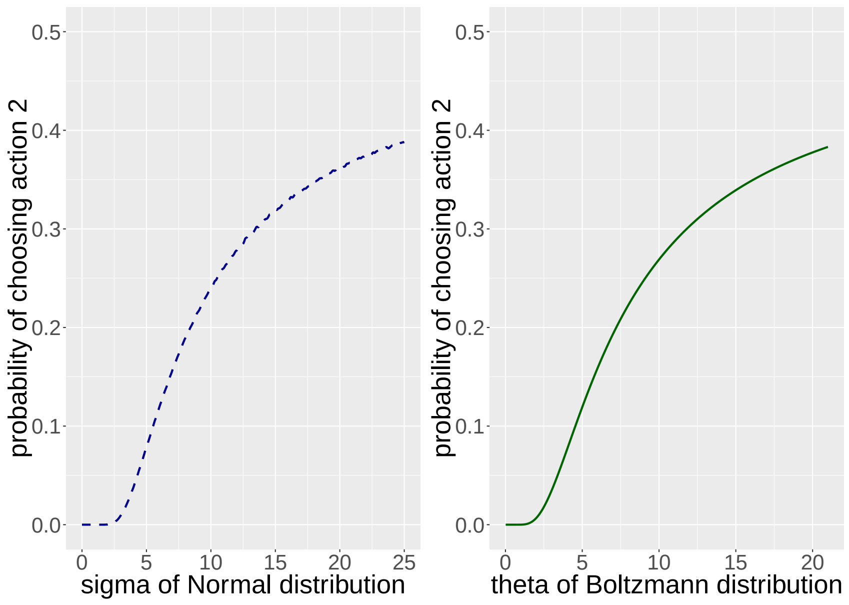

Figure 33: Comparison of *noisy actions* (left) to *Boltzmann exploration* (right) given different values of σ and θ, where action 1 is preferred over action 2. The chart with *Boltzmann exploration* uses calculated values, while the chart with *noisy actions* has been numerically generated and averaged.

In Figure 33 one can see, that both, *noisy actions* and *Boltzmann exploration* show a very similar behavior in regard to their parameter.
Using *noisy actions* gives a small speed advantage given our implementation in R:

```
> benchmark("noisy"={
+   rands <- rnorm(length(a),mean=0, sd=2)
+   act.values <- a + rands
+   res <- which.is.max(act.values)
+ },
+ "boltz"={
+   P.a <- sapply(a,FUN=function(x){exp(x/20)/sum(exp(a/20))})
+   res <- sample(length(a),size=1,prob=P.a)
+ }, replications=1000000
+ )
    test replications elapsed relative user.self sys.self
2  boltz      1000000   59.63    4.411     58.39     0.06
1  noisy      1000000   13.52    1.000     13.23     0.00
```

E. Compare.Exploration

In Section 3.5.5 and Section 3.5.6 we discuss the strategy *compare.exploration*, which is shown here.

Pseudo-Code

```
If First period:
    StateIsNice ← TRUE
    Cooperate
If Period number $\leq 10$ AND defection of opponent in the previous
period observed:
    StateIsNice ← FALSE
    Defect
If StateIsNice == TRUE
    If Current period is one of the numbers 11, 15, 17, 23, 26, 28, 30,
    34, 37, 38, 40, 51, 53, 56, 58
        Cooperate # Independent of opponent
    Else
        If Observe cooperation of opponent in the previous period
            Cooperate
        Else
            Defect
Else # Defection of opponent within the first 9 periods observed
    Defect
```

F. Testing Pool

Here, the specific strategies which are the basis for the analysis as described in Section 3.3 as well as their proposed counter-strategies.

For each strategy 7 types of scores are given

1. Best Answer Score (60 periods) - Measures the average performance $\bar{R}_T^{\text{best}}$ of the proposed counter-strategy against the strategy with a maximum number of periods of $T = 60$.
2. Algorithm Best Score (60 periods) - Measures the average performance of the final best policy generated by the algorithm according to the performance test of Section 3.2 against the strategy with a maximum number of periods of $T = 60$. The best of both function approximators, gradient boosting and recurrent neural network is displayed.
3. Q-Learning Score (60 periods) - Measures the average performance of the strategy against the basic Q-Learning algorithm with a maximum number of periods of $T = 60$.
4. Score against itself (60 periods) - Measures the average performance of the strategy against itself with a maximum number of periods of $T = 60$.
5. Best Answer Score (no period limit) - Measures the average performance $\bar{R}_T^{\text{best}}$ of the proposed counter-strategy against the strategy without a period limit.
6. RNN Best Score (no period limit)- Measures the average performance recurrent neural network run of the final best policy generated by the algorithm according to the performance test of Section 3.2 against the strategy without a period limit.
7. Score against itself (no period limit) - Measures the average performance of the strategy against itself without a maximum number of periods.

Each score has been calculated by running a *StratTourn*-tournament (Kranz and Kies, 2019) over 1000 games after setting the random seed to 234567.

F.1. strat.a

Basic facts

Best Answer Score (60 periods): 1.321
Algorithm Best Score (60 periods): 1.317
Q-Learning Score (60 periods): 1.291
Score against itself (60 periods): 0.920
Best Answer Score (no period limit): 1.325
RNN Best Score (no period limit): 1.319
Score against itself (no period limit): 0.919
Short Summary: Starts with a cooperation and counts the difference between cooperations and defections of the opponent in each period. If the number of observed cooperations is greater than the number of observed defections, cooperate.

Pseudo-Code

```
If First period:
    CooperationDiff ← 0
    Cooperate
If Observe cooperation of opponent in previous period
    CooperationDiff ← CooperationDiff + 1
Else
    CooperationDiff ← CooperationDiff - 1
If CooperationDiff $\geq 0$
    Cooperate
Else
    Defect
```

Best Answer

```
If First period:
    CooperationDiff ← 0
    Cooperate
If Observe cooperation of myself in previous period
    CooperationDiff ← CooperationDiff + 1
Else
    CooperationDiff ← CooperationDiff - 1
If CooperationDiff $\geq$ 2
    Defect
Else
    Cooperate
```

This strategy effectively calculates the state of mind of *strat.a* and defects if there is a sufficient margin of observed self-cooperations.

F.2. strat.b

Basic facts

Best Answer Score (60 periods): 0.747
Algorithm Best Score (60 periods): 0.747
Q-Learning Score (60 periods): 0.684
Score against itself (60 periods): 0.495
Best Answer Score (no period limit): 0.746
RNN Best Score (no period limit): 0.703
Score against itself (no period limit): 0.487
Short Summary: This strategy is a variant of *generous.tit.for.tat.* It starts with a cooperation. After each observed cooperation of the opponent the strategy answers with a cooperation. Every second defection is answered with a guaranteed defection, every other defection there is a 15% chance of there being a cooperation.

Pseudo-Code

```
If First period:
    defection.counter ← 0
    Cooperate
If Observation of Cooperation of Opponent
    Cooperate
Else
    defection.counter ← defection.counter+1
    If defection.counter == 3
        defection.counter ← 1
        Defect
    Else
        Cooperate (with 15%) or Defect (with 85%)
```

Best Answer

This strategy has no opening to sneak in defections. Even in the case of there being less than three defections, defection is not a fruitful action. It is thus optimal to play *always.coop*.

F.3. strat.c

Basic facts

Best Answer Score (60 periods): 1.156
Algorithm Best Score (60 periods): 1.1
Q-Learning Score (60 periods): 0.941
Score against itself (60 periods): 0.979
Best Answer Score (no period limit): 1.159
RNN Best Score (no period limit): 1.097
Score against itself (no period limit): 0.98
Short Summary: In contrast to the other strategies, this strategy does not try to induce cooperation. The goal of this strategy is to play perfect cooperation if vetted against another version of itself, but defect otherwise. It uses Bayesian updating to calculate the probability of playing against itself instead of a counterfactual strategy, which cooperates with a certain probability.

Pseudo-Code

```
If period 1:
    # Initialize parameters and variables
    $P_i \leftarrow 0.5$ # A-priori probability of being me. Will be updated based on
    my observed actions about me to accurately reflect the state of mind of
    my opponent (under the assumption, that the opponent plays the same
    strategy
    $P_a \leftarrow 0.5$ # A-priori probability of the opponent playing the same strategy
    as I am. Will be updated based on my observed actions about his actions.
    threshold $\leftarrow 0.25$ # Cooperate, if I assume that I am playing with at
    least this probability against myself.
    Fratio $\leftarrow 0.4$ # Measuring the probability of cooperation against myself
    of the counter factual opponent strategy.
    $P_D \leftarrow 0.25$ # Probability to erroneously perceive a D due to an observa-
    tion error. This parameter is set to the value of the game.
    $P_C \leftarrow 0$ # Probability to erroneously perceive a C due to an observation
    error. This parameter is set to the value of the game.
Else # Update variables
    If $P_i \geq$ threshold # Given I play against myself the opponent assumes
    (correctly) that I am me
        If I observe cooperation of opponent in previous period # What was
        expected
            $P_a \leftarrow \frac{P_a\cdot(1-P_D)}{P_a\cdot(1-P_D)+(1-P_a)\cdot(P_C+\text{Fratio}\cdot(1-P_D-P_C))}$
        Else Observation error or not playing against myself
            $P_a \leftarrow \frac{P_a\cdot(P_D)}{P_a\cdot P_D+(1-P_a)\cdot(P_D+(1-\text{Fratio})\cdot(1-P_D-P_C))}$
    Else The opponent strategy should assume that I am not me
        If I observe cooperation of opponent in previous period # Other
        strategy or observation error [May never happen with $P_C == 0$]
            $P_a \leftarrow \frac{P_a\cdot P_C}{P_a\cdot P_C+(1-P_a)\cdot(P_C+(1-\text{Fratio})\cdot(1-P_D-P_C))}$
        Else What was expected
            $P_a \leftarrow \frac{P_a\cdot(1-P_C)}{P_a\cdot(1-P_C)+(1-P_a)\cdot(P_D+\text{Fratio}\cdot(1-P_D-P_C))}$
Update $P_i$ analogously to $P_a$ from the view of the opponent.
If $P_a \geq$ threshold I assume I play against myself
    Cooperate
Else
    Defect
```

Best Answer

```
If period 1:
    Use the same parameters and variables as strat.c and initialize them
    identically
Else
    Update $P_a$ and $P_i$ identical to strat.c
If $P_a > 0.9999999$: $P_a \leftarrow 0.99$
If $P_i \geq 0.85$: Defect
If $P_a \geq$ threshold: Cooperate
Defect
```

This strategy copies the internal mechanism of *strat.c* to effectively pose as this strategy. If *strat.c* assumes, that it is playing against itself, a defection is sneaked in, otherwise the same behavior as *strat.c* is shown. The limitation of P_a does not have a game-theoretic motivation, but rather a numerical one.

F.4. strat.d

Basic facts

Best Answer Score (60 periods): 1.317
Algorithm Best Score (60 periods): 1.312
Q-Learning Score (60 periods): 1.114
Score against itself (60 periods): 0.98
Best Answer Score (no period limit): 1.319
RNN Best Score (no period limit): 1.315
Score against itself (no period limit): 0.98
Short Summary: After a starting phase of five periods this strategy rewards consecutive cooperations with cooperation, punishes consecutive defections with a defection and cooperates otherwise, given a sufficiently good ratio of observed cooperations of 50%.

Pseudo-Code

```
If period 1-5: Cooperate
If Observed consecutive defects in the last 3 periods: Defect
If Observed consecutive cooperations in the last 2 periods: Cooperate
If Observed actions of opponent are more than 50% cooperations: Cooperate
Else Defect
```

Best Answer

```
If period 1: Cooperate
If Number of observed cooperations of myself $\leq$ Number of observed defections
of myself +2: Cooperate
Else Defect
```

This strategy aims to have at least 50% cooperations with some safety margin and thus nearly always triggers the “50% cooperation” condition, except in the case of bad luck with the observation errors.

F.5. strat.e

Basic facts

Best Answer Score (60 periods): 0.743
Algorithm Best Score (60 periods): 0.743
Q-Learning Score (60 periods): 0.395
Score against itself (60 periods): 0.672
Best Answer Score (no period limit): 0.708
RNN Best Score (no period limit): 0.696
Score against itself (no period limit): 0.638
Short Summary: This strategy resembles *grim.trigger*, but the trigger are in total 4 observed defections of the opponent.

Pseudo-Code

```
If period 1: Cooperate
If 4 or more observed defections of opponent in total:
    Cooperate
Else
    Defect
```

Best Answer

```
If period 1: Cooperate
If Opponent has not yet observed 4 total defections of myself:
    Cooperate
Else
    Defect
```

This strategy cooperates until the trigger condition of *strat.e* is met and defects afterwards, as *strat.e* can't be appeased.

F.6. strat.f

Basic facts

Best Answer Score (60 periods): 1.503
Algorithm Best Score (60 periods): 1.499
Q-Learning Score (60 periods): 1.358
Score against itself (60 periods): 0.833
Best Answer Score (no period limit): 1.503
RNN Best Score (no period limit): 1.497
Score against itself (no period limit): 0.831
Short Summary: This strategy cooperates in the first period and has two different states: "skeptic" and "content". It starts "skeptic" and switches to "content" as soon as a cooperation of the opponent is observed. As long as no cooperation is observed, the strategy defects. Once switched to "content", the strategy cooperates no matter what for the following three actions.

Pseudo-Code

```
If period 1:
    Switch to status "Skeptic"
    Cooperate
If Skeptic:
    If Cooperation of opponent in last period is observed:
        Switch to "Content"
    Else
        Defect
If "Content" within both last two periods: Switch to "skeptic" # Note that
this condition and the following parts of the code are only executed if the
current status is "Content"
Cooperate
```

Best Answer

```
If period 1:
    count ← 2
    Cooperate
If count = 0:
    count ← 2
    Cooperate
If count = 1:
    count ← 0
    Defect
If count = 2:
    If Opponent observed a defection from me in last period:
        count ← 2
        Cooperate
    Else
        count ← 1
        Defect
```

The parameter *count* keeps track of the opponents internal state. If the opponent is skeptic, one wants cooperate and defect otherwise.

F.7. strat.g

Basic facts

Best Answer Score (60 periods): 1.328
Algorithm Best Score (60 periods): 1.328
Q-Learning Score (60 periods): 1.321
Score against itself (60 periods): 0.971
Best Answer Score (no period limit): 1.327
RNN Best Score (no period limit): 1.327
Score against itself (no period limit): 0.969
Short Summary: This is a *tit.for.tat* like strategy, which forgives a single "misstep", i.e. needs two defections in a row before retaliation. Additionally after having observed five defections of the opponent in a row a single peace offering in form of a cooperation is offered.

Pseudo-Code

```
If period 1:
    reset.state ← TRUE
    Cooperate
If Observe Cooperation of opponent:
    reset.state ← TRUE
    Cooperate
If Observe Defection of opponent but reset.state is TRUE:
    reset.state ← FALSE
    Cooperate
If Observe fifth Defection of opponent in a row:
    reset.state ← TRUE
    Cooperate
Else
    reset.state ← FALSE
    Defect
```

Best Answer

```
If period 1:
    count ← 0
    Defect
If My observed action in the previous period has been a Defection
  AND count < 5:
    count ← count+1
    Cooperate
Else:
    count ← 0
    Defect
```

This strategy starts with a defection and then tries to alternate between observed defection and cooperation. In the off chance, that due to observation errors five defections have been observed by the opponent, a real defection is played to utilize the peace offering of *strat.g*, as it resets its memory anyway.

F.8. strat.h

Basic facts

Best Answer Score (60 periods): 1.083
Algorithm Best Score (60 periods): 1.054
Q-Learning Score (60 periods): 0.917
Score against itself (60 periods): 0.962
Best Answer Score (no period limit): 1.046
RNN Best Score (no period limit): 1.018
Score against itself (no period limit): 0.941
Short Summary: This strategy starts by cooperating and generally continues the cooperation unless the opponent defects three times in a row (based on the observation of *strat.h*. After at least 10 defections of the opponent in total have been observed, the strategy is triggered and defects until the game ends. Effectively this strategy is thus a more robust variant of a combination of *tit.for.tat* and *grim.trigger*.

Pseudo-Code

```
If period 1: Cooperate
If Observed opponent to defect the last three actions in a row OR Number of
observed defections of opponent is at least 10: Defect
Else: Cooperate
```

Best Answer

```
If period 1: Defect
If Number of observed defections of myself greater or equal to 10: Defect
If Observed own action to defect the last period OR Number of observed
defections of myself is between 5 and 9: Cooperate
Else: Defect
```

This strategy starts by defecting and aims to alternate observed defections and cooperations of itself. After raking up 5 defections, it switches to cooperation to avoid to hit the 10 defection limit of *strat.h*. If due to observation errors the defection limit is hit anyway, the strategy protects itself from the trigger of its opponent by defecting too.

F.9. strat.i

Basic facts

Best Answer Score (60 periods): 1.321
Algorithm Best Score (60 periods): 1.356
Q-Learning Score (60 periods): 1.320
Score against itself (60 periods): 0.986
Best Answer Score (no period limit): 1.281
RNN Best Score (no period limit): 1.34
Score against itself (no period limit): 0.983
Short Summary: Start with a cooperation and copy the observed move of the opponent in the first period in the second period. Afterwards, cooperate after each cooperation, except if the opponent has earned sufficient mistrust. Mistrust is earned if the opponent defects sufficiently often in a row. What constitutes as "sufficiently" depends on the number of such observed rows; Starting with 3 necessary defects, with each occurrence the number is decreased by one until each defect increases the mistrust parameter. As long as the number of sufficient defects in a row is not reached, defects go unpunished, as the strategy reacts with a cooperation.

Pseudo-Code

```
If First period:
    loss.conf ← 1 # Parameter which measures whether there is an extraor-
    dinary amount of opponent defections.
    con.def ← 0 # Parameter which mostly measures consecutive defections
    of opponent.
    mistrust ← 0 # Parameter measuring “mistrust” towards the opponent
    Cooperate
If Second period:
    If Observe cooperation of opponent in first period
        Cooperate
    Else
        con.def ← 1; mistrust ← 1
        Defect
If Observe cooperation of opponent:
    con.def ← 0; mistrust ← mistrust -1
    If mistrust>2
        Defect
    Else
        Cooperate
Else # Opponent defected
    If mistrust>3
        con.def ← 0
        Defect
    Else
        If con.def > 2 - loss.conf
            loss.conf ← loss.conf+1; con.def ← 0; mistrust ← mistrust+2
            Defect
        Else
            con.def ← con.def+1
            Cooperate
```

Best Answer

```
If First period: Cooperate
If Opponent observes Defection
    Cooperate
Else
    Defect
```

This very simple strategy effectively alternated between cooperation and defection except in the case of an observation error, where again a cooperation is offered.
Doing a more in-depth analysis in regards to Feature Selection using this strategy, one can see, that an encoding with only two periods leads to very fast good results. We find, that the more powerful encoding effectively leads the algorithm to become less robust and thus vulnerable to the random fluctuations due to the observation error. To achieve a comparable result it is thus necessary to have significantly more training data, so that the randomness of the observation error evens out and the algorithm is able to accurately estimate the true expected value even of less often visited states. Using just two periods on the other hands leads to a fast detection of exactly the algorithm as seen above.
Comparing the performance of this strategy against the superior strategy as found by the RNN one can see, that the first 20 periods are played at approximately the same level, but that the ability of the handcrafted anti-strategy to exploit the strategy tethers out. The strategy of the algorithm on the other hand is able to keep up the exploitation, probably manipulating the *mistrust* system.

G. Parameters and Specifics

In this section the parameters are given to generate the corresponding figures and analyses within the main body of the thesis. Note that these may vary between the different figures. As a general rule the parameters have been chosen to allow sufficiently many runs to receive significance even though this often puts a ceiling on the power of the result. Unless noted otherwise the results discussed in the main body of the thesis are expected to generalize.
All parameters refer to the Package *Reinforcement Learning with R* (Kies, 2019). It holds for all specifications, that *game.object* is the name of the game object (default parameters per *Get.Game.Object.PD()*), *algo.par* is the list containing the main parameters of the algorithm (default parameters per *Get.Def.Par.QLearningPersExpPath()*) and *model.par* the list with parameters regarding the function approximator.
Unless noted otherwise, the parameters parameters which are not specified in neither the common parameters nor the specific section are irrelevant for the respective figures.

G.1. Common Parameters

Unless noted otherwise, the parameters as defined here hold true for all figures.

1. General Parameters

    memory.init: none

    blocks: 100

    start.w.training: FALSE

2. game.object

    encoding.state: Main.real

    game.pars$uCC: 1

    game.pars$uCD: -1

    game.pars$uDC: 2

    game.pars$uDD: 0

    game.pars$err.D.prob: 0.15

    game.pars$err.C.prob: 0

    game.pars$delta: 0.95

    game.pars$T: 60

    game.pars$T.max: 60

    game.pars$intermed: 0

game.pars$direct.rewards: TRUE

3. algo.par

   curio.beta: 0

   gamma: 0.95

   a: 0.25

   action.policy: exploration.path

   expl.path.multi.start: 0.8

   expl.path.multi.end: 0.99

   expl.path.multi.decay.type: exponential

   expl.path.multi.best.db: 10

   expl.path.multi.best.disc: 0.9

   expl.path.multi.best.Kp.var: 0.025

   expl.path.multi.best.Ki.var: 0.0025

   expl.path.multi.best.Kd.var: 0.01

   expl.path.multi.best.Kp.shock: 0.0025

   expl.path.multi.best.Ki.shock: 0.00025

   expl.path.multi.best.Kd.shock: 1e-04

   expl.path.multi.best.Kp.db: 10

   expl.path.multi.best.Ki.db: 100

   expl.path.multi.best.Kd.db: 10

   expl.path.multi.best.Kp.disc: 0.9

   expl.path.multi.best.Ki.disc: 0.98

   expl.path.multi.best.Kd.disc: 0.9

   block.curr: 1

   block.best: 0

   block.expl.var: 0

   block.expl.shock: 0

   block.expl.surp: 0

   block.expl.fam: 0

   block.expl.vs: 0

   block.expl.multi: 1

replay.intensive: 1

remove.memory: 0.1

mem.type: game.encoded

batch.size: 6000

max.mem: 12000

force.last: 120

use.rnn: FALSE

only.experienced: TRUE

hybrid.Q: TRUE

Q.Learning: FALSE

MC: FALSE

hybrid.switch: TRUE

hybrid.decay: 0.9

hybrid.Q.apply: always

hybrid.Q.a.MC: 0.25

4. model.par

   name: Gradient.Boosting.XGBoost

   eta: 0.3

   gamma: 0.1

   colsample: 0.95

   subsample: 0.9

   min_child_weight: 1

   single.train: TRUE

   nthread: 6

G.2. Q-Switching

The following parameters have been chosen to generate Figures 16, 17 and 18:

1. Common Parameters

   a) algo.par

      expl.path.multi.start.var: 3

      expl.path.multi.start.shock: 0.2

  - expl.path.multi.start.frac.var: 0.8
  - expl.path.multi.start.frac.shock: 0.2
  - expl.path.multi.start.frac.surp: 0
  - expl.path.multi.start.frac.fam: 0
  - expl.path.multi.end.frac.var: 0.8
  - expl.path.multi.end.frac.shock: 0.2
  - expl.path.multi.end.frac.surp: 0
  - expl.path.multi.end.frac.fam: 0
  - batch.size: 12000
  - force.last: 600

b) model.par

  - nrounds: 10
  - max_depth: 10

2. Varied Parameters

a) Q-Learning

  - algo.par$hybrid.Q: FALSE
  - algo.par$Q.Learning: TRUE
  - algo.par$MC: FALSE
  - algo.par$hybrid.switch: FALSE

b) Q-Switching

  - algo.par$hybrid.Q: TRUE
  - algo.par$Q.Learning: FALSE
  - algo.par$MC: FALSE
  - algo.par$hybrid.switch: TRUE
  - algo.par$hybrid.decay: 0.9
  - algo.par$hybrid.Q.apply: always
  - algo.par$hybrid.Q.a.MC: 0.25

c) Monte-Carlo

  - algo.par$hybrid.Q: FALSE
  - algo.par$Q.Learning: FALSE
  - algo.par$MC: TRUE
  - algo.par$hybrid.switch: FALSE

G.3. Multi-Exploration

The following parameters have been chosen to generate Figure 19, Figure 21 and Figure 22:

1. Common Parameters
   a) algo.par

      expl.path.multi.start.var: 0.05

      expl.path.multi.start.surp: 0.05

      expl.path.multi.start.shock: 0.05

      expl.path.multi.start.fam: 0.05

   b) model.par

      nrounds: 20

      max_depth: 5

   c) model.par.surp with identical parameters to model.par, except

      nrounds: 30

      max_depth: 3

   d) model.par.fam with identical parameters to model.par.surp

2. Varied Parameters
   a) ε-exploration = eps100.noisy0

      algo.par$expl.path.multi.start.frac.var: 0

      algo.par$expl.path.multi.start.frac.shock: 1

      algo.par$expl.path.multi.start.frac.surp: 0

      algo.par$expl.path.multi.start.frac.fam: 0

      algo.par$expl.path.multi.end.frac.var: 0

      algo.par$expl.path.multi.end.frac.shock: 1

      algo.par$expl.path.multi.end.frac.surp: 0

      algo.par$expl.path.multi.end.frac.fam: 0

   b) *noisy actions* = eps0.noisy100

      algo.par$expl.path.multi.start.frac.var: 1

      algo.par$expl.path.multi.start.frac.shock: 0

      algo.par$expl.path.multi.start.frac.surp: 0

      algo.par$expl.path.multi.start.frac.fam: 0

algo.par$expl.path.multi.end.frac.var: 1

algo.par$expl.path.multi.end.frac.shock: 0

algo.par$expl.path.multi.end.frac.surp: 0

algo.par$expl.path.multi.end.frac.fam: 0

c) *maximizing surprise*

algo.par$expl.path.multi.start.frac.var: 0

algo.par$expl.path.multi.start.frac.shock: 0

algo.par$expl.path.multi.start.frac.surp: 1

algo.par$expl.path.multi.start.frac.fam: 0

algo.par$expl.path.multi.end.frac.var: 0

algo.par$expl.path.multi.end.frac.shock: 0

algo.par$expl.path.multi.end.frac.surp: 1

algo.par$expl.path.multi.end.frac.fam: 0

d) *minimizing familiarity*

algo.par$expl.path.multi.start.frac.var: 0

algo.par$expl.path.multi.start.frac.shock: 0

algo.par$expl.path.multi.start.frac.surp: 0

algo.par$expl.path.multi.start.frac.fam: 1

algo.par$expl.path.multi.end.frac.var: 0

algo.par$expl.path.multi.end.frac.shock: 0

algo.par$expl.path.multi.end.frac.surp: 0

algo.par$expl.path.multi.end.frac.fam: 1

e) eps50.noisy50

algo.par$expl.path.multi.start.frac.var: 0.5

algo.par$expl.path.multi.start.frac.shock: 0.5

algo.par$expl.path.multi.start.frac.surp: 0

algo.par$expl.path.multi.start.frac.fam: 0

algo.par$expl.path.multi.end.frac.var: 0.5

algo.par$expl.path.multi.end.frac.shock: 0.5

algo.par$expl.path.multi.end.frac.surp: 0

algo.par$expl.path.multi.end.frac.fam: 0

  f) noisy33.fam33.surp33

    algo.par$expl.path.multi.start.frac.var: $\approx 0.\bar{3}$

    algo.par$expl.path.multi.start.frac.shock: 0

    algo.par$expl.path.multi.start.frac.surp: $\approx 0.\bar{3}$

    algo.par$expl.path.multi.start.frac.fam: $\approx 0.\bar{3}$

    algo.par$expl.path.multi.end.frac.var: $\approx 0.\bar{3}$

    algo.par$expl.path.multi.end.frac.shock: 0

    algo.par$expl.path.multi.end.frac.surp: $\approx 0.\bar{3}$

    algo.par$expl.path.multi.end.frac.fam: $\approx 0.\bar{3}$

  g) all25

    algo.par$expl.path.multi.start.frac.var: 0.25

    algo.par$expl.path.multi.start.frac.shock: 0.25

    algo.par$expl.path.multi.start.frac.surp: 0.25

    algo.par$expl.path.multi.start.frac.fam: 0.25

    algo.par$expl.path.multi.end.frac.var: 0.25

    algo.par$expl.path.multi.end.frac.shock: 0.25

    algo.par$expl.path.multi.end.frac.surp: 0.25

    algo.par$expl.path.multi.end.frac.fam: 0.25

The parameters to generate Figure 20 have been chosen identical to Figure 19, except for the following deviations:

1. Common Parameters

  a) game.object

    game.pars$err.D.prob: 0.0

  b) algo.par

    batch.size: 12000

  c) model.par (To reflect that overfitting is a non-issue without noise)

    nrounds: 10

    max_depth: 10

    eta: 1

    gamma: 0.00001

    colsample: 1

subsample: 1

2. Varied Parameters
   a) eps100.noisy0

      algo.par$expl.path.multi.start.frac.shock: 1

      algo.par$expl.path.multi.start.frac.var: 0

      algo.par$expl.path.multi.end.frac.var: 0

      algo.par$expl.path.multi.end.frac.shock: 1

   b) eps50.noisy50

      algo.par$expl.path.multi.start.frac.shock: 0.5

      algo.par$expl.path.multi.start.frac.var: 0.5

      algo.par$expl.path.multi.end.frac.var: 0.5

      algo.par$expl.path.multi.end.frac.shock: 0.5

   c) eps0.noisy100

      algo.par$expl.path.multi.start.frac.shock: 0

      algo.par$expl.path.multi.start.frac.var: 1

      algo.par$expl.path.multi.end.frac.var: 1

      algo.par$expl.path.multi.end.frac.shock: 0

G.4. Exploration Pathing

The following parameters have been chosen to generate Figures 23:

1. Common Parameters
   a) model.par

      nrounds: 20

      max_depth: 5

2. Varied Parameters
   a) Decr85, Decr90, Decr95, Decr99

      algo.par$block.expl.shock: 1

      algo.par$block.expl.multi: 0

      algo.par$epsilon.start: 1

      algo.par$epsilon.decay: as specified (e.g. 0.85 for Decr85)

      algo.par$epsilon.min: 0

b) Path85, Path90-99

algo.par$expl.path.multi.start.shock: 0.5

algo.par$expl.path.multi.start.frac.var: 0

algo.par$expl.path.multi.start.frac.shock: 1

algo.par$expl.path.multi.start.frac.surp: 0

algo.par$expl.path.multi.start.frac.fam: 0

algo.par$expl.path.multi.end.frac.var: 0

algo.par$expl.path.multi.end.frac.shock: 1

algo.par$expl.path.multi.end.frac.surp: 0

algo.par$expl.path.multi.end.frac.fam: 0

algo.par$expl.path.multi.start: 0.85 for Path85, 0.9 for Path90-99

algo.par$expl.path.multi.end: 0.85 for Path85, 0.99 for Path90-99

algo.par$expl.path.multi.decay.type: linear

G.5. Feature Selection

The following parameters have been chosen to generate Figures 24 and 25:
Unless noted otherwise, the same parameters have been chosen as with Q-Switching (see Appendix G.2.)

When generating the game.object with Get.Game.Object.PD

encoding.state: "last.X.rounds" for only using the last X periods, "Main.real" for the main encoding and "Harper" for the encoding of Harper et al. (2017).

encoding.params

real: TRUE for all graphs except "Last 4 periods /wo private history", where FALSE has been chosen.

rounds: number of periods, which should be considered

The following parameters have been chosen to generate Figure 26:
Unless noted otherwise, the same parameters have been chosen as with Q-Switching (see Appendix G.2.)

When generating the game.object with Get.Game.Object.PD

encoding.state: "TimeSeries.flexible" for the encoding "Main LSTM Encoding", "TimeSeries.minimal" for the encoding "Minimal LSTM Encoding".

encoding.params (only relevant if TimeSeries.flexible) last.rounds: 10

rounds.bin: TRUE

av.def: TRUE

diff.bin: TRUE

prev.val.as.seen: TRUE

algo.par

batch.size: 360

use.rnn: TRUE

model.par

epochs: 10

batch.size.train: 100

give.up.precision: 5

hidden.nodes: (128,64)

drop.out: (0,0)

recurrent.dropout: 0

input.dropout: 0

activation.hidden: (sigmoid, sigmoid)

single.dimensional: TRUE

enforce.increasing.precision: TRUE

give.up.precision: 5

G.6. Experience Replay

In regards to Figures 27 and 28 it holds that

1. the parameters for the Neural Network are identical to the optimal Neural Network of Appendix G.7 unless noted otherwise
2. the parameters for the RNN are identical to the “Minimal LSTM Encoding” Network of Appendix G.5 unless noted otherwise
3. the parameters for the Gradient Boosting algorithm are identical to the optimal Gradient Boosting algorithm of Appendix G.7 unless noted otherwise
4. insofar as the size of the replay data set has been changed the parameter “algo.par$batch.size” is set to $2 \cdot 60 \cdot$ “size of replay data set” with a default of 360 if the replay data set is not varied.

5. insofar as the size of the number of most recent experiences has been forced, it holds that the parameter “algo.par$force.last” is set to $2 \cdot 60 \cdot$ “number of forced blocks” with a default of 120 (i.e. one forced block) if the number of forced blocks is not varied.

G.7. Choice of Function Approximator

The following parameters have been chosen to generate Figure 29 and Figure 30. If not noted otherwise, the same parameters have been chosen as with Q-Switching (Appendix G.2).
With Figure 29 the following parameters have been chosen with *model.par*, unless otherwise noted:

1. Gradient Boosting

   nrounds: 50

   max_depth: 5

2. Neural Network

   (algo.par), batch.size: 600

   epochs: 10

   batch.size.train: 100

   hidden.nodes: (128,64)

   activation.hidden: (sigmoid, sigmoid)

   activation.output: linear

   optimizer: optimizer_adam(lr=0.001)

   dropout: (0,0)

   input.dropout: 0

   single.dimensional: TRUE

   enforce.increasing.precision: TRUE

   give.up.precision: 5

3. Recurrent Neural Network has been chosen identically to the “Main encoding” of the LSTM from Appendix G.5.

The calculation time has been calculated based on the following hardware:

- CPU: Intel Core i7-3630QM (2.40 GHz)
- RAM: 8 GB

G.8. Model Persistence

The following parameters have been chosen to generate Figure 31.

1. The same parameter set as with the optimal Gradient Boosting parameter set has been used from Appendix G.7, except
2. algo.par

   block.cur: 2 for “Explo.2-Cur.2-Best.0”, 1 for “Explo.2-Cur.1-Best.1”

   block.best: 0 for “Explo.2-Cur.2-Best.0”, 1 for “Explo.2-Cur.1-Best.1”

   block.expl.multi: 2

G.9. Memory Initialization

The following parameters have been chosen to generate Figure 32.

1. Common Parameters identical to Section 3.4 (see Appendix G.2)
2. Varied Parameters
   a) No Initialization identical to Q-Switching, i.e.

      start.w.training: FALSE

   b) Initialization 1

      start.w.training: TRUE

      With Initialise.QLearningPersExpPath:

         memory.init: “self.play”

         memory.param: list(no=1)

   c) Initialization 50

      start.w.training: TRUE

      With Initialise.QLearningPersExpPath:

         memory.init: “self.play”

         memory.param: list(no=50)

H. Source Code for Final Results

This code may be used to generate the results behind Figure 12 and Figure 10. These files are also available at Kies (2019).

H.1. Recurrent Neural Network

```
library(ReinforcementLearningwithR)
require(compiler)

strat <- c("strat.a","strat.b","strat.c","strat.d","strat.e","strat.f",
    ↪ "strat.g","strat.h","strat.i") #"strat.a" to "strat.i" with get.
    ↪ antistrat or any other strategy (vector) without "self" implies
    ↪ self play.
antistrat <- "none" #or "none" if none of the strategies of above or
    ↪ several strategies are given.
file.name <- paste0("opt.run.",paste0(strat,collapse="."),".",Sys.Date
    ↪ (),".RNN") #File, where the results are saved

#Parameters of game. Currently supported: "BattleOfStrategiesThesis.
    ↪ Baseline" and "BattleOfStrategies2019"
game.setting <- "BattleOfStrategiesThesis.Baseline"

block.no <- 500 #Number of Blocks to Play. More should be always better
    ↪ , but time increases somewhat linear (given the memory is
    ↪ sufficiently full), while we have strong diminishing return in
    ↪ the final performance depending on the complexity of the strategy
    ↪ .
eval.no <- 1000 #Number of played matches to evaluate final performance
    ↪  of model with the model StratTourn
T.max <- 60 #Number of periods of game. Note: If one wants to change
    ↪ this, it is recommended, that algo.par$batch.size is changed as
    ↪ well.

#If Memory initilization through self.play is wished it may be set here
    ↪ . 0 means no initialization.
memory.initialization <- 100

#Set the most important parameters of Recurrent Neural Network here.
## nodes.layer.1 measures the number of nodes in the first hidden layer
    ↪ . More means more complex strategies may be tackled, but risks
    ↪ overfitting and might need more training data and more epochs.
## nodes.layer.2 measures the number of nodes in the second hidden
    ↪ layer. More means more complex strategies may be tackled, but
    ↪ risks overfitting and might need more training data and more
    ↪ epochs.
## batch.size.train is the RNN internal size of how big a neural
    ↪ network batch should be (see https://stats.stackexchange.com/
    ↪ questions/153531/what-is-batch-size-in-neural-network). More
    ↪ means more stable, but slower/more epochs needed.
##rnn.epochs is the number of how often the complete training data
    ↪ should be propagated through the network. More means more
    ↪ overfitting but better accuracy in describing the training data.
    ↪ Very relevant for speed.
##give.up.precision controls a method used in the thesis of Martin Kies
    ↪ : After training epochs times it is checked whether the new LOSS
    ↪ is better than the one in the block before. If not the training
    ↪ is repeated up to give.up.precision times. The actual number of
    ↪ epochs such may vary between rnn.epochs and rnn.epochs*give.up.
    ↪ precision
func.approx.params <- list(nodes.layer.1=126, nodes.layer.2=64, batch.
    ↪ size.train=600, rnn.epochs=5, give.up.precision=10)

#This defines the function which allows an easy access to the package
generate.best.strat <- function(strat, antistrat="none", game.setting,
    ↪ func.approx.params, memory.initialization, block.no, eval.no, T.
    ↪ max, file.name){
  restore.point("generate.best.strat")

  game.object <- Get.Game.Object.PD(encoding.state="TimeSeries.minimal"
      ↪ ,game.setting=game.setting,strats=strat, eval.strategy = "Model
      ↪ .strat.RNN.TimeSeries.minimal")
  assign("game.object",game.object,envir=.GlobalEnv) #necessary for the
      ↪  tournament

  #Define the non-changing parameters of the algorithm like which
      ↪ features and parameters to be used.
  algo.par <- Get.Def.Par.QLearningPersExpPath(setting="ThesisOpt.RNN")
  algo.par$gamma <- game.object$game.pars$delta #recommended as the
      ↪ algorithm otherwise optimises for a different setting as the
      ↪ game itself

  #Define the function approximator and its parameters
  model.par <- Get.Def.Par.RNN(setting="ThesisOpt")
  model.par$hidden.nodes[1] <- func.approx.params$nodes.layer.1
  model.par$hidden.nodes[2] <- func.approx.params$nodes.layer.2
  model.par$batch.size.train <- func.approx.params$batch.size.train
  model.par$epochs <- func.approx.params$rnn.epochs
  model.par$give.up.precision <- func.approx.params$give.up.precision

  #Setup the model and other variational aspects
  evaluator <- Setup.QLearningPersExpPath(game.object, algo.par=algo.
      ↪ par, model.par=model.par)
  if(memory.initialization==0){
    memory.init <- "none"
  } else {
    memory.init <- "self.play"
  }
  algo.var <- Initialise.QLearningPersExpPath(game.object, algo.par,
      ↪ memory.init=memory.init, memory.param=list(no=memory.
      ↪ initialization), model.par=model.par)

  #Execute the algorithm
  res <- Train.QLearningPersExpPath(evaluator=evaluator, model.par=
      ↪ model.par, algo.par=algo.par, algo.var=algo.var, game.object =
      ↪ game.object, blocks=block.no, eval.only=FALSE, start.w.training
```

```
     ↪ = TRUE, out.file=paste0(file.name,".tmp"))

#Save Memory & model
evaluator <- res$evaluator
algo.var <- res$algo.var
idio.name <- paste0("opt.run.RNN.full.",paste0(strat,collapse="."))
file.name <- paste0(idio.name, format(Sys.time(), "%d-%b-%Y␣%H.%M"),"
     ↪ before.StratTourn", sep="␣")
save(evaluator, algo.var, algo.par, game.object, model.par, file=file
     ↪ .name)

# Do the StratTourn evaluation
game = make.pd.game(uCC=game.object$game.pars$uCC, uCD=game.object$
     ↪ game.pars$uCD, uDC=game.object$game.pars$uDC, uDD=game.object$
     ↪ game.pars$uDD, err.D.prob=game.object$game.pars$err.D.prob, err
     ↪ .C.prob=game.object$game.pars$err.C.prob, delta=game.object$
     ↪ game.pars$delta)

#Prepare list of strategies for StratTourn
if(antistrat!="none"){
    strat.tourn = nlist(Model.strat.RNN.TimeSeries.minimal,get(strat)
        ↪ , get(antistrat))
    names(strat.tourn)[2] <- strat
    names(strat.tourn)[3] <- antistrat
  } else {
    strat.tourn = c(Model.strat.RNN.TimeSeries.minimal,lapply(strat,
        ↪ FUN=function(x){
      if(x!="self"){
        return(get(x))
      } else {
        return(NULL)
      }
    }))
    names(strat.tourn) <- seq_along(strat.tourn)
    strat.tourn[sapply(strat.tourn, is.null)] <- NULL
    names(strat.tourn)[1] <- "Model.strat.RNN.TimeSeries.minimal"
    names(strat.tourn)[2:length(names(strat.tourn))] <- strat[strat!=
        ↪ "self"]
}

#Initialize tournament
tourn = init.tournament(game=game, strat=strat.tourn)
tourn = run.tournament(tourn=tourn, R = eval.no, T.max=T.max)

#Calculate a single relevant statistic. If against a single strategy
     ↪ this is the return against this strategy. If against a
     ↪ tournament this is the tournament performance.
if(length(strat)==1){
  r.limit <- get.matches.vs.matrix(tourn$dt)["Model.strat.RNN.
      ↪ TimeSeries.minimal",strat]
} else {
  srfm <- strat.rank.from.matches(tourn$dt)
  r.limit <- srfm[srfm$strat=="Model.strat.RNN.TimeSeries.minimal",
      ↪ mean]
  }

  file.name <- paste0(idio.name, format(Sys.time(), "%d-%b-%Y␣%H.%M"),
     ↪ sep="␣")

  #Save Memory & model
  save(evaluator, algo.var, algo.par, game.object, model.par, r.limit,
     ↪ tourn, file=file.name)

  #Show Tournament
  show.tournament(tourn)
}

disable.restore.points(TRUE)
enableJIT(3)
generate.best.strat(strat=strat, antistrat=antistrat, game.setting=game
   ↪ .setting, func.approx.params=func.approx.params, memory.
   ↪ initialization=memory.initialization, block.no=block.no, eval.no=
   ↪ eval.no, T.max=T.max, file.name=file.name)
```

H.2. Gradient Boosting

```
library(ReinforcementLearningwithR)
require(compiler)

strat <-c("strat.a","strat.b","strat.c","strat.d","strat.e","strat.f","
   ↪ strat.g","strat.h","strat.i") #"strat.a" to "strat.i" with get.
   ↪ antistrat or any other strategy (vector) without "self" implies
   ↪ self play.
antistrat <- get.antistrat()[strat] #or "none" if none of the
   ↪ strategies of above or several strategies are given.
file.name <- paste0("opt.run.",paste0(strat,collapse="."),".",Sys.Date
   ↪ (),".XGB") #File, where the results are saved

#Parameters of game. Currently supported: "BattleOfStrategiesThesis.
   ↪ Baseline" and "BattleOfStrategies2019"
game.setting <- "BattleOfStrategiesThesis.Baseline"

block.no <- 150 #Number of Blocks to Play. More should be always better
   ↪ , but time increases somewhat linear (given the memory is
   ↪ sufficiently full), while we have strong diminishing return in
   ↪ the final performance depending on the complexity of the strategy
   ↪ .
eval.no <- 100 #Number of played matches to evaluate final performance
   ↪ of model with the model StratTourn
T.max <- 60 #Number of periods of game. Note: If one wants to change
   ↪ this, it is recommended, that algo.par$batch.size is changed as
   ↪ well.

#If Memory initilization through self.play is wished it may be set here
   ↪ . 0 means no initialization.
memory.initialization <- 100

#Set the most important parameters of Gradient Boosting here.
```

```
## xgb.rounds measures the number of build trees
## xgb.depth measures the depth of each tree.
## In general it holds more is more precise (which might imply more
    ↪ training data is necessary) which is good but costly in time.
func.approx.params <- list(xgb.rounds=50, xgb.depth=5)

#This defines the function which allows an easy access to the package
generate.best.strat <- function(strat, antistrat="none", game.setting,
    ↪ func.approx.params, memory.initialization, block.no, eval.no, T.
    ↪ max, file.name){
  restore.point("generate.best.strat")

  game.object <- Get.Game.Object.PD(game.setting=game.setting,strats=
      ↪ strat)
  assign("game.object",game.object,envir=.GlobalEnv) #necessary for the
      ↪ tournament

  #Define the non-changing parameters of the algorithm like which
      ↪ features and parameters to be used.
  algo.par <- Get.Def.Par.QLearningPersExpPath(setting="ThesisOpt.XGB")
  algo.par$gamma <- game.object$game.pars$delta #recommended as the
      ↪ algorithm otherwise optimises for a different setting as the
      ↪ game itself

  #Define the function approximator and its parameters
  model.par <- Get.Def.Par.XGBoost(setting="ThesisOpt")
  model.par$nrounds <- func.approx.params$xgb.rounds
  model.par$max_depth <- func.approx.params$xgb.depth

  #Setup the model and other variational aspects
  evaluator <- Setup.QLearningPersExpPath(game.object, algo.par=algo.
      ↪ par, model.par=model.par)
  if(memory.initialization==0){
    memory.init <- "none"
  } else {
    memory.init <- "self.play"
  }
  algo.var <- Initialise.QLearningPersExpPath(game.object, algo.par,
      ↪ memory.init=memory.init, memory.param=list(no=memory.
      ↪ initialization), model.par=model.par)

  #Execute the algorithm
  res <- Train.QLearningPersExpPath(evaluator=evaluator, model.par=
      ↪ model.par, algo.par=algo.par, algo.var=algo.var, game.object =
      ↪ game.object, blocks=block.no, eval.only=FALSE, start.w.training
      ↪  = TRUE,out.file=paste0(file.name,".tmp"))

  #Save Memory & model
  evaluator <- res$evaluator
  algo.var <- res$algo.var
  idio.name <- paste0("opt.run.XGB.full.",paste0(strat,collapse="."))
  file.name <- paste0(idio.name, format(Sys.time(), "%d-%b-%Y␣%H.%M"),"
      ↪ before.StratTourn", sep="␣")
```

```
save(evaluator, algo.var, algo.par, game.object, model.par, file=file
    ↪ .name)

# Do the StratTourn evaluation
game = make.pd.game(uCC=game.object$game.pars$uCC, uCD=game.object$
    ↪ game.pars$uCD, uDC=game.object$game.pars$uDC, uDD=game.object$
    ↪ game.pars$uDD, err.D.prob=game.object$game.pars$err.D.prob, err
    ↪ .C.prob=game.object$game.pars$err.C.prob, delta=game.object$
    ↪ game.pars$delta)

#Prepare list of strategies for StratTourn
if(antistrat!="none"){
     strat.tourn = nlist(Model.strat.Main.real.Exp.Path,get(strat),
         ↪ get(antistrat))
     names(strat.tourn)[2] <- strat
     names(strat.tourn)[3] <- antistrat
  } else {
     strat.tourn = c(Model.strat.Main.real.Exp.Path,lapply(strat,FUN=
         ↪ function(x){
       if(x!="self"){
         return(get(x))
       } else {
         return(NULL)
       }
     }))
     names(strat.tourn) <- seq_along(strat.tourn)
     strat.tourn[sapply(strat.tourn, is.null)] <- NULL
     names(strat.tourn)[1] <- "Model.strat.Main.real.Exp.Path"
     names(strat.tourn)[2:length(names(strat.tourn))] <- strat[strat!=
         ↪ "self"]
}

#Initialize tournament
tourn = init.tournament(game=game, strat=strat.tourn)
tourn = run.tournament(tourn=tourn, R = eval.no, T.max=T.max)

#Calculate a single relevant statistic. If against a single strategy
    ↪ this is the return against this strategy. If against a
    ↪ tournament this is the tournament performance.
if(length(strat)==1){
  r.limit <- get.matches.vs.matrix(tourn$dt)["Model.strat.Main.real.
      ↪ Exp.Path",strat]
} else {
  srfm <- strat.rank.from.matches(tourn$dt)
  r.limit <- srfm[srfm$strat=="Model.strat.Main.real.Exp.Path",mean]
}

file.name <- paste0(idio.name, format(Sys.time(), "%d-%b-%Y␣%H.%M"),
    ↪ sep="␣")

#Save Memory & model
save(evaluator, algo.var, algo.par, game.object, model.par, r.limit,
    ↪ tourn, file=file.name)
  #Show Tournament
  show.tournament(tourn)
}

disable.restore.points(TRUE)
enableJIT(3)
generate.best.strat(strat=strat, antistrat=antistrat, game.setting=game
    ↪ .setting, func.approx.params=func.approx.params, memory.
    ↪ initialization=memory.initialization, block.no=block.no, eval.no=
    ↪ eval.no, T.max=T.max, file.name=file.name)
```

H.3. Q-Learning

```
library(ReinforcementLearningwithR)
require(compiler)

strat <- c("strat.a") #"strat.a" to "strat.i" with get.antistrat or any
    ↪  other strategy (vector) without "self" implies self play.
antistrat <- "none" #or "none" if none of the strategies of above or
    ↪ several strategies are given.
file.name <- paste0("opt.run.",paste0(strat,collapse="."),".",Sys.Date
    ↪ (),".NN") #File, where the results are saved

#Parameters of game. Currently supported: "BattleOfStrategiesThesis.
    ↪ Baseline" and "BattleOfStrategies2019"
game.setting <- "BattleOfStrategiesThesis.Baseline"

block.no <- 150 #Number of Blocks to Play. More should be always better
    ↪ , but time increases somewhat linear (given the memory is
    ↪ sufficiently full), while we have strong diminishing return in
    ↪ the final performance depending on the complexity of the strategy
    ↪ .
eval.no <- 1000 #Number of played matches to evaluate final performance
    ↪  of model with the model StratTourn
T.max <- 60 #Number of periods of game. Note: If one wants to change
    ↪ this, it is recommended, that algo.par$batch.size is changed as
    ↪ well.

#If Memory initilization through self.play is wished it may be set here
    ↪ . 0 means no initialization.
memory.initialization <- 0

#Set the most important parameters of a Neural Network here.
## nodes.layer.1 measures the number of nodes in the first hidden layer
    ↪ . More means more complex strategies may be tackled, but risks
    ↪ overfitting and might need more training data and more epochs.
## nodes.layer.2 measures the number of nodes in the second hidden
    ↪ layer. More means more complex strategies may be tackled, but
    ↪ risks overfitting and might need more training data and more
    ↪ epochs.
## batch.size.train is the NN internal size of how big a neural network
    ↪  batch should be (see https://stats.stackexchange.com/questions/
    ↪ 153531/what-is-batch-size-in-neural-network). More means more
    ↪ stable, but slower/more epochs needed.
```

```
##epochs is the number of how often the complete training data should
    ↪ be propagated through the network. More means more overfitting
    ↪ but better accuracy in describing the training data. Very
    ↪ relevant for speed.
##give.up.precision controls a method used in the thesis of Martin Kies
    ↪ : After training epochs times it is checked whether the new LOSS
    ↪ is better than the one in the block before. If not the training
    ↪ is repeated up to give.up.precision times. The actual number of
    ↪ epochs such may vary between epochs and epochs*give.up.precision.
    ↪  As this is the showcase for a benchmark case the default uses no
    ↪  give.up.precision
func.approx.params <- list(nodes.layer.1=126, nodes.layer.2=64, batch.
    ↪ size.train=32, epochs=50, give.up.precision=0)

#This defines the function which allows an easy access to the package
generate.best.strat <- function(strat, antistrat="none", game.setting,
    ↪ func.approx.params, memory.initialization, block.no, eval.no, T.
    ↪ max, file.name){
  restore.point("generate.best.strat")

  game.object <- Get.Game.Object.PD(encoding.state="Harper",game.
      ↪ setting=game.setting,strats=strat, eval.strategy = "Model.strat
      ↪ .NN.Harper")
  assign("game.object",game.object,envir=.GlobalEnv) #necessary for the
      ↪  tournament

  #Define the non-changing parameters of the algorithm like which
      ↪ features and parameters to be used.
  algo.par <- Get.Def.Par.QLearningPersExpPath(setting="QLearning.Basic
      ↪ ")
  algo.par$gamma <- game.object$game.pars$delta #recommended as the
      ↪ algorithm otherwise optimises for a different setting as the
      ↪ game itself

  #Define the function approximator and its parameters
  model.par <- Get.Def.Par.Neural.Network(setting="ThesisBasic")
  model.par$hidden.nodes[1] <- func.approx.params$nodes.layer.1
  model.par$hidden.nodes[2] <- func.approx.params$nodes.layer.2
  model.par$batch.size.train <- func.approx.params$batch.size.train
  model.par$epochs <- func.approx.params$epochs
  model.par$give.up.precision <- func.approx.params$give.up.precision
  if(func.approx.params$give.up.precision==0){
    model.par$enforce.increasing.precision <- FALSE
  } else {
    model.par$enforce.increasing.precision <- TRUE
  }


  #Setup the model and other variational aspects
  evaluator <- Setup.QLearningPersExpPath(game.object, algo.par=algo.
      ↪ par, model.par=model.par)
  if(memory.initialization==0){
    memory.init <- "none"
    start.w.training <- FALSE
} else {
  memory.init <- "self.play"
  start.w.training <- TRUE
}
algo.var <- Initialise.QLearningPersExpPath(game.object, algo.par,
    ↪ memory.init=memory.init, memory.param=list(no=memory.
    ↪ initialization), model.par=model.par)

#Execute the algorithm
res <- Train.QLearningPersExpPath(evaluator=evaluator, model.par=
    ↪ model.par, algo.par=algo.par, algo.var=algo.var, game.object =
    ↪ game.object, blocks=block.no, eval.only=FALSE, start.w.training
    ↪ = start.w.training,out.file=paste0(file.name,".tmp"))

#Save Memory & model
evaluator <- res$evaluator
algo.var <- res$algo.var
idio.name <- paste0("opt.run.NN.full.",paste0(strat,collapse="."))
file.name <- paste0(idio.name, format(Sys.time(), "%d-%b-%Y␣%H.%M"),"
    ↪ before.StratTourn", sep="␣")
save(evaluator, algo.var, algo.par, game.object, model.par, file=file
    ↪ .name)

# Do the StratTourn evaluation
game = make.pd.game(uCC=game.object$game.pars$uCC, uCD=game.object$
    ↪ game.pars$uCD, uDC=game.object$game.pars$uDC, uDD=game.object$
    ↪ game.pars$uDD, err.D.prob=game.object$game.pars$err.D.prob, err
    ↪ .C.prob=game.object$game.pars$err.C.prob, delta=game.object$
    ↪ game.pars$delta)

#Prepare list of strategies for StratTourn
if(antistrat!="none"){
  strat.tourn = nlist(Model.strat.NN.Harper,get(strat), get(antistrat
      ↪ ))
  names(strat.tourn)[2] <- strat
  names(strat.tourn)[3] <- antistrat
} else {
  strat.tourn = c(Model.strat.NN.Harper,lapply(strat,FUN=function(x){
    if(x!="self"){
      return(get(x))
    } else {
      return(NULL)
    }
  }))
  names(strat.tourn) <- seq_along(strat.tourn)
  strat.tourn[sapply(strat.tourn, is.null)] <- NULL
  names(strat.tourn)[1] <- "Model.strat.NN.Harper"
  names(strat.tourn)[2:length(names(strat.tourn))] <- strat[strat!="
      ↪ self"]
}

#Initialize tournament
tourn = init.tournament(game=game, strat=strat.tourn)
tourn = run.tournament(tourn=tourn, R = eval.no, T.max=T.max)
```

```
  #Calculate a single relevant statistic. If against a single strategy
      ↪ this is the return against this strategy. If against a
      ↪ tournament this is the tournament performance.
  if(length(strat)==1){
    r.limit <- get.matches.vs.matrix(tourn$dt)["Model.strat.NN.Harper",
        ↪ strat]
  } else {
    srfm <- strat.rank.from.matches(tourn$dt)
    r.limit <- srfm[srfm$strat=="Model.strat.NN.Harper",mean]
  }

  file.name <- paste0(idio.name, format(Sys.time(), "%d-%b-%Y␣%H.%M"),
      ↪ sep="␣")

  #Save Memory & model
  save(evaluator, algo.var, algo.par, game.object, model.par, r.limit,
      ↪ tourn, file=file.name)

  #Show Tournament
  show.tournament(tourn)
}

disable.restore.points(TRUE)
enableJIT(3)
generate.best.strat(strat=strat, antistrat=antistrat, game.setting=game
    ↪ .setting, func.approx.params=func.approx.params, memory.
    ↪ initialization=memory.initialization, block.no=block.no, eval.no=
    ↪ eval.no, T.max=T.max, file.name=file.name)
```

2 Impacts of Sponsored Data on Infrastructure Investments and Welfare

Source:

Martin Kies (2017). “Impacts of Sponsored Data on Infrastructure Investments and Welfare”. In: *Available at SSRN*. DOI: `10.2139/ssrn.3042563`. URL: `https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3042563`

Impacts of Sponsored Data on Infrastructure Investments and Welfare[*]

Martin Kies[§]

December 13, 2017

With increasing demand for wireless data and new requirements to uphold "net neutrality", internet service providers try new methods to ensure their profits. Sponsored content, the archetype of "Zero-Rating", allows the content provider to pay for the accrued traffic instead of the consumer.
This paper shows, using a theoretical model, that allowing sponsored content has ambiguous results both on infrastructure investments and net welfare.
Sponsored content may be used by the ISP to internalize additional profits of the content provider and thus motivate higher infrastructure investments, especially given very high network costs. However, given a sufficiently high profitability of the content provider, the internet service provider might be more interested in optimizing the income stream from the content provider than in the satisfaction of the consumer. This not only decreases effective network capacity but might also lead to a decrease in net welfare.

Keywords: Net Neutrality, Two-Sided Markets, Sponsored Content, Sponsored Data, Zero Rating, Zero-Rating, ISP, Internet Service Provider
JEL Classification: D42, D60, L51, L86, L96

[*]I thank Sebastian Kranz for excellent feedback. I am grateful to Frederik Collin and Maria Baier for helpful discussions and comments.

[§]Department of Mathematics and Economics, Ulm University. E-mail: martin.kies@uni-ulm.de

1. Introduction

Net neutrality as coined by T. Wu (2003) wants to ensure, that all traffic within the internet receives “fair” treatment and that Internet Service Providers (ISPs) do not earn money by discriminating the speed or accessibility of content. The literature review paper Krämer, Wiewiorra, and Weinhardt (2013) defines net neutrality the following way:

> “Net neutrality prohibits Internet service providers from speeding up, slowing down or blocking Internet traffic based on its source, ownership or destination.”

Net neutrality remains a controversial topic, especially in the USA. Internet service providers tried to establish innovations like fast-track internet lanes (see e.g. Sasso (2014)) leading to a public outcry. Consequently several initiatives like the “Internet Slowdown” (FightForTheFuture, 2015) formed, demanding that net neutrality should be enforced by the law. President Obama positioned himself as an advocate of net neutrality (The White House, 2015). Due to this political pressure the Federal Communications Commission (FCC) published several new laws and guidelines (Federal Communications Commission, 2015), defining internet broadband as a “Common Carrier” thereby ensuring net neutrality. However, after the change of presidency in the U.S., the new chairman of the FCC Ajit Pai aggressively targets consumer protection regulations and got the FCC to vote for rolling back the net neutrality rules (Fiegerman, 2017), starting the process to change the rules once again.

A similar discussion took place in the European Union, which also implemented net neutrality rules (European Commission, 2017). Since April 30th, 2016 blocking, throttling and discrimination of internet traffic is not permitted within the EU, save for some exceptions due to special circumstances.

Even though both in the U.S. and the EU - regarding normal internet traffic - no speed discrimination may take place and thus the ISP may not increase the Quality-of-Service of favored content providers this way, both markets currently allow a certain form of economic discrimination: The practice of “zero-rating”.

Not uncommon for broadband internet in the U.S. and the most common form of contract for mobile internet in both the U.S. and the EU is a form of fixed data cap contract, where the consumer pays a fee per period and receives in return the right to consume a fixed amount of internet traffic. With a zero-rating contract certain content is exempted from this data cap. Given that the data cap otherwise doesn’t change and the ISP is able to absorb the additional traffic, this type of feature is very attractive both to the consumer and the content service provider. The consumer can now consume as much of this content as she wants and the content provider can use the additional traffic to increase its advertisement earnings or can transform the additional publicity and attractiveness to more sales or subscriptions. A special form of zero-rating contracts are sponsored data contracts, where the content provider whose content is exempted from the data cap pays the ISP to do so.

Such contracts are on the rise and Welsh de Grimaldo (2015) estimates a strong growth of such business models in the future. Notable examples include AT&T (2017) in the U.S., where the content provider pays for the traffic induced through their services instead of the consumer. The ISP T-Mobile US offers for most of their plans that the streaming of music from a lot of different specific content providers does not count toward the data cap of the consumer (T-Mobile US, 2017b) as well as a similar service called "Binge On" with regard to video streaming (T-Mobile US, 2017a).
Within the EU, or more specific Germany, T-Mobile offered for several months the option that the music streaming service Spotify can be included as zero-rated content by the consumer (Schuhmacher, 2016). The current state of zero-rating with a strong focus on Germany can be found with Goldhammer, Wiegand, and Birkel (2016).
In other countries similar contracts have been standard for quite some time, as for example with the case of the ISP Bharti Airtel, which was sponsered to a great extent by Google (airtel India, 2015) before the Indian regulator Trai adopted a very strong stance in favor of net neutrality, forbidding this practice (Gustin, 2016).

With this paper I show that the possibility of sponsored content strongly influences a monopolistic internet provider in his decision on how much network capacity to provide. In my model, the consumer faces a single data cap contract with a usage-independent price and -if beneficial to the the respective companies- unlimited content for one content provider. The ISP has two potential income streams: The payment of the consumer and the payment of the content provider. He finds himself in a dilemma, where increasing one decreases the other. By deciding how much network capacity to build, and which contracts to offer, the ISP determines whether sponsored content proves beneficial or detrimental to total welfare.
With a sufficiently high profitability of the content provider, sponsored content will be implemented, exclusively so, if the profitability is very high. Given that non exclusive sponsored content is implemented and the profitability of the content provider is comparatively low, sponsored content increases network capacity relative to the non-sponsored case. With increasing profitability of the content provider the network capacity decreases and can fall below the respective network capacity of the non sponsored situation. Within our model the consumer and the non sponsoring content provider never profit from sponsored content. Net welfare only increases if network capacity increases and always decreases if the preference of the consumer towards the sponsored content is sufficiently low and non exclusive sponsored content is implemented. Given that both, network costs and profitability of the content provider, are very high, sponsored content leads to a situation of exclusive sponsored content, where net welfare increases.

We start in section 2 with an overview of the model. Section 3 and section 4 set baseline points by solving the first best solution and the setting without sponsored content, respectively. Afterwards section 5 introduces sponsored content and highlights which fundamentally different situations can happen with sponsored content. The effects on network capacity are discussed in section 6. Section 7 examines the welfare effects of allowing sponsored content. In section 8 the model is changed so that the bargaining power now lies with the content provider, thus providing a robustness check of the model. Finally, in section 9 related literature is discussed and section 10 offers some concluding remarks.

2. The Model

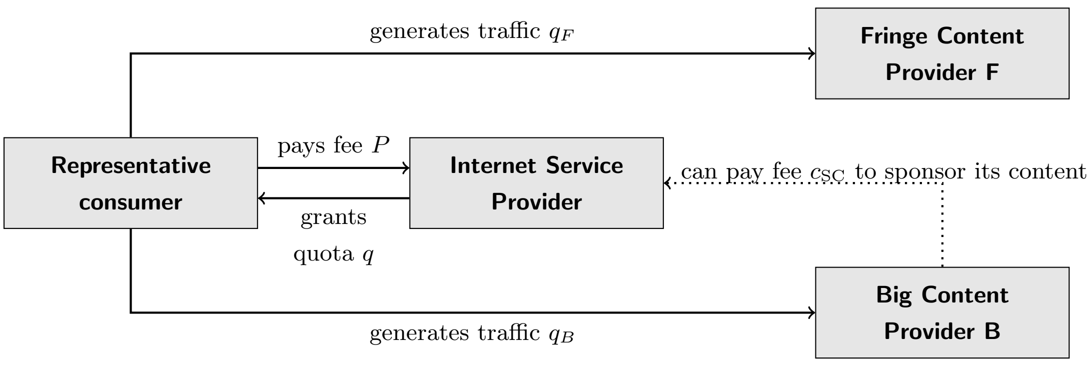

Figure 1: Interactions between the different actors. If content provider B pays the fee, than the consumer generates traffic q_B independent from the received quota q.

We assume that there exists one monopolistic last-mile internet service provider (ISP) which builds internet capacity $\kappa > 0$ with constant marginal costs $c_\kappa > 0$. The single, representative consumer signs a contract (q, P) with the ISP to access the internet and creates traffic for the two content service providers (CSP) B and F as seen in Figure 1. Quota $q > 0$ is the allowed maximum traffic the consumer can generate, while $P > 0$ is the internet access fee she pays to the ISP. We assume that this data cap q is final and no additional traffic can be generated, neither through overcharge fees nor through throttled speeds.
If sponsored content is permitted, content provider B is eligible to sponsor its traffic, while content provider F can be seen as a stand-in for the fringe content providers. One can image B to be a big content provider with market power and a sizable traffic volume, as for example YouTube or Netflix, while F are all other forms of internet traffic, including Peer-2-Peer, voice over internet and other services. The ISP can offer content provider B that for a fee $c_{SC} > 0$[1] its content will be sponsored. If CSP B agrees, B pays the fee to the ISP and the traffic of B does not count toward the quota of the consumer. The timeline of the model is thus

1. The ISP decides to build internet capacity κ.
2. The ISP can offer B a take-it-or-leave-it offer to pay the fee c_{SC} to sponsor its content.

[1] One can show that this is equivalent to a payment per traffic unit, as all information are known to all parties and there are no fixed costs within the model.

3. B can accept or decline that offer and pays c_{SC} if the offer has been accepted.
4. The ISP offers the consumer a take-it-or-leave-it contract with a specific quota q for a fixed fee P.
5. The consumer accepts the contract, if sensible, and creates traffic for B and F.

We assume that all firms want to maximize their profits. As we have perfect and complete information, we can restrict our attention to those cases, where only contracts are offered which will be accepted.

The utility function of the consumer if the contract of the ISP has been accepted is defined as

$$U(q_F, q_B) = q_F + q_B - \frac{1}{2(1-\beta)\alpha} q_F^2 - \frac{1}{2\beta\alpha} q_B^2 - P \tag{1}$$

with $q_B \geq 0$ and $q_F \geq 0$ being the traffic generated for CSP B or F, respectively. We assume, that the consumer receives zero utility if the contract of the ISP is rejected.
If no content is sponsored, the consumer faces the usage restriction

$$q_B + q_F \leq q \tag{2}$$

In the case that the content of CSP B is sponsored the usage restriction

$$q_F \leq q \tag{3}$$

applies.
The exogenously given parameter β describes the preference of the consumer towards content provider B. It holds, that $0 < \beta < 1$ so the consumer will always be interested in the content of both CSPs. The parameter $\alpha > 0$ denotes the internet affinity of the consumer and equals her saturation point given unlimited quota.

The profits of the CSPs are defined as

$$\pi_F = d \cdot q_F \qquad \pi_B = \begin{cases} d \cdot q_B & \text{if content is not sponsored} \\ d \cdot q_B - c_{\text{SC}} & \text{if content is sponsored} \end{cases} \tag{4}$$

with $d > 0$ being the marginal profit of greater traffic on their pages and $c_{\text{SC}} > 0$ the payment from the content provider to the internet service provider.

The profit function of the ISP given all his offers are accepted is given by

$$\pi_{\text{ISP}}(\kappa, c_{\text{SC}}, P) = \begin{cases} P - c_\kappa \cdot \kappa & \text{if content is not sponsored} \\ P + c_{\text{SC}} - c_\kappa \cdot \kappa & \text{if content is sponsored} \end{cases} \tag{5}$$

The ISP has thus to optimize between costs of network c_κ, the extraction from the consumer P and the extraction from the CSP B c_{SC}. We assume, that $c_\kappa < 1$, which guarantees that the ISP will enter the market. Note, that there is never any incentive for the ISP to shut out the consumer, so we will assume that the ISP always offers a fee P which is acceptable to the consumer.

The ISP has to ensure network stability, i.e. given the offered contract and resulting consumer choices it has to hold that

$$q_F + q_B \leq \kappa \tag{6}$$

To keep case differentiations to a minimum, we will assume that the ISP already incorporates the utility maximization of the consumer and thus always builds internet capacity be above the saturation point of the consumer, i.e. it holds that

$$\kappa \leq \alpha \tag{7}$$

This condition is motivated by the idea, that the ISP has no way to force the consumer to consume certain internet content, even if this might be welfare enhancing due to high profits of the CSP. Additionally he may not pay the consumer to use the internet.

3. First Best Solution

To understand the effects of the model and to have a baseline comparison, we will have a look at the first best solution, where everybody acts to maximize total welfare given by

$$\begin{aligned} W(q_F, q_B, \kappa) &= U + \pi_F + \pi_B + \pi_{\text{ISP}} \\ &= (q_F + q_B)(1+d) - \frac{1}{2(1-\beta)\alpha} q_F^2 - \frac{1}{2\beta\alpha} q_B^2 - c_\kappa \kappa \end{aligned}$$

Note, that sponsored content does not matter in this scenario, as the payments c_{SC} and P are welfare neutral. The only thing that matters is therefore the choice of the consumer on how to optimally divide her traffic and the choice of the ISP on the capacity κ.

First, we study the case, where the ISP has already implemented a specific network capacity $\kappa \leq \alpha$.

The welfare optimal consumption choices solve

$$(q_F^o, q_B^o) = \underset{q_F > 0,\, q_B > 0}{\arg\max}\; W(q_F, q_B, \kappa) \quad \text{w.r.t.} \quad q_F + q_B \leq \kappa \tag{8}$$

As W is monotonously increasing in q_F and q_B we can use Lagrange optimization leading to

$$q_F^o(\kappa) = (1-\beta)\kappa, \qquad q_B^o(\kappa) = \beta\kappa \tag{9}$$

Both CSPs have the same profitability d which makes a shift between the CSPs welfare neutral except for the effects on the consumer. Optimizing total welfare is thus equivalent to optimizing the benefit of the consumer, given κ. Intuitively it can be seen, that q_B^o increases in consumer preference β.

Now, we allow capacity to be endogenous:

$$\begin{aligned}\kappa^o &= \underset{0<\kappa\leq\alpha}{\arg\max}\, W(q_F^o(\kappa), q_B^o(\kappa), \kappa) \\ &= \underset{0<\kappa\leq\alpha}{\arg\max} \left\{ (1 - c_\kappa + d)\,\kappa - \frac{\kappa^2}{2\alpha} \right\}\end{aligned}$$

which results in

$$\kappa^o = \begin{cases} \alpha(1 - c_\kappa + d) & \text{if } d \leq c_\kappa \\ \alpha & \text{if } d > c_\kappa \end{cases} \tag{10}$$

It can be seen that a full network $\kappa = \alpha$ is implemented if and only if the profitability of the CSPs d is greater or equal than the network costs c_κ. Near the saturation point of the consumer the welfare increase due to the consumer is negligible, so the only relevant forces are the marginal network costs and the marginal profit from the CSPs.

Total welfare is given by

$$W^o = W(q_F^o(\kappa^o), q_B^o(\kappa^o), \kappa^o) = \begin{cases} \frac{1}{2}\alpha\,(1 - c_\kappa + d)^2 & \text{if } d \leq c_\kappa \\ \alpha\left(\frac{1}{2} - c_\kappa + d\right) & \text{if } d > c_\kappa \end{cases} \tag{11}$$

Optimal welfare is increasing in internet affinity α, CSP profitability d and decreasing in network costs c_κ.

4. No sponsored content

In this subsection we will examine the equilibrium outcome given that sponsored content is forbidden. This is necessary to provide a baseline for the ISP and content provider B. Even if sponsored content is allowed, the ISP can always enforce this setting by not offering a contract to content provider B. CSP B on the other hand can simply decide not to accept the contract of the ISP. This situation can't arise by design, as the ISP already incorporates this decision process of B in his decision to built network capacity and offers a corresponding contract.

Assuming sponsored content is forbidden, the timeline can be shortened to

1. The ISP decides to build internet capacity κ.
2. The ISP offers the consumer a take-it-or-leave-it contract with a specific quota q for a fixed fee P.
3. The consumer accepts the contract, if sensible, and creates traffic for B and F. If the consumer does not accept the contract she receives a utility of 0.

Given an accepted contract $(q,\, P)$ with $q \leq \kappa \leq \alpha$, the consumer will thus maximize her utility by optimizing her traffic towards the content providers q_{F} and q_{B}:

$$(q_{\mathrm{F,NSC}}(q),\, q_{\mathrm{B,NSC}}(q)) = \underset{\substack{q_{\mathrm{F}}>0 \\ q_{\mathrm{B}}>0}}{\arg\max}\, U(q_F, q_B, P) \quad \text{given that } q_{\mathrm{F}} + q_{\mathrm{B}} \leq q \tag{12}$$

leading to

$$q_{\mathrm{F,NSC}}(q) = (1-\beta)q, \qquad q_{\mathrm{B,NSC}}(q) = \beta q \tag{13}$$

which can be calculated using Lagrange optimization. The setting of q itself is costless to the ISP. The utility of the consumer, and thus possible price P is monotonously increasing in q. The ISP will thus always set $q = \kappa$. Hence we find

Proposition 1 For any given capacity κ it holds that

$$W_{\mathrm{NSC}}(\kappa) = W(q^o_{\mathrm{F}}(\kappa), q^o_{\mathrm{B}}(\kappa), \kappa)$$

with $W_{\mathrm{NSC}}(\kappa)$ being the total welfare realized in the case of "no sponsored content" (NSC) given κ.
In other words the case of no sponsored content and the first best solution provide the same total welfare with κ given exogenously.

Given exogenous network capacity κ the ISP can calculate the maximal extractable price

$$\bar{P}_{\mathrm{NSC}}(q = \kappa) = \kappa\left(1 - \frac{\kappa}{2\alpha}\right), \tag{14}$$

setting the utility of the consumer to zero. He has to balance this income against the costs of providing the network, thus solving

$$\kappa^*_{\mathrm{NSC}} = \underset{\kappa}{\arg\max}\ \pi_{\mathrm{ISP}}(\kappa) = \underset{\kappa}{\arg\max}\ \left\{\bar{P}_{\mathrm{NSC}}(\kappa) - c_\kappa \cdot \kappa\right\} \tag{15}$$

with κ^*_{NSC} being the implemented network capacity when the ISP can not offer a sponsored content contract to the CSP or does not want to do so. Optimizing κ^*_{NSC} results in

$$\kappa^*_{\mathrm{NSC}} = \alpha(1 - c_\kappa) \tag{16}$$

Proposition 2 It holds that

$$\kappa^*_{\mathrm{NSC}} < \kappa^o \tag{17}$$

i.e. if sponsored content is forbidden, the ISP will always set the network capacity below the optimal network capacity of the first best solution.

Given that the ISP is not allowed to receive additional payments, the CSPs can not subsidize the ISP, so it doesn't matter for the ISP how much welfare is lost on their side. Consequently

the ISP does not internalize the positive externatilities on the CSPs.
It can be seen, that the ISP builds more network capacity with increasing internet affinity α but does not care about the distribution of the traffic, making κ^*_{NSC} independent of β. Unsurprisingly the ISP builds less capacity if the costs rise, and in fact builds no capacity at all in the limit case $c_\kappa \to 1$.

Definition 1 We define **investment distortion** ID to be the difference in total welfare between the first best solution and the best solution achievable given κ

$$\mathrm{ID}(\kappa) = W^o - W(q^o_{\mathrm{F}}(\kappa), q^o_{\mathrm{B}}(\kappa), \kappa)$$

Due to Proposition 1 the investment distortion is the only welfare reducing effect in the NSC case. It holds that

Proposition 3 If there is no sponsored content the investment distortion is strictly increasing in marginal CSP profitability d and consumer internet affinity α and weakly increasing in network costs c_κ as it holds that

$$\mathrm{ID}(\kappa^*_{\mathrm{NSC}}) = \begin{cases} \frac{1}{2} \cdot \alpha \cdot d^2 & \text{if } d \leq c_\kappa \\ \frac{1}{2}\alpha c_\kappa (2d - c_\kappa) & \text{if } d > c_\kappa \end{cases} \tag{18}$$

In contrast to the first best solution, the ISP is not inclined to built more network capacity due to an increase in d, but rather keeps it constant, thus creating a bigger investment distortion.
As long as network costs c_κ are smaller than the marginal CSP profitability d, the first best solution results in the full network independent on the actual value of c_κ. Given that the ISP chooses the network capacity profit maximizing, he will reduce the network capacity κ with increasing c_κ, thus increasing the investment distortion effect.
The internet affinity α of the consumer simply acts as a multiplier to those effects.
Seeing that the ISP clearly lacks the incentive to account for the needs of the CSPs, thus creating the investment distortion, internalizing some of the profits of the CSPs might lead to a higher network and thus to a higher total welfare. One possible instrument might be sponsored content.

5. Introducing Sponsored Content

In this subsection we will solve the complete model with (optional) sponsored content as introduced in Section 2. To do so, we will use backwards induction, starting with the consumer.
The consumer solves, given a contract (q, P) with sufficiently low price P and $q \leq (1-\beta)\alpha$,

$$(q_{\mathrm{F,SC}}(q),\, q_{\mathrm{B,SC}}(q)) = \underset{\substack{q_{\mathrm{F}}>0 \\ q_{\mathrm{B}}>0}}{\arg\max}\; U(q_F, q_B, P) \quad \text{under the condition that} \quad q_{\mathrm{F}} \leq q \tag{19}$$

leading to

$$q_{\mathrm{F,SC}}(q) = q, \qquad q_{\mathrm{B,SC}}(q) = \beta\alpha \tag{20}$$

which means that the complete quota is used for the fringe content provider and the consumer will generate traffic for content provider B according to her saturation point with respect to B. Given that the ISP has already built a specific network capacity κ, he has the choice to either

a) Offer the quota $q = \kappa$ without sponsored content, or

b) Offer the quota $q = \kappa - \beta\alpha$ with sponsored content, with $\beta\alpha$ being the saturation point of the consumer in respect to B.

Offering less quota would unnecessarily reduce the profits of the ISP and offering more would imply that the network stability is no longer warranted.
The de facto effect of sponsored content on the consumer, given that network capacity κ is constant, is that she is "forced" to depart from her optimal traffic allocation. This change implies a lower utility of the consumer and has thus to be countered by a lower price P. We find the ISP will offer

$$\bar{P}_{\mathrm{SC}}(q = \kappa - \beta\alpha) = \frac{\kappa}{1-\beta} - \frac{\kappa^2 + \alpha^2\beta}{2\alpha(1-\beta)} \tag{21}$$

with $\bar{P}_{\mathrm{SC}}(q)$ being the price which sets consumer utility to zero given sponsored content and a given quota q.

Definition 2 We define **choice distortion** to be the difference in optimal contract price P with and without sponsored content, given a fixed network capacity κ, i.e.

$$\mathrm{CD}(\kappa) = \bar{P}_{\mathrm{NSC}}(q = \kappa) - \bar{P}_{\mathrm{SC}}(q = \kappa - \beta\alpha) \tag{22}$$

Proposition 4 It holds for $\kappa < \alpha$ that

$$\mathrm{CD}(\kappa) = \frac{\beta\,(\alpha-\kappa)^2}{2\alpha\,(1-\beta)} > 0 \tag{23}$$

so unless the demand of the consumer is completely met, the choice distortion is always positive.

In other words, given fixed network capacity $\kappa < \alpha$, the price with sponsored content has always to be lower than without sponsored content. Additionally it can be seen, that the choice distortion decreases when increasing network capacity - as the need of the consumer regarding the fringe content is already reasonably met.

This choice distortion is, given network capacity κ, the only welfare distorting effect of sponsored content, since we assumed that both content providers have the same profitability. Combining Proposition 1 and Proposition 4 leads to

Proposition 5 It holds that

$$W^o - W_{\text{SC}}(\kappa) = \text{ID}(\kappa) + \text{CD}(\kappa)$$

with $W_{\text{SC}}(\kappa)$ being the realized total welfare given sponsored content is implemented and network capacity κ is given.
This means, that the investment distortion and the choice distortion make up the difference in total welfare to the first best solution given sponsored content.

As a direct result, should the investment distortion be zero, i.e. $\kappa = \kappa^o$ the choice distortion is the only welfare distorting effect. It also means that even if sponsored content would lead to an optimal network, the choice distortion guarantees that the first best solution is not implemented.

Content provider B observes the built network capacity κ and the reactions of the consumer and is thus able to calculate the maximal price $c_{\text{SC}}(\kappa)$ it is willing to pay for the sponsoring of its content:

$$\begin{aligned} c_{\text{SC}}(\kappa) &= \pi_{\text{B,SC}}(\kappa) - \pi_{\text{B,NSC}}(\kappa) && (24) \\ &= d\beta(\alpha - \kappa) && (25) \end{aligned}$$

The content provider is thus willing to pay more, if the consumer is more interested in its content, it is more profitable and if the network capacity κ is small. The ISP is able to come to the same conclusions and will thus offer c_{SC} as the price CSP B has to pay for the privilege of sponsored content.

Given those intermediary results based on network capacity κ the ISP chooses the network capacity profit optimizing:

$$\arg\max_{\kappa} \pi_{\text{ISP}}(\kappa) = \begin{cases} \pi_{\text{ISP, NSC}}(\kappa) & \text{if } \kappa \leq \beta\alpha \\ \max\{\pi_{\text{ISP, NSC}}(\kappa),\, \pi_{\text{ISP, SC}}(\kappa)\} & \text{if } \kappa > \beta\alpha \end{cases} \tag{26}$$

with

$$\pi_{\text{ISP, NSC}}(\kappa) = \bar{P}_{\text{NSC}}(q = \kappa) - c_\kappa \cdot \kappa \tag{27}$$

and

$$\pi_{\text{ISP, SC}}(\kappa) = \bar{P}_{\text{SC}}(q = \kappa - \beta\alpha) + c_{\text{SC}}(\kappa) - c_\kappa \cdot \kappa \tag{28}$$

being the optimal profits of the ISP in the given case with network capacity κ.

Before we will go into detail regarding the actual network and welfare effects, we will analyze the specific routes the ISP can take:

- In the **No Sponsored Content** (NSC) case the ISP will simply implement the solution of section 4 without sponsored content. This is the case if the profitability d is sufficiently low such that the CSP B isn't able to pay a meaningful payment c_{SC}, thus making sponsored content unattractive to the ISP.

- In the **Sponsored Content** (SC) case the ISP will build a network strong enough to allow sponsored content and make the corresponding arrangements. Here two cases have to be distinguished:
  - In the basic case the consumer is able to satisfy all her needs toward CSP B and can otherwise use the internet for CSP F using her offered quota.
  - In the **Exclusively Sponsored Content** (ESC) case the ISP will build network capacity *only* to the extent that sponsored content is possible. The consumer thus receives a quota of $q = 0$ and the offer to use the content of CSP B unlimitedly for a fixed price. This is the best course of action for the ISP if profitability d and network costs c_κ are sufficiently high. In this case one could imagine, that the CSP B is so profitable, that it simply pays the ISP to build its network infrastructure.

Which scenario is implemented depends on the specific relation between profitability d and network costs c_κ, given that all other parameters are held constant. For a graphical illustration of a numerical example see Figure 2.

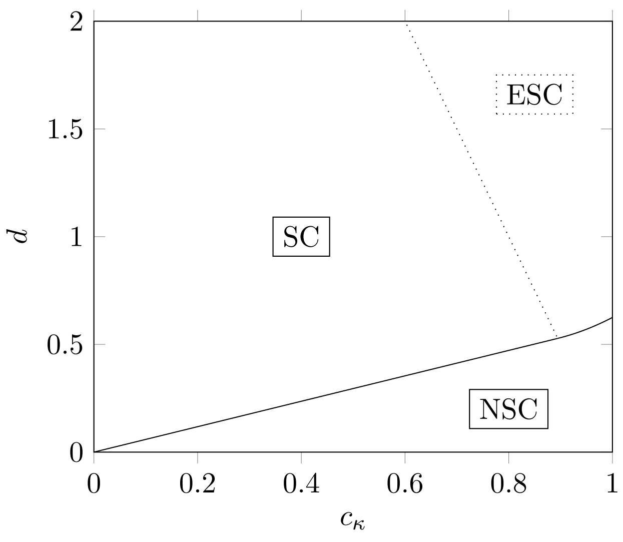

Figure 2: Different optimal decisions of the ISP given network costs c_κ and marginal CSP profitability d with $\beta = 0.2$ and internet affinity $\alpha = 1$. *NSC* is the case where no sponsored content is implemented even if permitted. For all combinations above the NSC area sponsored content is implemented. In the the upper right corner the special situation of exclusively sponsored content (*ESC*), i.e. an offered quota of $q = 0$, occurs.

The following propositions show, that the numerical illustration of Figure 2 is representative for a wide array of parameter combinations. The specific formulas to calculate π_{ISP} and the chosen case can be found in the Appendix with Proposition I.1.

Proposition 6 There exists a minimal profitability $\underline{d}(c_\kappa) > 0$ such that the ISP will implement sponsored content content if and only if it holds that marginal profitability $d > \underline{d}(c_\kappa)$. $\underline{d}(c_\kappa)$ is strictly monotonically increasing in marginal network costs c_κ with $\lim_{c_\kappa \to 0} \underline{d}(c_\kappa) = 0$.

If marginal network costs c_κ are negligible, the optimal provided network is very large and the consumer near her saturation point. The choice distortion of switching is therefore negligible. Even if the CSP is only willing to pay a small fee c_{SC}, this would be sufficient for the ISP to pursue the route of sponsored content.
On the other hand, if marginal network costs c_κ are fairly high, the optimal network capacity in the NSC case is very low. In fact, it can be so low, that more network capacity is needed to even offer sponsored content. Even if that is not the case a low network capacity implies that offering sponsored content leads to a comparatively bigger change in consumer behavior, thus increasing choice distortion. Consequently the ISP will only pursue the route of sponsored content, if he expects a high fee c_{SC} from the content provider B.
CSP B will only agree to sponsored content if the additional traffic offsets the fee c_{SC}. The higher the profitability d, the higher B values the additional traffic and thus the payment c_{SC} increases as well, leading to the implementation of sponsored content. As the threshold for the necessary c_{SC} increases with higher network costs, the minimal necessary profitability $\underline{d}$ increases accordingly as well.

Proposition 7 There exists a minimal profitability $\bar{d}(c_\kappa) \geq \underline{d}(c_\kappa)$ such that the ISP implements exclusively sponsored content if and only if $d > \bar{d}(c_\kappa)$.

Given fixed network costs c_κ the ISP always offers exclusive contracts if the marginal profitability of CSP B is high enough. If the CSP has an extremely high profitability, the utility of the consumer becomes irrelevant as it is much more important to optimize the payment c_{SC}. c_{SC} decreases in network size as the advantage due to the sponsored content contract itself is decreased. The ISP thus tries to generate a situation, where the consumer generates the least amount of traffic for CSP B, so that sponsoring content is as attractive for the CSP as possible.

Proposition 8 There exists a minimal profitability $c_\kappa^* = \sqrt{1-\beta}$ such that the ISP only implements either the case of no sponsored content or the case of exclusively sponsored content for all $c_\kappa > c_\kappa^*$.
For $c_\kappa < c_\kappa^*$ it holds that $\underline{d}(c_\kappa) < \bar{d}(c_\kappa)$, i.e. there exists a range of marginal profitability d with $\underline{d}(c_\kappa) < d \leq \bar{d}(c_\kappa)$ where non exclusive sponsored content is implemented.

If the network is relatively cheap, sponsored content is attractive to the ISP, as he can simply counteract the choice distortion effect by building more network capacity and thus profiting both

from a relatively high P and the payment c_{SC}. With higher network costs however, the marginal increase in P is not sufficient to justify an increase in network capacity over the necessary minimal network capacity $\kappa = \beta\alpha$. Consequently, the ISP switches directly from the NSC case to the ESC case.
The exact position of c_κ^* depends on the consumer preference β: If β is fairly low, i.e. the consumer is not that interested in B, implementing the case of exclusively sponsored content is a very drastic step. This is only justified by extremely high network costs in combination with a relatively high profitability d. Note, that the relatively high profitability d follows directly out of the strict monotonous increase of $\underline{d}(c_\kappa)$.

6. Network Sizes

We have argued that sponsored content might be a viable instrument to mitigate the effect of the investment distortion. In this section we will thus examine the effects on the realized and from the view of the ISP optimal network capacity κ^*. A representative figure given comparatively small network costs $c_\kappa < \tilde{c}_\kappa = 1 - \beta$ can be found with Figure 3.

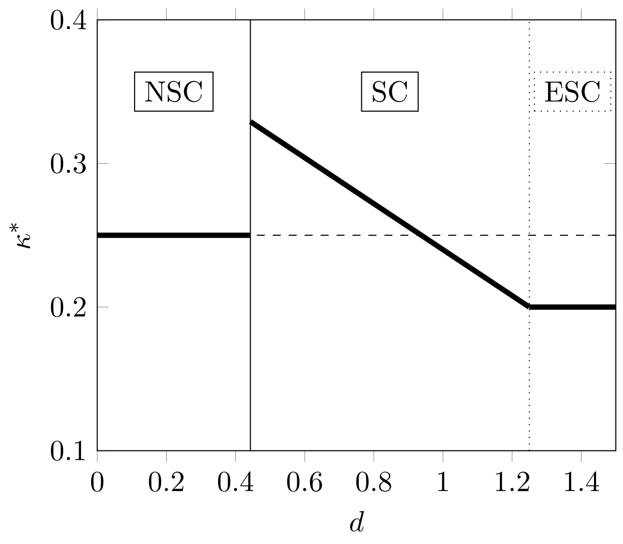

Figure 3: Realized network capacity κ^* given marginal CSP profitability d with $\beta = 0.2$, internet affinity $\alpha = 1$ and network costs $c_\kappa = 0.75$, i.e. $c_\kappa < \tilde{c}_\kappa = 1-\beta$. The dashed line denotes the case where sponsored content is forbidden. One can see, that with increasing profitability d at one point sponsored content is implemented, which results in a discrete jump in network capacity. With increasing profitability d the network capacity decreases until the case of exclusive sponsored content is reached.

Proposition 9 Given that $c_\kappa < \tilde{c}_\kappa = 1 - \beta$, for comparative statics in the CSP profitability d, it holds that

1. Within the case of non-exclusively sponsored content, the implemented network capacity

κ^* is strictly monotonously decreasing, as it holds here that

$$\kappa^*_{\text{SC}} = \alpha(1 - (1-\beta)(c_\kappa + d\beta)) \tag{29}$$

2. The network capacity κ^* of both the NSC and ESC case is constant with

$$\kappa^*_{\text{NSC}} = \alpha(1 - c_\kappa)$$
$$\kappa^*_{\text{ESC}} = \alpha\beta$$

   As it holds that $c_\kappa \leq 1 - \beta$ it follows that $\kappa^*_{\text{NSC}} > \kappa^*_{\text{ESC}}$.

3. There exists a minimal profitability $\underline{d}(c_\kappa) < d^*(c_\kappa) < \bar{d}(c_\kappa)$, i.e. within the case of non-exclusive sponsored content, where sponsored content leads to less network capacity for all $d > d^*$

4. There is a discrete jump in network capacity κ^* of

$$\alpha c_\kappa(1 - \sqrt{1-\beta}) > 0 \tag{30}$$

   when switching from the NSC to the SC case.

5. The transition from the SC case to the ESC case is continuous.

As was argued in section 5, the ISP has to weigh up the payment c_{SC} against the resulting choice distortion. Both streams depend on the network capacity:

- An increase in network capacity implies that the choice distortion is *decreased*, as the utility difference for the consumer between a contract with and without sponsored content shrinks. The extractable price P thus *increases* with an increase in network capacity.

- An increase in network capacity implies that the payment c_{SC} is *decreased* . More network capacity implies less additional traffic which has to be attributed to the sponsoring of the content itself.

The ISP would like to set the network to a small value if the payment c_{SC} is important relative to the choice distortion and increase network capacity otherwise.
The consumer does not care about the profits of the content providers, holding the choice distortion constant with changes in d. However the payment c_{SC} is strongly sensitive in the profitability d with a higher profitability allowing a higher payment.
In other words an increase in d means that the payment c_{SC} becomes more important to the ISP relative to the price of the consumer contract, thus leading to a decrease in network capacity. This effect continues with an increase in d until the ESC case is reached, where the network capacity stagnates. A representation of the importance of the two income streams can be found with Figure 4.

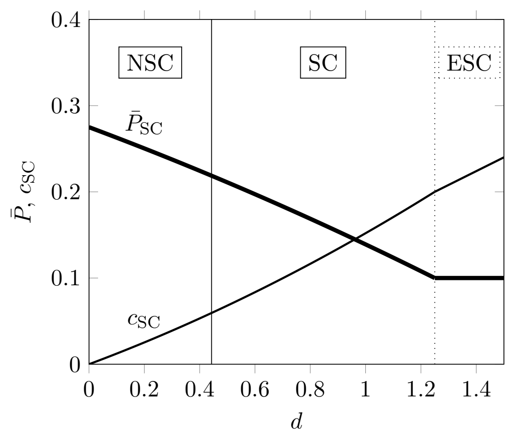

Figure 4: Numerical comparison of earnings of the ISP due to consumer price $\bar{P}_{\mathrm{SC}}$, with sponsored content (thick line) and contract price between CSP B and the ISP c_{SC} (thin line) given marginal CSP profitability d. We used CSP affinity $\beta = 0.2$, internet affinity $\alpha = 1$ and network costs $c_\kappa = 0.75$. In all cases network capacity κ has been chosen to be optimal for the ISP, i.e. $\kappa = \kappa *$.
One can see, that with increasing profitability d the importance of the contract between CSP B and the ISP increases in relation to the payment of the consumer.

Analyzing increased network costs above the threshold of $\tilde{c}_\kappa = 1 - \beta$ gives us Proposition 10.

Proposition 10 Given that $c_\kappa > \tilde{c}_\kappa = 1 - \beta$ and the implementation of sponsored content, the built network capacity is higher than with forbidden sponsored content.

Given that $c_\kappa > \tilde{c}_\kappa$ and forbidden sponsored content, the ISP would like to build network capacity below the saturation point of the consumer in respect to content of CSP B. The implementation of sponsored content is thus only possible if the network capacity is increased at least to this level. Depending on the specific combination of network costs c_κ, marginal profitability of CSP B d and the preference for its content β non-exclusive sponsored content will be implemented or - in the case of extremely high network costs - a direct switch to exclusive sponsored content might take place.
Generally, if we increase c_κ, the network capacity of the ESC case stays constant, the network capacity of the NSC case decreases and the SC area shrinks as can be seen in Figure 5, where only the value of c_κ has been changed in relation to the settings of Figure 3.

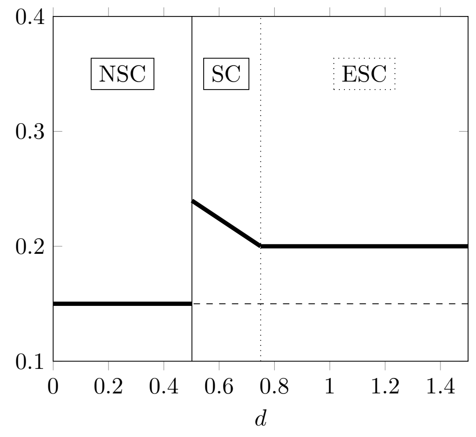

Figure 5: Network capacity κ given marginal CSP profitability d with $\beta = 0.2$, internet affinity $\alpha = 1$ and network costs $c_\kappa = 0.85$, i.e. $1 - \beta = \tilde{c}_\kappa < c_\kappa$. The dashed line denotes the case where sponsored content is forbidden.

Regarding the jump in network capacity when implementing sponsored content it should be kept in mind, that choice distortion can be mitigated by simultaneously increasing the network capacity. If network costs c_κ are high, the NSC case provides a low network capacity. Switching to sponsored content with a fixed demand, a low network capacity implies an especially high choice distortion, thus making an higher increase necessary.
If no corner-solution-scenario is implemented, the increase in network capacity is directly paid for by the CSP, as Proposition 11 shows.

Proposition 11 We define $\kappa_1 = \kappa^*(\underline{d}(c_\kappa))$ to be the profit maximizing network capacity of the non-sponsored content case and $\kappa_2 = \lim_{d \to \underline{d}(c_\kappa)+} [\kappa^*(d)]$ to be the profit maximizing network capacity after switching to sponsored content by increasing profitability d incrementally from the case of non-sponsored content to the case of non-exclusively sponsored content. It holds that the costs of the increase in network capacity equals the payment c_{SC} from CSP B to the ISP, i.e. formally

$$c_\kappa \cdot (\kappa_2 - \kappa_1) = c_{\text{SC}}\,(\kappa_2) \quad \text{given} \quad c_\kappa \leq c_\kappa^* = \sqrt{1 - \beta}$$

From the view of the ISP one necessary condition to switch away from non-sponsored content is, that there are no effective additional costs to do so. The difference in costs of optimal network capacities has thus to be paid by the contract price c_{SC}.
The effective utility of the consumer is not changed by the behavior of the ISP: While she receives a higher quota due to the increase in network capacity, having a contract with sponsored content simultaneously lowers her utility due to not being able to structure her traffic optimally. The increase in network capacity cancels the higher quota, leaving the effective price constant.

The same does not hold in the case of changing directly to the case of non-exclusive sponsored content, as here the ISP is not as flexible with his optimal network and thus price $\bar{P}_{\text{SC}}$, as here always the network $\kappa = \alpha\beta$ is implemented. The consumer might thus enjoy so much more network capacity, that she shoulders some of the network costs.

Even though some of the gains of B are internalized by allowing sponsored content, this does not hold for the possible earnings of the fringe content provider F.

Proposition 12 When allowing sponsored content the realized network capacity k^* is smaller than the welfare optimizing network capacity k^o.

To summarize one can say, that sponsored content, as well as the case of forbidden sponsored content never achieves the desired welfare-optimizing network capacity. However, given very high network costs c_κ and/or a sufficiently low profitability d sponsored content might increase the built network. Quite a few scenarios exist though, were allowing sponsored content not only implies the welfare decreasing effect of increasing choice distortion but also directly decreases the build network, thus increasing the investment distortion as well.

7. Welfare analysis

Allowing and implementing sponsored content has the following effects on the different parties:

1. The consumer changes her internet usage, thus creating choice distortion. Per design the ISP sets the utility of the consumer to zero, thus negating all direct effects of the consumer and transferring the utility loss due to choice distortion to the ISP.

2. The fringe content provider F is never explicitly addressed in the optimization process of the ISP, leading to:

   Proposition 13 Given that sponsored content is allowed and implemented the fringe content provider F always loses profits relative to the situation of forbidden sponsored content. In other words given that the ISP implements sponsored content it holds that

$$\pi^*_{\mathrm{F,NSC}} = d \cdot q_{\mathrm{F,NSC}}(q = \kappa^*_{\mathrm{NSC}}) > d \cdot q_{\mathrm{F,SC}}(q = \kappa^* - \beta\alpha) = \pi^*_{\mathrm{F,SC}} \tag{31}$$

3. Regarding content provider B it should be noted, that all effects of the payment c_{SC} are neutral in respect to total welfare. Given that the ISP makes the take-it-or-leave-it offer, the profit of B is always limited to the profit in the NSC case given built network capacity k^*, as all excess profit is taken by the ISP. Therefore, B profits exactly than, if sponsored content leads to more network capacity ($k^* > k^*_{\mathrm{NSC}}$) and makes less profits, if it leads to less network capacity ($k^* < k^*_{\mathrm{NSC}}$).

4. The ISP never loses by being presented with the additional option of sponsored content, as he can always choose to not draw this option. The effects on the ISP are the balance act between the payment c_{SC} and the choice distortion and have already been discussed in the previous chapters.

To reiterate the following effects influence total welfare:

a) The choice distortion leads to a decrease in the profits of the ISP.

b) The investment distortion depends on the chosen network capacity - a higher network capacity κ^* relative to the network capacity in the NSC model κ^*_{NSC} implies higher profits for the sum of the CSPs, a smaller choice distortion but higher network costs.

Definition 3 We define the total welfare given sponsored content is forbidden as

$$W^*_{\mathrm{NSC}} := W_{\mathrm{NSC}}(k^*_{\mathrm{NSC}})$$

and total welfare given sponsored content is allowed as

$$W^* := \begin{cases} W_{\mathrm{NSC}}(k^*) & \text{if the ISP does not implement sponsored content} \\ W_{\mathrm{SC}}(k^*) & \text{if the ISP does implement sponsored content} \end{cases}$$

The rather complicated explicit formulas can be found with Proposition I.4 in the Appendix.

Proposition 14 Having a higher network due to sponsored content is a necessary but not sufficient condition that total welfare increases when allowing sponsored content. It holds that

$$W^* > W^*_{\mathrm{NSC}} \quad \Rightarrow \quad \kappa^* > \kappa^*_{\mathrm{NSC}} \tag{32}$$

but the opposite direction does not hold.

As a direct consequence, in all situations where where sponsored content leads to a decrease in network capacity, e.g. as seen in Proposition 9 with relatively low network costs $c_\kappa < 1 - \beta$ and $d > d^*(c_\kappa)$, total welfare is reduced.

Proposition 15 Given that a non exclusive-sponsored content contract would be implemented, sponsored content leads to a loss in total welfare if the consumer preference β towards content of content provider B is less than $\tilde{\beta} = \frac{\sqrt{5}-1}{2} \approx 0.61$.

In other words under all conditions, where there is a comparatively small content provider B and sponsoring does not lead to a “CSP B only” network, the total welfare is reduced by allowing sponsored content. The investment distortion might be somewhat mitigated by a higher network capacity, but the choice distortion works against it, especially if the content of CSP B is not that well liked. However, should the consumer indeed prefer to predominately generate traffic for B, the choice distortion is not as bad, as the consumer uses most of her traffic quota for B anyway.

Proposition 16 There exists a minimal network cost $\tilde{c}_\kappa = 1 - \beta$ and a minimal content profitability $\bar{\bar{d}}(\tilde{c}_\kappa) > 0$, where for all combinations c_κ, d with $c_\kappa > \tilde{c}_\kappa$ and $d > \bar{\bar{d}}(c_\kappa)$ it holds that allowing sponsored content increases total welfare.

Note that $c_\kappa > (1 - \beta)$ implies per Proposition 10 that sponsored content leads to more network capacity.
If the content provider is so profitable, that it effectively pays the ISP to build up the network structure for its content, sponsored content indeed leads to an increase in total welfare. In this parameter subset the ISP always chooses to implement exclusively sponsored content, as the payment of the consumer becomes comparatively irrelevant. As a direct result the choice distortion no longer is the driving factor: Allowing sponsored content allows the ISP optimize the joint income of both CSP B and himself. Consequently, the investment distortion in this instance is greatly reduced.

8. Robustness check: Giving Bargaining Power to the Content Provider

One of the main reasons leading to welfare loss follows out of the exploitation of the content provider by the ISP due to throttled network capacity. The main force behind this behavior is the possibility of the ISP to dictate the details of the contract between the ISP and the content provider. One might argue, that a more realistic setting would be, that the ISP builds up network capacity and is approached by the content provider. The new timeline is thus

1. The ISP decides to build internet capacity κ.
2. Content provider B offers the ISP a take-it-or-leave-it offer to get its content sponsored.
3. The ISP can accept or decline that offer.
4. The ISP offers the consumer a take-it-or-leave-it contract with a specific quota q for a fixed fee P.
5. The consumer accepts the contract, if sensible, and creates traffic for B and F.

Proposition 17 If the content provider offers the take-it-or-leave-it offer, the ISP will always built the network capacity k^*_{NSC}, optimal for the case where sponsored content is forbidden.

The total welfare is reduced by the amount of the choice distortion $\mathrm{CD}(\kappa^*_{\mathrm{NSC}})$ when implementing sponsored content.

Given that the network has already been built, the content provider has no incentives to pay more than necessary to the ISP and will therefore structure the offer in a way, that the ISP barely accepts it. The ISP will therefore make the same amount of profits as in the NSC case. Consequently he will optimize the network capacity identical to the scenario, where sponsored content is forbidden. Here the ISP has no incentive to decrease the investment distortion, as he can't internalize the corresponding profits. Consequently we do not observe the increased network capacity of Propositions 9 and Proposition 10.

On the other hand the choice distortion still happens when implementing sponsored content. Assuming that both content providers have identical profitability d, changing the content from the fringe content provider to the big content provider is welfare neutral, but the consumer is still hurt by not being able to consume the content in her optimal distribution.

While in the main model the ISP internalized the choice distortion and had to balance between the payment of the content provider and the choice distortion, here the choice distortion is simply passed on to the content provider. CSP B will offer the contract if its expected additional traffic is greater than the choice distortion and will thus still be present in the total welfare.

9. Related Literature

An overview of the different effects of (non-existing) net neutrality in various forms can be found with Greenstein, Peitz, and Valletti (2016). For an extensive overview of existing literature regarding net neutrality see Krämer, Wiewiorra, and Weinhardt (2013).

Focusing on the specific notion of actually implemented zero-rating, one can find with Yoo (2016) a recent review of several zero-rating programs. He argues, that zero rating has to be judged on a case by case basis and that blanket statements are sub-optimal. An overview on the regulatory implementation of zero rating worldwide can be found with Marsden (2016). Saenz De Miera Berglind (2016) conducted an empirical study on the effects of zero-rating. He finds that zero-rating increases estimated consumer surplus.

Regarding zero rating and more specifically sponsored content several articles exists which use verbal argumentation to point out potential benefits or warn of potential dangers. Notable examples are R. S. Lee and T. Wu (2009), Eisenach (2015), Rogerson and Charles (2016), as well as Brake (2016) arguing largely in favor of sponsored content. They adduce positive effects as increased economic efficiency, increased competitiveness between internet service providers and the possibility to penetrate otherwise too costly markets. In particular this last argument will be strengthened by our model.

On the other side, Schewick (2015) takes a general stance against zero-rating and Schewick (2016) argues in detail why the service “Binge On” from T-Mobile US (T-Mobile US, 2017a) can be quite harmful, even though it is arguably constructed in a way that every video streaming provider can attach itself to this program without costs.

In the following paragraphs notable theoretic models are provided. Our model differs structurally insofar as all the following models either do not examine network capacity at all or assume within their welfare analysis that it is given exogenously and thus do not find the effect, that the ISP has an incentive to limit his network.
Zhang, W. Wu, and Wang (2015) model, given a fixed network capacity, sponsoring costs and a monopolistic ISP the effects of sponsored content with a two-stage Stackelberg Game: First the ISP decides on the contract details for the consumer and afterwards the CSPs can decide whether they want to participate in the sponsoring scheme. They find that while sponsored content can enlarge the unbalance in revenue distribution between content providers, at least in the short run consumers win through sponsored content independent on the congestion of the network.
Ma (2014) examines a monopolistic ISP with a quantity of content service providers each bringing in their own consumers. The number of consumers depends on the per-byte charge of the ISP, which can in part be subsidized by the CSP. The actual usage, and thus revenue of the ISP, depends on the congestion of the built network. Ma (2014) finds that in particular profitable CSPs will subsidize their content, thus increasing efficiency of the market and both the profit margins of the ISP as well as total welfare, at least in presence of pricing regulations.
Caron, Kesidis, and Altman (2010) model consumer demand in a world with multiple ISPs and CSPs through a consumer demand function, which depends on the price the consumer pays per byte to both the ISP and the CSP. To determine how the choices of content of the consumers are divided, Caron, Kesidis, and Altman (2010) account for consumer loyalty. Both the ISP and the CSP are allowed to make side payments to improve their situation. They find that paradoxically side payments can be detrimental to the receiving side by reducing the Nash equilibrium revenues.
Andrews, Özen, et al. (2013) model a single CSP and a single ISP with a Stackleberg Game, where the ISP can set the pricing parameters and the CSP can determine a maximum number of views which can be sponsored. It is assumed, that sponsored views will always be used, while non-sponsored views will or will not be used based on chance. Both, the situation where the consumer pays per byte and a fixed quota are analyzed. The demand of the consumer is modeled via a random variable of potential views. They show, that a coordinating contract in their setting can be developed which maximizes system profits.
Andrews (2013) introduces a general frame work for a monopolistic ISP and several CSPs and Users, where the CSP can decide to sponsor some data packets but not others and the ISP has the final decision on whether he will allow this or not. This framework does not cover network investments.
Somogyi (2017) uses a monopolistic ISP but two potentially sponsoring, competing CSPs and a

single outsider CSP. The consumers are assumed to be either capped by their allowed capacity or by their bliss point and anticipate congestion of the exogenously available network. Somogyi (2017) calculate that within their model the ISP can offer sponsored content contracts either if the content is very attractive or very unattractive but not in the intermediate region.
Supported by empirical data and numerical simulation, Joe-Wong, Ha, and Chiang (2015) argues that sponsored content favors less cost-constrained content providers and more cost-constrained consumers. They use a model with heterogeneous consumers, which pay for traffic on a per byte basis and receive this traffic in the form of both actual content and advertisement.

The following models do determine network capacity endogenously but not by a strategic decision of the ISP. It is assumed, that the ISP has access to an infinite network, but has to pay a marginal price for each byte of traffic.
Jullien and Sand-Zantman (2012) assume a mass of content providers which can be separated in two groups: low and high. High content providers are characterized by offering content which is valued at a higher price as the marginal network costs. Without sponsored content consumers can not discriminate between those contents, as they pay the same for traffic of both kinds. Sponsored content thus can be welfare enhancing as it allows to direct the focus of the consumer to higher valued content.
Zhang and Wang (2014) calculate which content providers and to which extent they would like to participate. They find that whether big or small content providers profit from sponsored content depends on whether one takes a short term view with fixed market rates or a long term view. Their article is extended by a empirical estimation of their parameters with Andrews, Bruns, and H. Lee (2014).

While the other articles assume complete knowledge of all participants, Andrews, Jin, and Reiman (2016) assume that this does not necessarily have to be the case at all times. Their focus lies on devising a pricing strategy for the ISP to incentivize the CSPs to truthfully report their parameters.

With a similar structure as my paper Pil Choi and Kim (2010) focuses on the dilemma of the ISP to optimize between the fee paid by the content provider versus the fee paid by the consumer and the subsequent effects on infrastructure investments. In contrast to my paper, however, they don't examine the effects of zero-rating with undiscriminating traffic: Here, the content provider sponsors a favorable speed to increase Quality-of-Service in a tiered, and therefore not neutral internet. Modeling network congestion using queuing theory they conclude that, similar to my paper, a low profitability of the content providers leads the ISP to prefer a neutral internet, while a higher profitability has ambiguous results.

10. Concluding Remarks

We assumed, that a content provider can pay the monopolistic internet service provider for the privilege of having its content sponsored. A contract with sponsored content has the feature, that this sponsored content does not count toward the data cap of the consumer.

Allowing sponsored content can lead to more infrastructure investments due to internalizing possible profits of the content provider. With sponsored content allowed, the ISP has to juggle two income streams - the traditional payment by the consumers and the payment from the content provider, who expects more traffic due to the sponsoring of his content. A higher network capacity implies more willingness to pay from the consumers, but a lower willingness to pay from the content provider, as the comparative advantage due to sponsored content shrinks. A high profitability of the content provider can thus lead to a lower network capacity due to the ISP trying to maximize this income stream. With a very high profitability and comparatively high network costs this can result in an internet, where the consumer de facto only pays for (unlimited) access to the sponsored content but does not receive any regular internet access.

By implementing sponsored content, our model predicts total welfare to be reduced in a wide array of cases. Given that we have a relatively normal "broadband" setting, i.e. some content might be sponsored for a sizable but not dominating content provider, total welfare decreases when allowing sponsored content. In the case of very high network costs and comparatively high profitability of the sponsoring content provider however, sponsored content might be beneficial. One might think about rural areas in less developed nations, where sponsored content might allow a big content provider, or a conglomerate thereof, to open up this market. This type of sponsored infrastructure might be profitable for the ISP, the content providers and the consumers who now have access to the internet which they otherwise wouldn't have. In developed areas one might justifiably be wary of the effects of sponsored content, especially given that a lot of negative effects are beyond the scope of this paper. For example one might argue, that sponsored content can lead to a monopolisation of the content provider and thus to market distorting effects, or that consumers are heterogenous and therefore suffer to a greater extent due to the lack of options as has been modelled within this paper.

The model might be extended by modeling the consumer(s) with heterogeneous preferences and the explicit wish for variety in their internet usage. I would expect, that this strengthens the negative effects of sponsored content. Similarly, heterogeneous consumers might make it sensible to allow the ISP to offer different kind of contracts, with changing prices, quotas and with or without sponsored content. The market of the EU might be better modelled by having a small number of internet service providers bidding to gain access to the consumers. One might supplement the model by using differing marginal profitabilities over an array of content providers as well as modeling efficiency effects of content providers due to increased traffic. It could be interesting to see how the concept of using sponsored content as a "loss-leader" for underdeveloped areas fares within this framework by letting the satisfaction point of the consumer depend on the traffic of previous periods and having multiple network investment decisions.

References

airtel India (2015). *Airtel partners with Google to lead India broadband growth story.* URL: `http://www.airtel.in/about-bharti/media-centre/bharti-airtel-news/telemedia/airtel-partners-with-google-to-lead-india-broadband-growth-story` (visited on 04/30/2015).

Andrews, Matthew (2013). “Implementing sponsored content in wireless data networks”. In: *Communication, Control, and Computing (Allerton), 2013 51st Annual Allerton Conference.* IEEE, pp. 1208–1212. DOI: `10.1109/Allerton.2013.6736663`.

Andrews, Matthew, G. Bruns, and Hyoseop Lee (2014). “Calculating the benefits of sponsored data for an individual content provider”. In: *2014 48th Annual Conference on Information Sciences and Systems (CISS)*, pp. 1–6. DOI: `10.1109/CISS.2014.6814145`.

Andrews, Matthew, Yue Jin, and Martin I Reiman (2016). “A truthful pricing mechanism for sponsored content in wireless networks”. In: *Computer Communications, IEEE INFOCOM 2016-The 35th Annual IEEE International Conference.* IEEE, pp. 1–9. DOI: `10.1109/INFOCOM.2016.7524559`.

Andrews, Matthew, Ulaş Özen, et al. (2013). “Economic models of sponsored content in wireless networks with uncertain demand”. In: *Computer Communications Workshops (INFOCOM WKSHPS), 2013 IEEE Conference.* IEEE, pp. 345–350. DOI: `10.1002/9781118899250.ch10`.

AT&T (2017). *Want to become a Sponsored Data provider?* URL: `https://www.att.com/att/sponsoreddata/en/index.html` (visited on 12/08/2017).

Brake, Doug (2016). “Mobile Zero Rating: The Economics and Innovation Behind Free Data”. In: *Net Neutrality Reloaded: Zero Rating, Specialised Service, Ad Blocking and Traffic Management.* Ed. by Luca Belli, pp. 132–154. ISBN: 978-8-5632-6576-0.

Caron, Stephane, George Kesidis, and Eitan Altman (2010). “Application Neutrality and a Paradox of Side Payments”. In: *Proceedings of the Re-Architecting the Internet Workshop.* ReARCH ’10. Philadelphia, Pennsylvania: ACM, 9:1–9:6. ISBN: 978-1-4503-0469-6. DOI: `10.1145/1921233.1921245`.

Eisenach, Jeffrey A (2015). “The Economics of Zero Rating”. In: URL: `http://www.nera.com/content/dam/nera/publications/2015/EconomicsofZeroRating.pdf` (visited on 12/12/2017).

European Commission (2017). *Open Internet.* URL: `https://ec.europa.eu/digital-single-market/en/open-internet-net-neutrality` (visited on 12/08/2017).

Federal Communications Commission (2015). “Protecting and Promoting the Open Internet”. In: *Federal Register* 80.FR 19737. URL: `https://www.federalregister.gov/articles/2015/04/13/2015-07841/protecting-and-promoting-the-open-internet` (visited on 12/08/2017).

Fiegerman, Seth (2017). *FCC votes to move forward with net neutrality rollback.* URL: `http://money.cnn.com/2017/05/18/technology/fcc-net-neutrality-vote/index.html` (visited on 12/08/2017).

FightForTheFuture (2015). *Sept. 10th is the Internet Slowdown.* URL: `https://www.battleforthenet.com/sept10th/` (visited on 12/08/2017).

Goldhammer, Klaus, André Wiegand, and Mathias Birkel (2016). *Marktstudie Zero-Rating.* German. Goldmedia GmbH Strategy Consulting. URL: `https://www.blm.de/files/pdf1/goldmedia-marktstudie-zero-rating-2016-3.pdf`.

Greenstein, Shane, Martin Peitz, and Tommaso Valletti (2016). “Net neutrality: A fast lane to understanding the trade-offs”. In: *The Journal of Economic Perspectives* 30.2, pp. 127–149. DOI: `10.1257/jep.30.2.127`.

Gustin, Sam (2016). *India Just Banned Zero-Rating. Your Move, FCC.* URL: `https://motherboard.vice.com/en_us/article/indias-new-open-internet-law-is-stronger-than-the-united-states` (visited on 12/08/2017).

Joe-Wong, Carlee, Sangtae Ha, and Mung Chiang (2015). “Sponsoring mobile data: An economic analysis of the impact on users and content providers”. In: *Computer Communications (INFOCOM), 2015 IEEE Conference.* Kowloon, Hong Kong: IEEE. DOI: `10.1109/INFOCOM.2015.7218528`.

Jullien, Bruno and Wilfried Sand-Zantman (2012). “Internet Regulation, Two-Sided Pricing, and Sponsored Data”. In: *TSE Working Paper* 12.327. revised March 2017.

Krämer, Jan, Lukas Wiewiorra, and Christof Weinhardt (2013). “Net neutrality: A progress report”. In: *Telecommunications Policy* 37. 9, pp. 794–813. DOI: `10.1016/j.telpol.2012.08.005`.

Lee, Robin S and Tim Wu (2009). “Subsidizing creativity through network design: Zero-pricing and net neutrality”. In: *The Journal of Economic Perspectives* 23.3, pp. 61–76. DOI: `10.1257/jep.23.3.61`.

Ma, Richard T.B. (2014). “Subsidization competition: Vitalizing the Neutral Internet”. In: *Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies.* ACM. Sydney, Australia, pp. 283–294. DOI: `10.1145/2674005.2674987`.

Marsden, Christopher T. (2016). “Comparative Case Studies in Implementing Net Neutrality: A Critical Analysis of Zero Rating”. In: *SCRIPTed* 13, p. 1. DOI: `10.2966/scrip.130116.1`.

Pil Choi, Jay and Byung-Cheol Kim (2010). “Net neutrality and investment incentives”. In: *The RAND Journal of Economics* 41.3, pp. 446–471. DOI: `10.1111/j.1756-2171.2010.00107.x`.

Rogerson, William P and E Charles (2016). “The economics of data caps and free data services in mobile broadband”. In: URL: http://www.ctia.org/docs/default-source/default-document-library/081716-rogerson-free-data-white-paper.pdf.

Saenz De Miera Berglind, Oscar (2016). “The Effect of Zero-Rating on Mobile Broadband Demand: An Empirical Approach and Potential Implications”. In: *International Journal of Communication* 10, p. 18. URL: http://ijoc.org/index.php/ijoc/article/view/4651.

Sasso, Brandon (2014). *On Net Neutrality, Verizon Leads Push for ’Fast Lanes’.* URL: http://www.nationaljournal.com/tech/on-net-neutrality-verizon-leads-push-for-fast-lanes-20140718 (visited on 05/04/2015).

Schewick, Barbara van (2015). “Network Neutrality and Zero-Rating”. In: *Attachment to Barbara van Schewick’s Ex Parte in the Matter of Protecting and Promoting the Open Internet submitted February* 19.2015, pp. 14–28. URL: https://cyberlaw.stanford.edu/files/publication/files/vanSchewick2015NetworkNeutralityandZerorating.pdf.

– (2016). “T-Mobile’s Binge On violates key net neutrality principles”. In: URL: https://cyberlaw.stanford.edu/downloads/vanSchewick-2016-Binge-On-Report.pdf.

Schuhmacher, Merlin (2016). *Spotify: Telekom stampft unlimitiertes Datenvolumen für Musikstreaming ein.* German. URL: https://www.heise.de/newsticker/meldung/Spotify-Telekom-stampft-unlimitiertes-Datenvolumen-fuer-Musikstreaming-ein-3280776.html (visited on 03/21/2017).

Somogyi, Robert (2017). *The Economics of Zero-Rating and Net Neutrality.* Working Paper, Version November 27, 2017. Université catholique de Louvain, Center for Operations Research and Econometrics (CORE). URL: https://sites.google.com/site/robertsomogyi/Zero_rating_Somogyi.pdf.

The White House (2015). *Net Neutrality - President Obama’s Plan for a Free and Open Internet.* URL: https://www.whitehouse.gov/net-neutrality (visited on 04/29/2015).

T-Mobile US (2017a). *BINGE ON. Video now streams FREE.* URL: https://www.t-mobile.com/offer/binge-on-streaming-video.html (visited on 12/13/2017).

– (2017b). *Stop burning data when you stream music.* URL: https://www.t-mobile.com/offer/free-music-streaming.html (visited on 12/13/2017).

Welsh de Grimaldo, Susan (2015). *Measuring the Success of Sponsored Data.* Strategy Analytics. URL: https://www.strategyanalytics.com/access-services/service-providers/service-providers-strategies/reports/report-detail/measuring-the-success-of-sponsored-data.

Wu, Tim (2003). “Network Neutrality, Broadband Discrimination”. In: *Journal of Telecommunications and High Technology Law* 2, pp. 141–160. DOI: 10.2139/ssrn.388863.

Yoo, Christopher S (2016). “Avoiding the Pitfalls of Net Uniformity: Zero Rating and Nondiscrimination”. In: *Review of Industrial Organization* 50 (4), pp. 509–536. DOI: 10.1007/s11151-016-9555-7.

Zhang, Liang and Dan Wang (2014). “Sponsoring content: Motivation and pitfalls for content service providers”. In: *2014 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pp. 577–582. DOI: 10.1109/INFCOMW.2014.6849295.

Zhang, Liang, Weijie Wu, and Dan Wang (2015). “Sponsored data plan: A two-class service model in wireless data networks”. In: *ACM SIGMETRICS Performance Evaluation Review*. Vol. 43. 1. ACM. Portland, Oregon, USA, pp. 85–96. ISBN: 978-1-4503-3486-0. DOI: 10.1145/2745844.2745863.

I. Appendix

Proof of Proposition 1. This follows directly out of

$$q_{\mathrm{F,NSC}}(\kappa) = q_F^o(\kappa) \qquad q_{\mathrm{B,NSC}}(\kappa) = q_B^o(\kappa) \tag{33}$$

and the fact that any chosen price of the ISP is welfare neutral. □

Proof of Proposition 2. The utility of the consumer is

$$U(q_F, q_B) = q_F + q_B - \frac{1}{2(1-\beta)\alpha} q_F^2 - \frac{1}{2\beta\alpha} q_B^2 - P \tag{34}$$

with

$$(q_{\mathrm{F,NSC}}(q),\, q_{\mathrm{B,NSC}}(q)) = ((1-\beta)q,\, \beta q) \tag{35}$$

Solving

$$U(q_{\mathrm{F,NSC}}(q), q_{\mathrm{B,NSC}}(q)) \overset{!}{=} 0 \tag{36}$$

leads to

$$\bar{P}_{\mathrm{NSC}}(q) = q\left(1 - \frac{q}{2\alpha}\right) \tag{37}$$

The ISP will chose $q = \kappa$ because $\bar{P}_{\mathrm{NSC}}(q)$ is increasing in q, given that $q \le \kappa \le \alpha$ which holds true per assumption.
The ISP thus optimizes his profit by chosing κ

$$\pi_{\mathrm{ISP}}(\kappa) = \bar{P}_{\mathrm{NSC}}(\kappa) - c_\kappa \cdot \kappa \tag{38}$$

$$= \kappa\left(1 - \frac{\kappa}{2\alpha}\right) - c_\kappa \cdot \kappa \tag{39}$$

where it holds that

$$\frac{\partial \pi_{\mathrm{ISP}}(\kappa)}{\partial \kappa} = 1 - \frac{\kappa}{\alpha} - c_\kappa \overset{!}{=} 0 \tag{40}$$

resulting in

$$\kappa^*_{\mathrm{NSC}} = (1 - c_\kappa)\alpha \tag{41}$$

For the first best solution it holds that

$$\kappa^o = \begin{cases} \alpha(1 - c_\kappa + d) & \text{if } d \le c_\kappa \\ \alpha & \text{if } d > c_\kappa \end{cases} \tag{42}$$

This means the proposition holds iff

$$\kappa^o - \kappa^*_{\mathrm{NSC}} = \begin{cases} \alpha d & \text{if } d \le c_\kappa \\ \alpha c_\kappa & \text{if } d > c_\kappa \end{cases} > 0 \tag{43}$$

which is always the case due to $d > 0$, $c_\kappa > 0$ and $\alpha > 0$. □

Proof of Proposition 3. Keeping in mind Proposition 1 we can directly calculate

$$\begin{aligned} \mathrm{ID}(\kappa^*_{\mathrm{NSC}}) &= W(q^o_{\mathrm{F}}(\kappa^*_{\mathrm{NSC}}), q^o_{\mathrm{B}}(\kappa^*_{\mathrm{NSC}}), \kappa^*_{\mathrm{NSC}}) - W^o \\ &= \begin{cases} \frac{1}{2} \cdot \alpha \cdot d^2 & \text{if } d \leq c_\kappa \\ \frac{1}{2}\alpha c_\kappa (2d - c_\kappa) & \text{if } d > c_\kappa \end{cases} \end{aligned}$$

This function is continuous, as it holds that for $d = c_\kappa$ the second part is

$$\frac{1}{2}\alpha c_\kappa (2d - c_\kappa) = \frac{1}{2}\alpha d(2d - d) = \frac{1}{2}\alpha d^2 \tag{44}$$

As it holds that $\alpha > 0$ and $d > 0$ the monotony in d and α is straightforward. If $d \leq c_\kappa$ the investment distortion does not change, when changing c_κ. If $d > c_\kappa$ it holds that

$$\frac{\partial ID}{\partial c_\kappa} = \alpha d - \alpha c_\kappa = \alpha \underbrace{(d - c_\kappa)}_{>0} > 0 \tag{45}$$

thus making ID weakly increasing in c_κ. □

Proof of Proposition 4. It holds that

$$\begin{aligned} \mathrm{CD}(\kappa) &= \bar{P}_{\mathrm{NSC}}(q = \kappa) - \bar{P}_{\mathrm{SC}}(q = \kappa - \beta\alpha) \\ &= \kappa\left(1 - \frac{\kappa}{2\alpha}\right) - \frac{\kappa}{1-\beta} + \frac{\kappa^2 + \alpha^2\beta}{2\alpha(1-\beta)} \\ &= \frac{\beta(\alpha - \kappa)^2}{2\alpha(1-\beta)} > 0 \end{aligned}$$

as $\alpha \neq \kappa$, $0 < \beta < 1$ and $\alpha > 0$. □

Proof of Proposition 5. It holds that

$$\begin{aligned} W^o - W_{\mathrm{SC}}(\kappa) &= \mathrm{ID}(\kappa) + W(q^o_{\mathrm{F}}(\kappa), q^o_{\mathrm{B}}(\kappa), \kappa) - W_{\mathrm{SC}}(\kappa) \\ &= \mathrm{ID}(\kappa) + W_{\mathrm{NSC}}(\kappa) - W_{\mathrm{SC}}(\kappa) \\ &= \mathrm{ID}(\kappa) + (d\kappa(1-\beta) + d\kappa\beta + \bar{P}_{\mathrm{NSC}}(\kappa) - c_\kappa\kappa) \\ &\quad - (d(\kappa - \alpha\beta) + d\alpha\beta + \bar{P}_{\mathrm{SC}}(\kappa - \alpha\beta) - c_\kappa\kappa) \\ &= \mathrm{ID}(\kappa) + \mathrm{CD}(\kappa) \end{aligned}$$

Note that the payment c_{SC} is welfare neutral and the consumer always receives a utility of 0. □

Proposition I.1 The optimal profit of the ISP is

$$\pi^*_{\mathrm{ISP}} = \begin{cases} \frac{1}{2}\alpha(1 - c_\kappa)^2 & \text{in the NSC case} \\ \frac{1}{2}\alpha(1 + c_\kappa^2(1-\beta) + d^2\beta^2(1-\beta) - 2c_\kappa(1 - d\beta(1-\beta))) & \text{in the SC case with } q > 0 \\ \frac{1}{2}\alpha\beta(1 - 2c_\kappa + 2d(1-\beta)) & \text{in the ESC case} \end{cases} \tag{46}$$

where the case is determined by the ISP himself based on the following parameter constellations:

$$\begin{cases} \text{Sponsored Content with } q > 0 & \text{if } d+\sqrt{d(4c_\kappa+d)}>2(c_\kappa+d\beta) \text{ and } c_\kappa+d\beta\leq 1 \\ \text{Exlusively Sponsored Content} & \text{if } c_\kappa^2-2c_\kappa(1-\beta)+(1-\beta)(1-2d\beta)<0 \text{ and } c_\kappa+d\beta>1 \\ \text{No Sponsored Content} & \text{otherwise} \end{cases} \tag{47}$$

Proof. Based on the results of section 4 it holds that

$$\pi^*_{\text{ISP, NSC}} = \pi_{\text{ISP, NSC}}(\kappa^*_{\text{NSC}}) \tag{48}$$
$$= \bar{P}_{\text{NSC}}(\kappa^*_{\text{NSC}}) - c_\kappa \kappa^*_{\text{NSC}} \tag{49}$$
$$= \kappa^*_{\text{NSC}}\left(1 - \frac{\kappa^*_{\text{NSC}}}{2\alpha}\right) - c_\kappa \kappa^*_{\text{NSC}} \tag{50}$$
$$= \frac{1}{2}\alpha(1-c_\kappa)^2 \tag{51}$$

as it holds that

$$\kappa^*_{\text{NSC}} \overset{\text{Eq. 16}}{=} \alpha(1-c_\kappa) \tag{52}$$

In case of sponsored content it holds that

$$\pi_{\text{ISP, SC}}(\kappa) = \bar{P}_{\text{SC}}(q = \kappa - \beta\alpha) + c_{\text{SC}}(\kappa) - c_\kappa \cdot \kappa \tag{53}$$
$$= \frac{1}{2}\left(\alpha - 2c_\kappa\kappa - \frac{(\alpha-\kappa)^2}{\alpha(1-\beta)} + 2d\beta(\alpha-\kappa)\right) \tag{54}$$

given that $\beta\alpha \leq \kappa \leq \alpha$ and $c_{\text{SC}}(\kappa) = d\beta(\alpha-\kappa)$ as seen in Equation 25. For $\bar{P}_{\text{SC}}$ see Equation 21. $\beta\alpha \leq \kappa$ is a direct consequence of Equation 6 ensuring network stability while $\kappa \leq \alpha$ is Equation 7 as the consumer can't be forced to generate traffic.

It is left to specify the optimal network capacity κ in the case of sponsored content and to compare it with the alternative of having no sponsored content.

The ISP chooses out of the following four cases:

1. $\pi_{\text{ISP, SC}}(\kappa = \beta\alpha)$ [Sponsored only, left side solution when choosing κ]
   $\pi_{\text{ISP, SC}}(\kappa = \beta\alpha) = \frac{1}{2}\alpha\beta(1 - 2c_\kappa + 2d(1-\beta))$

2. $\pi_{\text{ISP, SC}}(\kappa = \kappa_{\text{opt, SC}})$ [Inner solution],
   $\pi_{\text{ISP, SC}}(\kappa = \kappa_{\text{opt, SC}}) = \frac{1}{2}\alpha(1 + c_\kappa^2(1-\beta) + d^2\beta^2(1-\beta) - 2c_\kappa(1 - d\beta(1-\beta)))$
   $\kappa_{\text{opt, SC}} = \alpha(1 - (1-\beta)(c_\kappa + d\beta))$
   $\kappa_{\text{opt, SC}}$ can be determined by calculating $\frac{\partial \pi_{\text{ISP, SC}}}{\partial\kappa} \overset{!}{=} 0$.
   To make sponsored content feasible it has to hold that, $\beta\alpha \leq \kappa_{\text{opt, SC}} \leq \alpha$ which holds if $c_\kappa + d\beta \leq 1$

3. $\pi_{\text{ISP, SC}}(\kappa = \alpha)$ [Full network, right side solution when choosing κ]
   $\pi_{\text{ISP, SC}}(\kappa = \alpha) = \frac{1}{2}\alpha(1 - 2c_\kappa)$.

4. $\pi^*_{\text{ISP, NSC}} = \frac{1}{2}\alpha(1-c_\kappa)^2$ [Outside option without sponsored content]

We can now determine Equation 47 by comparing the different possible profits which can be achieved under the given conditions.

It holds that

$$\pi_{\text{ISP, SC}}(\kappa = \alpha) > \pi_{\text{ISP, SC}}(\kappa = \beta\alpha) \quad \Leftrightarrow \quad c_\kappa + d\beta < \frac{1}{2} \tag{55}$$

so if this is the case the ISP will chose the full network case over the ESC case conditional on the outside option and the inner solution being worse or not applicable. It should be noted, that $c_\kappa + d\beta < \frac{1}{2}$ implies that $c_\kappa + d\beta \leq 1$, so the inner solution is applicable. We have thus to check whether the inner solution is always better as $\pi_{\text{ISP, SC}}(\kappa = \alpha)$ and indeed it is. In other words if sponsored content is permitted the ISP will never implement the full network solution.

Furthermore it holds that

$$\pi_{\text{ISP, SC}}(\kappa = \beta\alpha) > \pi_{\text{ISP, SC}}(\kappa = \kappa_{\text{opt, SC}}) \tag{56}$$

is always false, so the ISP would always chose the inner option if applicable over the ESC option.

Given that the inner solution is applicable (i.e. $c_\kappa + d\beta \leq 1$), the inner solution is preferable to the outside option if and only if

$$d + \sqrt{d(4c_\kappa + d)} > 2(c_\kappa + d\beta) \tag{57}$$

If the inner solution is not applicable (i.e. $c_\kappa + d\beta > 1$) then it holds that

$$\begin{aligned} &\pi_{\text{ISP, SC}}(\kappa = \alpha\beta) > \pi^*_{\text{ISP, NSC}} \\ \Leftrightarrow \quad & c_\kappa^2 - 2c_\kappa(1-\beta) + (1-\beta)(1-2d\beta) < 0 \end{aligned}$$

so with these conditions the ESC case is implemented. □

Proposition I.2 The ISP will implement the following network capacity κ^*:

$$\kappa^* = \begin{cases} \alpha(1-c_\kappa) & \text{in the NSC case} \\ \alpha(1-(1-\beta)(c_\kappa + d\beta)) & \text{in the SC case with } q > 0 \\ \alpha\beta & \text{in the ESC case} \end{cases} \tag{58}$$

where the case is determined by the ISP himself based on the following parameter constellations:

$$\begin{cases} \text{Sponsored Content with } q > 0 & \text{if } d+\sqrt{d(4c_\kappa+d)}>2(c_\kappa+d\beta) \text{ and } c_\kappa+d\beta\leq 1 \\ \text{Exlusively Sponsored Content} & \text{if } c_\kappa^2-2c_\kappa(1-\beta)+(1-\beta)(1-2d\beta)<0 \text{ and } c_\kappa+d\beta>1 \\ \text{No Sponsored Content} & \text{otherwise} \end{cases} \tag{59}$$

Proof. This Proposition is a direct result of the proof of Proposition I.1. □

Proof of Proposition 6. We define $\underline{d}(c_\kappa)$ as

$$\underline{d}(c_\kappa) = \begin{cases} c_\kappa \frac{\beta+\sqrt{1-\beta}-1}{(1-\beta)\beta} & \text{if } c_\kappa \leq \sqrt{1-\beta} \\ \frac{c_\kappa^2+(1-2c_\kappa)(1-\beta)}{2(1-\beta)\beta} & \text{if } c_\kappa > \sqrt{1-\beta} \end{cases} \tag{60}$$

The proof is separated into eight parts:

1. We show, that

$$d > c_\kappa \frac{\beta+\sqrt{1-\beta}-1}{(1-\beta)\beta} \quad \Leftrightarrow \quad d + \sqrt{d(4c_\kappa + d)} > 2(c_\kappa + d\beta) \tag{61}$$

2. We show that

$$d > \frac{c_\kappa^2 + (1-2c_\kappa)(1-\beta)}{2(1-\beta)\beta} \quad \Leftrightarrow \quad c_\kappa^2 - 2c_\kappa(1-\beta) + (1-\beta)(1-2d\beta) < 0 \tag{62}$$

3. We show that

$$d > \underline{d}(c_\kappa) \wedge c_\kappa > \sqrt{1-\beta} \quad \Rightarrow \quad c_\kappa + d\beta > 1 \tag{63}$$

4. We show that

$$c_\kappa \leq \sqrt{1-\beta} \wedge c_\kappa + d\beta > 1 \quad \Rightarrow \quad c_\kappa^2 - 2c_\kappa(1-\beta) + (1-\beta)(1-2d\beta) < 0 \tag{64}$$

5. We show that

$$c_\kappa \leq \sqrt{1-\beta} \wedge d \leq c_\kappa \frac{\beta+\sqrt{1-\beta}-1}{(1-\beta)\beta} \quad \Rightarrow \quad c_\kappa + d\beta \leq 1 \tag{65}$$

   and that

$$c_\kappa > \sqrt{1-\beta} \wedge d > c_\kappa \frac{\beta+\sqrt{1-\beta}-1}{(1-\beta)\beta} \quad \Rightarrow \quad c_\kappa + d\beta > 1 \tag{66}$$

6. We combine the earlier parts to proof that $\underline{d}(c_\kappa)$ indeed separates the cases.

7. We show that $\underline{d}(c_\kappa)$ is strictly monotonously increasing.

8. We show that indeed $\lim\limits_{c_\kappa \to 0} \underline{d}(c_\kappa) = 0$.

9. We conclude that $d > 0$.

First, we show that

$$d > c_\kappa \frac{\beta+\sqrt{1-\beta}-1}{(1-\beta)\beta} \quad \Leftrightarrow \quad d + \sqrt{d(4c_\kappa + d)} > 2(c_\kappa + d\beta) \tag{67}$$

We will focus in the notation on "⇒", but the other direction holds analogously by showing that $\leq$ on the left side implies $\leq$ on the right side as well.
If we define $\psi := \frac{d}{c_\kappa} > 0$ it holds that

$$d + \sqrt{d(4c_\kappa + d)} > 2(c_\kappa + d\beta) \tag{68}$$

$$\Leftrightarrow \quad \frac{d + \sqrt{d(4c_\kappa + d)}}{2c_\kappa} > 1 + \frac{d}{c_\kappa}\beta \tag{69}$$

$$\Leftrightarrow \quad \frac{\frac{d}{c_\kappa} + \sqrt{4\frac{d}{c_\kappa} + \frac{d^2}{c_\kappa^2}}}{2} > 1 + \frac{d}{c_\kappa}\beta \tag{70}$$

$$\Leftrightarrow \quad \frac{\psi + \sqrt{4\psi + \psi^2}}{2} > \psi\beta + 1 \tag{71}$$

If this holds true, than this concludes the first part of the proof.
It holds that

$$\psi\beta + 1 = \frac{d}{c_\kappa} + 1 > \frac{\beta - 1 + \sqrt{1-\beta}}{1-\beta} + 1 = -1 + \frac{1}{\sqrt{1-\beta}} + 1 = \frac{1}{\sqrt{1-\beta}} \tag{72}$$

But here it holds that

$$\begin{aligned}
& & \psi\beta + 1 &> \frac{1}{\sqrt{1-\beta}} \\
&\overset{\psi > 0}{\Leftrightarrow} & \psi^2\beta^2 + 2\psi\beta + 1 &> \frac{1}{1-\beta} \\
&\overset{0<\beta<1}{\Leftrightarrow} & (1-\beta)(\psi^2\beta^2 + 2\psi\beta + 1) - 1 &> 0 \\
&\Leftrightarrow & \beta(\psi^2\beta^2 + \beta(2\psi - \psi^2) + (1 - 2\psi)) &< 0
\end{aligned}$$

As $\beta(\psi^2\beta^2 + \beta(2\psi - \psi^2) + (1 - 2\psi))$ is a polynomial of third degree it is sufficient to calculate where it vanishes and testing where it is above and below 0.

It holds that

$$\beta_1 = \frac{\psi - 2}{2\psi} - \frac{1}{2}\sqrt{\frac{\psi + 4}{\psi}} \tag{73}$$

$$\beta_2 = 0 \tag{74}$$

$$\beta_3 = \frac{\psi - 2}{2\psi} + \frac{1}{2}\sqrt{\frac{\psi + 4}{\psi}} \tag{75}$$

It is straightforward to see, that $\beta_1 < 0$ and thus irrelevant as $0 < \beta < 1$. It holds that

$$\begin{aligned}
& & \beta_3 = \frac{\psi - 2}{2\psi} + \frac{1}{2}\sqrt{\frac{\psi + 4}{\psi}} &> 0 \\
&\Leftrightarrow & \psi - 2 &> -\psi\sqrt{\frac{\psi + 4}{\psi}}
\end{aligned}$$

which is obvious if $\psi \geq 2$. If $\psi < 2$ we can see that

$$\begin{aligned}
&& \psi - 2 > -\psi\sqrt{\frac{\psi+4}{\psi}} \\
\Leftrightarrow && \psi^2 - 4\psi + 4 < \psi^2\frac{\psi+4}{\psi} = \psi(\psi+4) = \psi^2 + 4\psi \\
\Leftrightarrow && 4 < 8\psi \\
\Leftrightarrow && \frac{1}{2} < \psi
\end{aligned}$$

It is easy to check, that Inequality 72 holds exactly if $0 < \beta < \beta_3$, thus making $\psi > \frac{1}{2}$ a necessary condition for $d > c_\kappa\frac{\beta+\sqrt{1-\beta}-1}{(1-\beta)\beta}$. As Inequality 72 is true due to $d > c_\kappa\frac{\beta+\sqrt{1-\beta}-1}{(1-\beta)\beta}$ we know that $\beta < \beta_3$ also holds true.
We can now check, whether Inequality 71 holds true. It does indeed, as

$$\psi\beta + 1 < \psi\beta_3 + 1 = \frac{\psi-2}{2} + \frac{1}{2}\sqrt{\psi(\psi+4)} + 1 = \frac{\psi + \sqrt{4\psi+\psi^2}}{2} \tag{76}$$

The direction "$\Leftarrow$" follows analogously. Here we can show, that it has to follow that $\beta > \max\{\beta_3, 0\}$ is a necessary condition, thus proofing an inequality analogously to Inequality 71. This concludes the first part of the proof.

For the second part we show that

$$d > \frac{c_\kappa^2 + (1-2c_\kappa)(1-\beta)}{2(1-\beta)\beta} \quad \Leftrightarrow \quad c_\kappa^2 - 2c_\kappa(1-\beta) + (1-\beta)(1-2d\beta) < 0 \tag{77}$$

This follows directly as

$$\begin{aligned}
&& d > \frac{c_\kappa^2 + (1-2c_\kappa)(1-\beta)}{2(1-\beta)\beta} \\
\Leftrightarrow && 2d\beta(1-\beta) > c_\kappa^2 - 2c_\kappa(1-\beta) + (1-\beta) \\
\Leftrightarrow && 0 > c_\kappa^2 - 2c_\kappa(1-\beta) + (1-\beta)(1-2d\beta)
\end{aligned}$$

which concludes the second part of the proof.

For the third part we show that with $c_\kappa > \sqrt{1-\beta}$ it holds that

$$d > \underline{d}(c_\kappa) \quad \Rightarrow \quad c_\kappa + d\beta > 1 \tag{78}$$

This follows directly as

$$\begin{aligned}
c_\kappa + d\beta &> c_\kappa + \underline{d}(c_\kappa)\cdot\beta = c_\kappa + \frac{c_\kappa^2 + (1-2c_\kappa)(1-\beta)}{2(1-\beta)} > c_\kappa + \frac{(1-\beta) + (1-2c_\kappa)(1-\beta)}{2(1-\beta)} \\
&= c_\kappa + \frac{1+1-2c_\kappa}{2} = c_\kappa + 1 - c_\kappa = 1
\end{aligned}$$

which concludes the third part.

For the fourth part we show that $c_\kappa \leq \sqrt{1-\beta}$ and $c_\kappa + d\beta > 1$ implies

$$c_\kappa^2 - 2c_\kappa(1-\beta) + (1-\beta)(1-2d\beta) < 0$$

It holds that

$$\begin{aligned} & 0 > c_\kappa^2 - 2c_\kappa(1-\beta) + (1-\beta)(1-2d\beta) \\ \Leftrightarrow \quad & d > \frac{c_\kappa^2 + (1-2c_\kappa)(1-\beta)}{2(1-\beta)\beta} \end{aligned}$$

so it is sufficient to show this inequality. This holds as

$$\begin{aligned} \frac{c_\kappa^2 + (1-2c_\kappa)(1-\beta)}{2(1-\beta)\beta} &\leq \frac{(1-\beta) + (1-2c_\kappa)(1-\beta)}{2(1-\beta)\beta} \\ &= \frac{1+1-2c_\kappa}{2\beta} = \frac{1-c_\kappa}{\beta} < d \end{aligned}$$

which concludes the fourth part.

In the fifth part we show that

$$c_\kappa \leq \sqrt{1-\beta} \wedge d \leq c_\kappa \frac{\beta + \sqrt{1-\beta} - 1}{(1-\beta)\beta} \quad \Rightarrow \quad c_\kappa + d\beta \leq 1 \tag{79}$$

and that

$$c_\kappa > \sqrt{1-\beta} \wedge d > c_\kappa \frac{\beta + \sqrt{1-\beta} - 1}{(1-\beta)\beta} \quad \Rightarrow \quad c_\kappa + d\beta > 1 \tag{80}$$

We will only show the first statement, as the other one follows analogously:

$$\begin{aligned} c_\kappa + d\beta &\leq \sqrt{1-\beta} + \sqrt{1-\beta} \cdot \beta \cdot \frac{\beta + \sqrt{1-\beta} - 1}{(1-\beta)\beta} \\ &= \frac{\sqrt{1-\beta} - \beta\sqrt{1-\beta} + \sqrt{1-\beta}\beta + (1-\beta) - \sqrt{1-\beta}}{(1-\beta)} \\ &= \frac{1-\beta}{1-\beta} = 1 \end{aligned}$$

which concludes the fifth part.

In the sixth part we combine the earlier parts to show the statement.
Combining the second and the third part shows, that it holds that for all $d > \underline{d}(c_\kappa)$ and $c_\kappa > \sqrt{1-\beta}$ exclusively sponsored content is implemented.
Combining the first and fourth part implies that either normal sponsored content with $q > 0$ or exclusively sponsored content is implemented if $c_\kappa \leq \sqrt{1-\beta}$. We know from the first part that the first condition of normal sponsored content is satisfied, but the second condition ($c_\kappa + d\beta \leq 1$) is not shown. There are two possibilities:

1. If the condition is met, normal sponsored content is implemented.
2. If the condition is not met, it follows directly, that the second condition of exclusively sponsored content is satisfied. The first condition follows out of part four. Exclusively sponsored content is implemented.

No sponsored content with $d \leq \underline{d}(c_\kappa)$ is implemented, when neither condition of the sponsored content cases is met.
Assuming, that $c_\kappa \leq \sqrt{1-\beta}$ it holds due to the first part, that normal sponsored content is not implemented. Exclusively sponsored content is not implemented, as the first statement in the fifth part shows that the second condition of this case is not met. Consequently with $c_\kappa \leq \sqrt{1-\beta}$ no sponsored content is implemented.
Assuming, that $c_\kappa > \sqrt{1-\beta}$ it holds due to the second part, that exclusively sponsored content is not implemented. Now two cases have to be examined:

1. $$d \leq c_\kappa \frac{\beta + \sqrt{1-\beta} - 1}{(1-\beta)\beta} \tag{81}$$
   In this case the first part can be used and the first condition of normal sponsored content is not met. As exclusively sponsored content has been ruled out, no sponsored content is implemented.

2. $$d > c_\kappa \frac{\beta + \sqrt{1-\beta} - 1}{(1-\beta)\beta} \tag{82}$$
   In this case the second statement of the fifth part can be used and the second condition of normal sponsored content is not met. As exclusively sponsored content has been ruled out, no sponsored content is implemented.

In the seventh part we show that $\underline{d}(c_\kappa)$ is strictly monotonously increasing in c_κ.

1. Assume $c_\kappa \leq \sqrt{1-\beta}$
   $$\frac{\partial}{\partial c_\kappa}\left(c_\kappa \frac{\beta + \sqrt{1-\beta} - 1}{(1-\beta)\beta}\right) = \frac{\beta + \sqrt{1-\beta} - 1}{(1-\beta)\beta} > 0$$
   as
   $$\begin{aligned} & \beta + \sqrt{1-\beta} > 1 \\ \Leftrightarrow \quad & \sqrt{1-\beta} > 1-\beta \end{aligned}$$
   which is true as $0 < \beta < 1$. $\underline{d}(c_\kappa)$ is thus monotonously increasing within the first case.

2. Assume $c_\kappa > \sqrt{1-\beta}$
   $$\begin{aligned} \frac{\partial}{\partial c_\kappa}\left(\frac{c_\kappa^2 + (1-2c_\kappa)(1-\beta)}{2(1-\beta)\beta}\right) &= \frac{2c_\kappa}{2(1-\beta)\beta} - \frac{2(1-\beta)}{2(1-\beta)\beta} \\ &= \frac{c_\kappa}{(1-\beta)\beta} - \frac{1}{\beta} \overset{c_\kappa > \sqrt{1-\beta}}{>} \frac{\sqrt{1-\beta}}{(1-\beta)\beta} - \frac{1-\beta}{\beta(1-\beta)} \\ &= \frac{\sqrt{1-\beta} - (1-\beta)}{\beta(1-\beta)} > 0 \end{aligned}$$
   with the same argumentation as in the case above. We have thus shown, that $\underline{d}(c_\kappa)$ is also monotonously increasing within the second case.

3. It is left to show, that the $\underline{d}(c_\kappa)$ is increasing when switching from the first into the second case.
   If $c_\kappa = \sqrt{1-\beta}$ it holds that
$$\underline{d}(c_\kappa) = \sqrt{1-\beta}\frac{\beta+\sqrt{1-\beta}-1}{(1-\beta)\beta} = \frac{\beta+\sqrt{1-\beta}-1}{\sqrt{(1-\beta)}\beta} = \frac{\sqrt{1-\beta}-(1-\beta)}{\beta\sqrt{1-\beta}} \tag{83}$$
   on the other hand it holds that
$$\begin{aligned}\lim_{c_\kappa \to \sqrt{1-\beta}+} \underline{d}(c_\kappa) &= \frac{(1-\beta)+(1-2\sqrt{1-\beta})(1-\beta)}{2(1-\beta)\beta} = \frac{1+1-2\sqrt{1-\beta}}{2\beta} \\ &= \frac{1-\sqrt{1-\beta}}{\beta} = \frac{\sqrt{1-\beta}-(1-\beta)}{\beta\sqrt{1-\beta}}\end{aligned}$$
   $\underline{d}(c_\kappa)$ is thus continuous which concludes the seventh part.

Now we show that $\lim_{c_\kappa\to 0} \underline{d}(c_\kappa) = 0$. This follows directly as for small c_κ it holds that
$$\underline{d}(c_\kappa) = c_\kappa \frac{\beta+\sqrt{1-\beta}-1}{(1-\beta)\beta} \tag{84}$$
which approaches 0 when c_κ approaches 0.

It is left to show that $\underline{d}(c_\kappa) > 0$ which follows out of the monotony and the fact that $\lim_{c_\kappa\to 0} \underline{d}(c_\kappa) = 0$ but $c_\kappa > 0$ and $0 < \beta < 1$. □

Proof of Proposition 7. This proof largely draws from the proof of Proposition 6.
We define
$$\bar{d}(c_\kappa) := \begin{cases} \frac{1-c_\kappa}{\beta} & \text{if } c_\kappa \le c_\kappa^* = \sqrt{1-\beta} \\ \underline{d}(c_\kappa) & \text{if } c_\kappa > c_\kappa^* \end{cases} \tag{85}$$
Let there be $c_\kappa \le c_\kappa^*$. It holds that
$$d > \bar{d}(c_\kappa) \quad \Leftrightarrow \quad d > \frac{1-c_\kappa}{\beta} \quad \Leftrightarrow \quad c_\kappa + d\beta > 1 \tag{86}$$
so the second condition of the exclusively sponsored content case is met. The first condition of the exclusively sponsored content case is met due to Statement 64. This shows that for all $d > \bar{d}(c_\kappa)$ exclusively sponsored content is implemented. For all $d \le \bar{d}(c_\kappa)$ the second condition is not met, which shows that in this case exclusive sponsored content is not implemented.
To show that in this case $\bar{d}(c_\kappa) \ge \underline{d}(c_\kappa)$ it is sufficient to show that
$$\frac{1-c_\kappa}{\beta} - c_\kappa\frac{\beta+\sqrt{1-\beta}-1}{(1-\beta)\beta} \ge 0 \tag{87}$$
This holds as
$$\begin{aligned}\frac{1-c_\kappa}{\beta} - c_\kappa\frac{\beta+\sqrt{1-\beta}-1}{(1-\beta)\beta} &= \frac{1-\beta}{(1-\beta)\beta} - \frac{c_\kappa(1-\beta)}{(1-\beta)\beta} - c_\kappa\frac{\beta+\sqrt{1-\beta}-1}{(1-\beta)\beta} \\ &= \frac{(1-\beta)-c_\kappa+c_\kappa\beta-c_\kappa\beta-c_\kappa\sqrt{1-\beta}+c_\kappa}{(1-\beta)\beta} \\ &= \frac{(1-\beta)-c_\kappa\sqrt{1-\beta}}{(1-\beta)\beta} \ge \frac{(1-\beta)-(1-\beta)}{(1-\beta)\beta} = 0\end{aligned}$$

Let there be $c_\kappa > c_\kappa^*$. Proposition 6 shows the proposition for sponsored content. It is left to show that this can only be exclusively sponsored content. This follows directly out of Statement 63.
It holds that $\bar{d}(c_\kappa) \geq \underline{d}(c_\kappa)$ per Construction. $\square$

Proof of Proposition 8. The first part of this proposition is a direct corollary of the proof of Proposition 7.

Regarding the second part it should be noted that $\bar{d}(c_\kappa)$ is monotonously decreasing for $c_\kappa \leq c_\kappa^*$ as here it holds that

$$d'(c_\kappa) = -\frac{1}{\beta} < 0 \tag{88}$$

From Proposition 6 we know, that $\underline{d}(c_\kappa)$ is strictly monotonously increasing. Therefore it holds that $\underline{d}(c_\kappa) < \bar{d}(c_\kappa)$ and thus the existence of d between those two values is shown as Proposition 7 holds. Combining Proposition 6 and Proposition 7 proves, that for those d non-exclusive sponsored content is implemented. $\square$

Proof of Proposition 9. First of all it should be noted, that $c_\kappa < \tilde{c}_\kappa = 1 - \beta < c_\kappa^* = \sqrt{1-\beta}$ and that here d exists for which non-exclusive sponsored content is implemented (see Proposition 8). Note, that to prove the existence of d^* we need the two last features of the proposition. The order of this proof is thus not identical to the order in the proposition.

Calculation of the jump based on the results of Proposition I.2:

$$\begin{aligned}
\left(\lim_{d \to \underline{d}(c_\kappa)+} \kappa^*(d)\right) - \kappa^*\left(d = \underline{d}(c_\kappa)\right) &= \alpha\left(1 - (1-\beta)(c_\kappa + \lim_{d \to \underline{d}(c_\kappa)+} d\beta)\right) - \alpha(1 - c_\kappa) \\
&= \alpha\left(c_\kappa - (1-\beta)\left(c_\kappa + c_\kappa \frac{\beta + \sqrt{1-\beta} - 1}{(1-\beta)\beta}\beta\right)\right) \\
&= \alpha c_\kappa\left(1 - (1-\beta)\left(1 + \frac{\beta + \sqrt{1-\beta} - 1}{(1-\beta)}\right)\right) \\
&= \alpha c_\kappa\left(1 - \left((1-\beta) + \beta + \sqrt{1-\beta} - 1\right)\right) \\
&= \alpha c_\kappa\left(1 - \sqrt{1-\beta}\right)
\end{aligned}$$

It holds that $\alpha c_\kappa\left(1 - \sqrt{1-\beta}\right) > 0$, as $\sqrt{1-\beta} < 1$.

The form of $\kappa^*(d)$ within the sponsored content case can be found within I.2. It is left to show, that indeed $\kappa^*(d)$ is monotonously decreasing. This holds as

$$\frac{\partial \kappa^*}{\partial d} = (1-\beta)\beta > 0 \tag{89}$$

The constant network capacity in the NSC and ESC cases are straightforward results of I.2. Furthermore it holds that

$$\kappa_{\text{NSC}}^* - \kappa_{\text{ESC}}^* = \alpha(1 - c_\kappa) - \alpha\beta > \alpha(1 - \tilde{c}_\kappa) - \alpha\beta = \alpha(1 - 1 - \beta) - \alpha\beta > 0 \tag{90}$$

which shows that the network capacity in the NSC case is higher.

We will now show, that the transition of the SC case to the ESC case is indeed continuous. It holds that

$$\lim_{d \to \bar{d}(c_\kappa)+} \kappa^*(d) = \alpha\beta \tag{91}$$

and

$$\begin{aligned}\kappa^*(d = \bar{d}(c_\kappa)) &= \alpha\left(1 - (1-\beta)(c_\kappa + \bar{d}(c_\kappa)\beta)\right) \\ &= \alpha\left(1 - (1-\beta)(c_\kappa + \frac{1-c_\kappa}{\beta}\beta)\right) \\ &= \alpha\beta\end{aligned}$$

which proofs this point.

It is left to show, that there exists a minimal profitability $d^*(c_\kappa)$ within the sponsored content case, where sponsored content leads to less network capacity for all $d > d^*$. We have already shown that the network capacity in the NSC and the ESC case are independent of d and that in the ESC case always a lower network capacity is implemented. It is thus sufficient to show for all d within the sponsored content case. The existence of d^* follows directly out of the fact, that κ^* is monotonously decreasing within the sponsored content case, the jump from the NSC to the SC case is always positive and that the transition from the SC and the NSC case is continuous. □

Proof of Proposition 11. Note, that the proposition implies, that $c_\kappa < c_\kappa^*$ as there is a transition from non-sponsored content to non-exclusive sponsored content.

It holds due to Proposition 9 that

$$\lim_{d \to \underline{d}(c_\kappa)+} [\kappa^*(d)] - \kappa^*(d = \underline{d}(c_\kappa)) = \alpha c_\kappa(1 - \sqrt{1-\beta}) \quad , \tag{92}$$

due to the proof of Proposition 6 and Proposition 7 that

$$\underline{d}(c_\kappa) = c_\kappa \frac{\beta + \sqrt{1-\beta} - 1}{(1-\beta)\beta} \tag{93}$$

given that we switch from NSC to SC, due to Equation 25 that

$$c_{\text{SC}} = d\beta(\alpha - \kappa^*) \tag{94}$$

and due to Proposition I.2 that

$$\kappa^* = \alpha(1 - (1-\beta)(c_\kappa + d\beta)) \tag{95}$$

in the SC case. We combine those features to show the statement. For κ and d marginally near the thresholds it holds that

$$\begin{aligned}
c_{\text{SC}} &= d\beta(\alpha - \kappa) \\
&= c_\kappa \frac{\beta + \sqrt{1-\beta} - 1}{1-\beta}(\alpha - \alpha(1-(1-\beta)(c_\kappa + d\beta)) \\
&= \alpha c_\kappa(\beta + \sqrt{1-\beta} - 1)(c_\kappa + d\beta) \\
&= \alpha c_\kappa^2 \left(\frac{(1-\beta) - \sqrt{1-\beta}(1-\beta)}{1-\beta} \right) = \alpha c_\kappa^2 (1 - \sqrt{1-\beta})
\end{aligned}$$

As it holds that

$$c_\kappa \cdot \left(\lim_{d \to \underline{d}(c_\kappa)+} [\kappa^*(d)] - \kappa^*(d = \underline{d}(c_\kappa)) \right) = \alpha c_\kappa^2 (1 - \sqrt{1-\beta}) \tag{96}$$

the statement is shown. □

Proposition I.3

a) Given that $c_\kappa > \tilde{c}_\kappa = 1 - \beta$ and $c_\kappa < c_\kappa^* = \sqrt{1-\beta}$ the properties of Proposition 9 hold with the following exception:

   Given sponsored content is actually implemented, sponsored content always leads to more network capacity.

b) Given that $c_\kappa > c_\kappa^*$ and that sponsored content is implemented it holds that the realized network capacity k^* increases by

$$\alpha(c_\kappa + \beta - 1) > 0 \tag{97}$$

   when sponsored content is allowed. The case of non-exclusive sponsored content is not implemented and identical to Proposition 9 it holds that

$$\begin{aligned}
\kappa_{\text{NSC}}^* &= \alpha(1 - c_\kappa) \\
\kappa_{\text{ESC}}^* &= \alpha\beta
\end{aligned}$$

Proof of Proposition I.3. a) Let there be $\sqrt{1-\beta} > c_\kappa > 1 - \beta$.

The shared properties as given in Proposition I.3 can be shown analogously as in the proof of Proposition 9 itself.

The network capacity of the NSC case is $\kappa_{\text{NSC}}^* = \alpha(1 - c_\kappa)$ and in the ESC case $\kappa_{\text{ESC}}^* = \alpha\beta$ as per Proposition I.2. Thus it holds that

$$\kappa_{\text{NSC}}^* = \alpha(1 - c_\kappa) < \alpha(1 - (1-\beta)) = \alpha\beta = \kappa_{\text{ESC}}^* \tag{98}$$

i.e. the network capacity of the NSC case is lower than in the ESC case.

We have shown, that there is a positive jump when switching from the NSC case to the ESC case by increasing c_κ. Within the cases of sponsored content the function of network capacity is monotonously decreasing in c_κ and continuous. As the ESC case has more network capacity than the SC case it follows that the network capacity is higher than the NSC case when sponsored content is implemented and c_κ is in the given range.

b) Given that $c_\kappa > c_\kappa^*$

$$\begin{aligned}\left(\lim_{d\to \underline{d}(c_\kappa)+} \kappa^*(d)\right) - \kappa^*\left(d = \underline{d}(c_\kappa)\right) &= \alpha\beta - \alpha(1-c_\kappa) \\ &= \alpha\left(c_\kappa + \beta - 1\right)\end{aligned}$$

This difference is greater than zero as it holds that

$$\alpha\left(c_\kappa + \beta - 1\right) > \alpha\left(\sqrt{1-\beta} + \beta - 1\right) = \alpha\left(\sqrt{1-\beta} - (1-\beta)\right) > 0 \tag{99}$$

due to $0 < \beta < 1$.

□

Proof of Proposition 10. This proposition is a direct corollary from Proposition I.3. □

Proof of Proposition 12. 1. Given that sponsored content is not attractive to the ISP the results of the non-sponsored case hold. Propositon 2 thus gives the statement in the NSC case, i.e. for all $d \leq \underline{d}(c_\kappa)$.

2. Given that $\bar{d}(c_\kappa) \geq d > \underline{d}(c_\kappa)$, non-exclusive sponsored content is implemented. Assume $c_\kappa < c_\kappa^* = \sqrt{1-\beta}$, so that Proposition 8 guarantees the existence of such a d.

   a) Assume $d \leq c_\kappa$. Based on the Proof of Proposition 6, Proposition I.2 and Equation 10 we have to show that

$$c_\kappa \frac{\beta + \sqrt{1-\beta} - 1}{(1-\beta)\beta} < d \leq c_\kappa \quad \overset{!}{\Rightarrow} \quad 1 - (1-\beta)(c_\kappa + d\beta) < 1 - c_\kappa + d \tag{100}$$

   which is equivalent to having to show that

$$d(\beta + \beta^2 + 1) - \beta c_\kappa \overset{!}{>} 0 \tag{101}$$

   It holds that

$$\begin{aligned} d(\beta + \beta^2 + 1) - \beta c_\kappa &> c_\kappa \frac{(\beta + \sqrt{1-\beta} - 1)(\beta + \beta^2 + 1)}{(1-\beta)\beta} - \beta c_\kappa \\ &= \frac{(\beta + \sqrt{1-\beta} - 1)(\beta + \beta^2 + 1) - (1-\beta)\beta}{(1-\beta)\beta} c_\kappa \end{aligned}$$

We know that $c_\kappa > 0$ and $(1-\beta)\beta > 0$ due to $0 < \beta < 1$, so it is left to show that

$$\begin{aligned}&(\beta + \sqrt{1-\beta} - 1)(\beta + \beta^2 + 1) - (1-\beta)\beta \\ &\quad = 2\beta^3 - \beta^2 - 1 + \beta\sqrt{1-\beta} + \beta^2\sqrt{1-\beta} + \sqrt{1-\beta} \overset{!}{>} 0\end{aligned}$$

which can be written as

$$\frac{-2\beta^3 + \beta^2 + 1}{\beta^2 + \beta + 1} \overset{!}{<} \sqrt{1-\beta} \tag{102}$$

Note that it holds that $-2\beta^3 + \beta^2 + 1 > 0$, as

$$-2\beta^3 + \beta^2 + 1 = \underbrace{(1-\beta)}_{>0}\underbrace{(2\beta^2 + \beta + 1)}_{>0} \tag{103}$$

so both sides are > 0 and can without further case differentiations be squared:

$$\begin{aligned}& & \frac{-2\beta^3 + \beta^2 + 1}{\beta^2 + \beta + 1} &\overset{!}{<} \sqrt{1-\beta} \\ &\Leftrightarrow & \left(\frac{-2\beta^3 + \beta^2 + 1}{\beta^2 + \beta + 1}\right)^2 &\overset{!}{<} 1-\beta \\ &\Leftrightarrow & (-2\beta^3 + \beta^2 + 1)^2 - (\beta^2 + \beta + 1)^2 + \beta(\beta^2 + \beta + 1)^2 &< 0 \\ &\Leftrightarrow & -(1-\beta)\beta((4\beta^2 + \beta + 3)\beta^3 + 1) &< 0\end{aligned}$$

which concludes this part of the proof.

b) Assume $d > c_\kappa$. In this case it holds that $\kappa^o = \alpha$. We know due to Propositon 6 and 7 that the highest network capacity in the SC is necessarily implemented for $\lim_{d \to \underline{d}^+} d$, so if we are able tho show that the condition holds in this case, we have shown it for all higher d as well. In other words due to Proposition 6 and I.2 it is sufficient to show that

$$\alpha c_\kappa(1 - \sqrt{1-\beta}) + \alpha(1 - c_\kappa) < \alpha$$

This is easy to see as it holds that

$$c_\kappa(1 - \sqrt{1-\beta}) + (1 - c_\kappa) = 1 \underbrace{-c_\kappa\sqrt{1-\beta}}_{<0} < 1 \tag{104}$$

.

c) For $c_\kappa > c_\kappa^*$ no d exist given the condition, as here only exclusively sponsored content is implemented.

3. In the case of exclusively sponsored content, i.e. $d > \bar{d}$ it holds that $\kappa^* = \alpha\beta$.

For $d > c_\kappa$ it directly holds, that

$$\kappa^* = \alpha\beta < \alpha = \kappa^o \tag{105}$$

Given that $d \leq c_\kappa$ it is sufficient to show the statement for $c_\kappa > c_\kappa^*$ as given $c_\kappa \leq c_\kappa^*$ the Propositions 9 and I.3 imply that the highest network capacity is implemented in the SC case. That in this case it always holds that $\kappa^* < \kappa^o$ has already been shown in an earlier section.

Let $c_\kappa > c_\kappa^* = \sqrt{1-\beta}$ and $\bar{d} < d \leq c_\kappa$. It is to show that

$$k^o = \alpha(1 - c_\kappa + d) \overset{!}{>} \beta\alpha = k^* \tag{106}$$

It holds that

$$1 - c_\kappa + d > 1 - c_\kappa + \bar{d} = 1 - c_\kappa + \frac{c_\kappa^2 + (1-2c_\kappa)(1-\beta)}{2(1-\beta)\beta} \tag{107}$$

It is thus sufficient to show that

$$\begin{aligned} & 1 - c_\kappa + \frac{c_\kappa^2 + (1-2c_\kappa)(1-\beta)}{2(1-\beta)\beta} - \beta \overset{!}{>} 0 \\ \Leftrightarrow \quad & 2(1-\beta)\beta - 2c_\kappa(1-\beta)\beta + c_\kappa^2 + (1-2c_\kappa)(1-\beta) - 2(1-\beta)\beta^2 \overset{!}{>} 0 \\ \Leftrightarrow \quad & 2\beta^3 + 2c_\kappa\beta^2 - 4\beta^2 + \beta + c_\kappa^2 - 2c_\kappa + 1 \overset{!}{>} 0 \end{aligned}$$

As $c_\kappa^2 - 2c_\kappa + 1 = (c_\kappa - 1)^2 > 0$ and $\beta < 1$ it holds that

$$2\beta^3 + 2c_\kappa\beta^2 - 4\beta^2 + \beta + c_\kappa^2 - 2c_\kappa + 1 > 2\beta^3 + 2c_\kappa\beta^2 - 4\beta^2 + \beta + \beta(c_\kappa^2 - 2c_\kappa + 1) \tag{108}$$

which implies that after dividing by β it is sufficient to show that

$$f(\beta, \kappa) := 2\beta^2 + 2c_\kappa\beta - 4\beta + 1 + c_\kappa^2 - 2c_\kappa + 1 \overset{!}{>} 0 \tag{109}$$

We show that ($\beta = 1$, $c_\kappa = 0$) minimizes f with $f(\beta = 0, c_\kappa = 1) = 0$ which concludes our proof.
It holds that

$$\begin{aligned} f_\beta &= \frac{\partial f}{\partial \beta} = 4\beta + 2c_\kappa - 4 \overset{!}{=} 0 \quad \Rightarrow \quad \beta = 1 - \frac{1}{2}c_\kappa \\ f_{c_\kappa} &= \frac{\partial f}{\partial c_\kappa} = 2c_\kappa - 2 + 2\beta \overset{!}{=} 0 \quad \Rightarrow \quad \beta = 1 - c_\kappa \end{aligned}$$

which shows us that ($\beta = 1$, $c_\kappa = 0$) is a stationary point. It is a minimum as

$$f_{\beta\beta} = \frac{\partial f}{\partial\beta\,\partial\beta} = 4 \qquad f_{\beta c_\kappa} = \frac{\partial f}{\partial\beta\,\partial c_\kappa} = 2 \qquad f_{c_\kappa c_\kappa} = \frac{\partial f}{\partial c_\kappa\,\partial c_\kappa} = 2 \tag{110}$$

and so it holds that

$$f_{\beta\beta}f_{c_\kappa c_\kappa} - f_{\beta c_\kappa}^2 = 4 \cdot 2 - 2^2 > 0 \quad \text{and} \quad f_{\beta\beta} > 0 \quad \text{and} \quad f_{c_\kappa c_\kappa} > 0 \tag{111}$$

Finally, indeed it holds that

$$f(\beta = 1, c_\kappa = 0) = 2 + 0 - 4 + 1 + 0 - 0 + 1 = 0 \tag{112}$$

□

Proof of Proposition 13. It is to show that

$$\pi^*_{\text{F,NSC}} = d \cdot q_{\text{F,NSC}}(q = \kappa^*_{\text{NSC}}) > d \cdot q_{\text{F,SC}}(q = \kappa^* - \beta\alpha) = \pi^*_{\text{F,SC}} \tag{113}$$

It holds that

$$q_{\text{F,NSC}}(q = \kappa^*_{\text{NSC}}) = (1-\beta)\kappa^*_{\text{NSC}} = (1-\beta)\alpha(1-c_\kappa) > 0 \tag{114}$$

on the other hand it holds that (see Proposition I.2)

$$q_{\text{F,SC}}(q = \kappa^* - \beta\alpha) = \kappa^* - \beta\alpha = \begin{cases} \alpha(1-(1-\beta)(c_\kappa + d\beta)) - \alpha\beta & \text{in the SC case} \\ 0 & \text{in the ESC case} \end{cases} \tag{115}$$

It is straightforward to see, that the profit suffers in the ESC case. It is thus left to show, that

$$(1-\beta)\alpha(1-c_\kappa) - (\alpha(1-(1-\beta)(c_\kappa + d\beta)) - \alpha\beta) > 0 \tag{116}$$

This holds as

$$(1-\beta)\alpha(1-c_\kappa) - (\alpha(1-(1-\beta)(c_\kappa + d\beta)) - \alpha\beta) = \alpha\beta(1-\beta)d > 0$$

□

Proposition I.4 It holds that

$$W^*_{\text{NSC}} = \frac{1}{2}\alpha(1-c_\kappa)(1-c_\kappa+2d) \tag{117}$$

and

$$W^* = \begin{cases} \frac{1}{2}\alpha(1-c_\kappa)(1-c_\kappa+2d) & \text{in the NSC case} \\ \frac{1}{2}\alpha(1-2c_\kappa(1+d(1-\beta)) + c_\kappa^2(1-\beta) + d(2-d\beta(2-\beta-\beta^2))) & \text{in the SC case} \\ \frac{1}{2}\alpha\beta(1-2c_\kappa+2d) & \text{in the ESC case} \end{cases} \tag{118}$$

Proof. This proposition relies only on algebraic transformations of already proven and calculated results. □

Lemma I.0.1 The conditions of the ESC case imply $d > \frac{1}{2}$.

Proof of Lemma I.0.1. The first condition of the ESC case is

$$c_\kappa^2 - 2c_\kappa(1-\beta) + (1-\beta)(1-2d\beta) < 0 \tag{119}$$

It holds that

$$\begin{aligned} & c_\kappa^2 - 2c_\kappa(1-\beta) + (1-\beta)(1-2d\beta) < 0 \\ \Leftrightarrow \quad & 1-2d\beta < \frac{2c_\kappa(1-\beta) - c_\kappa^2}{(1-\beta)} \\ \Leftrightarrow \quad & 2d > \frac{1}{\beta} - \frac{2c_\kappa(1-\beta) - c_\kappa^2}{\beta(1-\beta)} \end{aligned}$$

It is thus sufficient to show that

$$\frac{1}{\beta} - \frac{2c_\kappa(1-\beta) - c_\kappa^2}{\beta(1-\beta)} > 1 \tag{120}$$

This is the case as

$$\begin{aligned}
\frac{1}{\beta} - \frac{2c_\kappa(1-\beta) - c_\kappa^2}{\beta(1-\beta)} &= \frac{1}{\beta} + \frac{c_\kappa^2 - 2c_\kappa(1-\beta) + (1-\beta)^2}{\beta(1-\beta)} - \frac{(1-\beta)^2}{(1-\beta)\beta} \\
&= \frac{1}{\beta} + \frac{(c_\kappa - (1-\beta))^2}{(1-\beta)\beta} - \frac{(1-\beta)}{\beta} \\
&= 1 + \underbrace{\frac{(c_\kappa - (1-\beta))^2}{(1-\beta)\beta}}_{>0} > 1
\end{aligned}$$

Which concludes the proof. □

Proof of Proposition 14.

a) Given that the parameters are in a way, that the NSC case is optimal for the ISP, the initial condition can never occur, as it holds that $W^* = W^*_{\text{NSC}}$

b) In the ESC case it holds that (see Proposition I.4)

$$\begin{aligned}
& W^* > W^*_{\text{NSC}} \\
\Leftrightarrow\quad & \frac{1}{2}\alpha\beta(1 - 2c_\kappa + 2d) > \frac{1}{2}\alpha(1-c_\kappa)(1-c_\kappa+2d) \\
\Leftrightarrow\quad & \beta(1-2c_\kappa+2d) > (1-c_\kappa)(1-c_\kappa+2d) \\
\Leftrightarrow\quad & \beta > (1-c_\kappa)\frac{(1-c_\kappa+2d)}{(1-2c_\kappa+2d)}
\end{aligned}$$

as Lemma I.0.1 with $d > \frac{1}{2}$ holds.

To show

$$\begin{aligned}
& \kappa^* > \kappa^*_{\text{NSC}} \\
\Leftrightarrow\quad & \beta\alpha > (1-c_\kappa)\alpha
\end{aligned}$$

It is sufficient to show that

$$\beta > (1-c_\kappa) \tag{121}$$

This is obvious as it holds that

$$\beta > (1-c_\kappa)\underbrace{\frac{(1-c_\kappa+2d)}{(1-2c_\kappa+2d)}}_{>1} > (1-c_\kappa) \tag{122}$$

due to $1 > c_\kappa > 0$. We have used that $d > \frac{1}{2}$ which holds due to Lemma I.0.1

c) In the SC case it holds that

$$\begin{aligned}
& & W^* &> W^*_{\text{NSC}} \\
&\Leftrightarrow & \frac{1}{2}\alpha(1 - 2c_\kappa(1 + d(1-\beta)) + c_\kappa^2(1-\beta) + d(2 - d\beta(2 - \beta - \beta^2))) &> \frac{1}{2}\alpha(1-c_\kappa)(1-c_\kappa+2d) \\
&\Leftrightarrow & 2c_\kappa d\beta - c_\kappa^2\beta - 2d^2\beta + d^2\beta^2 + d^2\beta^3 &> 0 \\
&\Leftrightarrow & -2c_\kappa d + c_\kappa^2 + d^2 &< d^2(-1+\beta+\beta^2) \\
&\Leftrightarrow & \left(\frac{d-c_\kappa}{d}\right)^2 &< -1+\beta+\beta^2 \\
&\Leftrightarrow & \left(1-\frac{c_\kappa}{d}\right)^2 &< -1+\beta+\beta^2 \\
&\Leftrightarrow & (1-t)^2 &< -1+\beta+\beta^2
\end{aligned}$$

if $t := \frac{c_\kappa}{d} > 0$.
It is to show, that

$$\begin{aligned}
& & \kappa^* &> \kappa^*_{\text{NSC}} \\
&\Leftrightarrow & \alpha(1-(1-\beta)(c_\kappa + d\beta)) &> (1-c_\kappa)\alpha \\
&\Leftrightarrow & c_\kappa &> d(1-\beta) \\
&\Leftrightarrow & t &> (1-\beta)
\end{aligned}$$

Which is the case, as

$$\begin{aligned}
& & (1-t)^2 &< -1+\beta+\beta^2 \\
&\Rightarrow & t &> 1-\sqrt{-1+\beta+\beta^2}
\end{aligned}$$

and thus

$$t > 1-\sqrt{-1+\beta+\beta^2} > 1-\sqrt{\beta^2} = 1-\beta \qquad (123)$$

because $\beta < 1$.

The other direction does not hold, as can be seen with the parameter set $d = 0.9$, $\beta = 0.8$, $\alpha = 1$ and $c_\kappa = 0.2$. Here it holds that the SC case is implemented if sponsored content is allowed. It holds that

$$\kappa^* = 0.816 > 0.8 = \kappa^*_{\text{NSC}}$$

so allowing sponsored content increases the network, but

$$W^* \approx 0.987 < 1.04 = W^*_{\text{NSC}}$$

so the total welfare decreases when allowing sponsored content. □

Proof of Proposition 15.

Given that the parameters imply the non-exclusive sponsored content case and our basic assumptions as e.g. $0 < \beta < 1$ it holds that

$$\begin{aligned} & W^* > W^*_{\text{NSC}} \\ \Leftrightarrow \quad & \frac{1}{2}\alpha(1 - 2c_\kappa(1 + d(1-\beta)) + c_\kappa^2(1-\beta) + d(2 - d\beta(2 - \beta - \beta^2)) > \frac{1}{2}\alpha(1-c_\kappa)(1 - c_\kappa + 2d) \\ \Leftrightarrow \quad & d(2c_\kappa + d(-2 + \beta + \beta^2)) > c_\kappa^2 \\ \Leftrightarrow \quad & \beta > \frac{\sqrt{9 + \frac{4c_\kappa(c_\kappa - 2d)}{d^2}} - 1}{2} \end{aligned}$$

To show the statement it is thus left to show, that

$$\sqrt{9 + \frac{4c_\kappa(c_\kappa - 2d)}{d^2}} \geq \sqrt{5} \tag{124}$$

or in other words

$$\frac{4c_\kappa(c_\kappa - 2d)}{d^2} \geq -4 \tag{125}$$

This is true independent on c_κ and d as

$$\frac{4c_\kappa(c_\kappa - 2d)}{d^2} = 4\frac{c_\kappa^2}{d^2} - 8\frac{c_\kappa d}{d^2} \overset{t:=\frac{c_\kappa}{d}}{=} 4t^2 - 8t \geq -4 \quad \forall t \tag{126}$$

□

Proof of Proposition 16. Let there be $c_\kappa > \tilde{c}_\kappa = 1 - \beta$ and $d > \bar{\bar{d}}$ with

$$d > \bar{\bar{d}} := \max\left\{\bar{d}(c_\kappa),\ \frac{(1-c_\kappa)^2 - \beta(1-2c_\kappa)}{2(\beta - 1 + c_\kappa)}\right\} \tag{127}$$

As $d > \bar{d}(c_\kappa)$ we know from Proposition 7 that the ESC case is implemented. Within the ESC case it holds due to Proposition I.4 that

$$\begin{aligned} & W^* > W^*_{\text{NSC}} \\ \Leftrightarrow \quad & \frac{1}{2}\alpha\beta(1 - 2c_\kappa + 2d) > \frac{1}{2}\alpha(1-c_\kappa)(1-c_\kappa+2d) \\ \Leftrightarrow \quad & \beta(1 - 2c_\kappa + 2d) > (1-c_\kappa)(1-c_\kappa+2d) \\ \Leftrightarrow \quad & \beta(1-2c_\kappa) + 2d\beta > (1-c_\kappa)^2 + 2d(1-c_\kappa) \\ \Leftrightarrow \quad & 2d\beta - 2d(1-c_\kappa) > (1-c_\kappa)^2 - \beta(1-2c_\kappa) \\ \overset{\star}{\Leftrightarrow} \quad & d > \frac{(1-c_\kappa)^2 - \beta(1-2c_\kappa)}{2(\beta - 1 + c_\kappa)} \end{aligned}$$

This does indeed hold per construction of $\bar{\bar{d}}$.
$\star$ Here we use that $c_\kappa > (1-\beta)$.

It holds that $\bar{\bar{d}} \geq \bar{d}(c_\kappa) > 0$ as per Proposition 6, which shows $\bar{\bar{d}} > 0$ □

Proof of Proposition 17. Who offers the contract does not change the behavior of the consumer. Thus, all results of the main model hold in regard to the consumer.
We deviate from the main model at the point, were the payment c_{SC} takes place. In contrast to the main model, where it optimal for the ISP to set the payment c_{SC} in a way that CSP B barely accepts the contract, CSP B will shape it in a way that the ISP barely accepts the contract, given that network capacitity κ has already been built. Consequently it has to hold that

$$\begin{aligned} & \pi_{\text{ISP, SC}}(\kappa) = \pi_{\text{ISP, NSC}}(\kappa) \\ \Leftrightarrow \quad & \bar{P}_{\text{SC}}(\kappa - \beta\alpha) + c_{\text{SC}}(\kappa) = \pi_{\text{ISP, NSC}}(\kappa) \\ \Leftrightarrow \quad & c_{\text{SC}}(\kappa) = \pi_{\text{ISP, NSC}}(\kappa) - \bar{P}_{\text{SC}}(\kappa - \beta\alpha) \end{aligned}$$

Note that the profit of CSP B is strictly increasing in c_{SC} meaning, that he will set it to the highest possible value. The ISP has no possibility within the model to punish the CSP for high payments, as he decreases his own profit when deciding not to offer the consumer the highest possible quota.
The ISP will thusly optimize his profit under the assumption, that he operates in the NSC case and will thus built $\kappa^*_{\text{NSC, main}} = \alpha(1 - c_\kappa)$.
Thus three situations are possible:

1. Profitability d is sufficiently high, that CSP offers a contract.
2. Profitability d is not high enough and no contract is offered
3. The built network capacity is not high enough (i.e. $\kappa < \alpha\beta$) and thus no contract can be accepted.

In the two last possibilities nothing changes relative to the NSC case. In the first case it holds that

$$\begin{aligned} W^* &= \pi_{\text{ISP}} + \pi_{\text{B}} + \pi_{\text{F}} + U \\ &= \bar{P}_{\text{SC}}(q{=}\kappa^*_{\text{NSC}}{-}\alpha\beta){+}c_{\text{SC}}(\kappa^*_{\text{NSC}}){-}c_\kappa\kappa^*_{\text{NSC}}{+}d{\cdot}(\alpha\beta){-}c_{\text{SC}}(\kappa^*_{\text{NSC}}){+}d{\cdot}(\kappa^*_{\text{NSC}}{-}\alpha\beta){+}0 \\ &= \bar{P}_{\text{SC}}(q = \kappa^*_{\text{NSC}} - \alpha\beta) - c_\kappa\kappa^*_{\text{NSC}} + d\kappa^*_{\text{NSC}} \\ &= \bar{P}_{\text{NSC}}(q = \kappa^*_{\text{NSC}}) - c_\kappa\kappa^*_{\text{NSC}} + d\kappa^*_{\text{NSC}} + \bar{P}_{\text{SC}}(q = \kappa^*_{\text{NSC}} - \alpha\beta) - \bar{P}_{\text{NSC}}(q = \kappa^*_{\text{NSC}}) \\ &= W^*_{\text{NSC}} - \text{CD}(\kappa^*_{\text{NSC}}) \end{aligned}$$

□

3 Impact of Near-Time Information for Prediction on Microeconomic Balanced Time Series Data using Different Machine Learning Methods

Source:

Frederik Collin and Martin Kies (2020). "Impact of Near-Time Information for Prediction on Microeconomic Balanced Time Series Data using Different Machine Learning Methods". In: *Available at SSRN.* DOI: `10.2139/ssrn.3559645`. URL: `https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3559645`

Co-Author:

Frederik Collin is a doctoral student at the Institute of Economics at Ulm University. He received his Masters of Science degree in Mathematics and Management from Ulm University in 2015.

He contributed in equal parts to the following article, including, but not limited to:

1. Development of the research questions
2. Literature review
3. Design and specification of used models
4. Development and programming of the used source code
5. Structuring and writing the article itself

Impact of Near-Time Information for Prediction on Microeconomic Balanced Time Series Data using Different Machine Learning Methods[*]

Frederik Collin[§], Martin Kies[‡]

March 23, 2020

Instead of relying solely on data of a single time series it is possible to use information of parallel, similar time series to improve prediction quality. Our data set consists of microeconomic data of daily store deposits from a large number of different stores. We analyze how prediction performance regarding a given store can be increased by using data from other stores. First we compare several machine learning methods, including Elastic Nets, Partial Least Squares, Generalized Additive Models, Random Forests, Gradient Boosting and Neural Networks using only data of a single time series. Afterwards we show that Random Forests are able to better utilize parallel time series data compared to Partial Least Squares. Using near-time data of parallel time series is highly beneficial for prediction performance. To allow a fair comparison between different machine learning methods, we present a novel hyper-parameter optimization technique using a regression tree. It enables a fast and flexible determination of optimal parameters for a given method.

Keywords: Time Series, Machine Learning, Forecasting, Nowcasting, Partial Least Squares, Random Forests, Neural Network, Hyperparameter Optimization

JEL Classification: C15, C32, C45, C53, C83

[*]We thank targens GmbH for the provision of the data set and a fruitfull collaboration. We are grateful for the feedback and support of our supervisors Georg Gebhardt and Sebastian Kranz.

[§]Ulm University, Institute of Economics, Helmholtzstraße 18, 89081 Ulm, Germany, e-mail: frederik.collin@uni-ulm.de

[‡]Ulm University, Institute of Economics, Helmholtzstraße 18, 89081 Ulm, Germany, e-mail: martin.kies@uni-ulm.de

1. Introduction

Forecasting microeconomic time series is of great interest for a broad range of applications. The prediction of cash flow for example can be used in liquidity planning.
The key contributions of this article are the following: We present an analysis of recent Machine Learning methods on a real microeconomic data set. Our data set consists of daily cash deposits of around 2000 European stores over a period of more than two years. We find that without Nowcasting all used methods except Elastic Net Regularization perform similar. The best method in this setting is the Neural Network which outperforms the worst by less than 6%. In addition we disentangle the (positive) Nowcasting effect from the (negative) effect of more noise due to more features. We show that combining Nowcasting and Random Forest outperforms the best classical approach by 24%. Finally, we present a novel technique to find hyperparameters in arbitrary settings. It can be used by all supervised learning methods and allows for variable objective functions. Our tree based technique improves upon Random Search by focusing the attention on particularly promising hyperparameter combinations in an assortment of hypercubes.
Nowcasting is the prediction of events close to or equal to the present. It has been utilized in weather forecasts for a long time and is still broadly used (for a recent Machine Learning approach see Xingjian et al. (2015)). Economists mainly use Nowcasting in the context of different release dates of macroeconomic indicators to forecast macroeconomic time series. Giannone et al. (2008) show that information contained in intra-monthly data releases helps to predict current-quarter real gross domestic product.
Throughout this paper we will refer to *Nowcasting*, whenever we use features that are close in time or simultaneous to the predicted response. More precisely we will use the term *Nowcasting* whenever we use information of other stores of the very same day to predict the deposits of a given store. This is in general of big interest as many money or cash streams, like bank account transfers, have to be checked and verified. One way to do so that does not rely on prior verification results is to build a prediction model. Comparing the observed with the predicted value allows to check for significant deviation. In our setting it is quite common that day deposits are already known but need to be checked prior to transfer. Checking a specific deposit can therefore be done utilizing information about same day deposits of other stores.
Several articles, such as Choi and Varian (2012), have used the advantage of Nowcasting to increase prediction power of economic forecasting tasks. Choi and Varian (2012) add Google Trends data of the present to predict near-term values of economic indicators. They find that adding Google Trends data increases prediction accuracy for all analyzed indicators. We add to this literature by showing that Nowcasting is also beneficial in predicting the next time step of a microeconomic time series. It might also be beneficial in similar microeconomic time series prediction tasks such as cash flow prediction.

Nowcasting as defined in this paper can reduce loss in at least two ways:
First, deposits of other stores of the same day potentially capture economic information that also impacts deposits of the store in question. On the one hand increasing sales of other stores might indicate an improving economic situation. Including these stores therefore helps us to capture the overall economic impact. On the other hand it could also be the case, that a change in sales of another store impacts the sales of the to be predicted store directly. If for example there are two rival restaurants and one of them introduces a new dish this could drive customers to them at the cost of its rival.
The other important factor is, that similar stores might be influenced similarly by exogenous non-economic shocks which are not part of the data set and potentially hard to quantify. Examples for such shocks are weather effects or regional events like a city wide festival.
Both, economic and non-economic factors, can be summarized as hidden or missing variables. Analogously to Ahmed et al. (2010) and Choi and Varian (2012) we use recent and broadly used methods and compare the prediction power of those. Additionally, we shed some light on the impact that Nowcasting has on different Machine Learning methods in predicting microeconomic time series. By adding features with different informational value we aim to disentangle the positive effect of recent information relative to the loss due to more noise.
All used Machine Learning methods need hyperparameter optimization. Many of them need different types (e.g. boolean or numeric or even abstract classes) of hyperparameters, but for a fair comparison the optimization process should be identical for all of them. To achieve this we developed a technique based on decision trees that searches for good hyperparameters without being restricted to a specific type. Another advantage of our approach is that it is not prone to the curse of dimensionality and that it is robust to local minima.
The outline of the paper is as follows. We start in Section 2 with a description of the data and present descriptive statistics. Section 3 explains our approach of analysis. A brief explanation of the used methods is given in Section 4. Section 5 explains the general procedure to find hyperparameters for these methods. We briefly elicit on our new approach of hyperparameter optimization. The main results are presented in Section 6. There, we also give a short note on the potential implications of the different prediction performances. We end the section by wrapping up the key findings. The paper ends with a discussion of potential drawbacks, future improvements and a research outlook in Section 7.

2. Data

Our data set has been provided through targens[1] by a business customer of theirs. The business model of this customer consists of allowing stores, mainly located in malls, to deposit their cash earnings into special safes. These safes register the deposited cash and electronically transfer the respective sum to the company of the store.
The data set we received consists of 2558 stores with data on a "per deposit"-level from 2016-06-20 to 2018-08-30, i.e. 802 days. We extracted our working data set with 1990 stores based on the following rules:

- There have been deposits within the first 5 weeks.
- There have been deposits within the last 5 weeks.
- The stores are located within Germany, Austria or Switzerland.

The first two rules aim to filter for stores which are part of the data set throughout the complete observation period. They aim to throw out those stores which have been added to or removed from the system. We did not filter based on seasonality, vacation time or similar effects. Stores which are closed for a longer period of time within the time frame do not receive a special treatment - the Machine Learning methods have to cope with such situations themselves.
Additionally to these filters, the data set was aggregated according to the following rules:

- Deposits which have been made in a currency other than Euro have been transformed to Euro based on daily exchange rates provided by Eurostat (2019).
- All deposits have been summed to daily data. In particular, the amount of deposited money on days without a deposit, including weekends and holidays, has been set to zero such that all stores have data points for all days of the observation period.

Figure 1 shows the allocation of stores according to a variety of classifications. It can bee seen, that with few exceptions most stores can be attributed to the industry classification *retail*. This is not the case on a company level. Nearly half of all featured companies belong to the industry classification *gastronomy*. While there is a sizable number of gastronomy companies they tend to have only one or very few associated stores. Retail companies on the other hand have a lot more stores per company within the data set. This assessment is supported by the fact that just over half of all stores belong to a single retail company. The data is geographically fairly evenly spread across different areas.
As most stores belong to the same industry, one might assume that they have deposits of a comparable size. This is not the case. The stores show considerable heterogeneity in their general deposit level. This can be seen in Figure 2 which depicts a density plot comparing the average deposits on Mondays.

[1]targens GmbH, Calwer Straße 33, 70173 Stuttgart, Germany, website: https://www.targens.de/en/

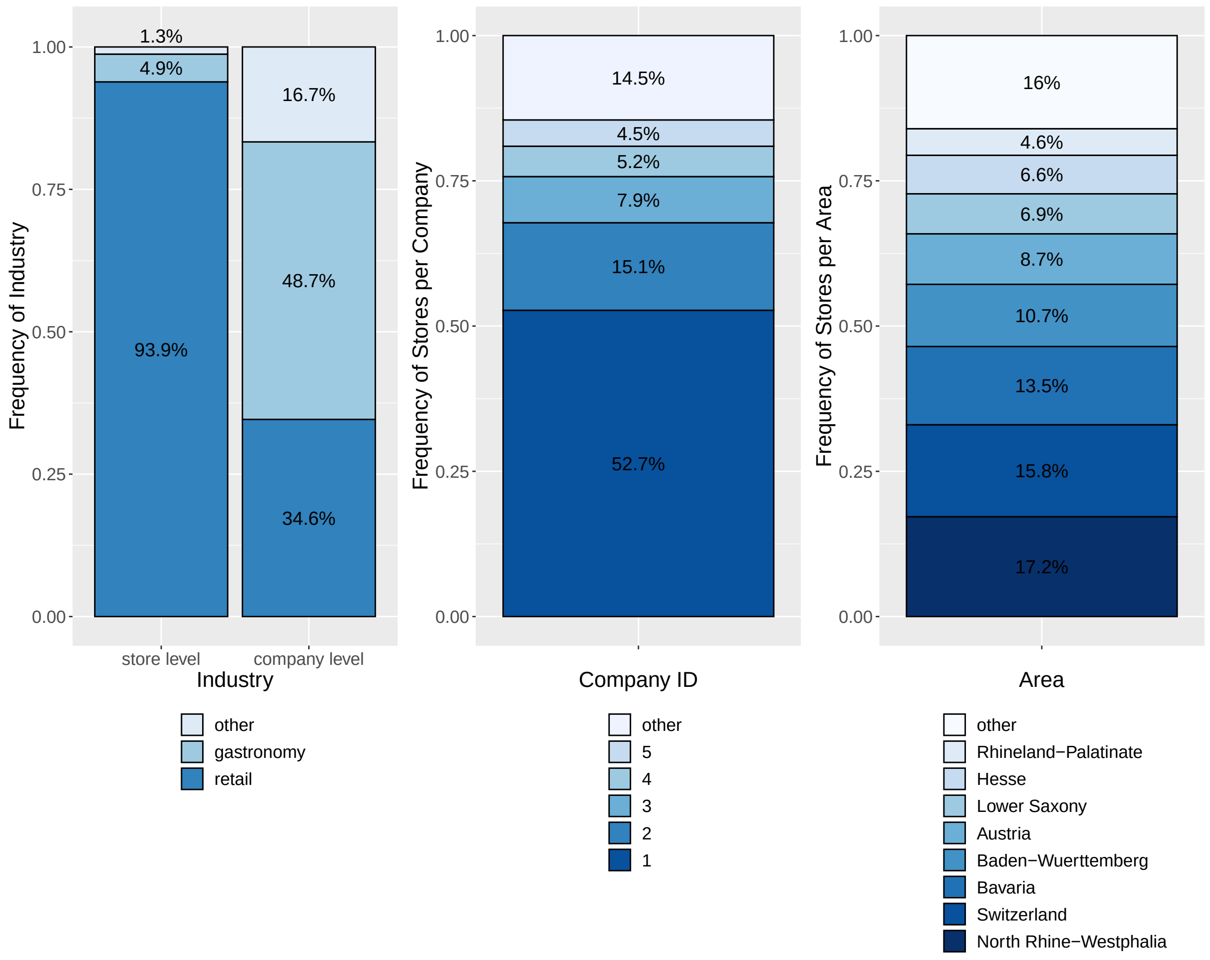

Figure 1: Descriptive statistic of the analysed data set after preparation. The first chart shows the frequency of the industry attribution on a per store and a per company basis. The second chart displays the frequency of stores per company, while the third one shows in which areas the stores are located.

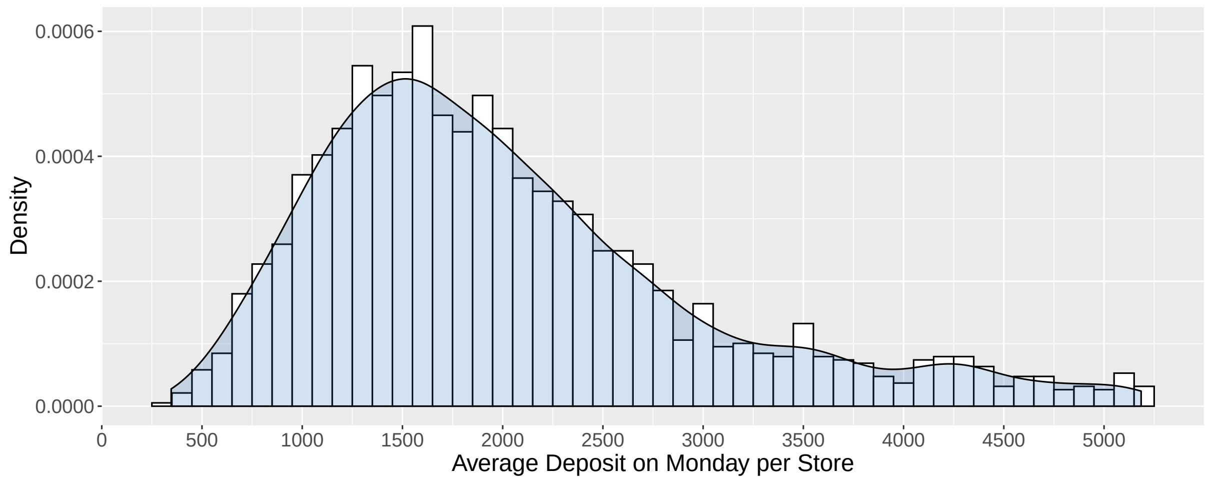

Figure 2: Density plot and histogram of the average deposit on Monday of all stores within the 95%-percentile.

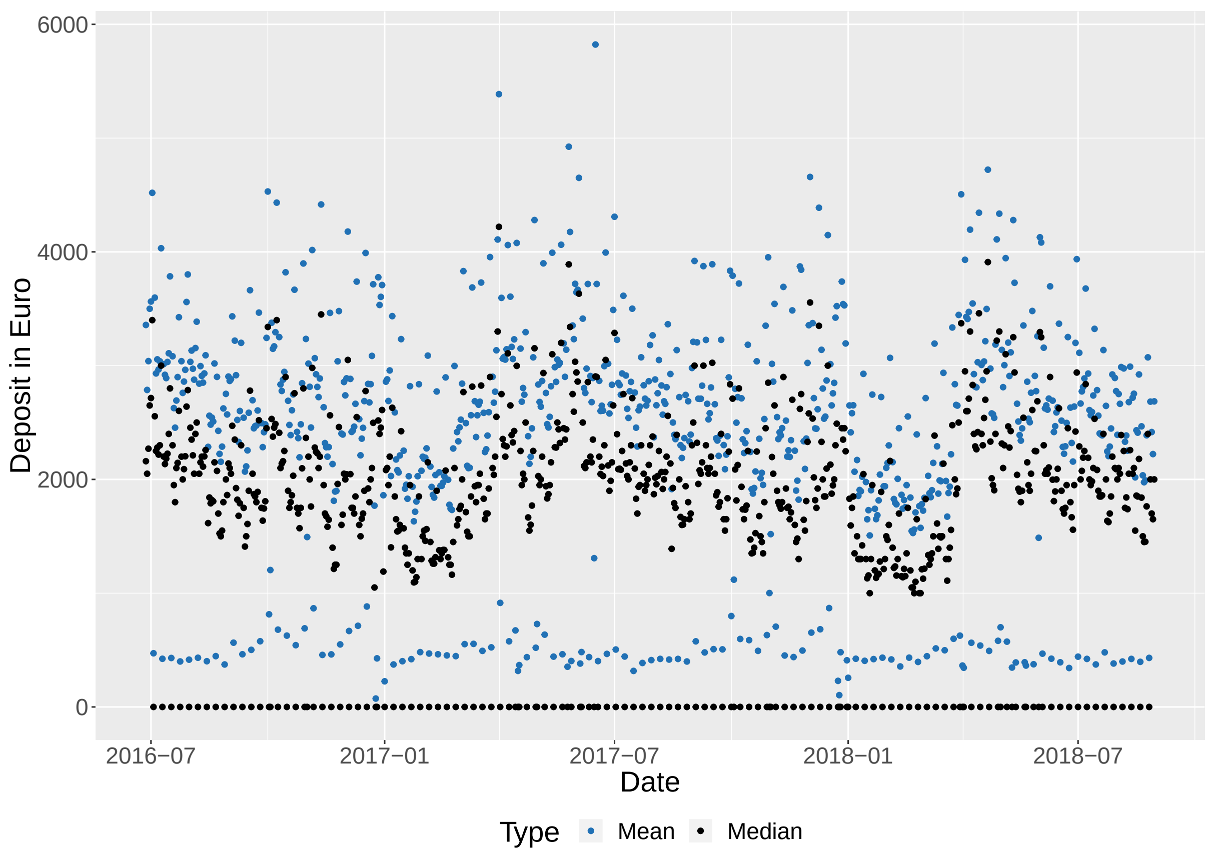

Figure 3: Time series of the representative, aggregated store.

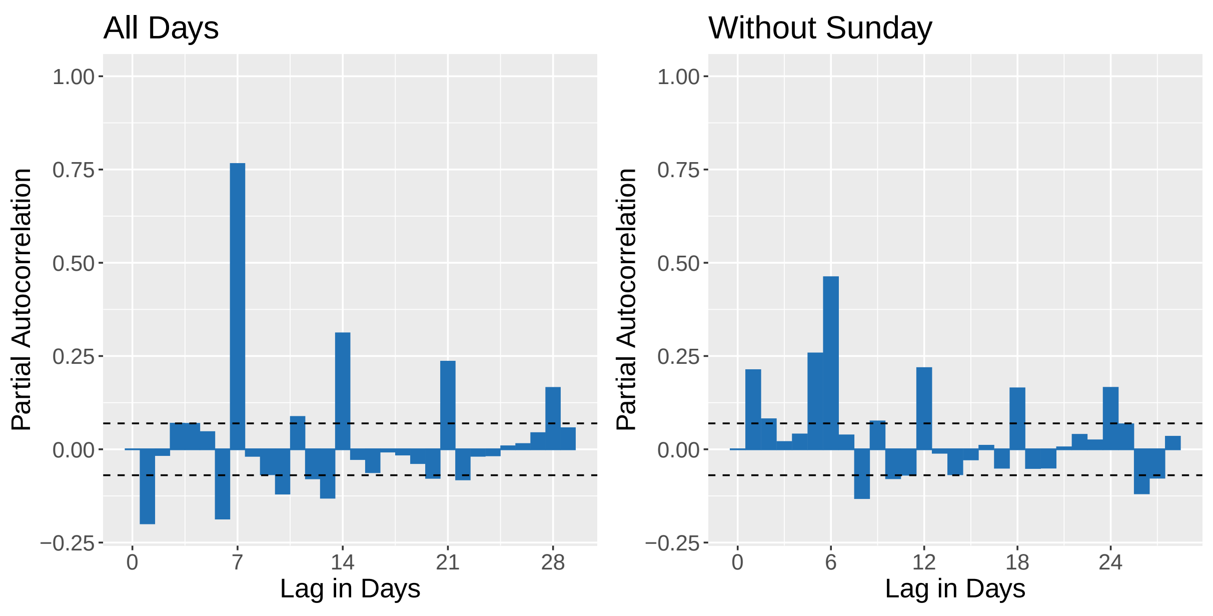

Figure 4: Partial autocorrelation based on the *stats::pacf()* function (R Core Team, 2018) with and without taking Sunday into account based on mean deposits. The dashed line is based on a white noise test using a 95% confidence interval.

A time series of the representative store, either as seen by the arithmetic mean or the median, is displayed in Figure 3. It can be seen that even when aggregating over all 1990 stores the deposits still show considerable variation. However, patterns over time can be identified. With the beginning of a new year for example there is a noticeable drop in the size of the deposits which persists over spring. Additionally, Figure 3 shows that several days exist, where the majority of stores do not make any deposits. Those days are holidays and Sundays.
Focusing on seasonality on a small time frame, Figure 4 depicts the partial autocorrelation of lagged days. It can be seen that deposits are significantly autocorrelated. Using all days as input there is particularly high correlation with a lag of 7 days, i.e. from the same weekday to the next. This is highly driven by the fact that most stores close on Sunday. It still holds true if one repeats the analysis on the data set after filtering out all Sundays. This analysis also shows that a higher amount of deposits the day before correlates with a higher deposit on the day in question. That this effect can not be seen in the analysis using all days can be attributed to the fact that Saturday has a higher deposit amount on average and is directly followed by Sunday.

3. Approach of Analysis

This section details how we set up our analysis. Our goal is to predict next day deposits for each store. Consequently, our response variable is the deposit of the next day. The models are trained on each store separately. All non-changing variables of the store, like industry or region, have therefore no relevance to the prediction task.
The first set of used features are weekday dummies. Figure 5 shows the mean and median deposits over all stores by weekday. It can be seen that there is variation during the week and in particular between workdays and Sunday. Adding weekday dummies allows the methods of Section 4 to capture these differences.

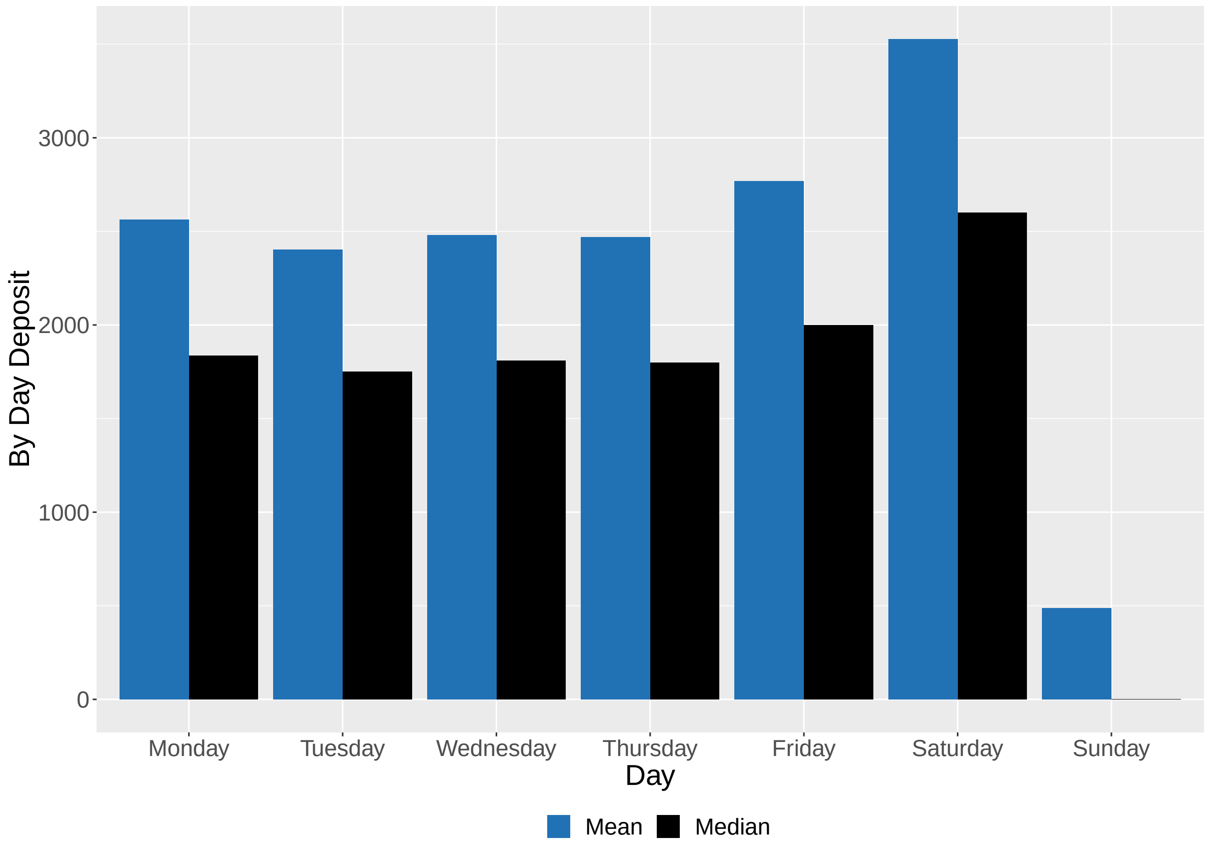

Figure 5: Aggregated daily deposits of the aggregated store based on the data set.

The second set of features are deposits of the past 7 days, aggregated on daily level. Recall Figure 4 from Section 2 showing partial autocorrelation of the past month. The highest correlation can be observed to the same day a week before. The second highest correlation is to the same day two weeks before but only about half in size. Considering the limited number of observations per store we decided to use 7 days. To further extract potential information incorporated in the data we added the change in deposits between each subsequent day as well as the sign of this change. The input available to the methods of Section 4 in the single time series prediction task is therefore:

- The day of the week, dummy coded in 7 different variables.
- The day-aggregated deposit of the past 7 days.
- The change in deposits between those previous 7 days, amounting to 6 different variables.
- The sign of change in deposits between those previous 7 days, amounting to 6 different variables.

In sum this forms a set of 26 features which will be referred to as the *standard* feature set for the remainder of the article. As each store is trained separately each combination of used Machine Learning method and store could have a separate hyperparameter optimization. We refrained from doing so to save computational resources. Instead we constructed a single hyperparameter set which is used for each store. To get a representative sample, we drew 20 out of the 1990 stores at random. Afterwards we optimized the hyperparameters of all methods of Section 4 over those 20 stores simultaneously. Based on the pairwise unique correlations among the 1990 stores we assumed that optimizing over 20 sample stores is sufficient. A density plot of those correlations is given in Figure 6. The average correlation between two unique stores time series in the training data is 0.54. The median correlation with a value of 0.62 is even higher, justifying our approach.

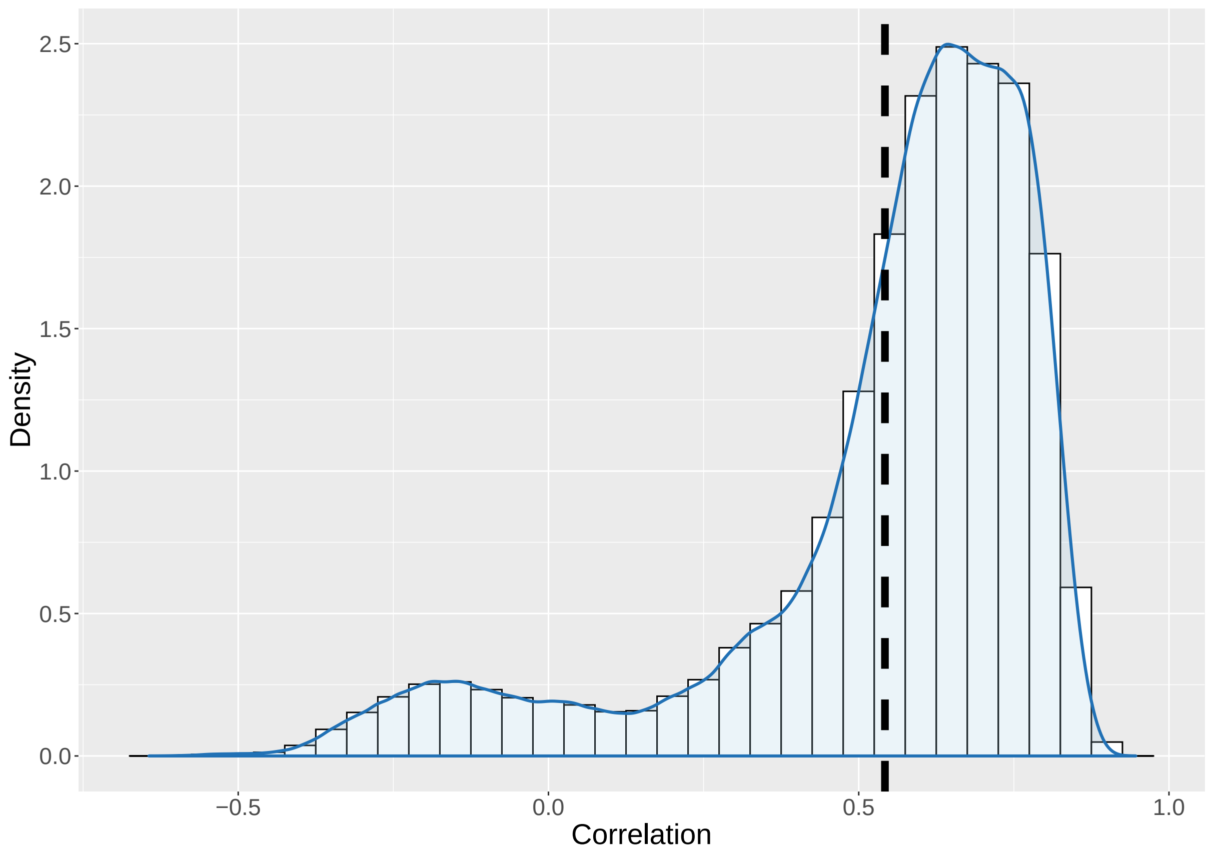

Figure 6: Pairwise unique correlation among stores in the training data. The dotted line depicts the average correlation.

The detailed approach of our hyperparameter optimization can be found in Section 5. Hyperparameter optimization takes place only on training data with further splitting within.

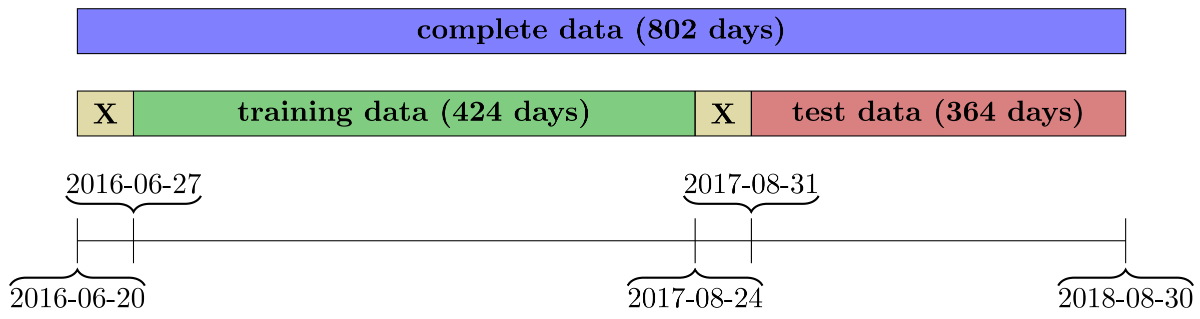

Figure 7: Visualization of the data splitting.

The splitting of our data set is visualized in Figure 7. Our data is a time series starting 2016-06-20, ending in 2018-08-30 and summing up to 802 days. We drop the first 7 days as the deposits of the previous 7 days are part of the feature space. To honor the time series structure we opted for the training and test data to be compact and the training dates to be chronologically before the test dates. We also drop 7 days between training and test data to ensure that any information used to predict a day in the test data is not contained in the training data. Finally we end up with training data from 2016-06-27 to 2017-08-24 and test data from 2017-08-31 to 2018-08-30. Thus we are left with 424 days to train and 364 days to test the methods of Section 4.

After optimizing hyperparameters for all methods using the training data and the 20 representative stores we trained one model for each method and each store. We used the same hyperparameters for each method/store combination and the standard feature set. Each model was then evaluated on the test data of their respective store. This will be called the *classical* prediction task and is equivalent to utilizing the standard feature set.

As mentioned in the introduction Nowcasting is beneficial in our setting. Recall that we defined Nowcasting in our specific context as using information of other stores from the very same day. To single out the effects of Nowcasting the above procedure is repeated in three variants on different feature sets for the methods Random Forest and Partial Least Squares. We chose Random Forest and Partial Least Squares for several reasons:

To cover potentially different impacts of Nowcasting on linear and non-linear methods we wanted representatives of both types. The methods had to be able to handle more features than observations and be comparatively fast in their execution, i.e. use a feasible amount of computational resources. Among those left we chose Random Forest as our non linear method due to its wide spread and its speed in high dimensional large feature sets. Partial Least Squares was the only linear method left in our setting which fulfilled these criteria.

The construction of the three additional different feature sets was done the following way:

- We added the deposits of all other stores of the 7 previous days to the standard feature set. We will refer to this as the *full* feature set.
- We added the deposits of all other stores of the to be predicted day to the standard feature set. We will refer to this as the *pure Nowcasting* feature set.
- We added the deposits of all other stores of the 7 previous days and of the to be predicted day to the standard feature set. We will refer to this as the *full Nowcasting* feature set.

The variation in contained information among the feature sets is depicted in Table 1.

Table 1: Feature Set Variation

Feature Set	**Past Information Own Store**	**Past Information Other Stores**	**Current Information Other Stores**
Standard	X	-	-
Full	X	X	-
Pure Nowcasting	X	-	X
Full Nowcasting	X	X	X

A priori it is not obvious which of these feature sets do have what kind of effect on our prediction performance: If the other stores contain important information, i.e. by smoothing out shocks within the last 7 days, one would expect their inclusion to improve prediction performance. On the other hand an inclusion implies an enormous increase in the number of free parameters for our models, i.e. less degrees of freedom, which might result in overfitting. Even with hyperparameter optimization spurious correlations might negatively impact performance. Similar considerations apply to the inclusion of up-to-the minute data, where the added informational value also has to be weighted against the decreasing degrees of freedom. Section 6 presents the results of these differentiations.

4. Studied Methods

In this section we briefly discuss the methods used in the analysis. Throughout this section we denote the number of features by k, the j-th feature by X_j, the response by Y and the set of all features by X. The total number of observations is denoted n. Time is denoted t. The t-th observation of feature j is therefore denoted X_{jt}.

4.1. Linear Regression

Linear Regression in general is perhaps the most often used Machine Learning method in economics. Ordinary Least Squares Regression, a specific variant of Linear Regression, is the best known variant. The key idea is to find the best linear combination of features to explain a response. The model takes the following form:

$$Y = \beta_0 + \sum_{j=1}^{k} \beta_j X_j$$

The main advantage is its robustness in prediction tasks and its interpretability. Linear Regression can be made much more powerful by using interaction terms between the features to allow more complex situations. To avoid overfitting it is generally advisable to only use those interactions that are economically plausible.
The specific form we choose for our model is the following:

$$Y_t = \alpha_0 + \sum_{i\in\{1,2,3,7\}} \beta_i Y_{t-i} + \sum_{j\in\{1,2,4,5,6,7\}} \gamma_j D_j + \sum_{j\in\{1,2,4,5,6,7\}} \delta_j Y_{t-1} D_j + \sum_{j\in\{1,2,4,5,6,7\}} \nu_j Y_{t-7} D_j$$

Where D_j is a dummy for weekday j ($j = 1$ corresponds to Monday).

In words, we interact the day before and the same day a week ago with weekday dummies and add the deposits made two and three days before today to explain today.
This form has been chosen due to the analysis given by Section 2 and Section 3. The most important days based on Figure 4, depicting autocorrelation, are the day before and the day exactly one week ago, respectively. We assumed this effect not to be identical for each day of the week. Due to the knowledge that the weekdays are quite influential, as can be seen in Figure 5, we interacted them with weekday dummies. The deposits of two and three days before the to be predicted day are added as they show substantial autocorrelation. We avoided using interaction effects here, as we wanted to keep the complexity of the model low and assumed any weekday effects to be sufficiently captured. Indeed, after-the-fact comparisons to alternative configurations show that this specific model is quite successful.

4.2. Generalized Linear Model with Elastic-Net Regularization

Penalized regressions such as Lasso or Ridge are a popular way of improving the out of sample prediction by reducing the variance of the estimator. Both methods rely on shrinking the regression coefficients towards zero. In contrast to Ordinary Least Squares Regression the methods work with a penalty that increases in the sum of absolute values (L_1-Norm) of the coefficients (Lasso) or the sum of squared (L_2-Norm) coefficients (Ridge).
While Ordinary Least Squares Regression minimizes the residual sum squared (RSS), i.e.

$$RSS = \sum_{i=1}^{n} \left(Y_i - \beta_0 - \sum_{j=1}^{k} \beta_j X_{ij} \right)^2$$

Ridge adds on this by inflicting a L_2-Norm penalty on the coefficients, thus minimizing

$$RSS + \lambda \sum_{j=1}^{k} \beta_j^2$$

where $\lambda \geq 0$ is a hyperparameter. Substituting the L_1-Norm for the L_2-Norm results in the method Lasso which minimizes

$$RSS + \lambda \sum_{j=1}^{k} |\beta_j|$$

where again $\lambda \geq 0$ is a hyperparameter.
The main difference between Lasso and Ridge is that with Lasso some coefficients will be set exactly to zero while with Ridge those coefficients will be small but typically non-zero. If one is interested in inference, Lasso might be easier to interpret, given the number of features k is large.
We use a method that combines Lasso and Ridge linearly called Elastic-Net (Zou and Hastie, 2005). The implementation used is the R-package *glmnet* (Friedman et al., 2019). The linear weight $0 \leq \alpha \leq 1$ is a hyperparameter balancing those two methods. The combined minimization problem can be written as:

$$RSS + \lambda \left((1-\alpha) \frac{1}{2} \sum_{i=1}^{k} \beta_i^2 + \alpha \sum_{i=1}^{k} |\beta_i| \right)$$

The extreme case $\alpha = 0$ corresponds to Ridge, $\alpha = 1$ to Lasso. Mixing both methods allows for more flexible solutions and thus potentially smaller loss.
With this method two hyperparameters, α and λ, have to be optimized. For both parameters we use the built-in optimizer provided by the R-package *glmnet* (Friedman et al., 2019) due to its computational efficiency.

4.3. Partial Least Squares Regression

Partial Least Squares Regression (PLSR) is a dimension reduction method using a linear model that can also be used for prediction tasks. It works similar to Principal Component Regression (PCR). Both methods use $M < k$ linear combinations $Z_1, ..., Z_M$ of the features as explanatory variables to describe the response Y. We will refer to those linear combinations, which are also known as latent variables or directions, as components. They can be written as:

$$Z_m = \sum_{j=1}^{k} \theta_{jm} X_j \quad \forall m \in \{1, ..., M\}$$

The weights θ_{jm} are real numbers. The response is predicted using

$$Y = \beta_0 + \sum_{i=1}^{M} \beta_i Z_i$$

PLSR uses both, variation in the features and in the response, to construct the weights θ_{jm} and thus the components. In this aspect PLSR can be seen as a supervised version of PCR. Several different implementations and variants of PLSR exist. Here, we present the algorithm as shown in James et al. (2013, p. 237f):
First all k features $X_1, ..., X_k$ and the response Y are standardized. As we update the features during this process we rename the standardized version of the feature X_j to $X_j^{(1)}$. Each of the features is regressed separately on the standardized response Y:

$$Y = \alpha_{j0} + \alpha_{j1} X_j^{(1)} \quad \forall j \in \{1, ..., k\}$$

The result are k slope coefficients $\hat{\alpha}_{j1}$. Those are used as weights θ_{j1} for the first component:

$$Z_1 = \sum_{j=1}^{k} \hat{\alpha}_{j1} X_j^{(1)} \quad \text{with} \quad \theta_{j1} = \hat{\alpha}_{j1} \quad \forall j \in \{1, ..., k\}$$

This guarantees that the first and as we will see all subsequent components place high weights on variables that have a high correlation with the response. After the first component has been computed one iteratively constructs weights that explain variation that has not been explained by previous components. This is done by subtracting the variation from the features that is already contained in the previous components.
To do so, first one regresses all corrected features separately on the most recently calculated component Z_{m-1}:

$$X_j^{(m-1)} = \gamma_{0j} + \gamma_{1j} Z_{m-1}$$

Based on the found γ one can calculate the estimated $\hat{X}_j^{(m-1)}$'s and thus the corresponding residuals $X_j^{(m)} = X_j^{(m-1)} - \hat{X}_j^{(m-1)}$. The information that has not been explained by the first $m-1$ components is captured by those residuals. They are taken as the corrected features for the consecutive steps.

Afterwards one regresses these k residuals separately on the standardized Y:

$$Y = \alpha_{j0} + \alpha_{jm} X_j^{(m)}$$

The result are k slope coefficients $\hat{\alpha}_{jm}$ that are used as weights in constructing the next component:

$$Z_m = \sum_{j=1}^{k} \theta_{jm} X_j^{(m)} = \sum_{j=1}^{k} \hat{\alpha}_{jm} X_j^{(m)} \quad m \geq 2$$

This procedure is repeated until all M components are found. Y can then be constructed as written in the beginning of this section. This model can now be used to predict response values in the same fashion as with Ordinary Least Squares Regression.
It remains the question on how to determine an optimal value of M. We used the so called *one-sigma* method to find the optimal number of components. This method is provided by the R-package *pls* (Mevik, 2019) which we used for Partial Least Squares Regression. It returns the model with the smallest number of components where the average root mean squared error on the prediction (RMSEP) on the validation set is within one standard deviation/error of the absolute optimum. This method is analogous to the *one standard error* rule described by Hastie, R. Tibshirani, and Friedman (2009, p. 244). Figure 8 illustrates the one-sigma method.

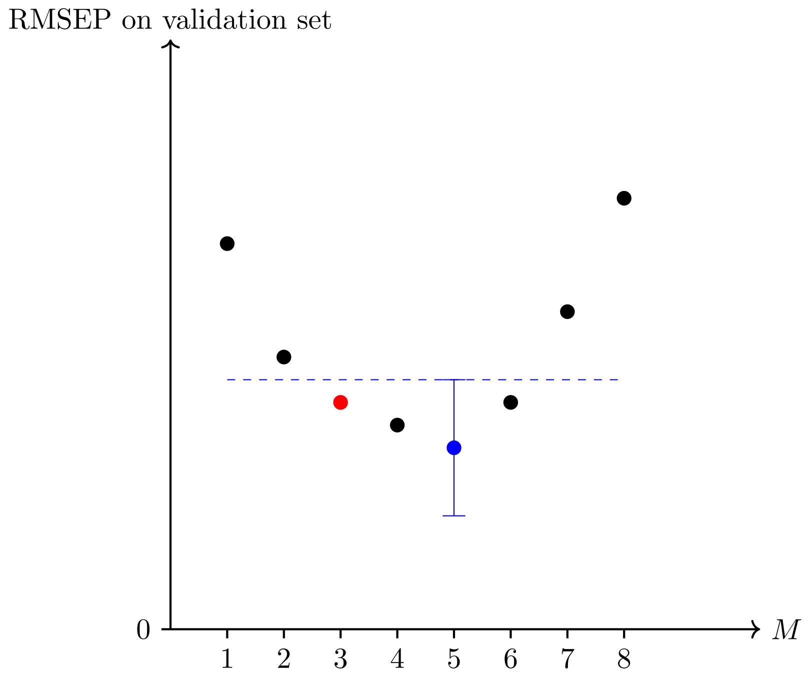

Figure 8: Visualization of the one-sigma method. In this case the optimal number of components M is chosen as 3.

In most cases there exists a number of components c^*, where for $M > c^*$ it holds that the RMSEP on the validation set increases with an increase in the number of components M. This effect is shown in Figure 8 for $M \geq 5$. Any model with more components than c^* is of no interest to us. Computing one additional component is not costless in time as it is necessary to execute $2k$ univariate regressions to do so. Combining those two insights we first selected a $M_{upper} < k$ that served as a maximum number of components. Afterwards we used the one-sigma method to choose the optimal number of components among those that have not more than M_{upper} components.

4.4. Generalized Additive Model

Generalized Additive Models (GAM) extend standard linear models by allowing non-linear functions on each feature. They have been developed by Hastie and R. Tibshirani (1986).
Given k features $X_j, \quad j \in 1, ..., k$ we switch from the linear form

$$Y = \beta_0 + \sum_{j=1}^{k} \beta_j X_j$$

to

$$Y = \beta_0 + \sum_{j=1}^{k} f_j(X_j)$$

by replacing each linear component $\beta_j X_i$ with a potentially non-linear but smooth function $f_j(X_j)$. The main advantage of GAM is providing non-linearity in the features while keeping the additivity of Linear Regression. This concept allows for a broad set of functions to be applied.
In contrast to Linear Regression it might not be possible to compute all functions f_j at once. Instead, they are computed iteratively. This procedure is called backfitting. One starts by calculating the first smoothing spline f_1. This is done using only information about the response Y and the first feature X_1. Afterwards one can calculate the estimator $\hat{f}_1(X_1)$ and repeat the following step to calculate f_l until all features have been considered:
Compute the residuals

$$\widetilde{Y} = Y - \sum_{i=1}^{l-1} \hat{f}_i(X_i)$$

Here one takes into account all functions f_j which were already calculated, i.e. up to up to $l-1$. Then one computes the l-th function f_l using information about $\widetilde{Y}$ and X_l only. The final model has the following form:

$$Y = \beta_0 + \sum_{j=1}^{k} \hat{f}_j(X_j)$$

For our implementation we used the R-package *gam* (Hastie, 2019). It uses smoothing splines for the functions f_j. The used smoothing spline can be interpreted as a natural cubic spline with a knot at every unique value of our feature. This amount of knots can lead to overfitting. For this reason they are controlled by setting the allowed degrees of freedom. Our only hyperparameter in this context therefore was the degrees of freedom of the smoothing splines.

4.5. Regression Tree

A Regression Tree is formed by iteratively dividing the feature space, consisting of values $X_{j1}, ..., X_{jn}$, into J non-overlapping regions $R_1, ..., R_J$. This is done via recursive binary splitting. One selects feature X_j and cut-point s such that

$$\sum_{i:X_{ji} \in R_1(j,s)} (Y_i - \hat{Y}_{R_1})^2 + \sum_{i:X_{ji} \in R_2(j,s)} (Y_i - \hat{Y}_{R_2})^2$$

is minimized. $R_1(j,s) = \{X|X_j < s\}$ and $R_2(j,s) = \{X|X_j \geq s\}$. The estimator $\hat{Y}_{R_1}$ is the mean response for the training observations in $R_1(j,s)$. Analogously, $\hat{Y}_{R_2}$ is the mean response for the training observations in $R_2(j,s)$. For the next splitting, one considers all features $X_1, ..., X_k$ and all possible values of the cut-point s for each of the features. Afterwards feature and cut-point are chosen such that the resulting tree has the lowest Residual Sum Squared (RSS). This procedure is repeated within already found regions until a stopping criterion is reached.
Prediction is done using the average over Y within the specific region R_r:

$$\hat{Y}(X|X \in R_r) = \hat{Y}_{R_r}$$

where $\hat{Y}_{R_r}$ is the mean response for the training observations in R_r to which the test observation X belongs. Specifically this means the prediction for each $X \in R_r$ is the same.
For the upcoming two sections we use the following notation for a tree:

$$f(X) = \sum_{i=1}^{J} \hat{Y}_{R_i} \mathbb{1}_{\{X \in R_i\}}$$

4.6. Random Forest

The idea of Random Forest is to combine basic tree learners to an ensemble. As base learners we used Regression Trees as described in Section 4.5. In particular we minimized RSS at each split of the basic tree learners.
Each tree is bootstrapped on samples from the training set. In contrast to Section 4.5 not all features are considered for each split. Instead the algorithm is restricted to only consider $m \leq k$ chosen features at random. The classical bagging method for trees is therefore a special variant of the Random Forest method with $m = k$. Using a subset of features reduces correlation between the trees relative to classical bagging.
After having grown l such trees ($\hat{f}_i$, $i \in \{1, ..., l\}$) they are averaged in order to obtain an aggregated estimate of Y given X. The predicted value for a given X is:

$$\hat{f}(X) = \frac{1}{l} \sum_{i=1}^{l} \hat{f}_i(X)$$

with $\hat{f}_i(X)$ being the prediction of the i'th tree given X.

We used the R-package *grf* (J. Tibshirani et al., 2019) for our main analysis of Random Forests in the context of Nowcasting. As a robustness-check the R-package *ranger* (Wright et al., 2019) has been used for the classical approach as well.
We considered the following tuning parameters:

- The sample fraction of each tree. This is the percentage of bootstrapped data used to construct the tree. Reducing the sample fraction reduces the correlation among trees at the cost of making each tree weaker.
- The number of randomly drawn feature to be considered at each split m. A lower m reduces the correlation among trees but makes them individually weaker.
- The minimum number of observations in each leaf. A lower minimum number of observations per leaf reduces the correlation among trees but might result in overfitting.
- The maximum imbalance of a split (only *grf*). Larger values result in smaller and therefore weaker trees but reduces the correlation between them.

The number of trees used to built the forest has been set exogeneously by us without using hyperparameter optimization. Increasing the number of trees always reduces model loss in expectation at the expense of training time. Our optimization algorithm will therefore find the optimal parameter to be at the right edge of the provided interval. Consequently we chose a value of 1000 trees manually.

4.7. Gradient Boosting

The idea of Boosting is to add basic tree learners sequentially. Each additional iteration aims to explain variation that has not yet been explained. In contrast to a Random Forest the trees are therefore not calculated independently from each other. For the method Gradient Boosting we also used Regression Trees (Section 4.5) as base learners, analogously to the method Random Forest.
One starts by setting $\hat{f}(X) = 0$ and $\widetilde{Y} = Y$. With each iteration $i \in \{1, ..., l\}$ a Regression Tree $\hat{f}_i(X)$ with features X and response $\widetilde{Y}$ is fitted. Within each iteration the function $\hat{f}(X) = \hat{f}(X) + \eta \hat{f}_i(X)$ is updated by adding a shrunken version of $\hat{f}_i$. The shrinking parameter $\eta \in (0, 1]$ can be interpreted as the learning rate. The response $\widetilde{Y} = \widetilde{Y} - \eta \hat{f}_i(X)$ is updated analogously. After l steps our final model is written as:

$$\hat{f}(X) = \sum_{i=1}^{l} \eta \hat{f}_i(X)$$

In words: Each additional tree learner tries to explain the variation still remaining in the response as measured by the still present residuals. Similarly to Random Forests each tree is generated on a subset of both training data and feature variables to allow for better generalization.

We used the implementation of the R-package *xgboost* (T. Chen et al., 2019) and considered the following tuning parameters:

- The maximum number of boosting iterations l. A higher number of boosting iterations increases the performance on the training data but might lead to to a worse performance on the test data, i.e. produce overfitting.
- The maximum depth of a single tree. Similarly to the number of iterations, a higher depth of the trees makes each individual tree stronger at the cost of overfitting. A minimum depth might be required if there are very complex relations among the feature space variables.
- The learning rate η. A higher rate of η reduces the number of necessary iterations. However, setting a lower rate allows the algorithm to generalize better.
- The number of randomly drawn feature to be considered for a specific tree. Similar to the approach of Random Forests, it is prudent not to use the complete feature space to avoid overfitting. On the other hand one has to use a sufficient number of feature variables to guarantee, that all relevant relations can be found by the base learners.
- The sample fraction of training data chosen before growing each tree. Using a lower sample fractions reduces the correlation among trees, which is especially useful in combination with a low learning rate.
- The minimum among the number of observations in each leaf. Increasing the minimum number of observations per leaf makes each tree individually weaker but allows them better to generalize, thus reducing overfitting.

4.8. Neural Networks

Neural Networks are a powerful non-linear supervised learning method which is used in a lot of different areas. Examples include image classification (Krizhevsky et al., 2012) and prediction tasks (see e.g. Rajkomar et al. (2018) for predicting the mortality of hospital patients). An excellent introduction is given in Nielsen (2019).

A standard Neural Network consists of several so called *neurons* which are arranged into different *layers*. The layers which are only used internally for computations are called *hidden*. Each neuron is connected to all other neurons of the previous layer using individual weights w_i, $i \in \{1, ..., \#$ of neurons in previous layer$\}$. Additionally to the weights, each neuron has an individual activation threshold b. There are several ways to combine the weights and the threshold using so called *activation functions*. We used the *sigmoid* function

$$\text{Output} = \frac{1}{1 + e^{-(wx+b)}}$$

for each hidden neuron and a simple linear combination for the output layer. In this case x is the output vector of the neurons of the previous layer.

The weights w and the activation threshold b are chosen to minimize the distance between the calculated output of the network and the desired one. By doing so, logic gates emerge. Providing a Neural Network with sufficiently many neurons these logic gates allow the approximation of arbitrarily complex functions (Nielsen, 2019). A schematic illustration of a Neural Network is depicted in Figure 9.

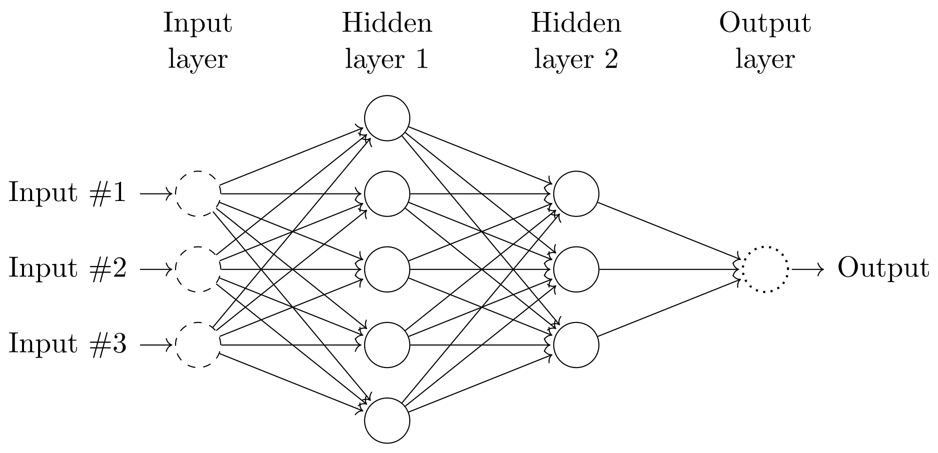

Figure 9: A schematic illustration of a Neural Network, as shown in Kies (2020), outputting a single value using two hidden layers of 5 and 3 hidden neurons.

In our case we used hyperparameter optimization to find the optimal number of neurons given up to three hidden layers. *Gradient Descent* was used to decide which of the weights should be changed how strongly. The specific variant chosen was the *Adam* optimizer (Kingma and Ba, 2014).

To increase the power of the Neural Network we used two common techniques:

- Neural Networks tend to work best with inputs which correspond to the output of their activation functions (i.e. between 0 and 1 with the sigmoid function). Thus we always normalized each input variable X_j as well as the response variable Y and re-normalized the response when predicting the actual output. This was done by linearly transforming each variable separately. The highest value of a given variable in the training data was mapped to 1. Correspondingly the lowest value in the training data was mapped to 0. In the test data identical transformations were applied so that here values below 0 and above 1 could be observed.
- We allowed the hyperparameter optimization to set dropout levels based on the method of Srivastava et al. (2014). The key idea is to randomly drop neurons, or inputs in the case of input dropout, within the training step. This forces the Neural Network to not rely to much on single connections which decreases overfitting.

In addition to the number of neurons within the layer and their dropout rates, the hyperparameter optimzation process optimized over the *batch size* and the number of *epochs*. Both parameters influence the duration of training and are very influential on how well the Neural Network generalizes.

5. Hyperparameter Optimization

Each of the Machine Learning methods discussed in Section 4 has several degrees of freedom in its exact functionality. These parameters which describe the functionality of the method, e.g. the number of neurons of a Neural Network, are called *hyperparameters*. Some methods, such as *glmnet* from Friedman et al. (2019) provide a built-in functionality to estimate good values for single parameters and several methods provide good default values for hyperparameters. However, most methods have the need to set some of their parameters explicitly and/or override some of their default values to increase performance. This section explains how we decided on the hyperparameters used to generate the results of Section 6.

Each store has its own optimal combination of hyperparameters for each method. It is very time consuming to find an idiosyncratic combination of hyperparameters for each store. As a more stable and efficient variant we used an identical combination of hyperparameters for each store. Given such a combination, we calculated the performance in regard to a given store by calculating an *out-of-sample root mean square error* (RMSE) of *nested backwards looking windows* using normalized values. A graphical schematic of the concept of nested backwards looking windows is given in Figure 10. The pseudo-code follows in Figure 11.

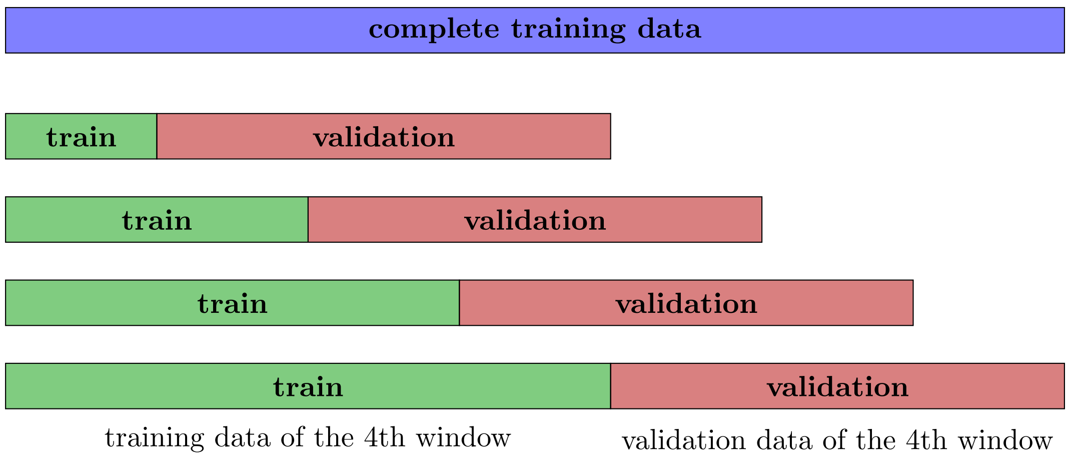

Figure 10: Sketch to show the evaluation of a hyperparameter combination on a single store using nested backwards looking windows. In this example we draw 4 windows.

Using these backwards looking windows allows us to avoid any possible contamination due to data points which can't be known at the time of training the model. We chose the length of the validation window to be 92 days. This way, we always considered 3 full months of data. The number of windows was chosen as $k_{\text{wind}} = 10$.

We wanted the representative hyperparameter set to be applicable to all stores. To counteract the considerable heterogeneity among them (see Figure 2) and in time (see Figure 3) we therefore normalized the (predicted) response variables before calculating individual RMSE. Not using normalized values would imply that a store with higher average deposits would get over weighted when determining optimal parameters.

Figure 11 Evaluation of a hyperparameter combination on store level.
Note that the functionality of the provided source code is greater and more flexible and thus able to deal with a greater variety of input data. For readability purposes, we opted to only display the functionality necessary for our final analysis. The same holds true for the other pseudo-code figures in this section.

Precondition:

h_i is a combination of hyperparameters.
$\mathcal{T}_{\text{train}}$ are the days available for training. We assume they are an ordered array with indexing starting at 1.
$X_{train} = \{x_{j,t},\ t \in \mathcal{T}_{\text{train}}\}$ is the training data of store j available at time t as defined by Section 3.
$Y_{train} = \{y_{j,t},\ t \in \mathcal{T}_{\text{train}}\}$ is the to be predicted values of store j at time t as defined by Section 3.
k_{win} is the number of windows which we want to use for the validation (in our setting 10)
len_{test} is the number of days in the test data set (in our setting 92).

```
1:  function EvaluateHyperParOnStore($h_i, X_{train}, Y_{train}$)
2:      for $k \leftarrow 1$ to $k_{\text{win}}$ do
3:          $t_{k,\text{start}} \leftarrow \mathcal{T}_{\text{train}}\left[1 + \left\lfloor \frac{|\mathcal{T}_{\text{train}}| - len_{\text{test}}}{k_{\text{win}}} k \right\rfloor\right]$
4:          $\mathcal{T}_k^{\text{val}} \leftarrow \{t \in \mathcal{T}_{\text{train}} : t_{k,\text{start}} \le t < t_{k,\text{start}} + len_{\text{test}}\}$   ▷ $\mathcal{T}_k^{\text{val}}$ defines the $k$-th window.
5:          $\mathcal{T}_k^{\text{train}} \leftarrow \{t \in \mathcal{T}_{\text{train}} : t < t_{k,\text{start}}\}$   ▷ $\mathcal{T}_k^{\text{train}}$ defines the training data available for the $k$-th window.
6:          Train the method using $h_i$ on training data $X_{train}$ and $Y_{train}$ limited to $\mathcal{T}_k^{\text{train}}$
7:          Predict $\{\hat{y}_{j,t},\ t \in \mathcal{T}_k^{\text{val}}\}$ using $\{x_{j,t},\ t \in \mathcal{T}_k^{\text{val}}\}$
8:          for $t \in \mathcal{T}_k^{\text{val}}$ do
9:              $y_{j,t,k}^{\text{norm}} \leftarrow \frac{y_{j,t} - \bar{y}_{j,t}}{sd\left(y_{j,t},\, t \in \mathcal{T}_k^{\text{val}}\right)}$
10:             $\hat{y}_{j,t,k}^{\text{norm}} \leftarrow \frac{\hat{y}_{j,t} - \bar{y}_{j,t}}{sd\left(y_{j,t},\, t \in \mathcal{T}_k^{\text{val}}\right)}$
11:                 ▷ $\bar{y}_{j,t}$ is the mean and $sd\left(y_{j,t},\ t \in \mathcal{T}_k^{\text{val}}\right)$ the standard deviation of $y_{j,t},\ t \in \mathcal{T}_k^{\text{val}}$
12:         $rmse_{j,k}^{\text{norm}} \leftarrow \sqrt{\frac{1}{|\mathcal{T}_k^{\text{val}}|} \sum_{t \in \mathcal{T}_k^{\text{val}}} (y_{j,t,k}^{\text{norm}} - \hat{y}_{j,t,k}^{\text{norm}})^2}$
13:     $rmse_j^{\text{val, norm}} \leftarrow \frac{1}{k_{\text{win}}} \sum_{k=1}^{k_{\text{win}}} rmse_{j,k}^{\text{val, norm}}$
14:     return $rmse_j^{\text{val, norm}}$
```

The actual evaluation of a given combination of hyperparameters is the averaged evaluation over single stores. Ideally, one would aggregate over all stores but this is very time-consuming. Instead we opted for drawing 20 random stores out of the data set as a *representative* example. The pseudo-code of the evaluation of a given combination of hyper parameters is given in Figure 12. Having defined a method to evaluate a combination of hyperparameters the question remains on which combinations should be tested. Besides manual tuning the two most widely used approaches to generate possible combinations are *grid search* and *random search* (Bergstra and Bengio, 2012). Grid search calculates intermediary values for each parameter using a specified step size. Afterwards all combinations of all parameters over all these intermediary values are evaluated. Adaptive, sequential variants circle in on the most promising combination by adjusting the step size and the range of parameters. The big drawback of this method is the curse of dimensionality - if one has a lot of hyperparameters, taking and evaluating the

Figure 12 Evaluation of a hyperparameter combination on data set level.

Precondition:

h_i is a combination of hyperparameters.
$\mathcal{T}_{\text{train}}$ are the days available for training. We assume they are an ordered array with indexing starting at 1.
$\mathcal{J} \subseteq \{1, ..., N\}$ is the set of stores chosen as the representative example of the N stores.
$\{x_{j,t},\ t \in \mathcal{T}_{\text{train}},\ j \in \mathcal{J}\}$ is the set of training data of the example stores.
$\{y_{j,t},\ t \in \mathcal{T}_{\text{train}},\ j \in \mathcal{J}\}$ is the set of to be predicted values of the example stores.

```
1: function EvaluateHyperPar($h_i, \{x_{j,t},\ t \in \mathcal{T}_{\text{train}},\ j \in \mathcal{J}\}, \{y_{j,t},\ t \in \mathcal{T}_{\text{train}},\ j \in \mathcal{J}\}$)
2:     for $j \in \mathcal{J}$ do
3:         $rmse_j^{\text{val, norm}} \leftarrow$ EvaluateHyperParOnStore($h_i, \{x_{j,t}, t \in \mathcal{T}_{\text{train}}\}, \{y_{j,t}, t \in \mathcal{T}_{\text{train}}\}$)
4:     $rmse^{\text{val, norm}} \leftarrow \frac{1}{|\mathcal{J}|} \sum_{j \in \mathcal{J}} rmse_j^{\text{val, norm}}$
5:     return $rmse^{\text{val, norm}}$
```

combination of all of them is quite resource intensive. A surprisingly effective alternative is simply drawing random combinations within the provided range (Bergstra and Bengio, 2012) and taking the best scoring combination.

Our approach, called *Hyperpar-Tree*, is a natural combination and extension of those two ideas. We wanted a hyperparameter optimization method which:

- Is easy to implement.
- Works for a large set of methods without making a priori assumptions about the mode of action and possible interactions of their parameters.
- Is able to work well with a wide range of different parameter types. In other words, we want to simultaneously optimize hyperparameters, where some of them might assume any numeric value, some are restricted to integers and some of them are chosen out of an arbitrary set of fixed strings.
- Is able to work well with complex interaction effects.
- Does not suffer (much) from the curse of dimensionality, e.g. is able to scale well to a large number of hyperparameters.
- Is able to circle in on the best combination without getting stuck in a local optimum.

Hyperpar-Tree aims to achieve all of those goals. As the starting input, the algorithm receives a range defining the minimum and maximum value or a list of allowed characterizations for each hyperparameter. These can be set based on general considerations like time constraints, the legal range of a parameter or a focus on promising areas. After having defined these inputs a random search is initialized to provide a data base for further calculations. On this data base a decision tree is executed that estimates a score, e.g. the root mean squared error, based on the given hyperparameters. Figure 13 depicts a graphical schematic of such a tree.

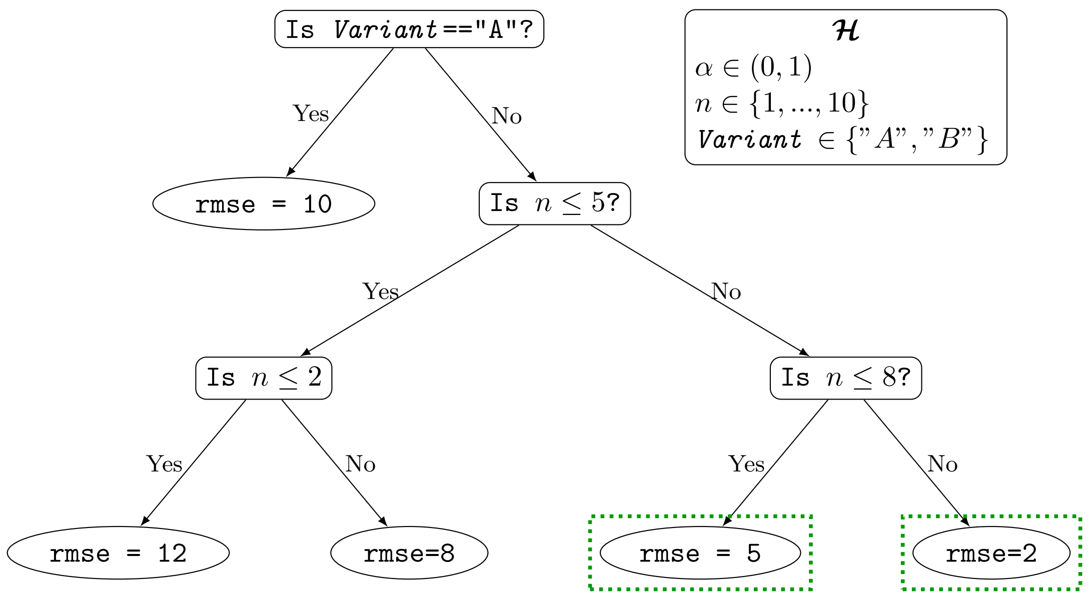

Figure 13: Sketch to show an example of a decision tree. In this example there are three hyperparameters of different types. α is assumed to have no significant effect on the *rmse*-value. If the two best leafs are chosen (highlighted in green), than the next combinations are drawn out of the ranges $\{\alpha \in (0,1), n \in \{6,7,8\}$, Variant $\in \{$“B”$\}\}$ and $\{\alpha \in (0,1), n \in \{9,10\}$, Variant $\in \{$“B”$\}\}$

By design each leaf of the decision tree defines a hypercube in the dimension space of the hyperparameters and thus defines new ranges. New combinations of hyperparameters are drawn randomly out of those ranges, defined by the most promising leafs, and are evaluated. Not just concentrating on the very best leaf allows us to focus on the most promising regions while still trying to minimize the chance of getting stuck in a local optimum. Broadly speaking using a higher number of leafs slows down learning but increases the probability to find the global optimum.

Afterwards the worst combinations are deleted to reduce the variance of the tree and allow for a circling in on the best answer without having to increase the complexity of the tree itself. The pseudo-code of executing *Hyperpar-Tree* on our specific situation is given in Figure 14.

In our case we drew 100 different combinations of hyperparameters out of the starting ranges and added 20 new ones each for 10 iterations based on the 3 best leafs. This resulted in 300 tested combinations total per method. Each iteration 15 combinations were removed. The final size of the data set from which the best RMSE was chosen was therefore 150.

The other specific parameters for the tree and the *Hyperpar-Tree* method are stated in Appendix A as well as the specific input ranges for the methods. These ranges have been chosen after a quick manual optimization to avoid spending computational resources on obviously low performing parameter settings.

Figure 14 Finding the optimal combination h^{opt} of hyperparameters for a given method.

Precondition:

$\mathcal{H}$ is the set of all possible combinations of hyperparameters as defined by the method itself.
$\mathcal{T}_{\text{train}}$ are the days available for training. We assume they are an ordered array with index starting at 1.
N is the number of stores in our data set.
$X^N_{train} = \{x_{j,t},\, t \in \mathcal{T}_{\text{train}},\, j \in \{1, ..., N\}\}$ is the training data.
$Y^N_{train} = \{y_{j,t},\, t \in \mathcal{T}_{\text{train}},\, j \in \{1, ..., N\}\}$ are the to be predicted values.
n_{start} is the number of randomly drawn combinations of hyperparameters (in our setting 100).
$n_{\text{best.splits}}$ is the number of the best leafs which should be considered (in our setting 3).
n_{add} is the number of new combinations of hyperparameters added in each cycle of *Hyperpar-Tree* (in our setting 20).
n_{rem} is the number of combinations of hyperparameters removed in each cycle of *Hyperpar-Tree* (in our setting 15).
n_{rep} is the number of cycles of *Hyperpar-Tree* (in our setting 10).

```
1:  procedure OptimizeHyperPar
2:      Manually define a set $\mathcal{H}^{\text{man}}$ of ranges and suitable options in regards to each hyperparameter based on $\mathcal{H}$.  ▷ We do not want our hyperparameters to have obviously bad values or values which are too time-consuming to evaluate
3:      Draw a number of random stores, defining $\mathcal{J}$
4:      ▷ Start Hyperpar-Tree
5:      Draw $n_{\text{start}}$ many combinations $h_i,\, i = 1, ..., n_{\text{start}}$ of hyperparameters out of $\mathcal{H}^{\text{man}}$.
6:      for $i \leftarrow 1$ to $n_{\text{start}}$ do
7:          $rmse_i^{norm} \leftarrow$ EvaluateHyperPar($h_i, X^{\mathcal{J}}_{train} = \{x_{j,t},\, t \in \mathcal{T}_{\text{train}},\, j \in \mathcal{J}\}, Y^{\mathcal{J}}_{train} = \{y_{j,t},\, t \in \mathcal{T}_{\text{train}},\, j \in \mathcal{J}\}$)
8:      $H \leftarrow$ List of combinations of $h_i$ and corresponding $rmse_i^{norm}$
9:      for $j \leftarrow 1$ to $n_{\text{rep}}$ do
10:         $Tree \leftarrow$ Tree($H$)  ▷ We use Wright et al. (2019), but other implementations can be used as well
11:         for $k \leftarrow 1$ to $n_{\text{add}}$ do
12:             $Leaf_k \leftarrow$ Random sample of $1, ..., \min\{n_{\text{best.splits}}; Tree.$NumberOfLeafs$\}$
13:             $\mathcal{H}_k \leftarrow$ Range spanned by $Tree.$RangeOfLeaf$[Leaf_k]$    ▷ Assuming RangeOfLeaf to deliver the ranges ordered from best to worst leaf and indexing starting with 1
14:             Draw one random combination of hyperparameters $h_k$ out of $\mathcal{H}_k$.
15:             $rmse_k^{norm} \leftarrow$ EvaluateHyperPar($h_k, X^{\mathcal{J}}_{train}, Y^{\mathcal{J}}_{train}$)
16:         Append $H$ with $h_k$ and corresponding $rmse_k^{norm}$, $k \in 1, ..., n_{\text{add}}$
17:         Delete the entries with the $n_{\text{rem}}$ worst $rmse^{norm}$ out of $H$.
18:     $h^{\text{opt}} \leftarrow h$ out of $H$ with the best $rmse^{norm}$
```

6. Results

We predicted daily deposits given the optimal parameters from Section 5, using the feature sets described in Section 3 based on the methods of Section 4. A graphical visualization of the performance of the different methods using the standard feature set, i.e. only backwards looking data from the same store, is given in Figure 15. The specific numbers can be found in Table 15 and Table 16 in Appendix B.

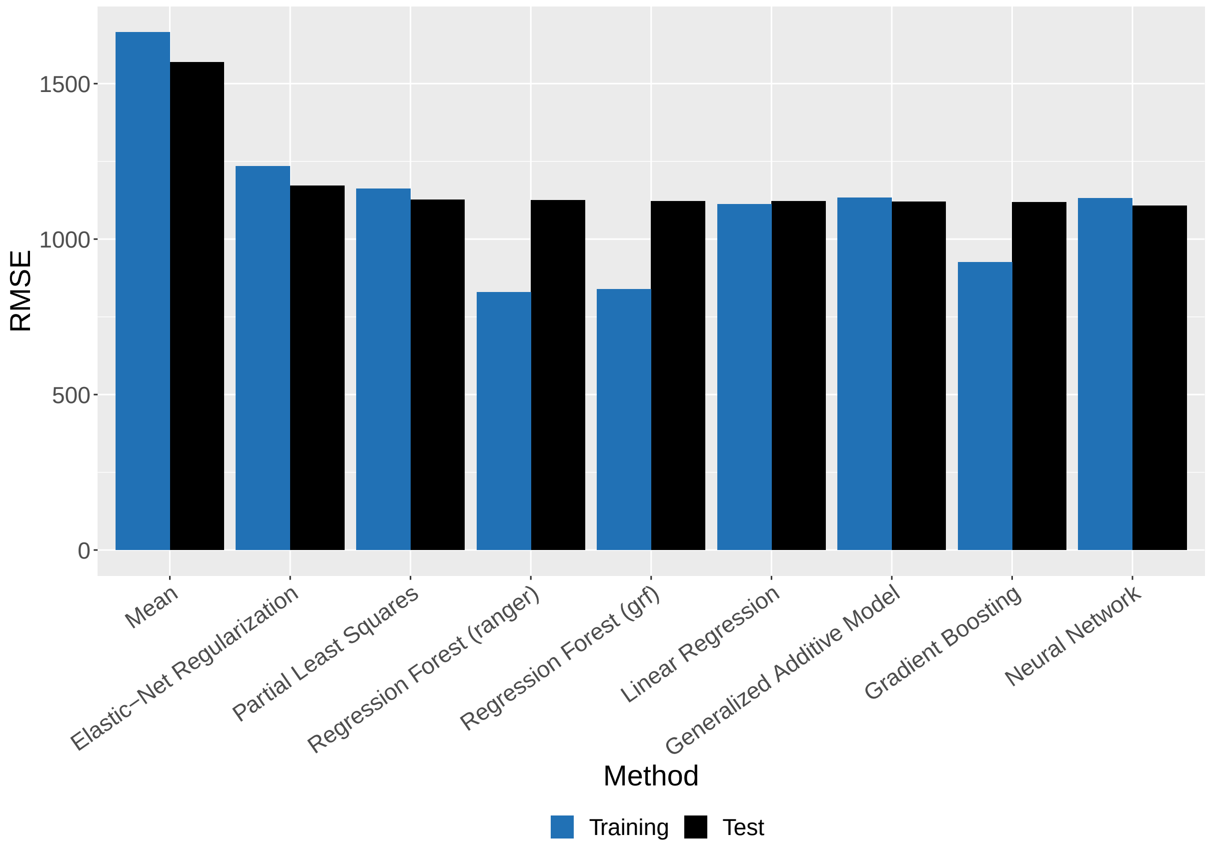

Figure 15: Mean of the root mean squared errors of all stores in the training and test data set for each analysed method. Only the standard feature set as specified by Section 3 has been used. The RMSEs have been ordered by their performance on the test data set. As a basic benchmark the method Mean is included, which predicts the deposit based on the arithmetic average across the training data set of the specific store.

Despite being highly significantly different from each other in practically all pair-wise comparisons (see Table 14 in Appendix B), all classical methods show approximately the same average performance. All methods are able to utilize some of the information and outperform the naive benchmark Mean by a considerable margin. The method Mean estimates for each day the same deposit per store. This deposit is chosen as the average deposit within the training data of the relevant store.
It should be noted that this result does not hold for training performance. All tree based methods are much more likely to overfit the training data. Interestingly, this does not negatively

impact the performance on the test data set relative to other methods. Most other methods show the curious result of the training data having a worse fit than the test data. This can partly be attributed to following fact: The test months show with ≈2290 € a lower average deposit level than the training period with ≈2470 €. The corresponding standard deviation of ≈4270 in the test and ≈6350 in the training data also differs in favor of the test data.

Looking at the 20 store where hyperparameter optimization took place (see Figure 18 in Appendix B) shows a very similar picture regarding the performance of the methods. Comparing the raw numbers, the achieved RMSEs on the 20 stores are a lot lower. This effect however is not achieved due to better models but can be explained by the generally different level of deposits with a mean of ≈2386€ for the complete data set and only ≈1640€ for the 20 sample stores. Using the normed RMSEs (see Table 17 and Table 18 in Appendix B) one can see that the performance across all stores is a bit better than across the sample stores. Interestingly this similarity does not hold in regard to the relative height of training to test data. With the 20 stores one can observe the expected effect of the training performance being better than the test performance. This implies that the 20 stores are at least somewhat different.

Having discussed the general results we move to the impact of more and recent data on prediction performance. The results regarding the variation of the feature sets given two Machine Learning methods are visualized in Figure 16.

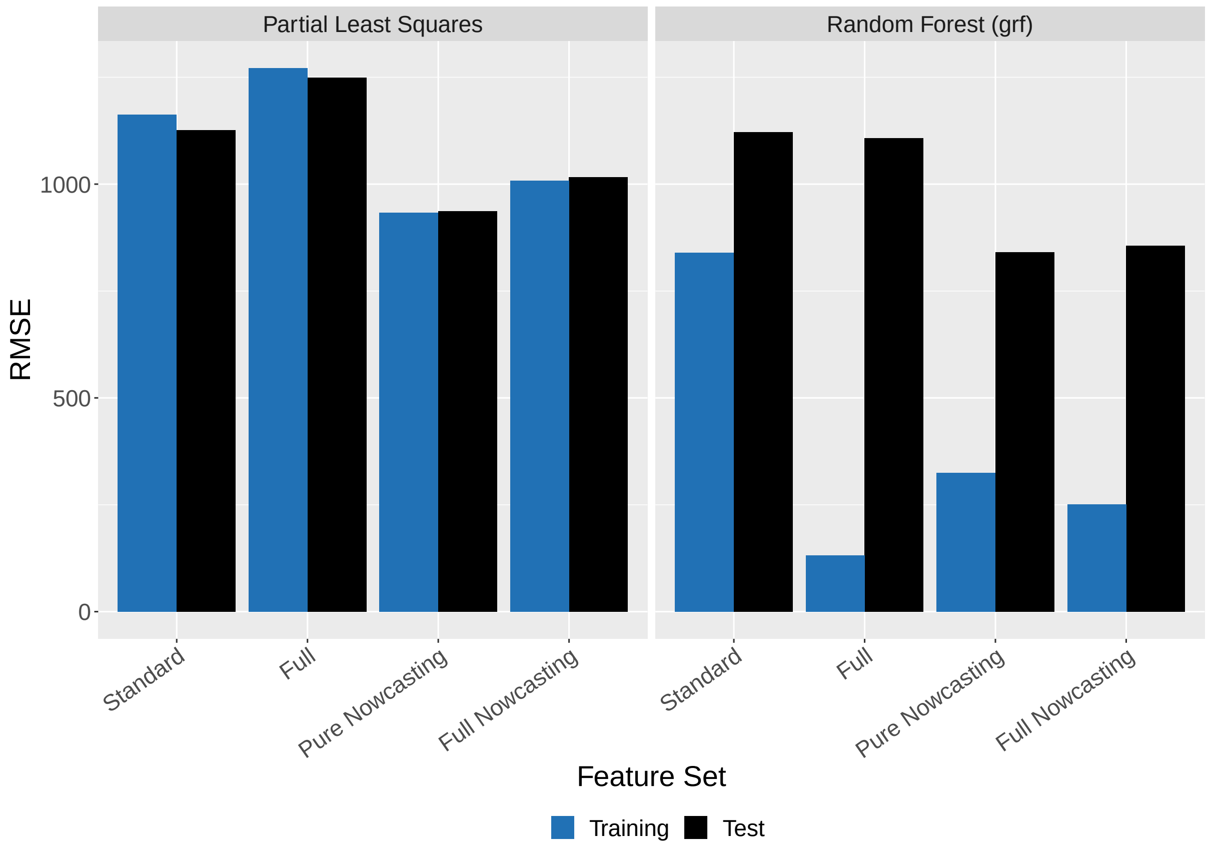

Figure 16: Mean of the root mean squared errors of all stores in the training and test data set for the methods Partial Least Squares and Random Forest given access to different feature sets according to Section 3.

For Random Forest we find that adding past information of other stores to the standard feature set decreases the root mean squared error of the prediction (RMSEP) on the test set from 1122 to 1108. Using this result as the baseline we find that adding current information of other stores reduces the RMSEP even further to 858. If we combine the standard feature set with current information of other stores only (omitting past information of other stores) the RMSEP is smallest with 841. In contrast we see that adding past information of other stores to the standard feature set harms Partial Least Squares. The RMSEP increases from 1127 to 1250. This is interesting as the Random Forest results suggest that the full feature set contains usable information. It seems to be that Partial Least Squares is not capable of filtering out relevant information as well as Random Forest in the full feature set. Adding current information of other stores impacts Partial Least Squares structurally the same way as Random Forest. In this case the RMSEP is reduced down to 1016. Omitting past information of other stores and just keeping their current information on top of the standard feature set reduces the RMSEP further to 937. This result is in line with the behavior of Random Forest.
Table 15 and Table 16 in Appendix B show more details including the results regarding the Median and a normed version of the RMSE/RMSEP. The structurally identical results for the 20 special stores are given by Figure 19, Table 17 and Table 18 in Appendix B.

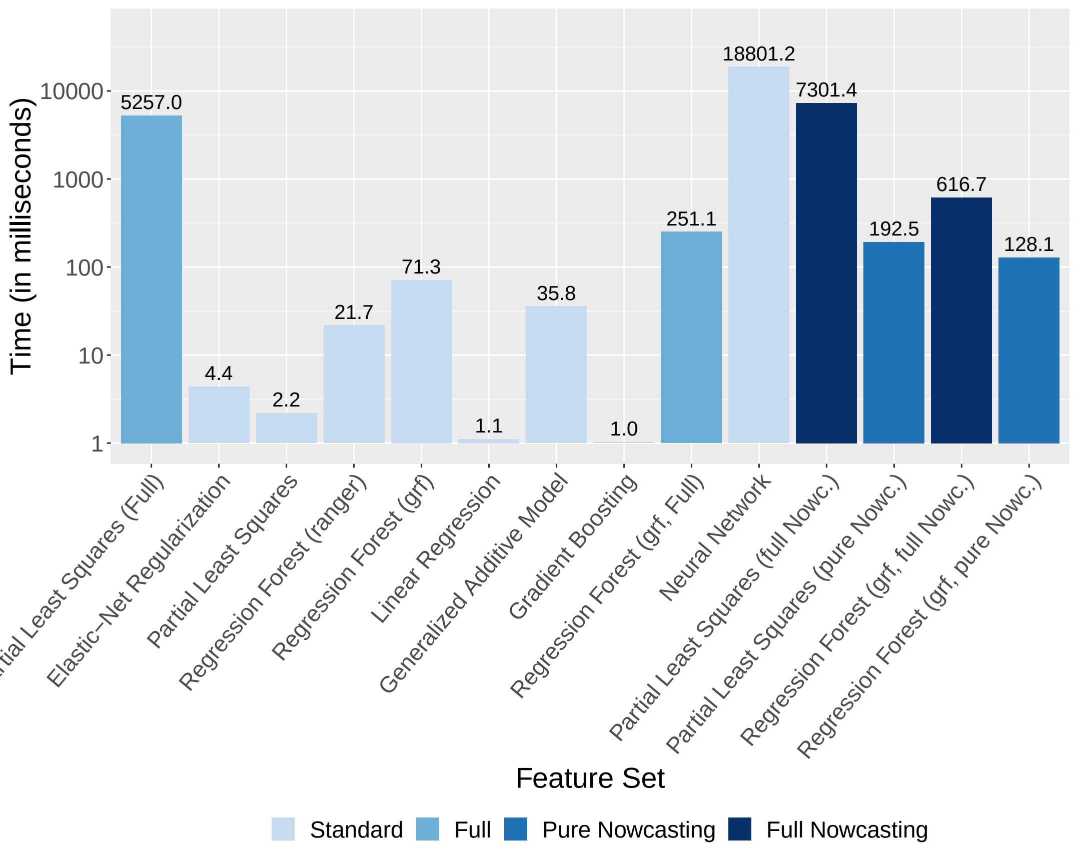

Figure 17: Comparison of average prediction time in milliseconds of the test data set per store, assuming the model has been trained. The order of the methods is according to their performance as measured by the RMSE on the test data set from worst (left) to best (right). Note that the y-axis is scaled logarithmically.

The performance increase of using pure Nowcasting is bought by an increase in computing time for the prediction. Figure 17 presents the different prediction times. All classical methods except the Neural Network are comparatively fast. On our machine they need less than 0.08 seconds computation time per store. Using pure Nowcasting increases this to approximately 0.13 seconds for the Random Forest and approximately 0.19 seconds in the case of Partial Least Squares. Comparing the times of Random Forest and Partial Least Squares on the classical set we see that Random Forest scales much better with the greater magnitude of input variables.
Similar effects can be found regarding the training times. Those are shown in Figure 20 in Appendix B. Here we observe that adding additional features increases training time as well. Interestingly, the calculation times of Gradient Boosting behave notably different to each other compared to the other methods. Gradient Boosting needs a comparatively long time to train relative to its fast prediction time.
Summing up we find that using Nowcasting for prediction is greatly beneficial. The best method/feature set combination Random Forest/Pure Nowcasting outperforms the best classical approach by 24% percent. Adding past information of other stores, which means moving from standard to full feature set or from pure to full Nowcasting, however, has no obvious positive effect on performance. On the contrary, these additional features might inject too much noise. This can be observed despite the fact that both analyzed methods should be able to handle additional features well and have gotten adapted hyperparameters for the specific feature sets. In the case of Partial Least Squares the additional features even have a noticeable detrimental effect. Random Forest is able to handle additional noise a lot better.

7. Discussion

There are several possible amendments to be discussed. We start by taking a critical look at our validation method. Afterwards we discuss weaknesses of our current implementation of hyperparameter optimization. Finally we end this section by pointing towards future research regarding a subtler analysis of the interaction of more recent information and different ML-algorithms.

Our validation method of backwards looking windows avoids any usage of future data to calibrate the model. However it has the following two weaknesses:

- As we want to use all data available to us, our algorithm always uses all available days to predict each validation window. The earlier days therefore have an influence on each single window-RMSE. Later ones are in the edge case only used for the prediction of the very last window. The resulting RMSE is calculated using the arithmetic mean over all window-RMSEs. Arguably less relevant earlier data points therefore have a comparatively higher impact than more recent ones.
- Always including the very first days results in a differing length of training data for each of the folds. This might be problematic if hyperparameters are sensitive to the length of training data. One example of such a sensitive hyperparameter is the number of neurons of a Neural Network. Generally speaking the provision of more training data justifies to use more neurons to keep the current level of overfitting. This allows to capture more complex interactions within the data, assuming the current model does not already sufficiently capture all relevant interactions. Consequently our generated hyperparameter sets might be optimized for shorter time-series than those which are used to build the final model.

One obvious solution to the second problem might be to use a fixed number of training days for each window. Doing so however throws away a lot of usable validation data if the number of days is to high. Using a low number of training days on the other hand might increase the severity of the problem that a found hyperparameter set might be unsuitable to be used on the complete training data.
An interesting alternative regarding the first problem might be to overweight the RMSEs which have been generated by more recent validation windows. This way one addresses that those have a higher information density due to their longer length and their inclusion of more recent data. Alternatively it is also possible to chose the weights in a way that the effect of the first problem is negated.

Similar to the basic implementations of *grid search* and *random search*, the current implementation of *Hyperpar-Tree* takes the best observed combinations of hyperparameters as the final result. This is not necessarily optimal, as the method might easily have stochastic elements in its training process. It might be possible, that this results in the usage of non-robust, suboptimal combinations of hyperparameters due to them scoring high by chance. This effect might be mitigated by using a smart kernel smoothing or repeating the most promising combinations to

confirm whether the high scoring is robust. Alternatively one might consider the leafs of the last tree itself as a form of smoothing. Assuming sufficiently many data points per leaf, the central value of the highest scoring leaf might be an alternative candidate for the final hyperparameter combination.
In the current implementation of *Hyperpar-Tree* the number of highest scoring leafs, which are the basis for the newly drawn combinations, is fixed. Instead of using a fixed number it might be sensible to use a one-sigma like approach and only discard those leafs which are significantly worse than the very best one.

Regarding the interaction of Nowcasting and Machine Learning methods we have seen that recent information impacts predictive power differently depending on the used method. It would be interesting to pinpoint how beneficial more recent information is for each method. One potential approach would be to rank other stores by a relevance measure such as for example correlation. Afterwards, one starts with the standard feature set and iteratively adds information of the most relevant stores one by one for each method. This approach could be used to compute a relevance-of-information coefficient in prediction for each method and store. Having obtained those coefficients it would be interesting to see in how far they differ between different methods.

References

Ahmed, Nesreen K., Amir F. Atiya, Neamat El Gayar, and Hisham El-Shishiny (2010). "An Empirical Comparison of Machine Learning Models for Time Series Forecasting". In: *Econometric Reviews* 29.5-6, pp. 594–621. DOI: 10.1080/07474938.2010.481556.

Atkinson, Beth (2019). *Package 'rpart'.* Version v4.1-15. URL: https://cran.r-project.org/web/packages/rpart/rpart.pdf (visited on 2019-10-23).

Bergstra, James and Yoshua Bengio (2012). "Random Search for Hyper-Parameter Optimization". In: *Journal of Machine Learning Research* 13.Feb, pp. 281–305.

Chen, Tianqi et al. (2019). *Package 'xgboost'.* Version 0.90.0.2. URL: https://cran.r-project.org/web/packages/xgboost/xgboost.pdf (visited on 2019-10-24).

Choi, Hyunyoung and Hal Varian (2012). "Predicting the present with Google Trends". In: *Economic Record* 88, pp. 2–9. DOI: 10.1111/j.1475-4932.2012.00809.x.

Eurostat (2019). *Exchange and interest rates - Overview.* URL: https://ec.europa.eu/eurostat/web/exchange-and-interest-rates (visited on 2019-11-19).

Falbel, Daniel, J. J. Allaire, François Chollet, RStudio, Google, Yuan Tang, Wouter Van Der Bijl, Martin Studer, and Sigrid Keydana (2019). *Package 'keras'.* Version 2.2.5.0. URL: https://cran.r-project.org/web/packages/keras/keras.pdf (visited on 2019-10-24).

Friedman, Jerome, Trevor Hastie, Rob Tibshirani, Noah Simon, Balasubramanian Narasimhan, and Junyang Qian (2019). *Package 'glmnet'.* Version 2.0-18. URL: https://cran.r-project.org/web/packages/glmnet/glmnet.pdf (visited on 2019-10-23).

Giannone, Domenico, Lucrezia Reichlin, and David Small (2008). "Nowcasting: The real-time informational content of macroeconomic data". In: *Journal of Monetary Economics* 55.4, pp. 665–676. DOI: 10.1016/j.jmoneco.2008.05.010.

Hastie, Trevor (2019). *Package 'gam'.* Version 1.16.1. URL: https://cran.r-project.org/web/packages/gam/gam.pdf (visited on 2019-10-24).

Hastie, Trevor and Robert Tibshirani (1986). "Generalized Additive Models". In: *Statistical Science* 1.3, pp. 297–310.

Hastie, Trevor, Robert Tibshirani, and Jerome Friedman (2009). *The Elements of Statistical Learning: Data mining, Inference, and Prediction.* 2nd ed. Springer Science & Business Media. Corrected 12th printing.

James, Gareth, Daniela Witten, Trevor Hastie, and Robert Tibshirani (2013). *An Introduction to Statistical Learning.* Vol. 112. Springer. ISBN: 9781461471370. DOI: 10.1007/978-1-4614-7138-7.

Kies, Martin (2020). "Finding Best Answers for the Iterated Prisoner's Dilemma Using Improved Q-Learning". In: *Available at SSRN.* DOI: 10.2139/ssrn.3556714. URL: https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3556714.

Kingma, Diederik P and Jimmy Ba (2014). "Adam: A method for stochastic optimization". In: *arXiv preprint arXiv:1412.6980.*

Krizhevsky, Alex, Ilya Sutskever, and Geoffrey E. Hinton (2012). "ImageNet classification with deep convolutional neural networks". In: *NIPS'12 Proceedings of the 25th International*

Conference on Neural Information Processing Systems. Vol. 1, pp. 1097–1105. URL: https://dl.acm.org/citation.cfm?id=2999257.

Mevik, Bjørn-Helge (2019). *Package 'pls'*. Version 2.7-2. URL: https://cran.r-project.org/web/packages/pls/pls.pdf (visited on 2019-10-23).

Nielsen, Michael (2019). *Neural Networks and Deep Learning*. URL: http://neuralnetworksanddeeplearning.com/index.html (visited on 2019-10-07).

R Core Team (2018). *R: A Language and Environment for Statistical Computing*. R Foundation for Statistical Computing. Vienna, Austria. URL: https://www.R-project.org/.

Rajkomar, Alvin et al. (2018). "Scalable and accurate deep learning with electronic health records". In: *NPJ Digital Medicine* 1.1, p. 18. DOI: 10.1038/s41746-018-0029-1.

Srivastava, Nitish, Geoffrey Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov (2014). "Dropout: a simple way to prevent neural networks from overfitting". In: *The journal of machine learning research* 15.1, pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

Tibshirani, Julie, Susan Athey, Rina Friedberg, Vitor Hadad, David Hirshberg, Luke Miner, Erik Sverdrup, Stefan Wager, and Marvin N. Wright (2019). *Package 'grf'*. Version 0.10.4. URL: https://cran.r-project.org/web/packages/grf/grf.pdf (visited on 2019-10-24).

Wright, Marvin N., Stefan Wager, and Philipp Probst (2019). *Package 'ranger'*. Version 0.11.2. URL: https://cran.r-project.org/web/packages/ranger/ranger.pdf (visited on 2019-10-24).

Xingjian, SHI, Zhourong Chen, Hao Wang, Dit-Yan Yeung, Wai-Kin Wong, and Wang-chun Woo (2015). "Convolutional LSTM network: A machine learning approach for precipitation nowcasting". In: *Advances in neural information processing systems*, pp. 802–810.

Zou, Hui and Trevor Hastie (2005). "Regularization and variable selection via the elastic net". In: *Journal of the royal statistical society: series B (statistical methodology)* 67.2, pp. 301–320. DOI: 10.1111/j.1467-9868.2005.00503.x.

Appendices

A. Hyperparameter Optimization

In this section we define the parameters and parameter ranges as explained in Section 5. Note that the type *boolean* is not implemented within the source code but only used for better readability. Using 0 for FALSE and 1 for TRUE in combination with the type *integer* is identical to allowing the parameter to be both TRUE and FALSE within the hyperparameter-optimization, albeit of course not at the same time.

A.1. Metaparameters

The following parameters of Table 2 have been chosen to find the hyper-parameters of the methods for both classical prediction and Nowcasting according to the methodology of section 5. In regard to all other control variables the default values have been used. The method used in *rpart* as defined by the parameter *method* is "anova".

Table 2: Meta-Parameters regarding the optimization of hyper-parameters.

Name of parameter within the sourcecode	**Value**	**Explanation**
Parameters regarding the type of hyperparameter-search.		
hyper.par.decision	"tree"	We do indeed use the tree method. The other method provided by the source code allows for a grid-search of hyper-parameters.
hyper.selection.method	"window.nested.cross"	We use the method as described in Section 5, where we use evenly spaced validation windows and all available data up to this point.
window.no	10	Number of validation windows used. Can be interpreted similarly to the number k of classical cross-validation, except that in this instance the size of the training data for each of the windows is different, as only historic data is used. Called k_{win} in Pseudo-Code 11.
length.period	92	Number of days which we use as the validation test set for each window. Identical to the number of days used in the actual test data. Called len_{test} in Pseudo-Code 11.

Name of parameter within the sourcecode	**Value**	**Explanation**
Parameters regarding the data set for the tree.		
start.n	100	Defines the number of random random parameter combinations which build the basis for the further tree search. Called n_{start} in Pseudo-Code 14.
add.n	20	How many new parameter combinations are newly drawn according to the most recent tree? Called n_{add} in Pseudo-Code 14.
repeat.no	10	How often do we want to draw new parameter combinations? The number of generated data points this way is $start.n + repeat.no \cdot add.n$. Called n_{rep} in Pseudo-Code 14.
best.splits	3	When drawing new parameter combinations, the parameter "best.splits" defines the number of "top"-leafs to be used. Using a higher numbers slows down convergence, but minimizes the chance to land in a local optimum. Called $n_{\text{best.splits}}$ in Pseudo-Code 14.
remove.n	15	How many parameter combinations do we want to delete in each cycle? The worst "remove.n" combinations are deleted to minimize variance of the tree and allow a better circling in on the optimal combination. Called n_{rem} in Pseudo-Code 14.
Parameters regarding the tree itself as per the R-package rpart *(Atkinson, 2019)*		
tree.param.cp	0.01	Corresponds to *cp* of *rpart.control*. "complexity parameter. Any split that does not decrease the overall lack of fit by a factor of cp is not attempted." (Atkinson, 2019)
tree.param.minsplit	10	Corresponds to *minsplit* of *rpart.control* and is "the minimum number of observations that must exist in a node in order for a split to be attempted." (Atkinson, 2019)
tree.param.minbucket	5	Corresponds to *minbucket* of *rpart.control* and is "the minimum number of observations in any terminal <leaf> node." (Atkinson, 2019)
tree.param.xval	1	Corresponds to *xval* of *rpart.control* and is the "number of cross-validations." (Atkinson, 2019). This implies, that we do not use the internal cross-validation of the method within our hyper-parameter search.
tree.param.maxdepth	10	Corresponds to *maxdepth* of *rpart.control* and it "[s]et[s] the maximum depth of any node of the final tree, with the root node counted as depth 0." (Atkinson, 2019).

A.2. Partial Least Squares

The following parameters have been used in regard to the method “Partial Least Squares” as described in Section 4.3 and according to the methodology of Section 5:

Table 3: Ranges of Hyperparameters considered for Partial Least Squares. For pls.ncomp we chose the maximum of 26 for the Standard case and a sufficiently high number for the other feature sets, so that it is non-limiting to the number of actually chosen components.

Name of param.	**Min**	**Max**	**Best**	**Type**	**Explanation**
pls.use.y.log	FALSE		-	boolean	Do we want to draw the logarithm of the to be predicted variable before training and reverse this operation after prediction?
pls.ncomp	26 (Standard), 50 (Full), 100 (pure N.), 50 (full N.)		-	integer	Corresponds to the parameter *ncomp* of the *plsr*-function of Mevik (2019) and describes the number of components which are considered for the model.
pls.validation	CV		-	factor	Corresponds to the parameter *validation* of the *plsr*-function of Mevik (2019) and describes what kind of internal validation should be used. “CV” results in cross-validation.
pls.method_nc	onesigma		-	factor	Corresponds to the parameter *method* of the *selectNcomp*-function of Mevik (2019) and describes what kind of selection criterion is used to determine the chosen number of components. “The approach “onesigma” simply returns the first model where the optimal CV is within one standard error of the absolute optimum” (Mevik, 2019), which is based on the method proposed by James et al. (2013, p. 244).

The number of components actually chosen can be found in Table 4.

Table 4: Frequency of choosing a certain number of components with the method Partial Least Squares given the different feature sets as defined by Section 3

Components	Standard	Full	pure Nowc.	full Nowc.
1	63	618	194	260
2	13	1332	56	192
3	4	31	410	226
4	1	3	522	641
5	1	0	201	519
6	0	5	75	95
7	0	0	125	13
8	42	0	92	26
9	1660	0	52	8
10	206	0	50	1
11	0	0	45	3
12	0	0	38	1
13	0	0	40	1
14	0	0	27	3
15	0	0	16	0
16	0	0	11	1
17	0	1	4	0
18	0	0	2	0
19	0	0	4	0
20	0	0	6	0
21	0	0	5	0
22	0	0	3	0
23	0	0	1	0
24	0	0	0	0
25	0	0	3	0
26	0	0	0	0
27	0	0	3	0
28	0	0	0	0
29	0	0	0	0
30	0	0	0	0
31	0	0	0	0
32	0	0	0	0
33	0	0	0	0
34	0	0	0	0
35	0	0	1	0
36	0	0	0	0
37	0	0	0	0
38	0	0	1	0
39	0	0	0	0
40	0	0	3	0

A.3. Generalized Linear Models (glmnet)

The following parameters and parameter ranges have been used regarding the method "Generalized Linear Models" as specified in Section 4.2. If not listed, the default values from Friedman et al. (2019) have been used.

Table 5: Ranges of Hyperparameters considered for the Generalized Linear Model.

Name of param.	**Min**	**Max**	**Best**	**Type**	**Explanation**
glmnet.use.y.log	FALSE		-	boolean	Do we want to draw the logarithm of the to be predicted variable before training and reverse this operation after prediction?
glmnet.alpha	0	1	0.4850	numeric	Corresponds to the parameter *alpha* of the *glmnet*-function of Friedman et al. (2019) which describes the elasticnet mixing parameter. If *alpha* equals 1 a *LASSO*-Regression is used, while *alpha* equals 0 results in a *Ridge*-Regression.
glmnet.lambda	NA		-	numeric	We use the default value of the parameter *lambda* of the *glmnet*-function defined by Friedman et al. (2019) which describes the regularization parameter. "NA" means we use the lambda value which has been calculated according to the internal crossvalidation.
glmnet.x.formula	"Full"		-	factor	All explanatory variables are used without any modification.

A.4. Generalized Additive Models (GAM)

The following parameters and parameter ranges have been used regarding the method "Generalized Additive Models" as specified in Section 4.4. If not listed, the default values from Hastie (2019) have been used.

Table 6: Ranges of Hyperparameters considered for the Generalized Additve Model.

Name of param.	**Min**	**Max**	**Best**	**Type**	**Explanation**
gam.use.y.log	FALSE		-	boolean	Do we want to draw the logarithm of the to be predicted variable before training and reverse this operation after prediction?
gam.family	gaussian		-	factor	Corresponds to the parameter *family* of the *gam*-function of Hastie (2019) which describes "the error distribution and link function to be used in the model" (Hastie, 2019).
gam.k	0	4	1	integer	Corresponds to the parameter *df* of the *s*-function of Hastie (2019) given to the formula of the *gam*-function. It is "the target equivalent degrees of freedom, used as a smooting parameter." Hastie (2019).

A.5. Random Forest (Package “grf”, Classical Prediction)

The following parameters and parameter ranges have been used for the classical prediction regarding the method “Random Forest” as specified in Section 4.6 in regards to the R-package *grf* (J. Tibshirani et al., 2019). If not listed, the default values from the function *regression_forest()* in J. Tibshirani et al. (2019) have been used.

Table 7: Ranges of Hyperparameters considered for the Random Forest as implemented by the R-package *grf* (J. Tibshirani et al., 2019).

Name of param.	Min	Max	Best	Type	Explanation
rfw.use.y.log	FALSE		-	boolean	Do we want to draw the logarithm of the to be predicted variable before training and reverse this operation after prediction?
rfw.sample.fraction	0.3	0.9	0.5496	numeric	Corresponds to *sample.fraction* of the *regression_forest()*-function of J. Tibshirani et al. (2019) which is the “[f]raction of the data used to build each tree.” (J. Tibshirani et al., 2019).
rfw.mtry	2	26	20	integer	Corresponds to *mtry* of the *regression_forest*-function of J. Tibshirani et al. (2019) which describes the “[n]umber of variables tried for each split.” (J. Tibshirani et al., 2019)
rfw.num.trees	1000		-	integer	Corresponds to the parameter *num.trees* of the *regression_forest*-function of J. Tibshirani et al. (2019) which describes the “[n]umber of trees grown in the forest.” (J. Tibshirani et al., 2019)
rfw.min.node.size	2	20	11	integer	Corresponds to *min.node.size* of the *regression_forest*-function of J. Tibshirani et al. (2019) which describes the “target for the minimum number of observations in each tree leaf.” (J. Tibshirani et al., 2019)
rfw.honesty	FALSE		-	boolean	Corresponds to the parameter *honesty* of the *regression_forest*-function of J. Tibshirani et al. (2019) which describes “[w]hether or not honest splitting (i.e., sub-sample splitting) should be used.” (J. Tibshirani et al., 2019)
rfw.alpha	0.001	0.2	0.1081	numeric	Corresponds to *alpha* of the *regression_forest*-function of J. Tibshirani et al. (2019) which is “tuning parameter that controls the maximum imbalance of a split.” (J. Tibshirani et al., 2019)

A.6. Random Forest (Package “grf”, Nowcasting)

The following parameters and parameter ranges have been used using Nowcasting and the method “Random Forest” as specified in Section 4.6 in regards to the R-package *grf* (J. Tibshirani et al., 2019). If not listed, the default values from the function *regression_forest()* in J. Tibshirani et al. (2019) have been used.

Table 8: Ranges of Hyperparameters considered for the Random Forest as implemented by the R-package *grf* (J. Tibshirani et al., 2019).

Name of param.	Min	Max	Best	Type	Explanation
rfw.use.y.log	FALSE		-	boolean	Do we want to draw the logarithm of the to be predicted variable before training and reverse this operation after prediction?
rfw.sample.fraction	0.3	0.9	0.8854	numeric	Corresponds to *sample.fraction* of the *regression_forest*-function of J. Tibshirani et al. (2019) which is the “[f]raction of the data used to build each tree.” (J. Tibshirani et al., 2019).
rfw.mtry	100	2000	1752	integer	Corresponds to *mtry* of the *regression_forest*-function of J. Tibshirani et al. (2019) which describes the “[n]umber of variables tried for each split.” (J. Tibshirani et al., 2019)
rfw.num.trees	1000		-	integer	Corresponds to the parameter *num.trees* of the *regression_forest*-function of J. Tibshirani et al. (2019) which describes the “[n]umber of trees grown in the forest.” (J. Tibshirani et al., 2019)
rfw.min.node.size	2	20	5	integer	Corresponds to *min.node.size* of the *regression_forest*-function of J. Tibshirani et al. (2019) which describes the “target for the minimum number of observations in each tree leaf.” (J. Tibshirani et al., 2019)
rfw.honesty	FALSE		-	boolean	Corresponds to the parameter *honesty* of the *regression_forest*-function of J. Tibshirani et al. (2019) which describes “[w]hether or not honest splitting (i.e., sub-sample splitting) should be used.” (J. Tibshirani et al., 2019)
rfw.alpha	0	0.2	0.1855	numeric	Corresponds to *alpha* of the *regression_forest*-function of J. Tibshirani et al. (2019) which is “tuning parameter that controls the maximum imbalance of a split.” (J. Tibshirani et al., 2019)

A.7. Random Forest (Package "grf", Full)

The following parameters and parameter ranges have been used using full and the method "Random Forest" as specified in Section 4.6 in regards to the R-package *grf* (J. Tibshirani et al., 2019). If not listed, the default values from the function *regression_forest()* in J. Tibshirani et al. (2019) have been used.

Table 9: Ranges of Hyperparameters considered for the Random Forest as implemented by the R-package *grf* (J. Tibshirani et al., 2019).

Name of param.	Min	Max	Best	Type	Explanation
rfw.use.y.log	FALSE		-	boolean	Do we want to draw the logarithm of the to be predicted variable before training and reverse this operation after prediction?
rfw.sample.fraction	0.5	0.99	0.9094	numeric	Corresponds to *sample.fraction* of the *regression_forest*-function of J. Tibshirani et al. (2019) which is the "[f]raction of the data used to build each tree." (J. Tibshirani et al., 2019).
rfw.mtry	1000	5000	1046	integer	Corresponds to *mtry* of the *regression_forest*-function of J. Tibshirani et al. (2019) which describes the "[n]umber of variables tried for each split." (J. Tibshirani et al., 2019)
rfw.num.trees	1000		-	integer	Corresponds to the parameter *num.trees* of the *regression_forest*-function of J. Tibshirani et al. (2019) which describes the "[n]umber of trees grown in the forest." (J. Tibshirani et al., 2019)
rfw.min.node.size	2	20	2	integer	Corresponds to *min.node.size* of the *regression_forest*-function of J. Tibshirani et al. (2019) which describes the "target for the minimum number of observations in each tree leaf." (J. Tibshirani et al., 2019)
rfw.honesty	FALSE		-	boolean	Corresponds to the parameter *honesty* of the *regression_forest*-function of J. Tibshirani et al. (2019) which describes "[w]hether or not honest splitting (i.e., sub-sample splitting) should be used." (J. Tibshirani et al., 2019)
rfw.alpha	0	0.4	0.2110	numeric	Corresponds to *alpha* of the *regression_forest*-function of J. Tibshirani et al. (2019) which is "tuning parameter that controls the maximum imbalance of a split." (J. Tibshirani et al., 2019)

A.8. Random Forest (Package "grf", Pure)

The following parameters and parameter ranges have been used using pure and the method "Random Forest" as specified in Section 4.6 in regards to the R-package *grf* (J. Tibshirani et al., 2019). If not listed, the default values from the function *regression_forest()* in J. Tibshirani et al. (2019) have been used.

Table 10: Ranges of Hyperparameters considered for the Random Forest as implemented by the R-package *grf* (J. Tibshirani et al., 2019).

Name of param.	Min	Max	Best	Type	Explanation
rfw.use.y.log	FALSE		-	boolean	Do we want to draw the logarithm of the to be predicted variable before training and reverse this operation after prediction?
rfw.sample.fraction	0.3	0.9	0.7374	numeric	Corresponds to *sample.fraction* of the *regression_forest*-function of J. Tibshirani et al. (2019) which is the "[f]raction of the data used to build each tree." (J. Tibshirani et al., 2019).
rfw.mtry	100	2015	593	integer	Corresponds to *mtry* of the *regression_forest*-function of J. Tibshirani et al. (2019) which describes the "[n]umber of variables tried for each split." (J. Tibshirani et al., 2019)
rfw.num.trees	1000		-	integer	Corresponds to the parameter *num.trees* of the *regression_forest*-function of J. Tibshirani et al. (2019) which describes the "[n]umber of trees grown in the forest." (J. Tibshirani et al., 2019)
rfw.min.node.size	2	20	3	integer	Corresponds to *min.node.size* of the *regression_forest*-function of J. Tibshirani et al. (2019) which describes the "target for the minimum number of observations in each tree leaf." (J. Tibshirani et al., 2019)
rfw.honesty	FALSE		-	boolean	Corresponds to the parameter *honesty* of the *regression_forest*-function of J. Tibshirani et al. (2019) which describes "[w]hether or not honest splitting (i.e., sub-sample splitting) should be used." (J. Tibshirani et al., 2019)
rfw.alpha	0	0.2	0.1908	numeric	Corresponds to *alpha* of the *regression_forest*-function of J. Tibshirani et al. (2019) which is "tuning parameter that controls the maximum imbalance of a split." (J. Tibshirani et al., 2019)

A.9. Random Forest (Package “ranger”)

The following parameters and parameter ranges have been used regarding the method “Random Forest” as specified in Section 4.6 in regards to the R-package *ranger* (Wright et al., 2019). If not listed, the default values from the function *ranger()* in J. Tibshirani et al. (2019) have been used.

Table 11: Ranges of Hyperparameters considered for the Random Forest as implemented by the R-package *ranger* (Wright et al., 2019).

Name of param.	**Min**	**Max**	**Best**	**Type**	**Explanation**
rfr.use.y.log	FALSE		-	boolean	Do we want to draw the logarithm of the to be predicted variable before training and reverse this operation after prediction?
rfr.sample.fraction	0.3	0.95	0.6407	numeric	Corresponds to the parameter *sample.fraction* of the *ranger*-function of Wright et al. (2019) which is the “[f]raction of observations to sample” (Wright et al., 2019).
rfr.mtry	2	26	21	integer	Corresponds to the parameter *mtry* of the *ranger*-function of Wright et al. (2019) which is the “[n]umber of variables to possibly split at in each node.” (Wright et al., 2019).
rfr.num.trees	1000		-	integer	Corresponds to the parameter *num.trees* of the *ranger*-function of Wright et al. (2019) which describes the “[n]umber of trees.” (Wright et al., 2019)
rfr.min.node.size	2	20	15	integer	Corresponds to *min.node.size* of the *ranger*-function of Wright et al. (2019) which describes the “[m]inimal node size.” (Wright et al., 2019)

A.10. Gradient Boosting

The following parameters and parameter ranges have been used regarding the method "Gradient Boosting" as specified in Section 4.7 using the R-package *xgboost* (T. Chen et al., 2019). If not listed, the default values from the function *xgboost()* in T. Chen et al. (2019) have been used.

Table 12: Ranges of Hyperparameters considered for Gradient Boosting as implemented by the R-package *xgboost* (T. Chen et al., 2019).

Name of param.	Min	Max	Best	Type	Explanation
xgboost.use.y.log	FALSE		-	boolean	Do we want to draw the logarithm of the to be predicted variable before training and reverse this operation after prediction?
xgboost.nrounds	50	1000	185	integer	Corresponds to *nrounds* of the function *xgboost()* which is the "max number of boosting iterations." (T. Chen et al., 2019).
xgboost.max_depth	2	15	3	integer	Corresponds to the value *max_depth* of the function *xgboost()* which is the "maximum depth of a tree." (T. Chen et al., 2019)
xgboost.eta	0.001	0.2	0.0178	numeric	Corresponds to *eta* of the function *xgboost()* which "control[s] the learning rate." (T. Chen et al., 2019). A lower value of eta prevents overfitting.
xgboost.gamma	0.01		-	numeric	Corresponds to *gamma* of the function *xgboost()* which is the "minimum loss reduction required to make a further partition on a leaf node of the tree." (T. Chen et al., 2019).
xgboost.colsample_bytree	0.4	1	0.5802	numeric	Corresponds to *colsample_bytree* of the function *xgboost()* which is the "subsample ratio of columns when constructing each tree." (T. Chen et al., 2019).
xgboost.subsample	0.4	1	0.5994	numeric	Corresponds to the value *subsample* of the function *xgboost()* which is the "subsample ratio of the training instance." (T. Chen et al., 2019).
xgboost.min_child_weight	2	10	3	integer	Corresponds to *min_child_weight* of the function *xgboost()* which is the " minimum sum of instance weight (hessian) needed in a child." (T. Chen et al., 2019).

A.11. Neural Network

The following parameters and parameter ranges have been used regarding the method "Neural Network" as specified in Section 4.8 using the R-package *keras* (Falbel et al., 2019). If not listed, the default values from Falbel et al. (2019) have been used. We use the *Adam*-optimizer with default values.

Table 13: Ranges of Hyperparameters considered for the Neural Network as implemented by the R-package *keras* (Falbel et al., 2019).

Name of param.	**Min**	**Max**	**Best**	**Type**	**Explanation**
batch.size.predict	24		-	integer	Corresponds to *batch_size* given to the function *predict()* and controls the number of input samples used within one batch.
batch.size.train	32	96	82	integer	Corresponds to *batch_size* given to the function *fit()*. *batch_size* is the "[n]umber of samples per gradient update." (Falbel et al., 2019) when training the model.
neurons.hidden.1	8	512	153	integer	Corresponds to *units* in regards to the function *layer_dense()* and is the number of neurons in the first hidden layer.
neurons.hidden.2	0	128	126	integer	Corresponds to *units* in regards to the function *layer_dense()* and is the number of neurons in the second hidden layer.
neurons.hidden.3	0	128	18	integer	Corresponds to *units* in regards to the function *layer_dense()* and is the number of neurons in the third hidden layer.
NN.verbose	FALSE		-	boolean	Corresponds to the parameter *verbose* of the function *fit()* and controls the displayed output. Should have no effects on performance.
train.epochs	100	2000	997	integer	Corresponds to *epochs* of the function *fit()* and is the number of epochs to train the model.

Name of param.	**Min**	**Max**	**Best**	**Type**	**Explanation**
activation1	sigmoid		-	factor	Corresponds to *activation* of the function *layer_activation()* and is the activation function of the first hidden layer.
activation2	sigmoid		-	factor	Corresponds to *activation* of the function *layer_activation()* and is the activation function of the second hidden layer.
activation3	sigmoid		-	factor	Corresponds to *activation* of the function *layer_activation()* and is the activation function of the third hidden layer.
activation.end	linear		-	factor	Corresponds to *activation* of the function *layer_activation()* and is the activation function of the final layer.
input.dropout	0	0.6	0.5450	numeric	Corresponds to *rate* of the function *layer_dropout()*, defining the drop out rate of the input layer.
dropout1	0	0.6	0.0837	numeric	Corresponds to *rate* of the function *layer_dropout()*, defining the drop out rate of the first hidden layer.
dropout2	0	0.6	0.2723	numeric	Corresponds to *rate* of the function *layer_dropout()*, defining the drop out rate of the second hidden layer.
dropout3	0		-	numeric	Corresponds to *rate* of the function *layer_dropout()*, defining the drop out rate of the third hidden layer.
NN.normalize	minmax		-	factor	Defines the normalization procedure as explained in Section 4.8.
NN.normalize.log	FALSE		-	boolean	Do we want draw the logarithm of the (normalized) and to be predicted variable before training and reverse this operation after prediction?

B. Results

Table 14: Pairwise p-Values based on a paired t-Test on logged RMSE values on the test data set for all methods on the standard feature set as defined by Section 3. The methods are ordered based on their performance on the test data set. We chose logged RMSE values, as the RMSEs across the stores are approximately log-normal distributed. Where a 0 is reported, the p-Value is below the floating point resolution. The paired t-Test corrects for the variance across stores and reports on the significance of the difference between the mean of two methods. Two methods might thus be non-significantly different (i.e. between Linear Regression and Neural Network) even though their aggregated performance differs. This can be explained by making structurally different errors across the stores.

	Mean	**Elastic-Net Regulariza-tion**	**Partial Least Squares**	**Regression Forest (ranger)**	**Regression Forest (grf)**	**Linear Regression**	**Generalized Additive Model**	**Gradient Boosting**	**Neural Network**
Mean	.	0	0	0	0	0	0	0	0
Elastic-Net Regularization		.	2.03e-233	4.00e-304	0	1.43e-282	5.26e-313	0	5.50e-320
Partial Least Squares			.	1.84e-55	7.81e-84	2.34e-35	9.26e-81	1.24e-61	1.25e-31
Regression Forest (ranger)				.	1.63e-54	5.67e-25	6.73e-12	0.69	6.78e-16
Regression Forest (grf)					.	1.00e-55	5.00e-39	1.59e-20	2.87e-33
Linear Regression						.	1.89e-13	4.01e-26	0.93
Generalized Additive Model							.	7.23e-13	1.55e-05
Gradient Boosting								.	5.78e-19
Neural Network									.

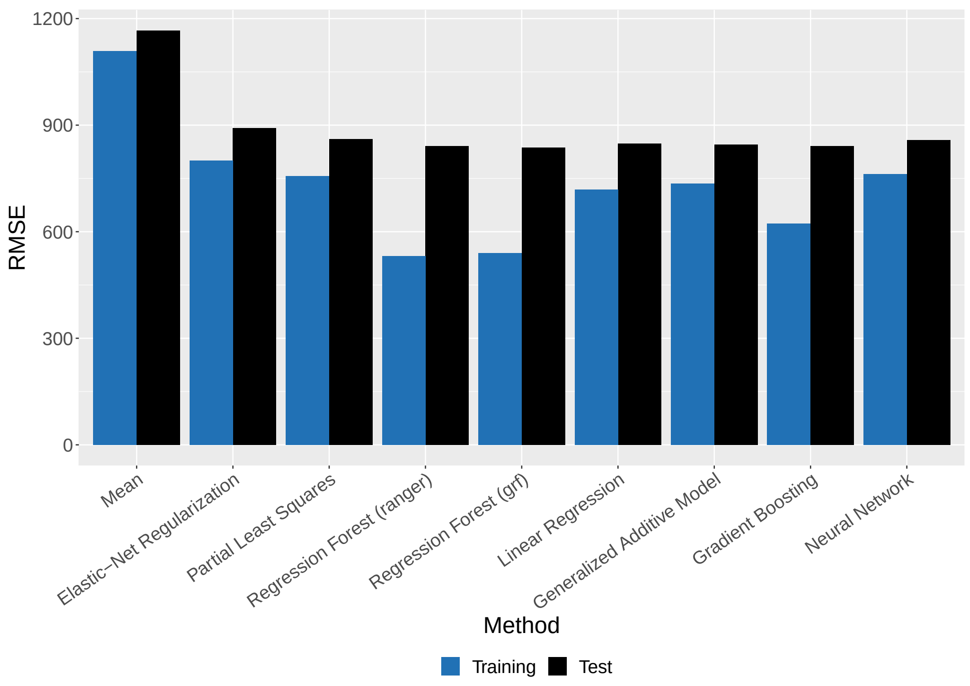

Figure 18: Mean of the root mean squared errors given the 20 stores, where hyperparameter optimization took place, on the training and test data set for each analysed method. Here, only the standard feature set as specified by Section 3 has been used. The RMSEs have been ordered by their performance on the test data set averaged over all stores. As a basic benchmark the method Mean is included, which predicts the deposit based on the arithmetic average across the training data set of the specific store.

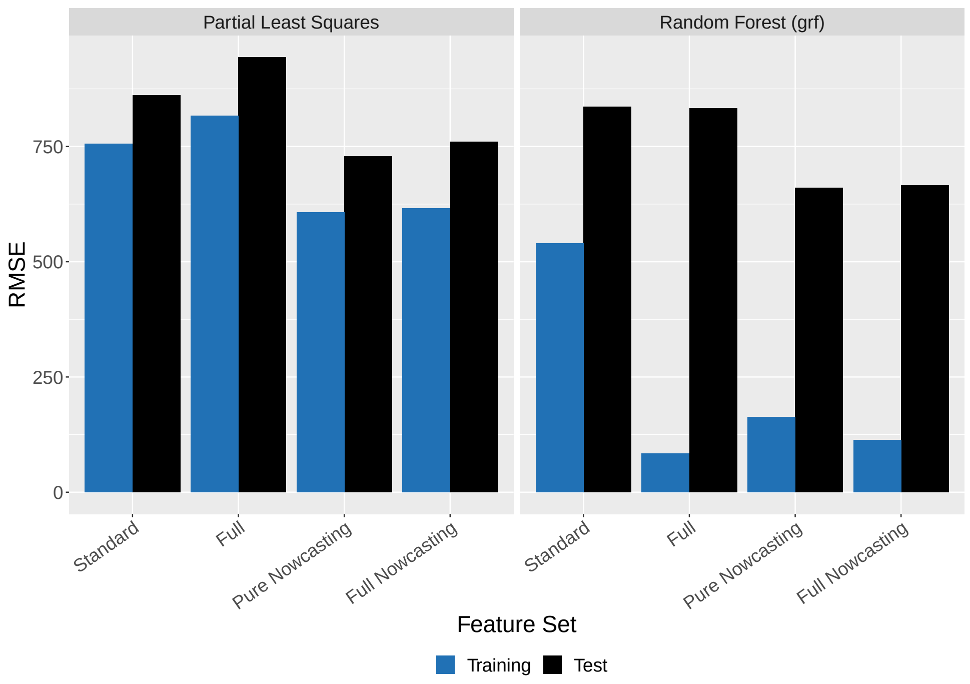

Figure 19: Mean of the root mean squared errors of the methods Partial Least Squares and Random Forest given access to different feature sets according to Section 3 using only data from the 20 stores, where hyperparameter optimization took place.

Table 15: Statistics of the complete training pool on the training data. The methods have been sorted by their performance on the test data set.
MRMSE = Mean root square error
MRMSEN = Mean root square error normed according to Section 5
MedRMSE = Median root square error
MedRMSEN = Median root square error normed according to Section 5

method	**MRMSE**	**MRMSEN**	**MedRMSE**	**MedRMSEN**
Mean	1665	1.0280	1110	0.9962
Partial Least Squares (Full)	1272	0.7696	803	0.7153
Elastic-Net Regularization	1234	0.7375	764	0.6834
Partial Least Squares	1162	0.6898	714	0.6417
Regression Forest (ranger)	829	0.4848	497	0.4455
Regression Forest (grf)	839	0.4929	507	0.4537
Linear Regression	1112	0.6615	685	0.6144
Generalized Additive Model	1134	0.6752	698	0.6281
Gradient Boosting	925	0.5620	587	0.5276
Regression Forest (grf, Full)	131	0.0796	81	0.0722
Neural Network	1132	0.6762	703	0.6295
Partial Least Squares (full Nowc.)	1008	0.5880	574	0.5134
Partial Least Squares (pure Nowc.)	934	0.5354	506	0.4790
Regression Forest (grf, full Nowc.)	251	0.1183	101	0.0912
Regression Forest (grf, pure Nowc.)	324	0.1633	145	0.1333

Table 16: Statistics of the complete training pool on the test data. The methods have been sorted by their performance on the test data set.
MRMSEP = Mean root square error on test set.
MRMSEPN = Mean root square error on test set normed according to Section 5
MedRMSEP = Median root square error on test set
MedRMSEPN = Median root square error normed on test set according to Section 5

method	MRMSEP	MRMSEPN	MedRMSEP	MedRMSEPN
Mean	1569	1.0155	1130	1.0056
Partial Least Squares (Full)	1250	0.7983	874	0.7756
Elastic-Net Regularization	1172	0.7470	813	0.7244
Partial Least Squares	1127	0.7151	779	0.6944
Regression Forest (ranger)	1124	0.7033	762	0.6778
Regression Forest (grf)	1122	0.6996	753	0.6722
Linear Regression	1121	0.7084	769	0.6873
Generalized Additive Model	1120	0.7053	767	0.6842
Gradient Boosting	1119	0.7019	762	0.6783
Regression Forest (grf, Full)	1108	0.6947	755	0.6637
Neural Network	1107	0.7074	770	0.6851
Partial Least Squares (full Nowc.)	1016	0.6334	657	0.5973
Partial Least Squares (pure Nowc.)	937	0.5855	597	0.5499
Regression Forest (grf, full Nowc.)	856	0.5200	521	0.4654
Regression Forest (grf, pure Nowc.)	841	0.5123	511	0.4594

Table 17: Statistics of the performance on the training data on the 20 stores, where the hyper-parameter optimization took place. The methods have been sorted by their performance on the test data set of all stores.
MRMSE = Mean root square error.
MRMSEN = Mean root square error normed according to Section 5
MedRMSE = Median root square error on test set
MedRMSEN = Median root square error normed according to Section 5

method	**MRMSE**	**MRMSEN**	**MedRMSE**	**MedRMSEN**
Mean	1109	0.9702	1107	0.9507
Partial Least Squares (Full)	816	0.7285	785	0.6949
Elastic-Net Regularization	800	0.7136	765	0.6696
Partial Least Squares	756	0.6736	711	0.6233
Regression Forest (ranger)	531	0.4751	493	0.4307
Regression Forest (grf)	540	0.4813	505	0.4484
Linear Regression	718	0.6431	671	0.5880
Generalized Additive Model	735	0.6563	695	0.6027
Gradient Boosting	623	0.5569	588	0.5093
Regression Forest (grf, Full)	84	0.0746	82	0.0710
Neural Network	762	0.6750	704	0.6409
Partial Least Squares (full Nowc.)	616	0.5636	543	0.5186
Partial Least Squares (pure Nowc.)	608	0.5507	513	0.5196
Regression Forest (grf, full Nowc.)	114	0.1012	101	0.0930
Regression Forest (grf, pure Nowc.)	163	0.1488	141	0.1367

Table 18: Statistics of the performance on the test data on the 20 stores, where the hyperparameter optimization took place. The methods have been sorted by their performance on the test data set of all stores.
MRMSEP = Mean root square error on test set.
MRMSEPN = Mean root square error on test set normed according to Section 5
MedRMSEP = Median root square error on test set
MedRMSEPN = Median root square error normed on test set according to Section 5

method	**MRMSEP**	**MRMSEPN**	**MedRMSEP**	**MedRMSEPN**
Mean	1166	1.0250	1130	1.0104
Partial Least Squares (Full)	944	0.8375	934	0.8152
Elastic-Net Regularization	892	0.7960	879	0.7817
Partial Least Squares	861	0.7627	822	0.7152
Regression Forest (ranger)	842	0.7553	806	0.7185
Regression Forest (grf)	837	0.7498	802	0.7130
Linear Regression	848	0.7555	816	0.7118
Generalized Additive Model	845	0.7516	805	0.7036
Gradient Boosting	841	0.7495	804	0.7143
Regression Forest (grf, Full)	833	0.7472	820	0.7204
Neural Network	858	0.7614	844	0.7157
Partial Least Squares (full Nowc.)	760	0.6957	733	0.6506
Partial Least Squares (pure Nowc.)	729	0.6706	687	0.6282
Regression Forest (grf, full Nowc.)	666	0.6109	595	0.5656
Regression Forest (grf, pure Nowc.)	661	0.6068	599	0.5615

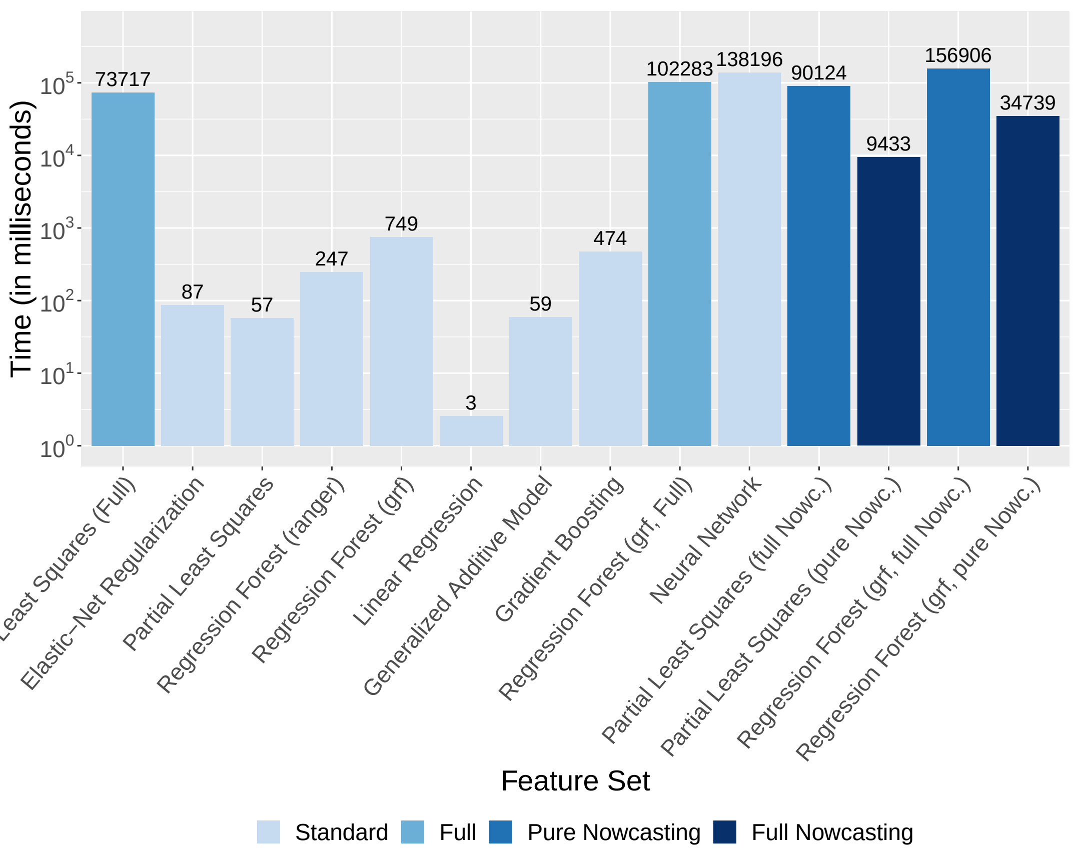

Figure 20: Comparison of average training time in milliseconds on the training data set per store, assuming the hyperparameter of the models have been selected. The order of the methods is according to their performance as measured by the RMSE on the test data set from worst (left) to best (right). Note that the y-axis is scaled logarithmically.

Ehrenwörtliche Erklärung

Ich erkläre hiermit ehrenwörtlich, dass ich die vorliegende Arbeit selbstständig angefertigt habe; die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher keiner anderen Prüfungsbehörde vorgelegt und auch noch nicht veröffentlicht.

Ich bin mir bewusst, dass eine unwahre Erklärung rechtliche Folgen haben wird.

Ulm, den 24.03.2020

(Unterschrift)