TU WIEN Informatics

Reinforcement Learning for Games with Imperfect Information

Teaching an Agent the Game of Schnapsen

MASTERARBEIT

zur Erlangung des akademischen Grades

Master of Science

im Rahmen des Studiums

Data Science

eingereicht von

Tobias Salzer, Bsc.
Matrikelnummer 01304720

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Assoc. Prof. Dipl.-Ing. Dr. techn. Clemens Heitzinger

Wien, 21. Jänner 2023

Tobias Salzer | Clemens Heitzinger

TU WIEN Informatics

Reinforcement Learning for Games with Imperfect Information

Teaching an Agent the Game of Schnapsen

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Master of Science

in

Data Science

by

Tobias Salzer, Bsc.
Registration Number 01304720

to the Faculty of Informatics

at the TU Wien

Advisor: Assoc. Prof. Dipl.-Ing. Dr. techn. Clemens Heitzinger

Vienna, 21st January, 2023

Tobias Salzer	Clemens Heitzinger

Erklärung zur Verfassung der Arbeit

Tobias Salzer, Bsc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 21. Jänner 2023

Tobias Salzer

Kurzfassung

In dieser Arbeit verwenden wir Reinforcement Learning, um einem Agenten das Spiel *Schnapsen*, das nationale Kartenspiel Österreichs, beizubringen. Es gehört zu den Spielen mit unvollständiger Information, da jeder Spieler nur die Karten in seiner Hand sieht, während jene des Gegners unbekannt sind.

Wir vergleichen verschiedene Methoden von Reinforcement Learning und evaluieren sie hinsichtlich ihre Fähigkeit, das Spiel zu erlernen. Die Ergebnisse unserer Arbeit können auch in anderen Bereichen interessant sein, etwa bei automatisierten Auktionen, da diese ebenfalls als Spiele mit einfachen Regeln und unvollständigen Informationen über die Gebote anderer Teilnehmer angesehen werden können.

Abstract

In this work we use reinforcement learning to teach an agent the game of *Schnapsen*, the national card game of Austria. It is a trick-taking card game with imperfect information, since each player can only see the cards in their hand while the cards held by the opponent are unknown.

We evaluate various methods of reinforcement learning regarding their suitability for this purpose. The results of our work may also be interesting in other areas like automated auctions, which may be viewed as games with simple rules and imperfect information about the bids of other participants.

Contents

Kurzfassung **vii**

Abstract **ix**

Contents **xi**

1 Introduction **1**
1.1 Problem Statement and Motivation . . . 1
1.2 Research Questions . . . 2
1.3 Methodology and Approach . . . 2
1.4 Structure of the Thesis . . . 3

2 Schnapsen **5**
2.1 Rules . . . 5

3 Game Theory **11**
3.1 Definitions . . . 11
3.2 Coin Toss Game . . . 12

4 Reinforcement Learning **15**
4.1 Definitions . . . 16
4.2 Comparison of Algorithms . . . 16

5 Related Work **21**
5.1 Reinforcement Learning for Perfect Information Games . . . 21
5.2 AI-based Approaches for Imperfect Information Games . . . 22
5.3 Reinforcement Learning for Imperfect Information Games . . . 22
5.4 Agents for Schnapsen . . . 23

6 AI for Mini-Schnapsen **25**
6.1 Rules of Mini-Schnapsen . . . 25
6.2 Considerations on the State Space of Mini-Schnapsen . . . 26
6.3 Monte Carlo . . . 26
6.4 Temporal-Difference Learning . . . 27

6.5 Comparison of the Tabular Methods 29
6.6 Approximate Action-Value Methods 31
6.7 Search Space 31

7 AI for Schnapsen **35**
7.1 Feature Vector 35
7.2 Neural Network Architecture 38
7.3 Update Target 40
7.4 Input Features and Markov Property 43
7.5 Actor Critic 45

8 Implementation **49**

9 Evaluation against Doktor Schnaps **51**

10 Conclusion **53**
10.1 Summary 53
10.2 Future work 54

List of Figures **55**

List of Tables **57**

List of Algorithms **59**

Bibliography **61**

CHAPTER 1

Introduction

1.1 Problem Statement and Motivation

Reinforcement Learning. Machine learning is a subfield of artificial intelligence that studies computer algorithms improving automatically through experience [Mit97]. One of its main paradigms is reinforcement learning (RL) [SB18].

In reinforcement learning, an agent is trained to make a sequence of decisions in a given environment while trying to maximize a cumulative reward. The machine employs trial and error to come up with a solution to the problem. The agent receives a representation of the current state of the environment and a set of actions, from which it can choose from. Each action potentially leads to a different state with a new set of admissible actions and a reward. In the beginning of the training process, the agent often explores the environment by acting randomly. Later, the agent can exploit the past experiences to come up with a sophisticated strategy to maximize the expected cumulative rewards.

Games and AI Being able to compete with and excel humans in game play has always been one of the key benchmarks of artificial intelligence (AI). Moreover, computer games use agents equipped with AI to serve, alongside humans, as players in multi-player games.

In the last years, AI agents reached super-human levels in various games with perfect information [SSS+17]. Perfect information refers to the fact that each player, when making any decision, is perfectly informed about all previous events, including the "initialization event" of the game (like the starting hands of each player in a card game). Examples of such games are Chess, Go and Shogi.

Recently, also games with imperfect information like Poker and Dota 2 have been tackled using RL [MSB+17]. Studying such games is particularly interesting, as many real life situations can be modeled as imperfect information games. However, this type of game requires more complex reasoning than similarly sized perfect information games.

Aims of the Thesis The aim of this work is to use RL to teach an agent *Schnapsen*, the national card game of Austria, or simplified versions thereof. It is a trick-taking card game with imperfect information, as each player can only see the cards in their hand, while the cards of the opponent are unknown.

To the best of our knowledge, there are no previous attempts of using reinforcement learning to teach an agent the game of *Schnapsen*. We will evaluate various strategies of RL regarding their suitability for this purpose. The results of our work may also be interesting in other areas such as automated auctions, which can be viewed as games with simple rules and imperfect information about the bids of other participants.

1.2 Research Questions

In our thesis, we address the following questions.

- Which algorithms for reinforcement learning are there? What are their key differences and for which problems are they suited?
- How does imperfect information affect the training of an agent? Which algorithms are suited for imperfect information settings?
- Is it possible to use reinforcement learning for teaching an agent the game of *Schnapsen*? If yes, can it compete with human players or with other bots using non-RL methods?

1.3 Methodology and Approach

To address our research questions, we apply the following methods.

1. **Literature Review:** To assess the state of the art, we systematically search for literature [KC07] on reinforcement learning in general and also specifically in the context of imperfect information.
2. **Comparison of RL Methods:** We explain the key features of different RL algorithms and give an overview for which problems they have been used in the past and what advantages/disadvantages each of them has [SB18].
3. **Implementing a Framework for *Schnapsen*:** In order to train an agent for the game of *Schnapsen*, we first have to implement a framework that keeps track of the game states and provides the agent with the set of admissible actions at each step.
4. **Implementing Training Algorithms:** We implement various RL algorithms that seem suited for learning *Schnapsen* and trained them by self-play [HDG+19].

5. **Evaluation:** We compare the agents in terms of playing-strength and in terms of other criteria, such as speed of convergence, robustness, and speed of decision making.

1.4 Structure of the Thesis

Chapter 2 summarizes the rules of the game of Schnapsen. In Chapter 3, we introduce the concepts of game theory relevant to this thesis. Chapter 4 gives an overview of reinforcement learning and discusses differences between RL methods. Chapter 5 discusses work related to reinforcement learning and games with imperfect information. In Chapter 6, we start our evaluation by comparing different methods on a simplified version of Schnapsen, which we then extend to the full game in Chapter 7. Chapter 8 specifies some details of our implementation. In Chapter 9, we compare our RL agents to the best AI that currently exists for the game of Schnapsen. Finally, Chapter 10 summarizes our work.

CHAPTER 2

Schnapsen

Schnapsen is the national card game of Austria [Tom15]. The earliest known written reference to the game dates back to 1715 [Cor15], but it was undoubtedly played long before that. Schnapsen is a trick-taking card game of the Bézique family that is popular in Bavaria and the territories of the former Austro-Hungarian Empire. Traditionally it is played with two players, but there are also variants for three (Dreierschnapsen) and four players (Bauernschnapsen). We consider the version for two players only. The precise rules of the game vary with place and time. For the thesis, we adhere to rules as specified by Tompa Martin [Tom15].

2.1 Rules

2.1.1 Aim

A game consists of a series of deals, where a deal is won by the first player to score 66 points. Points are obtained by winning tricks and declaring marriages. For brevity, we call them *trick points*.

Depending on the trick points at the end of a deal, the winner of the deal receives up to three points toward the game as a whole. We refer to these points as *game points*.

A game is won by the player that first scores seven game points.

2.1.2 Cards

Even though Schnapsen is often played with german-suited playing cards (suits of acorns, leaves, hearts, and bells), we will use the standard four French suits throughout this thesis: spades (♠), hearts (♡), clubs (♣), and diamonds (♢). Schnapsen is played with a deck of 20 cards, 5 cards in each suit. The cards are ranked according to their value, with a higher value corresponding to a higher rank:

Rank	Value
Ace (A)	11
Ten (T)	10
King (K)	4
Queen (Q)	3
Jack (J)	2

2.1.3 Dealing

The player dealing the first deal is decided at random, e.g. by each player drawing a card. The player with the higher card becomes the *dealer*. The other player is referred to as the *forehand*. In subsequent deals, the players alternate the roles of dealer and forehand. Likewise, in subsequent games the player dealing the first deal alternates with every game.

The dealer shuffles, cuts and deals the cards in the following order: three cards to the forehand, three cards to the dealer, one card face-up in the middle of the table indicating the *trump* suit and finally two cards are dealt to the forehand and two to the dealer, resulting in a hand of five cards for each player. The remaining nine cards are placed in a pile face-down on top of, and perpendicular to, the face-up trump card, leaving the trump card visible. These ten cards in the middle of the table form the *stock*.

These conventions aim at improving randomness when playing manually. For the purpose of the thesis, we may assume that the hands of the players and the card indicating the trump suit have been selected by any random process.

2.1.4 Tricks

Once the cards have been dealt, the players compete in a series of rounds, where each player plays a card from their hand face-up in the center of the table, forming a *trick* of two cards. Playing the first card of a trick is called *leading*, the first card is called the *lead*, and the player leading is said to be *on lead*. The forehand is the leader of the first trick. The winner of the trick collects the two cards and places them face-down on their side of the table.

The winner of a trick is determined as follows. If none of the two cards is in the trump suit, the higher ranked card in the suit of the card that was played first wins the trick. Otherwise, the player that played the higher ranked card in the trump suit wins the trick.

Next, both players draw a card from the top of the stock, starting with the winner of the last trick. The winner of the last trick also leads the next trick. The very last card drawn will be the face-up trump card.

2.1.5 Marriages

If you are the leading player of the upcoming trick and you have a king and a queen of the same suit in your hand, you can declare a *marriage*. This is done by showing both marriage cards to your opponent, followed by playing one of the two. This gives you extra trick points additionally to the ones that you have already collected by winning tricks:

Suit of the marriage	Extra trick points
Trump suit	40
Any other suit	20

2.1.6 Claiming 66

If a player has either just won a trick or declared a marriage and has gathered at least 66 trick points (included points obtained by marriages), this player can claim 66 and wins the current deal. The winner receives up to three game points (see Section 2.1.11), and the next deal starts.

2.1.7 Closing the Stock

If you are on the lead and the stock is neither exhausted nor closed, you can choose to *close the stock*. By this action you claim that you can win the current deal by obtaining 66 points without any player drawing additional cards. You and your opponent play cards one at a time from your hands, without refilling your hands from the stock. If you fail to win the deal by claiming 66, or if your opponent claims 66 first, you loose and your opponent is awarded bonus game points (see Section 2.1.11). When the stock is closed or exhausted, the non-leading player has to follow the suit of the lead card and has to try to win the trick.

2.1.8 Following Suit

Once the stock is exhausted (i.e., all cards have been drawn) or one of the player has closed the stock, the non-leading player (i.e., the player that plays the second card of the trick) has to follow suit and, if possible, win the trick. This means:

1. If your hand contains a card in the suit of the card played and ranked higher, you have to play such a card.
2. Otherwise, if your hand contains a card in the suit of the card played but all rank lower, you have to play such a card.
3. Otherwise, if you have a card of the trump suit, you have to play such a card.
4. Otherwise, you may play any card.

2.1.9 Exchanging Trumps

If

- you have the Jack of the trump suit in your hand and
- the stock is neither closed nor exhausted and
- you are on the lead,

then you may exchange the Jack of the trump suit for the face-up trump in the stock, prior to leading the next trick.

2.1.10 Last Trick

If the stock is exhausted (not closed) and all cards have been played from both hands, the winner of the last trick wins the entire deal.

2.1.11 Scoring Game Points

To determine how many game points are awarded to the winner of the deal, we distinguish two scenarios.

The Stock was not Closed

If the stock was not closed, a player either won by claiming 66 or by winning the last trick. In that case, the winner scores as follows.

Loser took no tricks	3 points
Loser's trick points (including declared marriages) add up to less than 33	2 points
Loser's trick points (including declared marriages) add up to at least 33	1 point

The Stock was Closed

When a player closes the stock, the game points at stake are only determined by the tricks and trick points of the non-closing player at the moment the stock was closed. If the player closing the stock manages to accumulate 66 trick points before the other player, he wins and scores game points as follows.

Non-closing player had no tricks when stock was closed	3 points
Non-closing player's trick points (including declared marriages) added up to less than 33 when stock was closed	2 points
Non-closing player's trick points (including declared marriages) added up to at least 33 when stock was closed	1 point

If the closing player fails to claim 66 or the non-closing player claims 66 first, the non-closing player scores game points as follows.

Non-closing player had no tricks when stock was closed	3 points
Non-closing player had at least one trick when stock was closed	2 points

CHAPTER 3

Game Theory

The first books on the mathematics of games already appeared 400 years ago, e.g. [Huy57] in 1657. However, it was until the 20th century that the research field of game theory was established. Often, the eminent book "Theory of Games and Economic Behavior" by John von Neumann and Oskar Morgenstern [vNM44], published in 1944, is considered the beginning of modern game theory.

3.1 Definitions

Strategy: defines how a player will act in every possible game state. For a *pure strategy*, the action depends deterministically on the given state. For a *mixed strategy*, the action is selected according to some probability distribution, in at least some states.

Nash equilibrium: a set of strategies, one for each player, where no player can gain by deviating from their initial strategy.

Payoff: the value associated with a possible outcome of a game. We use u_i to denote the payoff of player i.

Zero-sum game: The gain of one player is the loss of another one ($\sum_i u_i = 0$).

Two-player zero-sum game: zero-sum game with two players ($u_1 = -u_2$).

Extensive game form: a way of describing a sequential game using a game tree (as in Figure 3.1). The non-leaf nodes correspond to choices of a player at the given game state and the leaf nodes correspond to the resulting payoffs.

Subgame: any subtree of the extensive game form.

Perfect and imperfect information: In a *perfect information* game (PIG), each player, when making a decision, is perfectly informed about all previous events. Examples of PIGs are Chess, Go and Backgammon. *Imperfect information games (IIGs)* involve

information that is private to some players. Examples are card games like Poker, Skat and also Schnapsen, as each player only knows the cards in their own hand, but not the cards of the other players.

3.2 Coin Toss Game

We introduce a simple game with imperfect information to highlight two key difference between PIGs and IIGs for finding a Nash equilibrium.

1. In PIGs, there exists a Nash equilibrium with pure strategies for all players. In IIGs, Nash equilibria usually involve mixed strategies.
2. In PIGs, it is usually sufficient to look at the current state only and disregard past actions. For example, in Chess we only need to know the current positions of the pieces on the board and whose turn it is to be able to find the best move. In IIGs, on the contrary, past actions are important to take into account as they may reveal some private information of the opponent. Also, in IIGs, the values of subgames that did not occur affect the Nash equilibrium in a given state.

The following two-player zero-sum game, the *coin toss game*, illustrates the concepts[1].

> A coin gets tossed and shows with equal probability either Heads or Tails. Only P_1 observes the outcome of the coin toss and then decides between the actions Sell or Play. If P_1 chooses Sell, the game ends. P_1 receives a reward of 0.5 if the coin shows Heads and -0.5 if the coin shows Tails. If P_1 chooses Play, then it is P_2's turn and P_2 has to guess what the outcome of the coin toss was. If P_2 guesses correctly, P_1 receives a reward of -1, and 1 otherwise.

The extensive game form is shown in Figure 3.1. The dotted line between the two P_2-nodes signals that both nodes belong to the same *information set*, meaning that P_2 cannot tell in which of these two nodes P_2 is in. Therefore, the strategy has to be the same for these two nodes. What should P_2 guess when playing against a best response opponent?

The first option is to guess always heads. In this case, P_1 will maximize the reward by always selling when the initial coin flip resulted in heads and always playing when the initial flip resulted in tails. The expected value for P_1 is then $0.5 \cdot 0.5 + 0.5 \cdot 1 = 0.75$.

The second option for P_2 is to always guess tails. In this case, P_1 will maximize the reward by always playing when the initial coin flip resulted in heads and always selling when the initial flip resulted in tails. The expected value for P_1 is then $0.5 \cdot 1 + 0.5 \cdot (-0.5) = 0.25$, which is an improvement for P_2 compared to the previous scenario.

[1] Adapted from a presentation by Noam Brown `https://www.youtube.com/watch?v=2dX0lwaQRX0`.

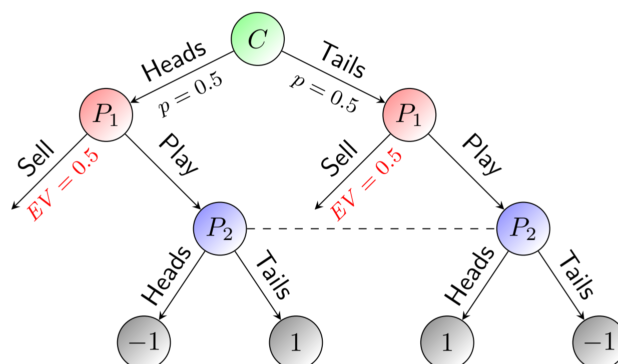


Figure 3.1: Extensive game form for the coin toss game

However, it turns out that the optimal strategy for P_2 is guessing heads with a probability of 0.25 and guessing tails with a probability of 0.75. This way, the expected values of choosing Play and Sell are the same for both possible outcomes of the initial coin flip and the value of the game for P_1 is $0.5 \cdot 0.5 + 0.5 \cdot (-0.5) = 0$. The same holds for the strategy of P_1: Any pure strategy enables P_2 to exploit P_1 by adjusting its own strategy. A mixed strategy is needed to ensure that the opponent cannot achieve a higher expected value than 0.

Next, assume that the values for selling change (e.g. $q(\mathsf{Sell} \mid \mathsf{Heads}) = -0.5$ and $q(\mathsf{Sell} \mid \mathsf{Tails}) = 0.5$). Clearly, this affects the optimal strategy of P_2 in the *play subgame* even though this subgame has not changed. P_2 will now guess heads with a probability of 0.25 and tails with a probability of 0.75.

Summarizing, in PIGs we can analyze a subgame perfectly given only the information within this subgame alone. In IIGs, we have in general to take into account past actions as well as the strategies and values of other subgames to come up with an optimal strategy.

CHAPTER 4

Reinforcement Learning

Machine learning is a subfield of artificial intelligence that studies computer algorithms improving automatically through experience [Mit97]. One of its main paradigms is reinforcement learning (RL) [SB18].

In RL, an agent is trained to make a sequence of decisions in a given environment while trying to maximize a cumulative reward. The machine employs trial and error to come up with a solution to the problem. The agent receives a representation of the current state of the environment and a set of actions to choose from. Each action potentially leads to a different state with a new set of legal actions and a reward. This cycle is demonstrated in Figure 4.1. At the start of training, the agent often explores the environment by random actions. Later, the agent exploits the past experiences to come up with a more sophisticated strategy to maximize the expected cumulative rewards.

The next sections introduce the RL methods that we are going to test for the game of Schnapsen. For a more thorough description of these methods, we recommend [SB18]. We will also adopt the notations from this book.

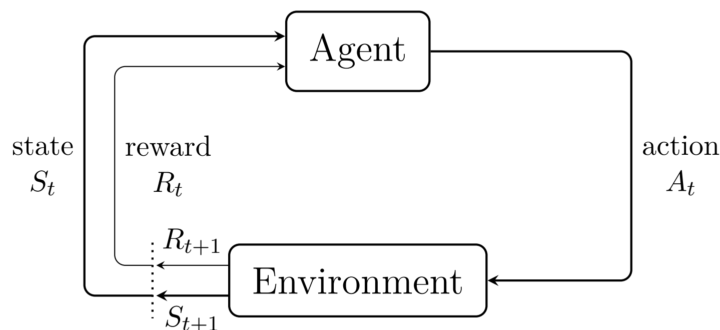


Figure 4.1: Cycle of reinforcement learning [SB18, p.54]

4.1 Definitions

4.1.1 Notations

The set of all possible states is denoted as $\mathcal{S}$.

The set of all possible actions in a given state $s \in \mathcal{S}$ is denoted as $\mathcal{A}(s)$.

4.1.2 Markov Decision Process

All the interactions between the agent and the environment can be recorded in a sequence:

$$\langle S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, \dots \rangle$$

This sequence is called a *Markov Decision Process* or is said to satisfy the *Markov property* if the random variables S_t and R_t provided by the environment depend only on the preceding state and action. Satisfying the Markov property is often a prerequisite for the theoretical properties of RL algorithms.

4.2 Comparison of Algorithms

We differentiate between *action-value* methods and *policy-gradient* methods.

4.2.1 Action-Value Methods

In action-value methods, the agent approximates the expected cumulative reward of each action in a given state s and chooses the action in a greedy manner, i.e. the action with the highest value. An exact description, which action is taken witch which probability in each state is called a *policy* and denoted as $\pi\colon \mathcal{S} \times \mathcal{A}(s) \to [0, 1]$.

For small state-action spaces, we can use a table where we store an value for each state-action pair. However, for decently large problems, this is not feasible anymore and we need to use some kind of approximation function, e.g. a neural network. We denote this table or approximation function as $q\colon \mathcal{S} \times \mathcal{A}(s) \to \mathbb{R}$.

During training, whenever the agent chooses the action a in state s, the approximated value $q(s, a)$ gets updated towards some target U, depending on the method used. As we want to make sure that all state-action pairs get updated over time, we usually use an ϵ-soft policy, e.g. the ϵ-greedy policy. The ϵ-greedy policy chooses with an probability of $\epsilon > 0$ an random action, i.e. in a given state s, the action A is chosen as

$$A = \begin{cases} \arg\max_{a \in \mathcal{A}(s)} q(s, a) & \text{with probability } 1 - \epsilon, \\ \text{random } a \in \mathcal{A}(s) & \text{with probability } \epsilon. \end{cases}$$

As already mentioned, the main difference between most action-value methods is only the choice of the update target during training. We will take a look at the following three action-value methods: *Monte Carlo*, *SARSA*, *Q-learning*.

Monte Carlo Method

For the Monte Carlo method, no updates happen during an episode, but only at the end of an episode when the agent actually receives a reward. So all states that have been visited and all actions that have been taken during an episode are stored in an array-like object. Then, at the end of the episode, the approximated values of all these states and actions get updated towards the actual reward received. Hence,

$$U = \sum_{t=0}^{T} \gamma^t R_t$$

where T denotes the end of the episode and γ is the discount factor, with $0 < \gamma \leq 1$. The discount factor is set to a value less than 1 if the agent is supposed to prefer early rewards over later ones.

One disadvantage of this method is, that it assumes that all actions contributed equally to the final reward. Hence, all the good actions as well as the bad actions taken by the agent get updated towards the same target.

However, Monte Carlo methods often still work fine when the Markov property is not satisfied.

SARSA Method

The SARSA method does not wait until the end of the episode before updating its policy, but rather updates it after each single time step. Whenever the agent is in a state s and chooses action a, it observes the potential reward r that it receives and the following state s'. According to the agent's policy, it will again choose some action a'. The value of the state-action $q(s, a)$ will then get updated towards r, the reward received, plus the estimated value of the next state-action pair $q(s', a')$. Hence,

$$U_t = R_{t+1} + \gamma q(S_{t+1}, A_{t+1}).$$

Updates happen online during the episodes and therefore, there is no need to keep track of all the states and actions taken during an episode.

However, as we often use an ϵ-greedy policy during training, the values of some state-action pairs will be updated towards the value of an state where the agent took a random action. This may result in a less accurate target.

Q-Learning Method

The Q-learning method is actually very similar to SARSA with the only difference being that the update target is not necessarily the estimated value of the following action taken, but rather the highest estimated value of all possible actions in the following state, i.e.

$$U_t = R_{t+1} + \gamma \max_a q(S_{t+1}, a).$$

For some problems, Q-learning converges faster than SARSA.

4.2.2 Policy-Gradient Methods

Policy-gradient methods calculate policies more directly than action-value methods. They learn a parameterized policy that can select actions without consulting a value function.

We use the notation $\theta \in \mathbb{R}^d$ for the policy's parameter vector and we write $\pi(a|s, \theta)$ for the probability that action a is taken given that the state is s and the parameter θ. We seek a parameter that corresponds to an optimal policy.

Again, the main difference between different policy-gradient methods is the update rule of θ. Additionally, some policy-gradient methods use also a state value approximation to improve the updating of the policy's parameter.

One key advantage of policy gradient methods is, that they are able to learn mixed strategies with arbitrary probabilities by design.

The only policy-gradient method we will try for the game of Schnapsen will the *actor-critic* method, however, we will also introduce *REINFORCE* and *REINFORCE with baseline* to better understand the idea behind actor-critic.

REINFORCE

REINFORCE can be seen as the policy-gradient version of the Monte-Carlo method, as we again keep track of all the states visited, actions chosen and rewards received and wait until the end of the episode before we do any updates. For each state-action pair, the policy's parameter is than updated as

$$\theta_{t+1} = \theta_t + \alpha\gamma G_t \frac{\nabla\pi(A_t|S_t, \theta_t)}{\pi(A_t|S_t, \theta_t)}$$

where $G_t = \sum_{i=t}^{T} R_i$, with T being again the time step of the end of the episode.

With this update rule, each increment is proportional to the product of the return G_t and a vector, namely the gradient of the probability of taking the action actually taken divided by the probability of taking that action. The vector is the direction in parameter space that most increases the probability of repeating the action A_t on future visits to the state S_t. The update increases the parameter proportional to the estimated strength of the action (estimated as the return), and inversely proportional to the action probability. The latter makes sense because otherwise actions that are selected frequently are at an advantage (the updates will be more often in their direction) and might win out even if they do not yield the highest return.

REINFORCE with Baseline

Generally, all actions may tend to have rather high values in one state, while having low values for another state. This can introduce a lot of variance to the updates in the REINFORCE algorithm, which slows down learning. To compensate for this, we can introduce a baseline $b\colon \mathcal{S} \to \mathbb{R}$ that can vary with the state, but is not allowed to vary with the action chosen. The update rule is

$$\theta_{t+1} = \theta_t + \alpha\gamma(G_t - b(S_t))\frac{\nabla\pi(A_t|S_t, \theta_t)}{\pi(A_t|S_t, \theta_t)}.$$

The natural choice is an approximation of the expected value of G_t, i.e. the state-value function.

Actor-Critic Method

In REINFORCE with baseline, the learned state-value function estimates the value of only the first state of each state transition. This estimate sets a baseline for the subsequent return, but is made prior to the transition's action and thus cannot be used to assess that action.

In actor-critic methods, on the other hand, the state-value function is applied also to the second state of the transition. The full return of REINFORCE is replaced with the one-step return δ and a learned state-value function $v\colon \mathcal{S} \to \mathbb{R}$ is used as a baseline:

$$\theta_{t+1} = \theta_t + \alpha\delta_t\frac{\nabla\pi(A_t|S_t, \theta_t)}{\pi(A_t|S_t, \theta_t)}$$

with

$$\delta_t = R_{t+1} + \gamma v(S_{t+1}) - v(S_t).$$

CHAPTER 5

Related Work

5.1 Reinforcement Learning for Perfect Information Games

TD-Gammon TD-Gammon is one of the biggest successes of reinforcement learning for games [Tes95, SB18]. It combines a nonlinear form of the TD(γ) algorithm and function approximation via a neural network. Only little Backgammon knowledge was incorporated into the feature vector of the neural network, yet TD-Gammon learned to play extremely well, close to the level of the world's strongest grandmasters and the best previous Backgammon computer programs. This was a striking result because all previous high-performance computer programs relied extensively on Backgammon knowledge. Subsequently, TD-Gammon was further improved by adding specific Backgammon features and by using heuristic search.

AlphaGo The ancient Chinese game of Go has challenged AI researchers for various decades. In 2016, a team at DeepMind developed the program AlphaGo [SHM$^+$16], which outperformed all previous Go programs and defeated the world champion Lee Sedol, winning 4 out of 5 games. The main innovation that made AlphaGo such a strong player is that it selected moves by a novel version of Monte Carlo Tree Search (MCTS) that was guided by both a policy and a value function learned by RL using a deep convolutional neural network. Furthermore, instead of starting RL from random weights within the network, the initial weights were the result of previous supervised learning from a collection of human expert moves.

AlphaGo Zero In contrast to AlphaGo, its successor AlphaGo Zero did not use any human data or guidance beyond the basic rules of the game [SSS$^+$17]. It learned exclusively from self-play RL, with the input consisting just of the positions of the stones on the Go board. Another significant difference between the two versions is that AlphaGo

Zero used MCTS to select moves throughout self-play RL, whereas AlphaGo used MCTS only for live play afterwards, but not during training. Even though AlphaGo learned quicker than AlphaGo Zero in the beginning of the training process, after 40 days of training AlphaGo Zero won against AlphaGo 100 games out of 100. AlphaGo Zero demonstrates that superhuman performance can be achieved by pure RL, augmented by a simple version of MCTS, without relying on human data or guidance.

5.2 AI-based Approaches for Imperfect Information Games

DeepStack Poker, the prototypical game of imperfect information, has been a long-standing challenge in artificial intelligence. DeepStack defeated professional poker players in heads-up no-limit Texas hold'em [MSB+17]. DeepStack does not compute and store a complete strategy prior to play. Like Alpha Zero, it uses heuristic search to choose an action in a given state. However, in imperfect information games past actions have to be taken into account. DeepStack does this by complementing the state with information about its own range (probabilities of different private holdings) and counterfactual values of the opponent (expected value if the opponent reaches the public state with a particular hand). It avoids reasoning about the entire remainder of the game by substituting the computation beyond a certain depth with a value estimate function. This function takes the form of a neural network and was trained on 10 millions of randomly generated states beforehand.

Libratus Libratus is another AI for heads-up no-limit Texas hold'em [BS18]. It managed to defeat professional poker players around the same time as DeepStack. It also uses counterfactual values and real time solving via heuristic search, but differs from DeepStack in algorithmic aspects. As both Poker programs do not rely on reinforcement learning, we refrain from discussing these differences in detail.

5.3 Reinforcement Learning for Imperfect Information Games

ReBeL ReBeL (Recursive Belief-based Learning) is a general RL and search framework that converges to a Nash equilibrium in two-player zero-sum games [BBLG20]. For perfect information games, ReBeL simplifies to an algorithm similar to AlphaZero, with the major difference being the type of search algorithm employed. ReBeL expands the notion of "state" to include the probabilistic belief distribution of all agents about what state they may be in, based on common knowledge observations and policies for all agents. One key difference to the algorithm of DeepStack is that the value function is trained via self-play RL instead of generating random public belief states. ReBeL also demonstrated superhuman performance in heads-up no-limit Texas hold'em.

5.4 Agents for Schnapsen

Doktor Schnaps Doktor Schnaps belongs to the strongest programs for the game of Schnapsen [Tom15]. It uses a modified version of Perfect Information Monte Carlo Sampling [Wis10, Wis15, Wis16]. In a given state, the agent runs many simulations with possible opponent hand cards and deck permutations and assumes that all private information is public. Hence, the game transforms to a perfect information game that is then solved. Actions are selected with a modified version of the minimax algorithm. Doktor Schnaps plays above expert human level.

CHAPTER 6

AI for Mini-Schnapsen

In this chapter we compare different RL algorithms for a simplified version of Schnapsen, *Mini-Schnapsen*, a made-up game we specify in the first section below. This simplification is necessary to be able to include tabular methods, which store information for each possible state and therefore do not scale to the state space of the full game of Schnapsen. However, they are interesting as they are able to find optimal strategies if they are trained for sufficiently many episodes. Next, we will look at function approximations, which are also applicable to large state spaces. Last but not least, we will consider policy-gradient methods, which allow the agent to use mixed strategies, important for imperfect information games.

6.1 Rules of Mini-Schnapsen

Mini-Schnapsen is played with 12 cards – three different symbols with four different values (J,Q,K,T) each.

Each player starts with three cards in the hand. As with Schnapsen, the last card of the deck determines the trump of the deal, but it is not faceup, hence the value of the last card is not known. Playing tricks and drawing cards follows the rules of the full game, except that there is no closing of the stock, exchanging of the trump or declaring of marriages. When the stock is exhausted, suit has to be followed by the non-leading player. Once all cards have been played, the player with the most trick points wins.

The ranking and the trick points awarded are the same as in Schnapsen:

Rank	Value
Ten (T)	10
King (K)	4
Queen (Q)	3
Jack (J)	2

6.2 Considerations on the State Space of Mini-Schnapsen

For the game of Schnapsen with 20 cards, we have $\sum_{i=0}^{10} \binom{20}{2i} = 524\,288$ combinations of cards played. For each of these combination, we need to consider all combinations of possible hand cards etc., which quickly exceeds available storage capacities.

For Mini-Schnapsen, this number of combinations reduces to $\sum_{i=0}^{6} \binom{12}{2i} = 2048$. Apart from the cards already played, we need to know how many points each player has so that we know how many points we still need to win and how risky we should play. Finally, we also have to consider our own hand and whether we lead the trick or not. If we do not, we need the information which card was led first to specify the state.

As Mini-Schnapsen is an imperfect information game and does not satisfy the Markov property, we would actually need to include the whole history of the previous tricks played: Past tricks may give us information on the probability distribution of the possible hand cards of our opponent. However, this would enlarge the state space even more, which is not feasible for tabular methods. We return to this problem later on when we look at approximation methods for the original game of Schnapsen.

6.3 Monte Carlo

The first algorithm that we will test for the game of Mini-Schnapsen is the Monte Carlo method. To derive the ϵ-greedy policy for a given state s, we need to keep track of how often we have chosen each action in this state s and how often this action led to a win or loss of the game. Therefore, we have two tables with an entry for each state-action pair:

- Action-value table: a table that stores for each action a in each state s a value, $Q(s, a)$, with $-1 < Q(s, a) < 1$. Normalizing $Q(s, a)$ to a value between 0 and 1 (i.e. $\frac{Q(s,a)+1}{2}$) gives an estimation of the probability of winning the game taking the given action in the given state.

- Count table: stores for each state-action pair a count, $Q_{count}(s, a)$, of how often we have already taken a given action in a given state and subsequently, how often we have updated this state-action pair. This is necessary for being able to incrementally update the averages of the action-value table.

Algorithm 6.1 specifies the Monte-Carlo method in pseudo-code. Throughout the thesis, we set $i_{\max}$, the number of episodes trained, to 5 million. This value ensures that the strategy converges while limiting training time (see Section 7.2).

Algorithm 6.1: Monte Carlo Self Play

Input: states $\mathcal{S}$, actions $\mathcal{A}$, $\epsilon_0 > 0$
Output: $Q^*(s, a)$

```
1  Q(s,a) ← 0 for all s ∈ S, a ∈ A(s)
2  Q_count(s,a) ← 0 for all s ∈ S, a ∈ A(s)
3  i_max ← 5 000 000
4  for i ← 1 to i_max do
5      ε ← ε_0(1 − i/i_max)
6      Play a game, where both players follow the ε-greedy policy derived from Q.
7      Store the state-action pairs of player P1 and player P2 in separate lists.
8      reward_P1 ← {  1  if P1 won
                      0  if draw
                     −1  if P2 won
9      reward_P2 ← −reward_P1
10     for P ∈ {P1, P2} do
11         for (a,s) in P's list of state-action pairs do
12             Q_count(s,a) ← Q_count(s,a) + 1
13             Q(s,a) ← Q(s,a) + (reward_P − Q(s,a)) / Q_count(s,a)
14 return Q
```

We trained various agents, varying the initial ϵ_0. Figure 6.1 shows the results of letting them play against a random agent. Apparently, the agent with $\epsilon_0 = 1$ performs best.

6.4 Temporal-Difference Learning

6.4.1 SARSA

In contrast to the Monte Carlo method, SARSA does not wait until the end of the episode to update the values of the Q-table. After each action, the corresponding value is updated a small step towards the estimated value of the resulting state. The size of this step is proportional to the difference between the estimated value of the state action pair and the estimated value of the following state. Additionally, it is proportional to a predefined learning rate α, which has to be tuned like the exploration rate ϵ. Furthermore, we can choose to let α decay over time, such that the estimated values converge more smoothly towards the end of training. Algorithm 6.2 specifies the SARSA method as pseudo-code.

We tested agents with various settings against a random agent. Again, agents with $\epsilon_0 = 1$ perform best. Figure 6.2 shows the results for different learning rates α. From all our tested settings, $\epsilon_0 = 1.0$ and $\alpha_0 = 0.7$ perform best.

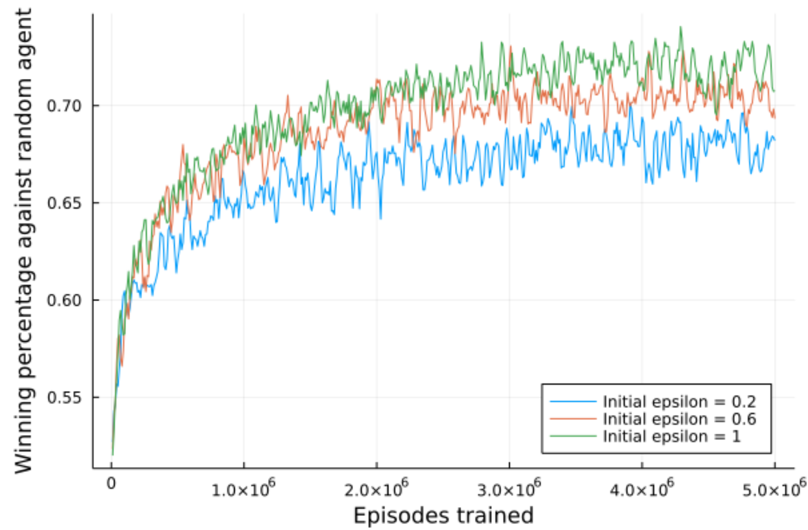


Figure 6.1: Comparison of three Monte-Carlo agents with initial ϵ_0 set to 0.2, 0.6 and 1.0, respectively. The agents' performance as tested every 10 000 episodes of training by playing a match of 1000 games against a random agent.

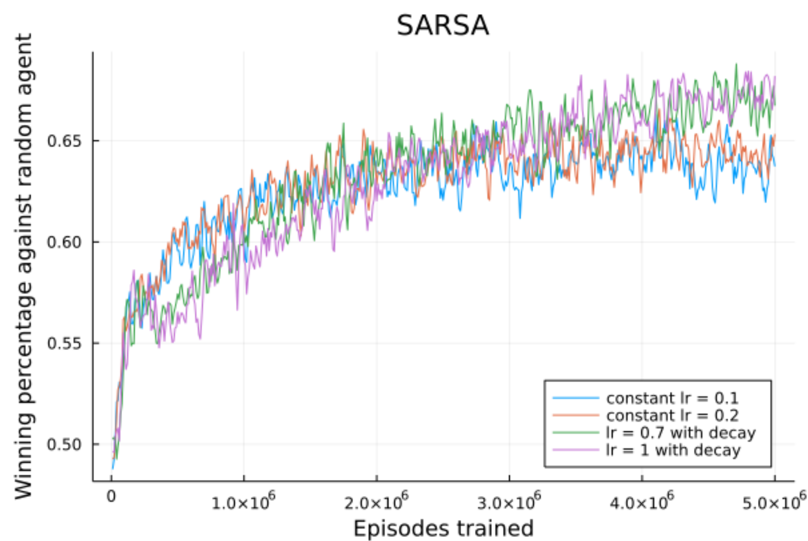


Figure 6.2: Comparison of four SARSA agents with the initial ϵ_0 set to 1 and various settings for the learning rate: $\alpha = 0.1$ and $\alpha = 0.2$ without decay as well as $\alpha = 0.7$ and $\alpha = 1.0$ with decay. The agents' performance was determined every 10 000 episodes by playing a match of 1000 games against a random agent.

Algorithm 6.2: SARSA Self Play

```
    Input: states S, initial states S_0 ⊆ S, actions A, ε_0 > 0, α_0 > 0, α-decay (Bool)
    Output: Q*(s, a)
 1  Q(s, a) ← 0 for all s ∈ S, a ∈ A(s)
 2  α ← α_0
 3  i_max ← 5 000 000
 4  for i ← 1 to i_max do
 5  |   ε ← ε_0(1 − i/i_max)
 6  |   if α-decay then
 7  |   |   α ← α_0(1 − i/i_max)
 8  |   Game gets initialized with a randomly chosen state s from S_0.
 9  |   Player to act (P_a) chooses action a according to the ε-greedy policy derived
    |     from Q.
10  |   for each step of the game do
11  |   |   Player observes the following state s' where again an action a' is chosen
    |   |     according to the ε-greedy policy derived from Q.
12  |   |   if s' final state then
    |   |   |                 ⎧  1   if P_a won
13  |   |   |   reward ←  ⎨  0   if draw
    |   |   |                 ⎩ −1   if P_a lost
14  |   |   |   Q(s, a) ← Q(s, a) + α(reward − Q(s, a))
15  |   |   else
16  |   |   |   Q(s, a) ← Q(s, a) + α(Q(s', a') − Q(s, a))
17  |   |   a ← a'
18  |   |   s ← s'
19  return Q
```

6.4.2 Q-Learning

Like SARSA, Q-learning does not wait for the end of the episode to update the Q-table. However, instead of using the value of the next action taken by the agent as an estimate for the new value, Q-learning uses the value of the next greedy action. This way, the agent can explore while still taking the greedy value for updating the Q-table. Algorithm 6.3 gives the pseudo-code for this variant. Pseudo code for the algorithm is given in 6.3. Again, testing different settings suggests that $\epsilon_0 = 1.0$ and $\alpha_0 = 0.7$ is a good choice.

6.5 Comparison of the Tabular Methods

In Table 6.1, we compare the fully trained agents implementing different algorithms. SARSA and Q-learning perform similarly well while they both seem to have a slight advantage over the Monte-Carlo method.

Algorithm 6.3: Q-learning Self Play

Input: states $\mathcal{S}$, initial states $\mathcal{S}_0 \subseteq \mathcal{S}$, actions $\mathcal{A}$, $\epsilon_0 > 0$, $\alpha_0 > 0$, α-decay (Bool)
Output: $Q^*(s,a)$

```
1  Q(s,a) ← 0 for all s ∈ S, a ∈ A(s)
2  α ← α_0
3  i_max ← 5 000 000
4  for i ← 1 to i_max do
5      ε ← ε_0(1 − i/i_max)
6      if α-decay then
7          α ← α_0(1 − i/i_max)
8      Game gets initialized with a randomly chosen state s from S_0.
9      Player to act (P_a) chooses action a according to the ε-greedy policy derived
        from Q.
10     for each step of the game do
11         Player observes the following state s'.
12         a'* ← arg max_{a'} Q(s', a')
13         if s' final state then
14             reward ← {  1  if P_a won
                          0  if draw
                         −1  if P_a lost
15             Q(s,a) ← Q(s,a) + α(reward − Q(s,a))
16         else
17             Q(s,a) ← Q(s,a) + α(Q(s', a'*) − Q(s,a))
18         Player choose an action a' according to the ε-greedy policy derived from Q.
19         a ← a'
20         s ← s'
21 return Q
```

	MC	SARSA	Q-lrn
MC	0.5015	0.4827	0.4857
SARSA	0.5025	0.4969	0.4928
Q-lrn	0.5150	0.5076	0.4982

Table 6.1: Comparison of the tabular methods Monte-Carlo, SARSA and Q-learning. For learning, ϵ_0 was set to 1 and the learning rate of SARSA and Q-learning, α_0, was set to 0.7. All agents were trained for $5 \cdot 10^6$ episodes. Both, ϵ and α, decayed over time. The entries give the rate of the method in the left column winning against the method in the top row in a match of 10 000 games. E.g., the entry in the third row and first column tells us that the Q-learning agent won 51.5% of the games against the Monte Carlo agent.

6.6 Approximate Action-Value Methods

The problem with large state spaces is not just the memory needed for large tables, but also the time and data needed to fill them accurately. In the game of Schnapsen, for example, even after 5 million games we will still mostly see states that we have never encountered before. Therefore, to make sensible decisions, it is necessary to generalize from previous encounters with different states that are in some sense similar to the current one.

Generalization from examples has already been extensively studied. So-called function approximation is an instance of supervised learning. However, reinforcement learning with function approximation raises a number of new issues that do not normally arise in conventional supervised learning, such as non-stationarity, bootstrapping, and delayed targets.

Let U_t be the target value, where we want to move the value of an state-action pair. In the Monte Carlo tabular method, for example, U_t was the received return after that state which was dependent on the result of the game. The update was then

$$Q(s,a) \leftarrow Q(s,a) + \alpha(U_t - Q(s,a))$$

where, for the Monte Carlo method, the learning rate α was set depending on how often we had already updated this state-action pair.

Now, for approximation methods we want to update the weights w_t of our value function such that the updated estimation of the state-action is closer to U_t. We achieve this by moving the weights in the direction of the gradient multiplied by the difference between the target value and the current estimation:

$$w_{t+1} \leftarrow w_t + \alpha(U_t - \hat{q}(S_t, A_t, w_t))\nabla\hat{q}(S_t, A_t, w_t)$$

6.7 Search Space

Besides reinforcement learning, there are many more parameters to tune and architectures to choose from. Table 6.2 gives an overview of some.

There are also many different function approximation methods. As testing several of them would exceed the scope of this thesis, we use the one that achieved the most significant results in reinforcement learning: neural networks. Regarding their architecture, we will explore different options for the full game of Schnapsen, but stick to one set of sensible choices for Mini-Schnapsen:

- 2 hidden layers,
- 512 nodes per layer,

RL-method	action-value methods: Monte-Carlo, SARSA, Q-learning, ... policy-gradient methods: REINFORCE, Actor-Critic, ...
method specific hyper-parameters	learning rate α, exploration rate ϵ, α/ϵ-decay, ...
Neural network architecture	number of hidden layers and nodes per hidden layer, activation function, optimizer, ...
feature vector representation	many ways to represent a state and the set of legal actions
number of neural networks	different neural networks may be used for different types of states

Table 6.2: Overview of some parameters to tune and architectures to choose.

- classic gradient descent, optimizer
- sigmoid activation for the last layer and ReLU activation for all other layers.

Our agent uses two neural networks:

- one for leading a trick (q_1),
- one for choosing an action when the opponent led the trick (q_2).

To compare the approximation methods with the tabular versions, we use the same state features for the input layer as we did for the Q-table in the tabular methods. We obtain a feature vector of size 39 for q_1 and a feature vector of size 51 for q_2, with the features being:

- 12 binary inputs for the cards that have been already played. For each card the corresponding bit is set to 1 when the card has been already played and 0 when not.
- 12 binary inputs for the cards held in the hand of the player. For each card the corresponding bit is set to 1 when the card is in the hand and 0 when not.
- 12 binary inputs for the intended action. The bit for the corresponding card is set to 1, all others to 0.
- 2 inputs for the trick points of the player and the opponent. All cards add up to 57 points, hence once a player has more than 28.5 points he is guaranteed to win. To get an similar range as the other inputs, both trick points are divided by 28.5.

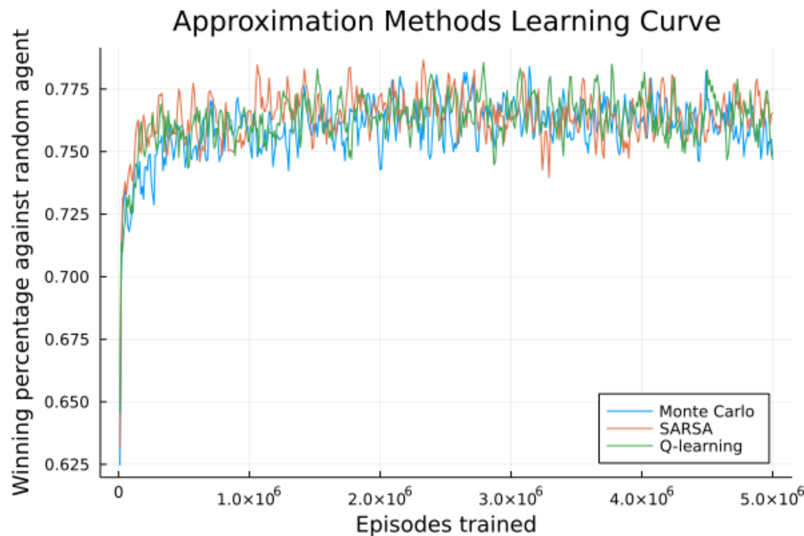


Figure 6.3: Comparison of the learning curves of the three approximation methods Monte-Carlo, SARSA, Q-learning. The agents' performances were tested every 10 000 episodes of training by playing a match of 1000 games against a random agent.

	MC	SARSA	Q-lrn
MC	0.5152	0.5030	0.4999
SARSA	0.5037	0.5044	0.4977
Q-lrn	0.5058	0.5007	0.5024

Table 6.3: Comparison of the approximation methods Monte-Carlo, SARSA and Q-learning. The entries are the rates of the agent in the left column winning against the agent in the top row in a match of 10 000 games.

- 1 input that signifies how many cards are left to draw and hence, whether suit has to be followed or not. After dealing three cards to each player, 6 are left on the stock. To adjust the range to the other inputs, the value fed into the neural network is divided by 6.

- Additionally, for q_2, there are 12 more binary inputs for the card led by the opponent. The bit for the corresponding card is set to 1, all others to 0.

Looking at the learning curves of the approximation methods against a random agent in Figure 6.3, we can see that they all perform similarly well. Already after 500k episodes they seem to reach the peak performance (at least against a random agent). Table 6.3 verifies that the fully trained agents of the different methods perform approximately equally well.

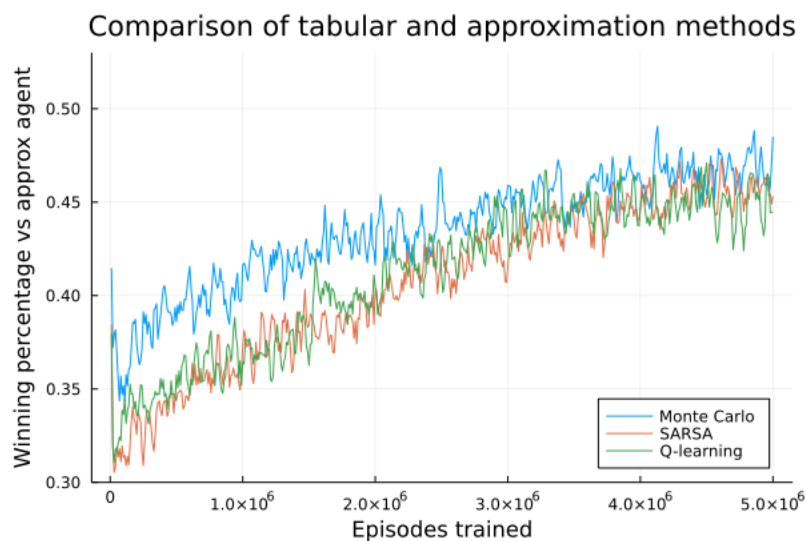


Figure 6.4: Comparison of the tabular methods vs. the approximation versions of the methods. Every 10 000 episodes the two agents played a match of 1000 games and the winning rate from the view of the tabular version is plotted.

The quicker learning in the initial episodes compared to the tabular versions is due to generalization from past experience. We further illustrate this aspect in Figure 6.4. After about 50k episodes of training, the agents that used function approximation have a huge advantage over the agents that do not. Then, slowly, the tabular methods approach the performance of the approximation methods but even after 5 million episodes they do not reach quite the same level. This already indicates how powerful function approximation can be in combination with reinforcement learning.

CHAPTER 7

AI for Schnapsen

As mentioned in the last chapter, the state space of Schnapsen is too large to apply tabular methods. Therefore, we will only consider the function approximation versions of Monte-Carlo, SARSA and Q-learning. Furthermore, we will test policy gradient methods for the game of Schnapsen and compare it to the action-value methods.

7.1 Feature Vector

Assembling a feature vector is often a tricky task when training RL agents. In general, the feature vector is one of the best places to incorporate domain-knowledge into the agent. Other common ways are training the agent initially via supervised learning with human-expert-level-games or modifying the rewards (like adding immediate rewards for accumulating points). However, the incorporation of human knowledge often hurts the performance of the agent in the long run [Sut19]. *"Rather than having the agents discover the world around them like babies, researchers restricted the detail of game states, crafting them only with a subset of information they deemed relevant. It turns out that researchers often had little idea what parts of the game state AI agents considered useful when attempting to win in a game."* Therefore, most successful AIs use raw features only [SB18].

The state at a particular point in a game of Schnapsen is completely determined by the following features.

- *Played cards:* 20 binary inputs for the cards that have been already played.
- *Hand cards:* 20 binary inputs for the cards held in the hand of the player.
- *Action:* 20 binary inputs for the intended action.

- *Action Marriage:* 1 binary input that signifies whether a marriage was declared in the current trick.
- *Last card value:* 5 binary inputs for the 5 possible values that the last faceup card can have.
- *Trick points:* 2 inputs for the current trick points of the player and the opponent (each divided by 66 to normalize.)
- *No tricks:* 1 binary input: 0 if the player already has won a trick in the current deal and 1 otherwise. E.g., if the player has declared a marriage in the first trick but has not won any tricks yet, the opponent may still win 3 game points in this deal.
- *Game points:* 2 inputs for the current game points of the player and the opponent (each divided by 7 to normalize.)
- *Stock Closed:* 1 binary input that signifies whether the stock has been closed by either player.
- *Stock Closed Payoff:* 2 inputs. If the stock has not been closed yet, both are set to 0. If the stock has been closed, the inputs store the game points that each player would gain if winning the deal. As these inputs range from 0 to 3, the inputs are normalized by dividing by 3.
- *Cards left:* 1 input that signifies how many cards are left to draw. After dealing 5 cards to each player, 10 are left on the stock. To adjust the range to the other inputs, the value fed into the neural network is divided by 10.
- *Led first trick:* 1 binary input: 1 if the player has led the first trick, and 0 otherwise. This information is probably not of particular importance, but it may slightly affect whether the player will lead the next deal or not.
- *Known cards of hands:* $2 \cdot 11$ binary inputs which indicate the handcards of the opponent known by each player. There are 11 opponent cards that a player may know of: Qs and Ks due to declared marriages, all trump cards due to exchanging trumps or drawing the last face up card.
- *n last tricks:* $21 \cdot n$ binary inputs for the past n tricks. For each past trick we use a binary input per card. Exactly two of them are set to 1 to indicate the two card of which the trick consisted. One more binary input indicates if the player has won the trick (and hence which player has played which card).
- *First card (for q_2 only):* 20 binary inputs for the card led by the opponent. The bit of the corresponding card is set to 1, the others to 0.

There are some more features that may be added or that may replace some features above.

- *Unseen cards:* 20 binary inputs for the cards that may still be in the deck. This information is already implicit in the features "Hand cards", "Played cards" and "Known cards of hands" from above. It may replace "Played cards".
- *Match suit:* binary feature whether suit has to be followed. Implied by "Cards left" and "Stock closed".
- *Belief state*: mau replace "n last tricks" (see Section 7.4.1).
- *Leads next trick (for q_2 after-states only):* 1 binary input that signals whether the player will lead the next trick or not (see below).

We can reduce the size of the feature vector by identifying strategically identical states.

For example, in the case of q_2 we can make use of so called *after-states.* Instead of estimating the value of the action directly, we estimate the value of the state that results from taking the action. So for each card in the hand, we imagine playing the card and evaluate the trick – we add the corresponding trick points to both players and remove the card in the feature vector from our hand. With after-states, we can remove the 20 inputs from the *action*-segment and the 20 inputs from the *first card*-segment of the feature vector. In return, we only have to add one more input, namely whether we won the trick and therefore lead the next trick, or not. The idea of after-states is described more detailed in [SB18, Section 6.8].

For q_1, we can reduce the number of features, too. For example, if hearts are trumps, it does not matter, from a strategical point of view, whether we get dealt $AK\diamondsuit JQK\spadesuit$ or $AK\spadesuit JQK\diamondsuit$. We can take advantage of such symmetric game states by replacing the 20 *action* inputs by 6 inputs, namely 5 for each possible value and one more signaling whether the card is trump or not. Then, we have to change all other features that refer to cards, such that the first 5 out of the 20 inputs correspond to the trump cards, the next five to the suit of the "action-card" if it was not the trump suit and the last 10 inputs to the other two suits.

If we lead the trick and the stock is still open, we may also use some kind of after-states to decide whether we want to close the stock and possibly exchange trump. We compare all values of possible cards played – once for closing the stock and once for not closing, once for exchanging and once for not exchanging.

Including the last tricks is a way to address the Markov property. To have it fully satisfied, we would need to include all past tricks. However, we assume that the first trick is rather inconsequential when we are close to the end of the deal: many cards have been drawn and played since the first trick, so its effect on the card distribution of the opponent's hand will be minimal. Therefore, we assume that it is sufficient to only take into account the past two or three tricks to fully specify the current state.

Another neat way to address the Markov property is by introducing belief states. The idea is to only keep track of the probability distribution of the possible cards in the

opponent's hand by putting oneself in the shoes of the opponent and by considering what one would have done with each possible hand. We will explain this method later in more detail.

7.2 Neural Network Architecture

We call an agent better than another one, if it performs better in terms of game playing strength, that is, if it wins more often than it looses. However, we also need to ensure fair training conditions, which is not trivial. There are two main aspects to consider.

- *Elapsed time:* We give all agents the same amount of time to train. As we use a linear decay for epsilon and learning rate, we would need to estimate the number of episodes that are possible within a given time span prior to training.
- *Data efficiency:* We train all agents for a fixed number of episodes. So all agents have the same amount of "data" for training.

The time needed for each episode during training correlates with the number of parameters of the neural network. Complex neural networks need more time for each episode but at the same time also need more episodes, i.e. more data, to converge to a good strategy.

We decide to compare the performance of agents with respect to data efficiency, but also limit training time by an upper bound of approximately 15 days, on the hardware at hand.[1] Each agent is trained with 5 million episodes. In the next section, we will empirically estimate the training time for different neural network architectures.

7.2.1 Training Time

Above, we assumed that the training time grows linearly with the number of parameters. The experimental evaluation below, however, shows that doubling the parameters of the network increases the training time less than twice. We perform a grid search with the following network parameters:

- *Number of nodes per hidden layer:* 128, 256, 512,
- *Number of hidden layers:* 2, 3, 4, 5, 6, 7.

For testing, we used the semi-gradient SARSA algorithm (see Section 7.3). Table 7.1 gives estimates for the time needed to train a model with 5 million episodes, for various neural network architectures. Each row shows, approximately, a linear correlation between

[1]The experiments where run on an AMD Ryzen Threadripper 3990X 64-core processor at a clock rate of 4GHz.

nodes	layers: 2	3	4	5	6	7
128	2.18	2.66	3.13	3.80	4.19	4.36
256	4.29	6.53	8.26	10.19	11.64	13.53
512	12.20	20.85	29.94	36.75	41.31	46.35

Table 7.1: Estimated total runtime (in days) for 5 million episodes, for NN architectures with varying numbers of hidden layers and of nodes per layer. For each combination, the runtime for 50 episodes was taken 40 times in a row and the average extrapolated to 5 million episodes.

nodes	layers: 2	3	4	5	6	7
128	6.64	5.12	4.66	4.22	3.83	3.39
256	4.13	3.63	3.22	3.07	2.85	2.79
512	3.40	3.14	3.10	2.89	2.62	2.47

Table 7.2: Average computation time per parameter (in seconds) for NN architectures with varying numbers of hidden layers and of nodes per layer.

layers and total runtime. For example, for 128 nodes per layer the computation takes $1.2 + 0.5n_{\text{hl}}$ days, with n_{hl} denoting the number of layers.

The total number of parameters, n_p, grows quadratically with the number of nodes. It is given by

$$n_p = n_{\text{nodes}}^2 \cdot (n_{\text{hl}} - 1) + n_{\text{nodes}} \cdot (n_{\text{hl}} + 1) + n_{\text{features}} \cdot n_{\text{nodes}} + 1,$$

where n_{nodes} is the number of nodes per hidden layer and n_{features} is the number of input features. Therefore, the time per parameter actually decreases with a growing number of nodes. Table 7.2 shows a breakdown of computation time per parameter ($n_{\text{features}} = 90$). While the overall training time doubles when changing the architecture from 2×128 to 7×128, the training time per network parameter is actually halved. A similar decrease can be observed when increasing the number of nodes per hidden layer.

7.2.2 Performance

We start by investigating how the number of hidden layers, the number of nodes per layers and the choice of activation function affects the performance of the agent. In literature, many different settings were used and there seems to be no magical setting that works best for all use cases. TD-Gammon for example, used a network with 7 hidden layers, 80 nodes per layer and the sigmoid activation function [Tes95]. DeepMind's Atari-AI on the

other hand, used a CNN with only 3 hidden layers and the ReLU activation function [SB18].

We chose three settings with approximately the same training time:

1. 2 hidden layers with 256 nodes each,
2. 4 hidden layers with 174 nodes each,
3. 7 hidden layers with 128 nodes each.

For reference, to see if adding hidden layers or nodes per layer increases performance, we also added a less complex agent:

4. 2 hidden layers with 128 nodes each.

Again, we used the semi-gradient SARSA algorithm for this experiment (Section 7.3). For each setting, we trained the network once with the ReLU activation function and once with the sigmoid activation function.

Figure 7.1 compares the performance of agents using the two activation functions. The first interesting thing to note is that the deep agent with 7 hidden layers and the sigmoid activation function does not seem to learn at all – throughout the training process it hardly ever won against the ReLU version. A possible explanation for this behavior is the vanishing gradient problem [RSW20]. Also for all other settings, the ReLU activation function performed better than the sigmoid activation function – the former had a winning probability between 52% and 60%. As a consequence of this analysis, we will use the ReLU activation function for subsequent experiments, even though, for other settings, the sigmoid activation function may perform better.

As a next step, we let the fully trained ReLU-agents play against each other to compare their performances. The winning rates are shown in Table 7.3. Overall, the more complex neural network architectures outperform the simpler ones. The agent performing best is the 4×174-agent, winning most often both matches against other. Hence, it seems that increasing the number of hidden layers as well as the number of nodes per layer both are important.

7.3 Update Target

As seen in Chapter 6, we can choose between various update targets for the neural network during training. We again focus on the following three:

1. SARSA: The estimated value of the next action is used to update the estimated value of the action before.

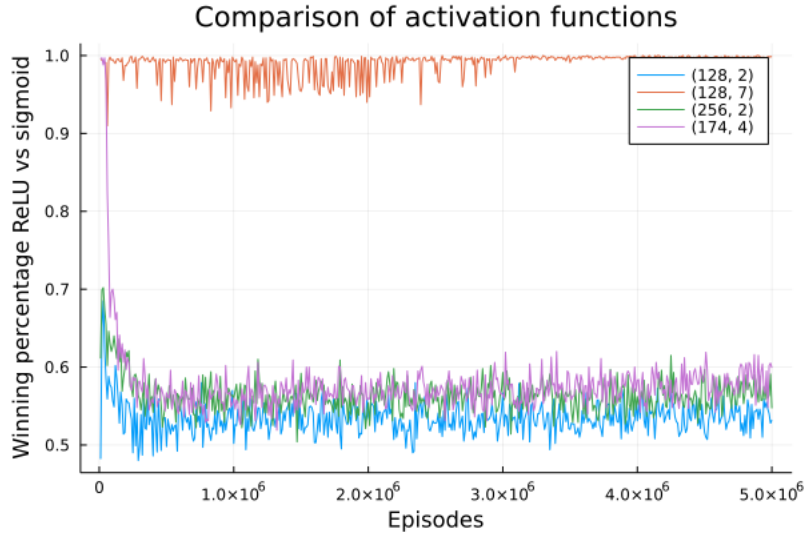


Figure 7.1: Comparison of the learning processes for the ReLU activation function and the sigmoid activation function for various neural network architectures $n_{\text{hl}}{\times}n_{\text{nodes}}$. Every 10 000 episodes the two agents played a match of 1000 games and the winning rate from the view of the ReLU-agent is plotted.

	2×128	7×128	2×256	4×174
2×128	0.5069	0.4784	0.4785	0.4666
7×128	0.5202	0.4930	0.4798	0.4801
2×256	0.5389	0.5093	0.5011	0.4862
4×174	0.5427	0.5135	0.5038	0.4996

Table 7.3: Comparison of various neural network architectures $n_{\text{hl}}{\times}n_{\text{nodes}}$, varying the number of hidden layers, n_{hl}, and the number of nodes per layer, n_{nodes}. Hyperparameter settings: $\epsilon_0 = 0.2$, $\alpha_0 = 0.1$, activation function ReLU. Each agent was trained for 5 million episodes. The entries are the rates of the row-element winning against the column-element in a match of 10 000 games.

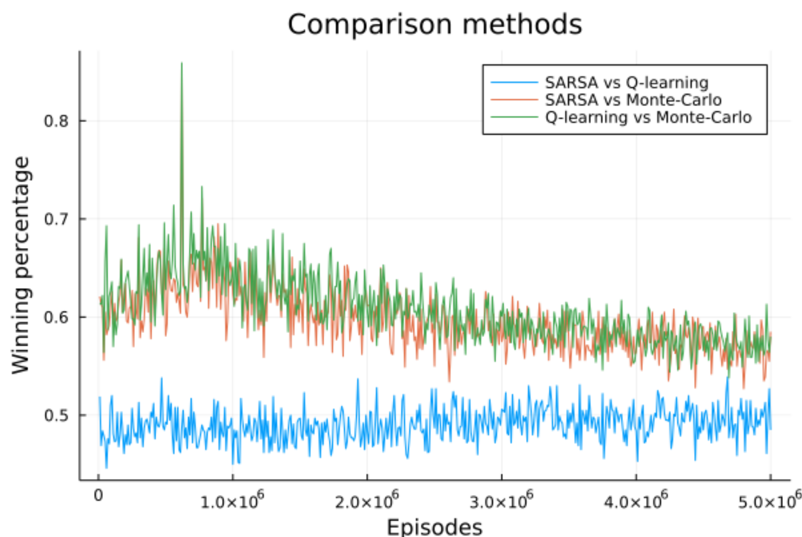


Figure 7.2: Comparison of the learning processes of SARSA, Q-learning and Monte-Carlo. Every 10 000 episodes, the agents play a match of 1000 games against each other.

	SARSA	Q-learning	Monte-Carlo
SARSA	0.5049	0.4959	0.5717
Q-learning	0.4967	0.4996	0.5788
Monte-Carlo	0.4324	0.4256	0.4978

Table 7.4: Comparison of the fully trained SARSA, Q-learning and Monte-Carlo agents (5 million episodes). The entries are the rates of the row-element winning against the column-element in a match of 10 000 games.

2. Q-learning: The highest estimated value of all possible next actions is used to update the estimated value of the action before.
3. Monte-Carlo: The updates of all actions are postponed until the end of the episode and the actual final outcome of the episode is used as the update target.

Figure 7.2 shows the training process of these three methods. We see that both SARSA and Q-learning clearly outperform the Monte-Carlo method. The performance difference between SARSA and Q-learning, on the other hand, is not significant.

In Table 7.4, we compared the performances of the fully trained versions of these agents. SARSA and Q-learning have a similar performance, while both outperform Monte-Carlo.

Later, we will also take into account actor-critic methods. However, hyper-parameters like epsilon and the learning rate cannot be transferred as easily and hence need an individual analysis first.

As SARSA is slightly more efficient than the Q-learning algorithm while yielding the same performance in our experiment, we continue to use the SARSA learning algorithm for investigating the effect of other hyperparameters, like different sets of input features.

7.4 Input Features and Markov Property

We tested various combinations of input features, among them:

- Inclusion of no past tricks, of only the last trick, or of the last two/three/four/five tricks. We tried different ways to represent past tricks: 21 binary input features for each trick (as described above) as well as 41 binary input features (distinguishing the card played by the agent and the card played by the opponent).
- Including and excluding redundant feature segments like *cards played*, *unseen cards*, *match suit*, etc.

All of these feature vectors resulted in agents with similar performances. This is quite surprising, as we expected that including past tricks would improve the performance, as it addresses the Markov Property. When playing Schnapsen, one often bases the decision which card to play on the past trick. For example, if the leading player plays the Jack of clubs and the other player does not take the trick with a higher club, it makes it less likely that the opponent has a higher club in his hand. Therefore, the leading player may be inclined to play another club in the following trick.

We present several reasons, why the inclusion of previous tricks does not improve performance.

- Past tricks actually do not matter, contrary to our intuition.
- The other input features (like *played cards* and *trick points*) already give enough information about the past tricks.
- The agent is not able to properly convert the raw information of the past tricks.

To address the last point, we try another way to take care of the Markov property, by introducing the concept of *belief states* [RPK99]. The idea is to approximate the card distribution in the opponent's hand throughout the game and feed this information into the neural network. The best AIs for the game of Poker use a similar concept, referred to as *counterfactual values* [MSB+17, BS18]. The following section describes our approach of estimating the belief state. Note, however, that these calculations are computationally very expensive and hence significantly increase the training time of an agent.

7.4.1 Belief State

The feature vector segment *belief state* consists of 20 input features (one for each card in the deck) ranging from 0 to 1 each. We will denote these input features by $\{p(c_i)\}_{i=1,\dots,20}$, where $p(c_i)$ approximates the probability of the card c_i being in the opponent's hand. The cards appear in arbitrary order.

At the beginning of a deal, each player gets dealt five cards and one card is face-up beneath the deck, determining the trump. Therefore, each player sees six cards – each of the other 14 unseen cards are equally likely to be in the opponent's hand. Hence, we can initialize these probabilities as

$$p(c_i) = \begin{cases} \frac{5}{14} & \text{if } c_i \text{ is unseen,} \\ 0 & \text{else.} \end{cases}$$

For the remainder of the game, the basic idea is, whenever the opponent plays a card, to put ourselves in the shoes of our opponent, and to calculate, for each possible combination of hand cards, the probability that we would have played that card.

For example, suppose the opponent leads the first trick with c_j without neither exchanging trump, closing the stock nor declaring a marriage before the trick. That means the opponent got initially dealt c_j and four cards out of the remaining 13 unseen cards. Hence, there are $\binom{13}{4} = 715$ possible hands. We will denote the combinations as $\{comb_i\}_{i=1,\dots,715}$. For every combination, we calculate the probability $p(c_j \mid comb_i)$ that we would neither have exchanged the trump, closed the stock nor declared a marriage (in case c_j corresponds to a queen or king) before the trick and then led the card c_j. For the first trick, all hand combinations are equally likely. In later tricks, however, the probabilities will vary, as we have already gathered information about the card distribution in our opponent's hand. Therefore, we weigh $p(c_j \mid comb_i)$ by multiplying it with $\prod_{c_k \in comb_i} p_{\text{old}}(c_k)$, where $\{p(c_i)\}_{i=1,\dots,20}$ represents the latest approximated belief state before the current trick. We denote this weighted product

$$p^w_{comb_i} = p(c_j \mid comb_i) \cdot \prod_{c_k \in comb_i} p_{\text{old}}(c_k).$$

Now for every card c_k, we sum up all $p^w_{comb_i}$, where $c_k \in comb_i$, and denote the sum by

$$l(c_k) = \sum_{\substack{comb_i \\ c_k \in comb_i}} p^w_{comb_i}$$

$l(c_k)$ is approximately proportional to the likelihood that c_k is in the hand of our opponent.

Now, we update the card distribution of the hand of our opponent as follows. First, for all cards that we know to be in the hand of our opponent (e.g. because of exchanging

trump or declaring a marriage) we set the probability to 1. We denote the number of these cards as n_{known}. For all cards that we know not to be in the hand of our opponent (e.g. because they have been already played or are in our own hand) we set the probability to 0. All other cards represent the set of potential hand cards of the opponent's hand, denoted by C_{pot}.

We want the probabilities to sum up to n_{hand}, the number of cards in our opponent's hand. We achieve this by setting the remaining probabilities to

$$p(c_i) = (n_{\text{hand}} - n_{\text{known}}) \frac{l(c_i)}{\sum_{c_j \in C_{\text{pot}}} l(c_j)} \quad \text{for all } c_i \in C_{\text{pot}}.$$

If the opponent does not lead the trick, we follow the same procedure after he played the second card.

After drawing, we also have to adjust the belief state. For the card that we draw ourselves, we set the corresponding probability in the opponents hand distribution to 0. We now want to update the other probabilities, such that they again sum up to the number of cards in our opponents hand. Also, unseen cards that have currently a lower estimated probability in our opponents hand distribution, have a higher probability to be drawn by our opponent (as that means the probability is higher that they are still in the draw pile). We update the probabilities as

$$p(c_i) = p_{\text{old}}(c_i) + (1 - p_{\text{old}}(c_i)) \frac{n_{\text{hand}} - \sum_j p_{\text{old}}(c_j)}{\sum_{c_j \in C_{\text{pot}}} (1 - p_{\text{old}}(c_j))} \quad \text{for all } c_i \in C_{\text{pot}}.$$

Note that this version of belief states does not take into account covariances between cards as this would make the computations even more complex and time consuming.

In Figure 7.3 we compare an agent that uses belief states for training to one that does not. We see that the former outperforms the latter, winning about 52% to 53% throughout the entire training process. Hence, it seems that the agent can process the information of past tricks better with belief states than with the raw input of past tricks.

7.5 Actor Critic

So far we have only looked at *action-value methods* – the agent learned a function that estimates the probability of winning for each state action pair and selects the action with the highest probability in a given state. Now, we will consider *policy-gradient methods*, which instead learn a parameterized policy that can select actions without consulting a value function. In particular, we will look at the well known *actor-critic method*, which actually uses a value function to learn the policy parameters, but the value function is not required for action selection.

One key advantage of policy gradient methods is that they are able to learn mixed strategies with arbitrary probabilities. In imperfect information games, it is often

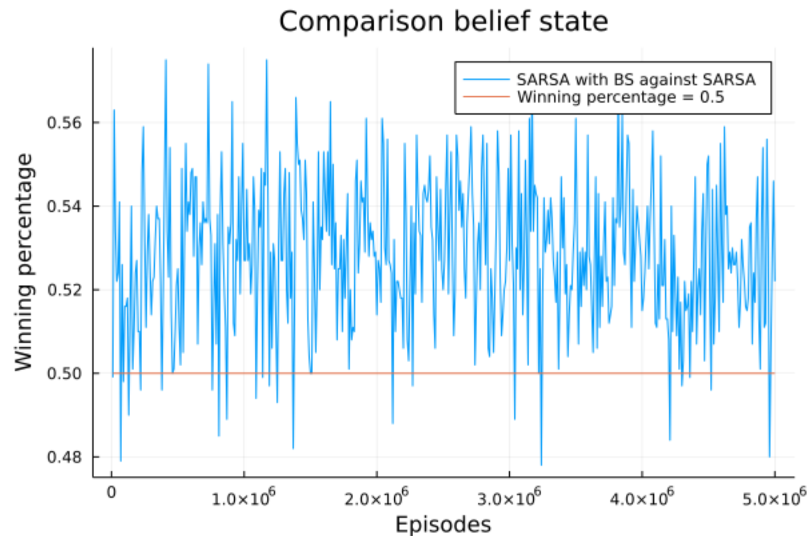


Figure 7.3: Comparison of learning process between the SARSA agent that computes belief states and the standard SARSA agent. Every 10 000 episodes the agents play a match of 1000 games against each other.

important to use mixed strategies, so that the opponent cannot make as many assumptions of one's private information (in the case of Schnapsen, the cards that you hold in your hand).

For a given state s and parameterization θ, we calculate a *numerical preference* $h(s, a, \theta)$ for each action a. Concrete probabilities are then calculated by applying the *soft-max* function to these numerical preferences, yielding the policy

$$\pi(a \mid s, \theta) = \frac{e^{h(s,a,\theta)}}{\sum_b e^{h(s,b,\theta)}}.$$

The action preference function $h(s, a, \theta)$ can be parameterized arbitrarily – we use once more a deep neural network.

As mentioned in Chapter 4, we also train a *critic*, a state value approximation $v(s)$, as we did with methods preceding this chapter. This critic network helps the actor to converge faster.

7.5.1 Results

Experiment Setup

We conduct a grid search over the following hyper-parameter search space:

- Learning rate of the critic network α_v: 0.001, 0.002, 0.004, 0.008, 0.016.

- Learning rate of the actor network α_θ: 0.001, 0.002, 0.004, 0.008, 0.016, 0.032.
- α_v-decay: true, false.
- α_θ-decay: true, false.
- Activation function for the neural networks: ReLU, Sigmoid.

For the size of the neural networks, we again use 4 hidden layers with 174 nodes each.

Conclusions

1. As with the action-value methods, the ReLU activation function outperforms the sigmoid activation function.
2. The other parameters have hardly any impact on the performance.
3. Best setting: $\alpha_v = 0.008$, $\alpha_\theta = 0.016$, no α_v-decay but with α_θ-decay, and as mentioned, with the ReLU activation function.

In Figure 7.4, we can see the learning curve of the actor-critic method (using the best setting as specified above). Clearly, the agent is improving, especially during the first one million episodes of training. Afterwards, however, only minimal improvements follow – the fully trained agent wins against a less trained version with one million trained episodes only about 52% of the times. When comparing the actor-critic agent to the SARSA agent, the actor-critic agent only wins about 2% of the games. Hence, it seems like the way we implemented the actor-critic method is not suitable for the game of Schnapsen, as the agent is not able to find a parameterization for a good strategy.

Challenges

First of all, the actor-critic method has some more hyper-parameters, which makes it harder to tune.

Secondly, in contrast to action-value methods, the policy network in policy-gradient methods usually has multiple outputs – one for each action. However, in the game of Schnapsen, the legal action space varies: In the beginning of the game one has five possible cards to choose from, while in the end of the game there are less than five cards left in the hand. Additionally, even if the number of actions would be constant, in each state the possible actions refer to different cards held in the hand. We tried to address this issues by having 20 outputs – one for each card in the deck. Then, for calculating the respective action probabilities for each card in the hand, we applied the softmax function to only those values that correspond to legal actions. Also, during training, we only updated the network with respect to the possible actions.

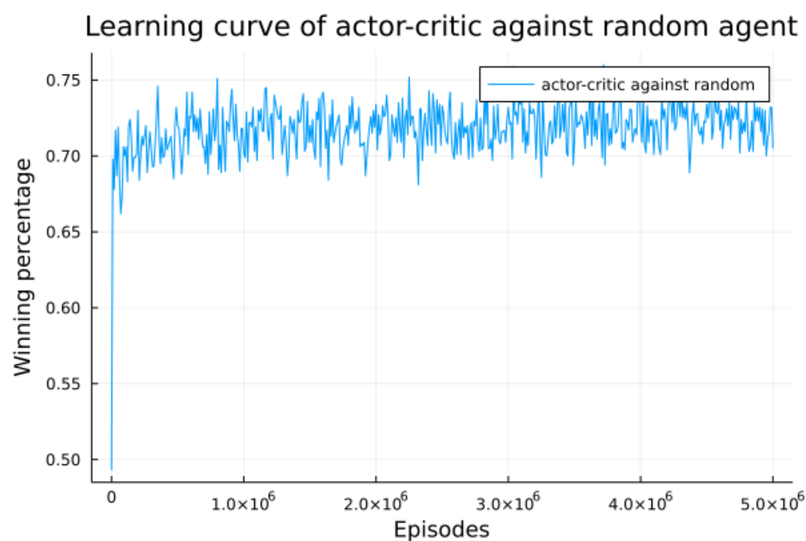
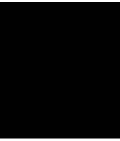


Figure 7.4: Learning curve of the actor-critic method. Every 10000 episodes the trained agent played a match of 1000 games against the initial agent before any learning (hence, a random agent) and the winning rate is plotted.

Thirdly, there is no obvious way to make use of after-states. We use only one feature vector to derive the action probabilities – for after-states we would need a feature vector for each possible action that describes the state after the action is taken.

Last but not least, sometimes it is just easier to estimate the state values, sometimes it is easier to estimate the policy parameterization. There are no clear rules to predict which one is going to work better in foresight.

CHAPTER 8



Implementation

Our implementation uses the programming language *Julia* for the environment and the RL algorithms and consists of about 4000 lines of code. For the neural networks, we use the package *Flux*. The code is structured as follows:

- *Helper modules* are used by all learning algorithms and implement constructors (for the players, deck, etc.), game rules, and functions for various game mechanics (updating points, drawing cards, etc.).
- *Learning algorithms:* Each RL method is implemented in a separate file, which depends on the helper modules. Each algorithm comes with a separate action file, as the logic of choosing an action may differ between algorithms.
- *Configuration files* allow us to specify, for example, the feature vector representation by setting feature segments to either true or false.

CHAPTER 9

Evaluation against Doktor Schnaps

Doktor Schnaps emerged during the doctoral thesis of Florian Wisser also at TU Wien. As mentioned in Chapter 5, Doktor Schnaps uses a fundamentally different approach and plays above human expert level. According to the book by Tompa on Schnapsen strategies [Tom15], it is the strongest published bot for the game of Schnapsen.

- *Speed of decision making:* Once the neural network of the RL-agent has been trained, decisions are calculated instantly, taking fractions of a second. Doktor Schnaps, on the other hand, which has to simulate hundreds of scenarios in a given state, takes up to 10 seconds per action.
- *Playing strength:* Unfortunately, we were not able to let the machines play automatically against each other. For each state, we had to feed the agent the current state by hand and execute the action of the RL-agent in the Doktor Schnaps interface. Therefore, evaluating these two bots against each other is time consuming, and it is hard to get a significant result as Schnapsen involves randomness.

  After 20 games, the score is 12:8 in favor of the RL-agent. The agent uses the SARSA algorithm with belief states, as it performed best in our comparisons.

CHAPTER 10

Conclusion

10.1 Summary

We started by comparing different RL-algorithms and mentioned some of their advantages and disadvantages in general and also with respect to imperfect information games. Monte-Carlo methods are interesting for the game of Schnapsen as they are often more robust if the Markov property is not satisfied. Temporal difference learning like SARSA and Q-learning perform well for a variety of problems and hence worth testing. Policy-gradient algorithms like actor-critic may be very suitable for imperfect information games as they use mixed strategies by design.

After implementing the environment consisting of all the rules of Schnapsen and for each state the set of legal actions as well as the different RL-algorithms, we tuned some of the hyper-parameters while setting others according to best practices and intuition. It turned out that complex neural networks outperformed simpler ones, the ReLU activation function outperformed the sigmoid activation function and regarding the feature vector representation, we improved performance by using the concept of after-states and we incorporated information about past tricks with the help of belief states.

In the end, we saw that temporal difference methods resulted in higher playing strength than the other methods and among them we chose the SARSA as the best suited algorithm for the game of Schnapsen as it is computationally slightly more efficient than Q-learning. Finally, we played some games against the best published Schnapsen bot that currently exists, named Doktor Schnaps, which emerged during the doctoral thesis of Florian Wisser at TU Wien. Our SARSA agent was able to play on a similar level as Doktor Schnaps, possibly even higher. However, we were not able to produce a significant winning result as playing these matches is very time consuming as we were not able to let these machines play against each other directly. Furthermore, the RL-agents outperformed Doktor Schnaps with respect to speed of decision making.

10.2 Future work

First of all, to get more insight which RL algorithms work best for imperfect information games in general, we would need to test these algorithms on a variety of imperfect information games which would exceed the scope of this master thesis.

Secondly, as the search space is basically indefinite, we could try out many more settings for each of the algorithms to try to improve performance. Furthermore, it would be interesting to compare our method of self-playing and use of belief state with the architecture explained in the paper [BBLG20].

One may also experiment with using reinforcement learning for the earlier game states of a the game when there is a lot of imperfect information present due to the many cards that are left to be drawn and use a similar approach to Doktor Schnaps for later states of the game.

List of Figures

3.1 Extensive game form for the coin toss game 13

4.1 Cycle of reinforcement learning [SB18, p.54] 15

6.1 Comparison of three Monte-Carlo agents with initial ϵ_0 set to 0.2, 0.6 and 1.0, respectively. The agents' performance as tested every 10 000 episodes of training by playing a match of 1000 games against a random agent. . . . 28
6.2 Comparison of four SARSA agents with the initial ϵ_0 set to 1 and various settings for the learning rate: $\alpha = 0.1$ and $\alpha = 0.2$ without decay as well as $\alpha = 0.7$ and $\alpha = 1.0$ with decay. The agents' performance was determined every 10 000 episodes by playing a match of 1000 games against a random agent. 28
6.3 Comparison of the learning curves of the three approximation methods Monte-Carlo, SARSA, Q-learning. The agents' performances were tested every 10 000 episodes of training by playing a match of 1000 games against a random agent. 33
6.4 Comparison of the tabular methods vs. the approximation versions of the methods. Every 10 000 episodes the two agents played a match of 1000 games and the winning rate from the view of the tabular version is plotted. . . . 34

7.1 Comparison of the learning processes for the ReLU activation function and the sigmoid activation function for various neural network architectures $n_{\text{hl}} \times n_{\text{nodes}}$. Every 10 000 episodes the two agents played a match of 1000 games and the winning rate from the view of the ReLU-agent is plotted. 41
7.2 Comparison of the learning processes of SARSA, Q-learning and Monte-Carlo. Every 10 000 episodes, the agents play a match of 1000 games against each other. 42
7.3 Comparison of learning process between the SARSA agent that computes belief states and the standard SARSA agent. Every 10 000 episodes the agents play a match of 1000 games against each other. 46
7.4 Learning curve of the actor-critic method. Every 10000 episodes the trained agent played a match of 1000 games against the initial agent before any learning (hence, a random agent) and the winning rate is plotted. 48

List of Tables

6.1 Comparison of the tabular methods Monte-Carlo, SARSA and Q-learning. For learning, ϵ_0 was set to 1 and the learning rate of SARSA and Q-learning, α_0, was set to 0.7. All agents were trained for $5 \cdot 10^6$ episodes. Both, ϵ and α, decayed over time. The entries give the rate of the method in the left column winning against the method in the top row in a match of 10 000 games. E.g., the entry in the third row and first column tells us that the Q-learning agent won 51.5% of the games against the Monte Carlo agent. 30
6.2 Overview of some parameters to tune and architectures to choose. 32
6.3 Comparison of the approximation methods Monte-Carlo, SARSA and Q-learning. The entries are the rates of the agent in the left column winning against the agent in the top row in a match of 10 000 games. 33

7.1 Estimated total runtime (in days) for 5 million episodes, for NN architectures with varying numbers of hidden layers and of nodes per layer. For each combination, the runtime for 50 episodes was taken 40 times in a row and the average extrapolated to 5 million episodes. 39
7.2 Average computation time per parameter (in seconds) for NN architectures with varying numbers of hidden layers and of nodes per layer. 39
7.3 Comparison of various neural network architectures $n_{\text{hl}} \times n_{\text{nodes}}$, varying the number of hidden layers, n_{hl}, and the number of nodes per layer, n_{nodes}. Hyperparameter settings: $\epsilon_0 = 0.2$, $\alpha_0 = 0.1$, activation function ReLU. Each agent was trained for 5 million episodes. The entries are the rates of the row-element winning against the column-element in a match of 10 000 games. 41
7.4 Comparison of the fully trained SARSA, Q-learning and Monte-Carlo agents (5 million episodes). The entries are the rates of the row-element winning against the column-element in a match of 10 000 games. 42

List of Algorithms

6.1 Monte Carlo Self Play 27

6.2 SARSA Self Play 29

6.3 Q-learning Self Play 30

Bibliography

[BBLG20] Noam Brown, Anton Bakhtin, Adam Lerer, and Qucheng Gong. Combining deep reinforcement learning and search for imperfect-information games. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[BS18] Noam Brown and Tuomas Sandholm. Superhuman ai for heads-up no-limit poker: Libratus beats top professionals. *Science*, 359(6374):418–424, 2018.

[Cor15] Gottlieb Siegmund Corvinus. *Nutzbares, Galantes, und Curiöses Frauenzimmer-Lexicon.* Leipzig, 1715.

[HDG+19] Daniel Hernandez, Kevin Denamganaï, Yuan Gao, Peter York, Sam Devlin, Spyridon Samothrakis, and James Alfred Walker. A generalized framework for self-play training. In *2019 IEEE Conference on Games (CoG)*, pages 1–8, 2019.

[Huy57] Christiaan Huygens. *De Ratiociniis in Ludo Aleae (On Reasoning in Games of Chance).* Franz van Schooten, 1657.

[KC07] Barbara Ann Kitchenham and Stuart Charters. Guidelines for performing systematic literature reviews in software engineering. Technical Report EBSE 2007-001, Keele University and Durham University Joint Report, 07 2007.

[Mit97] Tom M. Mitchell. *Machine learning, International Edition.* McGraw-Hill Series in Computer Science. McGraw-Hill, 1997.

[MSB+17] Matej Moravcík, Martin Schmid, Neil Burch, Viliam Lisý, Dustin Morrill, Nolan Bard, Trevor Davis, Kevin Waugh, Michael Johanson, and Michael H. Bowling. Deepstack: Expert-level artificial intelligence in no-limit poker. *CoRR*, abs/1701.01724, 2017.

[RPK99] Andres Rodriguez, Ronald Parr, and Daphne Koller. Reinforcement learning using approximate belief states. In S. Solla, T. Leen, and K. Müller, editors, *Advances in Neural Information Processing Systems*, volume 12. MIT Press, 1999.

[RSW20] Matías Roodschild, Jorge Gotay Sardiñas, and Adrián Will. A new approach for the vanishing gradient problem on sigmoid activation. *Prog. Artif. Intell.*, 9:351–360, 2020.

[SB18] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, second edition, 2018.

[SHM+16] David Silver, Aja Huang, Christopher Maddison, Arthur Guez, Laurent Sifre, George Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. Mastering the game of go with deep neural networks and tree search. *Nature*, 529:484–489, 01 2016.

[SSS+17] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, Yutian Chen, Timothy P. Lillicrap, Fan Hui, Laurent Sifre, George van den Driessche, Thore Graepel, and Demis Hassabis. Mastering the game of go without human knowledge. *Nat.*, 550(7676):354–359, 2017.

[Sut19] Richard Sutton. *The Bitter Lesson*. `http://www.incompleteideas.net/IncIdeas/BitterLesson.html`, 2019. Visited July 4, 2022.

[Tes95] Gerald Tesauro. Temporal difference learning and td-gammon. *Commun. ACM*, 38(3), 1995.

[Tom15] Martin Tompa. *Winning Schnapsen – From Card Play Basics to Expert Strategy*. CreateSpace Independent Publishing Platform, 2015.

[vNM44] John von Neumann and Oskar Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, 1944.

[Wis10] Florian Wisser. Creating possible worlds using sims tables for the imperfect information card game schnapsen. In *2010 22nd IEEE International Conference on Tools with Artificial Intelligence*, volume 2, pages 7–10, 2010.

[Wis15] Florian M. Wisser. An expert-level card playing agent based on a variant of perfect information monte carlo sampling. In *IJCAI*, 2015.

[Wis16] Florian M. Wisser. Evaluating the performance of presumed payoff perfect information monte carlo sampling against optimal strategies. In *AAAI Workshop: Computer Poker and Imperfect Information Games*, 2016.