Understanding reinforcement learning with *6nimmt!*

Group Project

Fabio Bertschi and Jean Mégret

`fabibert@ethz.ch | megretj@ethz.ch`

Distributed Computing Group
Computer Engineering and Networks Laboratory
ETH Zürich

Supervisors:
Béni Egressi
Prof. Dr. Roger Wattenhofer

July 18, 2022

Acknowledgements

We sincerely thank Beni Egressy for his support, giving us inspiration for new ideas during the weekly meetings. We also thank Johann Brehmer for initially creating the Repository containing the framework and some bots and also for getting in contact with us. This report took inspiration on the work of Peter Müller about a Tichu Bot(1)

Abstract

Inspired by recent breakthroughs in reinforcement learning especially for game environments and motivated by the wish to understand more about theses methods, we undertook the task of trying to implement such methods for the game of *6nimmt!*. As it is an imperfect information game, with a huge state-action space and with a varying number of players this is a very challenging test bed. We first reviewed multiple RL methods, then on one side we tried to extract the playing characteristics of the previously best available bot for this game and on the other side to implement new bot architectures. Finally, using this knowledge, we beat the previous best bot in three different ways.

Contents

Chapter 1

Introduction

Games are a classic setting for the development of (machine learning) algorithms. Since the dawn of computing, it has been an objective to beat the worlds best players in many well known games, most notably chess. Recently, with the rise of machine learning and more widely available computing power, we have seen a great increase in level as researchers implemented powerful reinforcement learning (RL) algorithms(2). Today, state of the art bots don't even need to be told the rules of a game to yield excellent results (3). This also lead to better understanding of the games as for example today's top chess players all use computers for their preparation.

Still, there lies a margin for improvement and some games even remain a big challenge. Indeed, partial information games, very stochastic games or collaborative games continue to oppose a certain resistance (see "The Hanabi challenge" (4)). A great example of this type of games is the card game of *6nimmt!*. As a less popular game, it hasn't been investigated much and the question of whether there is an optimal policy remains.

Therefore, this project has the objective of understanding more about how state of the art reinforcement learning algorithms manage to be so successful, how they can be implemented for the very challenging game of *6nimmt* and see whether this can help to understand this game better.

Chapter 2

Background

2.1 *6nimmt!*

6nimmt! is a popular multiplayer card game designed by Wolfgang Kramer in 1994. The goal of the game is to collect as few penalty points as possible. The game can be played by two to ten players. For the propose of this project, we decided to focus on 5 players; this will simplify the setting as different number of players might lead to different challenges and strategies . We will call the player of interest the agent and all the other players the opponents.

The game (5) consists of 104 cards. Each card has a unique value (ranging from 1 to 104) as well as a number of penalty points (a.k.a bullheads) associated with it. After shuffling the cards, 10 cards are dealt out to each player. Then, four cards are placed into four rows face up in the middle of the table from the leftover cards. Each of these cards is the first card in a row. Each row can only hold 5 cards, including the first card. A typical initial game situation can be seen in Figure 2.1.

There are no *individual turns* in this game, all players play synchronously in what we will call a playing turn. Each player takes one card from their hand and puts it face down in front of them. The cards are turned over after all players have made their choice. The player who played the card with the lowest value adds their card to one of the four rows, followed by the player with the second lowest card, etc. until all the cards played have been added to rows. Players repeat this process 9 more times until all 10 cards in every player's hand have been played. The players do not get to choose the row to place their card, each card only fits into one row, following two rules:

1. **Ascending Order:** The values of the cards in each row must always increase from left to right.

2. **Least Difference:** A card must always be played in the row that ends with the card that has the lowest value.

For example, in Figure 2.1 the card 89 would be placed in the second position

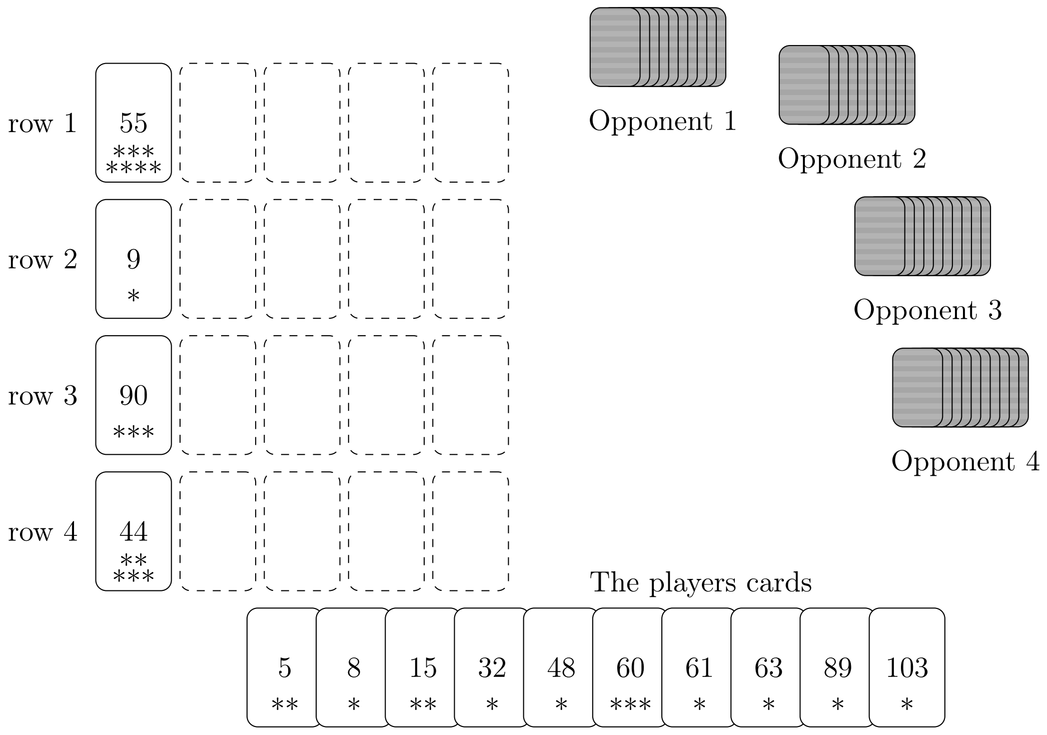


Figure 2.1: Example initial game situation

of the first row.

By the first rule, a card might not fit into any of the rows (i.e. the card is smaller than all of the rows highest cards, e.g. card 8 in Figure 2.1). In that case, the player must take all the cards of a row of their choice and play their card as the first card of that row. To simplify the game, we choose to alter this rule slightly. In our implementation, the player will automatically choose to replace the row with the lowest number of penalty points. This is the only deviation from the standard rules of the game.

A row is full when it has 5 cards. By the second rule, if a sixth card must be played on a row, the player who played this card must first pick up all five cards in that row, then play their card as the first card in that new row.

The round is over when all players have played their 10 cards. Each player then picks up their Bull Pile and counts the number of bullheads. Whoever has the least number of penalty points wins the round. Point can be added up over multiple rounds to determine the overall winner.

The cards have a number of penalty points depending on their value. All cards with values ending with five (5,15,25,...) have 2 bullheads, multiples of ten (10,20,30,...) have 3 bullheads, doublets (11,22,33,...) have 5 bullheads. Finally 55 is a doublet and ends with a 5 so it has 7 bullheads.

2.1.1 Challenges

Throughout this project, we noticed several challenges with respect to the game of 6nimmt!:

- **Imperfect information:** The game-state of 6nimmt! is only partially observable, which means the cards of the other players are hidden
- **Varying number of players:** As the number of players increase, the chances of collecting points becomes higher. This also leads to different strategies and makes it difficult for an agent to perform consistently. To help with this issue, we decided to focus solely on 5 players. This means all the following tests were performed with 5 agents, unless stated otherwise.
- **Evaluation of agent level:** This game inherently relies on a large amount of stochasticity, from the 15 novemdecillion (10^{60}) possible initial states in a game of 5 players, which, upon other things, means that the results of games are only consistent on a large amount of games.

2.2 Reinforcement learning

In the following, we will present a few basic reinforcement learning (RL) algorithms superficially, that will come in useful to understand the rest of this report, where we will detail how these are implemented in the context of the game of 6nimmt!

2.2.1 DQN

The basic principle of this learning algorithm comes from the very well known Q-learning algorithm which is an model-free reinforcement learning algorithm that seeks to learn a policy that maximises the total reward. The idea is to build a $Q : S \times A \to \mathbb{R}$ matrix whose elements $Q(s_i, a_j)$ represent the expected reward if we are in the state s_i and take action a_i. Learning is done by walking through the state-action space more or less randomly (exploration vs exploitation trade-off). The values of Q are updated by a weighted sum between the current value of Q, the reward obtained by taking action a_j and the projected future reward after a_j was taken. If the values of Q converge through training, the policy will simply be the action that maximises the reward in a certain state: $p(s_i) = \max_{a_j}(Q(s_i, a_j)$. However, if the Q Matrix is large, it will take very long until we have to go through all the states multiple times. This is where DQN, which stands form Deep-Q-Neural-Network, kicks in. Instead of storing every state-action pair in a matrix, a deep neural network is used to approximate $Q(s_i, a_j)$. The state and action are use as input of the network and the output is a reward. The network weights are updated via back propagation of the loss between the predicted reward and the actual reward.

2.2.2 Monte Carlo Method

A reinforcement learning agent based on the Monte Carlo method conducts game simulations by playing randomly until it reaches the end of an episode at which point a return value is stored. This is repeated enough times so that the average return from some game situation represents the true expected return from this game state(6).

2.2.3 Monte Carlo Tree Search

Like in the Monte Carlo method, the value of a state-action pair is estimated as the average of the returns from this pair. We do multiple simulations starting at some game state to predict the return, this state is mostly the initial game situation. However, we now use a more sophisticated way of choosing our moves than just randomly. We choose with one of two strategies either using the tree policy or the roll-out policy. The tree policy is used to navigate in game states which we have already seen, i.e. added to the tree. We call this set of states tree, because the states which we visit can be seen as nodes of a more and more explored game tree. The tree policy guides us through the tree nodes using criteria like previously encountered returns, number of visits to a certain state and other parameters. With every simulation, we add a new state to the tree starting with only the initial state. If we have used the tree policy to get to a state which has moves which are not yet tried out and thus not in the tree we choose one of them and use the roll-out policy from here on. The roll-out policy is a less sophisticated way of playing then the tree policy, because it is meant to give a fast approximation of the value of a given state, to know where to expand the tree and move with more sophisticated moves.

2.2.4 AlphaZero

The previous Section 2.2.3 was only a high level introduction to MCTS, since the tree policy and roll-out policy can be implemented in many different ways. Now we will look at one specific and very well working variation, proposed by Silver et al. in 2017 (2) for the games of Chess, Go and Shogi: the AlphaZero reinforcement learning algorithm.

The AlphaZero algorithm consists of multiple parts. First a double headed actor-critic neural network which learns the move probabilities P(s,a) and the return V(s) from playing real games. It also uses a separate data structure to store Q(s,a) the average achieved return for a given state s and action a. Finally, the tree policy is evaluated using Equation (2.1). The action which maximises this equation in a given state will be chosen as the next action to simulate. If the next node is already in the tree we again apply the tree policy again. If we have not seen the node yet, i.e. it is not in the tree, we apply the roll-out policy to get a fast estimate of the return value of this state. Here, the roll-out policy is the evaluation of the critic neural network which estimates the return value. Once the number of simulation is reached, the child with most visits will be chosen as the one with most visits count. When the end of a game is reached, the return value is recursively back-propagated and stored in the Q data structures corresponding to the states we passed through.

$$max(Q(s,a)+U(s,a)) \texttt{ with } U(s,a) \; \alpha \; \frac{P(s,a)}{1+N(s,a)} \tag{2.1}$$

Figure 2.2: Where:
$N(s,a)$: visit count of an state-action pair
$Q(s,a)$: average return value for a given state-action pair
$P(s,a)$: prediction of the probability for an action to yield highest return amongst all actions from state s

Chapter 3

Rl for 6nimmt!

3.1 Available framework

For this work, we based ourselves on a repository of Brehmer (7). This project is based on the OpenAI gym environment which is a toolkit for developing reinforcement learning algorithms (8). It has two basic components, the environment, which models the game, and the agents which are the instances who play the game. The environment API lets us get information about the game state and the actions taken.

The way the game is modelled is slightly simplified from the original game, when playing a lower card than the last card on all rows, the agent cannot freely choose which stack to replace, but instead will always take the stack with the smallest number of penalty points. Although this could potentially make a difference in the level of the player, we believe it doesn't change the game dynamics very much as it is very unlikely a player will picking a row with many points will turn out to be a good thing.

This project implements easy APIs that let different agents play against each other and evaluate they relative performance. Especially, a tournament can be created where a set of agents can play a predefined number of games with some number of players. Then, for each game, the number of penalty points of every player is recorded. An elo system is also built-in to track the evolution of the players. This leads us to a crucial point of this work and the question on how we went about to evaluate the level of the individual agents.

In addition to the game framework, Brehmer implemented a few bots that we will later discuss in Section 3.2.

Evaluating the level of the agents

Very quickly, it became apparent to us that being able to measure the level of an agent was very challenging. As mentioned before, the level of sochasticity is immense and in order to have a good idea of the relative level of different agents,

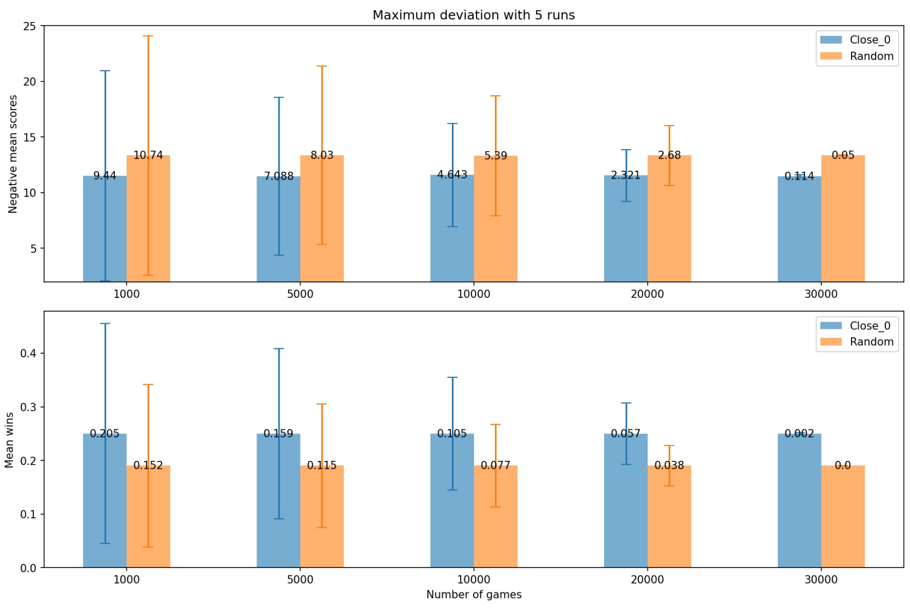


Figure 3.1: Bins represent the mean value after x games. The error bars shows how far the furthest run from the mean of all 5 runs was. The tournament included 4 random agents and one Close0 agent(See chapter 4)

many games must be played. This can be seen on Figure 3.1 where we see that at least $30'000$ games are necessary for the results to be consistent for random agents. (and thus comparable with each other.)

Also, we thought looking at the ELO from the tournament wasn't a good idea since it always depends on the level of the other agents. Elo is usually more interesting when looking a the evolution of an agents level but in many of our experiments we focus on fixed agents that aren't evolving anymore. We therefore decided to use score and sometimes wins as our metric for success.

3.2 Available bots

In this chapter, we will look at the best and most interesting bots that were already available in the repository.

3.2.1 MCSAgent

The MCSAgent conducts random rollouts, by using the Monte Carlo Method to approximate the value function in a given game state. Thus, to choose a move in a given game situation we simulate multiple possible game histories and average over the returns. The opponents cards, which are unkown to us are initialized

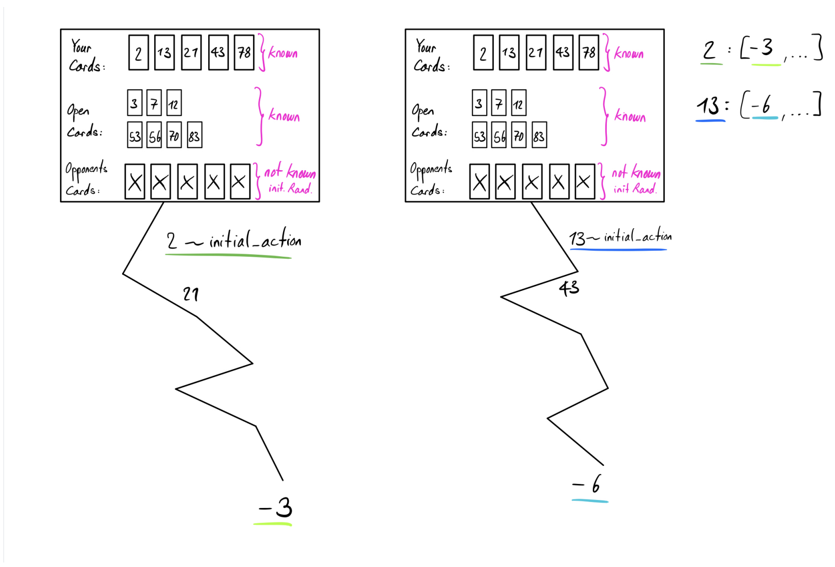


Figure 3.2: The Versions of the MCTS bot all have the same build up, but differ in the way they play out simulations, the MCSAgent plays random moves, the PolicyMCSAgent is guided by a neural network and the PUCTAgent is in the first step guided by the PUCT formula.

randomly. In the simulation, all of the players take turns and play a random card from their hand until the game has ended, then we store the return of the game and the initial card we played. This is called a simulation, since it does not influence the real game, instead we are just modelling different game histories. We play a given amount of simulations starting with different initial cards. Then we average the different returns which we received with the same initial card, which will then be our prediction for the return of this move, i.e. value-function. In the next move, the bot plays the card with the highest value prediction.

Todo: connect text with sketch

3.2.2 PolicyMCSAgent

The PolicyMCSAgent conducts actor network guided rollouts, it works similarly to the MCSAgent, but instead of that in the simulations to approximate the return value our agent plays a random card, we have a neural network which predicts the best card to play. The opponents play the same way using the same neural network. But, we still store the return values to the initial card we play and average for all received returns for a given card to get a prediction of the return value of playing a given card in this state. The neural network does not learn while we play the simulations, but only learns when playing real games.

3.2.3 PUCTAgent

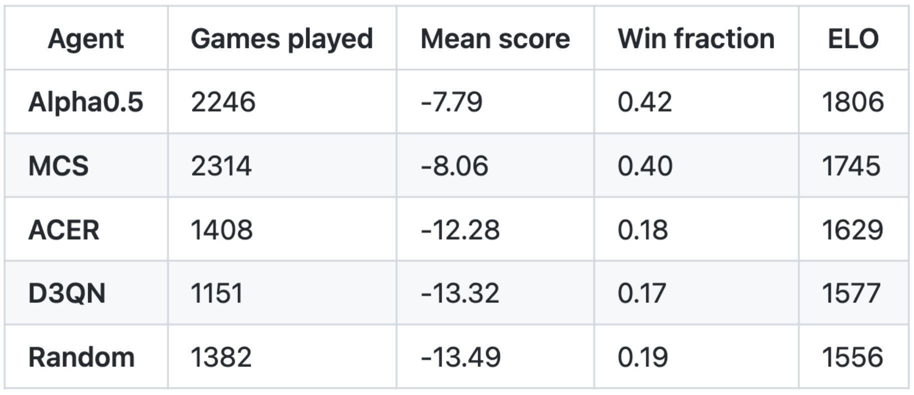
The PUCTAgent uses the Policy Upper Confidence bounded in Tree guided roll-out. Nevertheless, this bot differs to the PolicyMCSAgent only in the first move of every simulation. It chooses its move by maximising the PUCT formula in eq. (3.1).

$$PUCTS = q(a) + \alpha * P(s,a) * \frac{\sqrt{\texttt{n\_total} + 10^{-9}}}{1+n} \tag{3.1}$$

In the fromula, q denotes the average return the simulation scored when choosing a specific initial card, P(s,a) refers to the move probability obtained from the neural network, n refers to the number of times we already played this card in this game simulation, i.e. how many times we chose this card as the initial one for our simulation and `n_total` refers to the total number of simulations we have already played out. The variable q is on purpose written independent of the state, since in this implementation we only calculate and store the average return of the simulations q to the current game state. Thus, we can only calculate the PUCTFormula for the first step in the roll-out afterwards this method can not be used any more and the simulations will just be guided by the neural network which gives us the move probabilities like in the PolicyMCSAgent. Note, this bot implements the PUCT formula which is the key component of the AlphaZero and thus this is the creator's best attempt to get an AlphaZero like bot to work with this game. This is what we later want to build upon by implementing more components from AlphaZero in order to get a better performance. Because this bot is similar to AlphaZero, but missing some components the creator and also we some times refer to it as Alpha0.5 instead of PUCTAgent.

3.3 Performance of the existing bots

At the beginning of the project we had to analyse the performance of the existing bots. The creator of the Github repository which we took as a foundation for the project used a evolutionary tournament mode and ELO rating to analyse the performance of the bots. In fig. 3.4 and fig. 3.3, one can observe that the ACER agent (Actor-Critic) and the D3QN agent (Deep-Q-Neural-network) perform nearly as bad as random therefore we later stopped working with them. The MCS agent (Monte-Carlo-Method) and the Alpha0.5 agent (PUCTAgent) perform both very well.



Agent	Games played	Mean score	Win fraction	ELO
Alpha0.5	2246	-7.79	0.42	1806
MCS	2314	-8.06	0.40	1745
ACER	1408	-12.28	0.18	1629
D3QN	1151	-13.32	0.17	1577
Random	1382	-13.49	0.19	1556

Figure 3.3: ELO score of initialy untrained bots playing against each other, the different amount of games comes from the tournament mode which drops out poorly performing player.

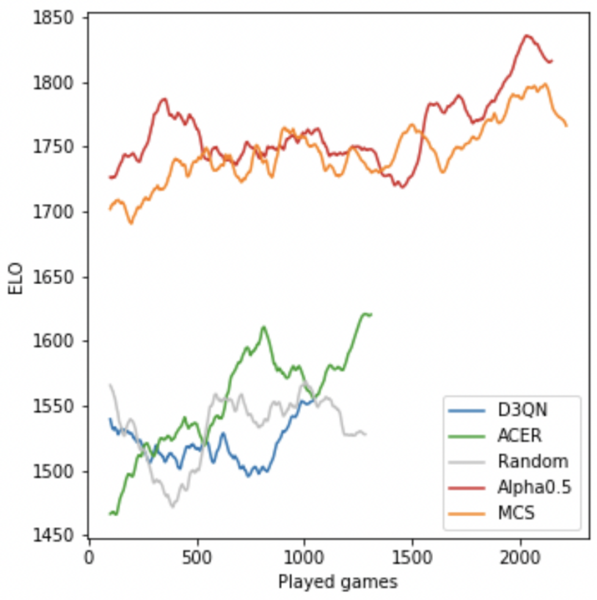


Figure 3.4: ELO score of initialy untrained bots playing against each other, .i.e. the learning speed and bot strategy play a role in the success.

Chapter 4

Game strategies and Explainability

Explainable artificial intelligence (XAI) is one of the main challenges in modern machine learning (ML). It consists in trying to understand the underlying mechanics behind a machine generated algorithm in order to make it more transparent and interpretable. The special case of explainable reinforcement learning (XRL) is particularly interesting since the system learns autonomously and being able to extract insight about how the system goes about learning can be very useful. A paper by Puiutta and Veith(9) builds a list of such XRL methods for different RL methods. Although these methods have shown to be effective, we went a more domain specific way when trying to explain the behaviour of the bot.

Our main idea for extracting information about the playing style of the best bot we encountered was to compare its playing style to hard-coded heuristics. The objective was to extract human usable/comprehensible strategies from the blackbox that is this bot. As mentioned before, although some other XRL (explainablility reinforcement learning) methods are available, we thought that doing this rigging method was a good starting point. As we later on discovered that the PUCTAgent did not learn much from the game (it does not perform much better than the basic MCS Agent see fig. A.1) we probably wouldn't have found anything meaningful with more standard XRL methods since we would have been trying to extract information from a clueless neural network.

The idea for this rigging strategy was composed of the following two steps:

1. Evaluate the heuristics against random agents to extract the relevant ones. This helped detecting what strategies where somewhat interesting to study. Especially because doing the testing on the alpha0.5 was very time consuming.

2. Rig alpha0.5 with the best heuristics and compare the percentage of moves that where the same between both the rigged and un-rigged agent.

The main motivation for this two step process is that in the first step we can

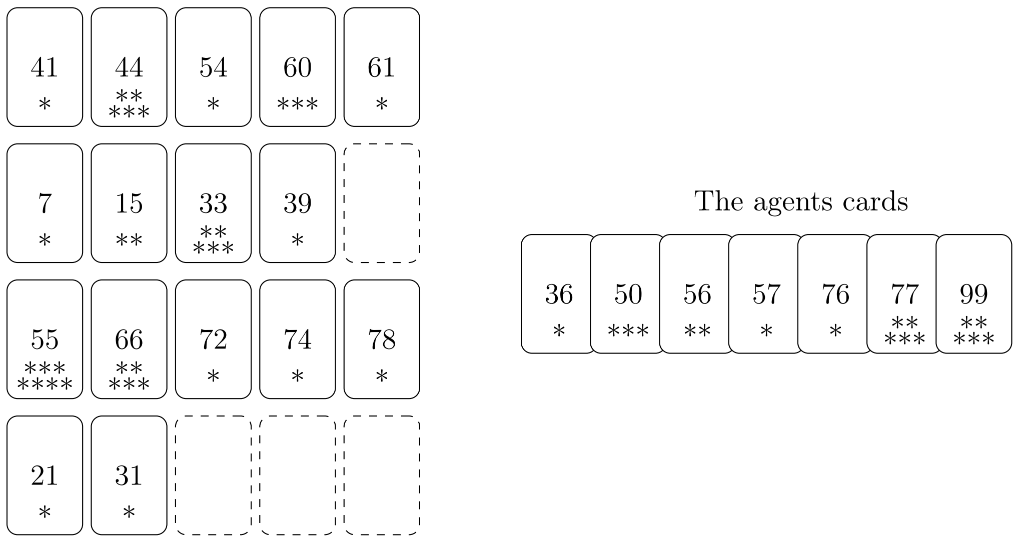


Figure 4.1: Example game situation after 3 turns, the agent must choose one of the 7 cards available in its hand

play tournaments with far much more games in much less time. As a reference, a tournament of 300 games with 3 MCS-based agents and 2 random agents takes approximately 20 hours to run (4 min/game). Whereas the same with only random agents takes about 1.5 seconds (200 games/sec). This let us identify the heuristics that improve on the random baseline, before taking the time and testing them against MCS-based agents.

The heuristics we decided to test where strategies that we could think of as being somewhat generally applicable strategies. A forum post gave us more ideas for good strategies (10). We will present them along with the result they gave when playing 30000 games against other 4 random agents. In order to explain how these rigs work, we will give examples with respect to Figure 4.1. Then, for the playing style comparison, we compared the playing styles on 300 games (i.e. 30000 moves) of one rigged Alpha0.5 agent playing against two "normal" Alpha0.5 agents and two random agents. Again, these results must be interpreted with care, since the variance on 300 games is still relatively high.

4.1 Point rig

This strategy encourages the bot to play cards depending on the number of bullheads according to a strength parameter τ. If the strength is positive, then it prefers to plays cards with high points first (e.g. 77 in Fig 4.1) and then with low points. As it can be seen on Figure 4.2, this did not lead to any significant difference with respect to the random baseline. So we did not investigate this rig

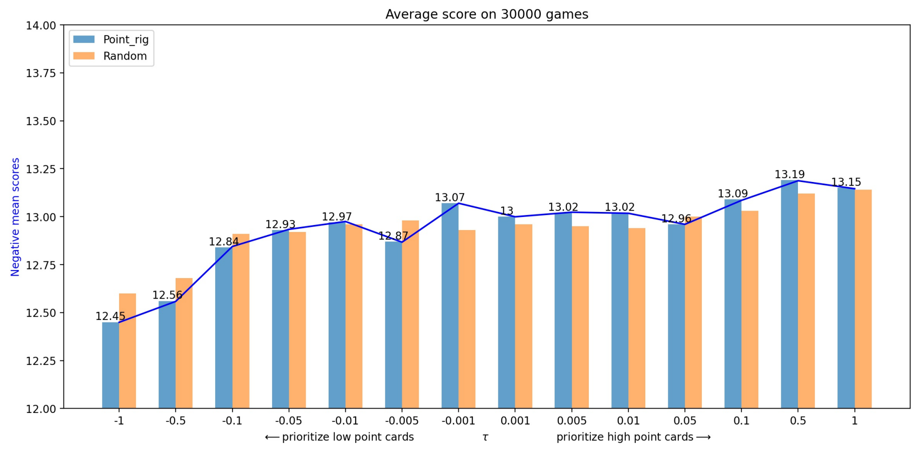


Figure 4.2: Average mean score of a random agent with point rig.

further.

4.2 Close rig

This strategy makes the bot immediately play cards that are safe upon row collection. This means, any cards that can be played and is close enough to the rows such that, even if opponents would place cards between the row and the players card, the agent would never take the row (in Fig 4.1 these would be the cards 32,33,34 and 40). In addition, the bot takes into account the cards that have already been played. This means if 32,33,34 or 40 had already been played then the next highest card becomes safe. In our example from Fig 4.1, 33 is already on the table and thus 35 would be considered as a safe card to play. Since it is quite unlikely that the opponents will coordinate to play all the cards between the row and the agents card, a "distance" parameter allows cards that are distance-cards away from the safe zone to be played directly. For example, if the distance parameter is set to 2, then the cards 36,37,~~41~~,42 and 43 would become safe. If multiple cards are safe, then the smallest is played.

The initial tests were relatively promising as it consistently outperformed the random agent, as it can be seen in Figure 4.3.

However, when compared to Alpha0.5, we were disappointed to see that there was no apparent correlation between the possibility of playing safe cards when it was possible and the playing style of Alpha0.5.

Indeed, the results summarised in Table 4.1 tell us that playing a close card when the opportunity to do so did not seem to interest Alpha0.5 very much. When compared to the random baseline (random played close 0.209 of the time when it was possible) it surely is a little improvement, but in order to say that the

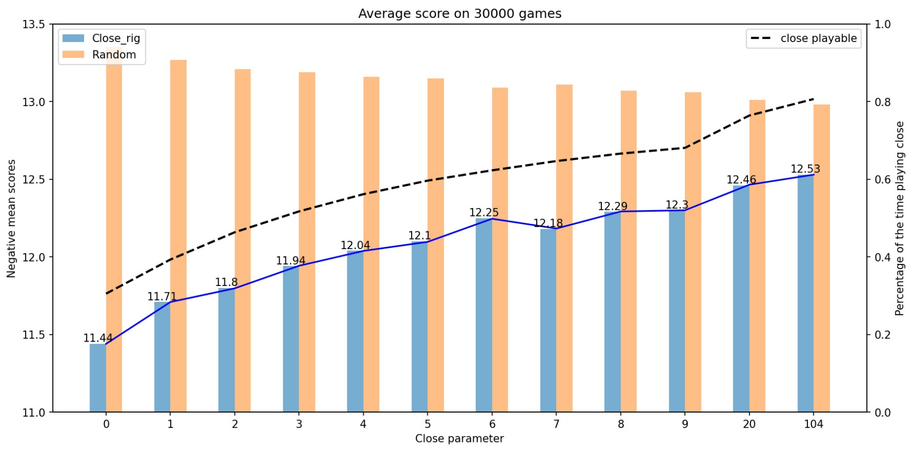


Figure 4.3: Average mean score of a random agent with close rig. The dashed line represent the average percentage of time the agent had the opportunity to play closely, i.e. one card in the agents hand was distance-close.

distance	0	1	2	3	4
α0.5 played close	0.315	0.32	0.342	0.365	0.372
similarity	0.772	0.708	0.661	0.653	0.635
performance	0.950	0.903	0.817	0.775	0.754

Table 4.1: Played close is the ratio of times alpha0.5 played close when it would have been possible to. Similarity is the number of moved played the same over the total number of moves played. Performance is the average mean score of the un-rigged agent over the average means score of the rigged agent ($< 1 \Rightarrow$ rigged worst than normal, $> 1 \Rightarrow$ rigged better than normal)

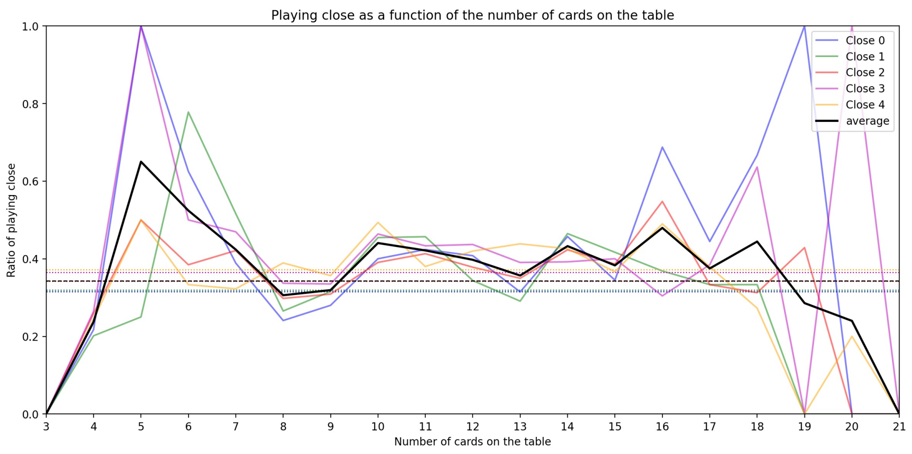


Figure 4.4: Ratio of moves where alpha0.5 played close when it was possible to do so. 1 means alpha0.5 always played close when x cards where on the table, 0 means it never played close.

Center	38	66	104
similarity	0.43888889	0.4762963	0.52666667
performance	0.67764737	0.65144795	0.78111455

Table 4.2: See Table 4.1 for reference.

bot follows the strategy of playing close, we would need much higher numbers. In an attempt to see whether the bot played more conservatively depending on the state of the game we plotted the ratio of playing close on Figure 4.4. But it does not seem to make any difference whether there are very few cards on the table or a lot.

4.3 Value rig

This rig encourages the agent to play cards near a certain value. To do this, we multiply a standard normal distribution with a centre and standard deviation with the playing probabilities of every card (for the random agent this probability is equal for every card, for Alpha0.5 this probability depends on the mean outcome of all simulations). The results shown in Figure 4.5 suggests that prioritising high cards decreases the number of penalty points drastically.

This was tested against Alpha0.5 and also does not seem to reflect how Alpha0.5 plays (Table 4.2). The standard deviation was set to a fixed value, lowering it would decrease the similarity and raising it would mean lowering the rigs impact on Alpha0.5.

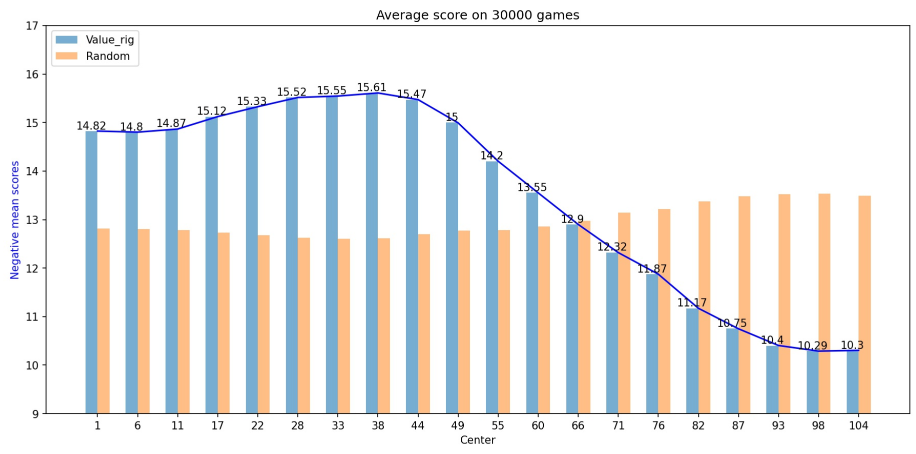


Figure 4.5: Average mean score of a random agent with value rig. The standard deviation was fixed of the normal

τ	-20	-10	-5	10
similarity	0.648	0.766	0.868	0.812
performance	0.88881513	1.06520105	1.02158961	0.87920635

Table 4.3: See Table 4.1 for reference.

4.4 Scattering rig

This rig encourages the bot to play cards that are either rather isolated or bunched together depending on a parameter τ. The fact of being isolated or not was determined by the average distance to the `n_neighbours` closest cards in the players hand. The initial results are shown in Figure 4.6 and seem to imply that prioritising cards that are isolated is a good thing.

When it came to comparing playing styles, we see on table 4.3 that the similarity decreases with the τ increasing, which just means if we force the rigged agent to play that way it will play less and less like the un-rigged agent. What is interesting however, is that with τ -5 and -10, the agents although playing relatively differently from the agent, still achieve a relatively similar performance. This seems to confirm the fact that playing isolated cards is a good strategy in general.

4.5 Risk rig

The previous rigs where all very naive and we tried to implement a somewhat more complicated heuristic as our final rig.

This one tries to make a balance between risking of taking a row and the

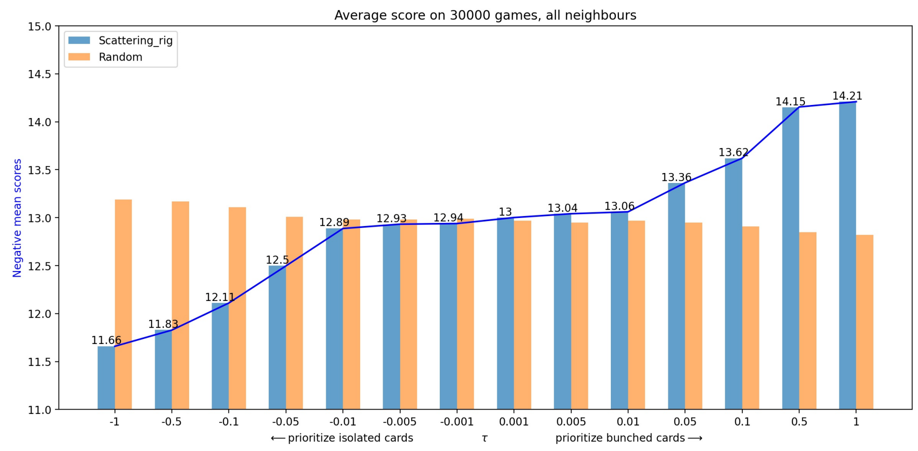


Figure 4.6: Average mean score of a random agent with scattering rig. The number of neighbours has been set to the maximum (average distance to all the other cards in the hand)

number of points that would be taken if the row was taken. For this, it first creates multiple matrices. Element i, j of these matrices represent the i-th card in the players hand being played on the on the j-th row. The distance matrix D and the point matrix P have an element respectively

$$d_{i,j} = \begin{cases} 1 + |i - top(j)|/104 & \text{if } i < top(j) \\ 1 - cl(j)/5 & \text{if } i > top(j) \\ 0 & \text{if } i \text{ is safe on } j \end{cases} \qquad p_{i,j} = pt(j) + pt(i) + 1.65cl(j)$$

where $cl(j)$ is the number of empty cards on row j, $top(j)$ the highest card of row j, $pt(\cdot)$ the number of penalty points in row j or card i and 1.65 is the average value of every card in the game. The risk matrix R is constructed as a weighted element-wise sum of both matrices. $r_{i,j} = \alpha d_{i,j} + p_{i,j}$ The parameter alpha determines the importance of the distance metric compared to the point metric. Finally, the minimum (or maximum for $\tau > 0$) entry or $R(i, j)$ of every card is added to the playing probabilities of every card.

As it can be seen on Figure 4.7, the given results are, in the best case (prioritising distance over points) somewhat similar to our previous best rig (value rig).

Putting this final rig under the test of the PUCT Agent, we see that it also leads to deceiving results as both similarity and performance are relatively low.

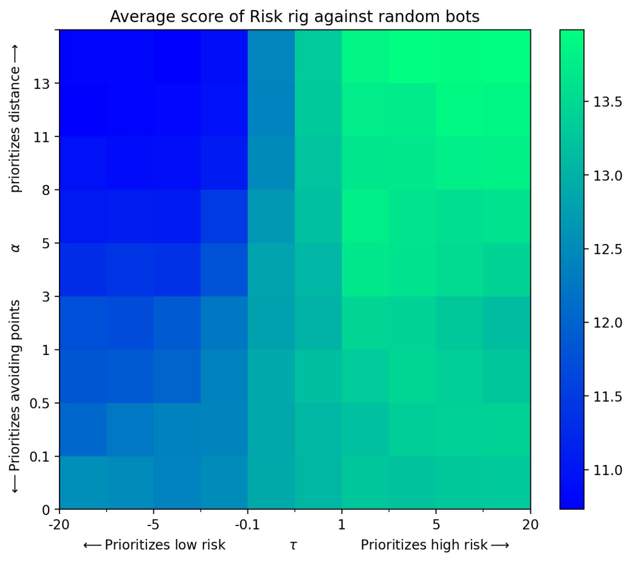


Figure 4.7: Average mean score of a random agent with risk rig.

τ	positive		negative	
α	0	15	0	15
similarity	0.408	0.491	0.694	0.710
performance	0.755	0.764	0.810	0.858

Table 4.4: See Table 4.1 for reference.

Chapter 5

Beating Alpha0.5

5.1 Problems Implementing AlphaZero

The main problem of implementing AlphaZero in this game, is that, from the perspective of any player, the state of the game is only partially known. This is a crucial point since, in particular, we cannot say that we are in one specific state at any given moment. To resolve this issue, we can restrict our view of a state to our cards and the ones on the table and ignore the cards of the opponents. However this means that in a given restricted state taking an action might lead to an enormous number of next states. As a matter of example let us be in the initial state pictured in fig. 2.1. The 4 opponents share 90 cards they could play $\binom{90}{4} = 2555190$ different combinations of cards. So any given restricted initial state (there are $\binom{104}{10}\binom{94}{4} \approx 10^{19}$ of them) might lead to 25551900 new restricted states. Each and every one of those will lead to new restricted states so on and so forth. This means the game tree, although relatively shallow (max. 10 moves) grows extremely large with billions of restricted game situations when totalling all the different levels of the tree. Even though MCTS (thus also AlphaZero) relies on learning only the parts of the game tree which seem to correspond to good strategies, the state space is so large that even learning which areas of the game tree are promising and exploring those is a difficult task given the constraints in training.

Another problem is that the basic AlphaZero implementation is not implemented for a multiplayer scenario and rather for the two-player game chess. The two-player AlphaZero algorithm is based on recursive playing with opposite incentives for the opponent player (Minimax theorem).The recursive algorithm lets us pass back the return, in order to store it to the visited states. This is not possible for more then two players, so it is harder to implement the search tree based AlphaZero for the 5 player setup we are analysing.

In the beginning, we wanted to implement the full AlphaZero algorithm as described in the paper(2). The challenges mentioned before in section 5.1 hindered us from building a ground up implementation of AlphaZero for *6nimmt!*.

After trying to look how to fix those problems, we recognised that extending it to multiple players, hiding the partial information and storing the rewards of the game tree which is done with negative and positive values which recursively backpropagate, would take us too much time for this project. We thus decided to focused on faster implementations, which we will present now.

5.2 Approaches

5.2.1 Directly Predict Q

One idea that we adapted from AlphaZero, was to not fully roll out a simulation if it encountered an unexplored state, but rather ask a neural network what the expected return would be. We still used the Monte Carlo averaging, meaning after we choose a move, we simulate a possible next game state by randomly drawing cards for the opponents. However instead of rolling out the full simulation we only went one step in depth. The choice for the move is done using the PUCT formula, which is similar to the real AlphaZero move formula. Once we have this new simulated next state, we ask a neural network to predict the return value for a given move, in other words, the expected return for this state-action pair, i.e. the Q-Value. Finally the predicted return, i.e. V-value for a given state, is computed using the mean over the Q-values for all actions (We used the detour over the Q-values, because we could predict them with the already available neural network architecture, which was used to predict the move probabilities). This saves a lot of time as only evaluating the neural network in the second step is about 5 times faster than the neural network guided rollout used by the PUCT agent. We performed a time comparison in section 5.5 to insist on that point. Therefore, not only can this agent play more simulations in the same amount of time, it also means it will train faster when trying to learn the value and move probability functions.

5.2.2 Directly Predict V

In order to simplify the previously presented Q-predicting bot, we also constructed a neural network which directly predicts the return, i.e. V-value of a state instead of averaging the Q-values. As with the Q-predicting bot, here we still start simulations with one step depth following the PUCT formula and then predict the return value. The difference is, here, we changed the whole neural network architecture and training such that we can directly predict a V-value and thus do not have to go over the averaging of Q-values. Since, we do not have to go over the Q-values and calculate their mean, we were able to simplify our infrastructure and got a more traceable learning. This also lead to increased performance, as we will show later on (see section 5.4). Because we directly predict V with a modified version of the PUCTAgent, we named this bot PUCTAgentModDirV.

5.2.3 Separate Value and Move Networks

Now, the previous implementations still use the prediction of the move probability to evaluate the PUCT formula. The prediction was extracted from a double headed actor-critic style network, which simultaneously predicts both variables. We assumed it might further benefit the learning if we separate the two networks, such that we have one actor network predicting move probabilities and one critic network that predicts the return, i.e. Q/V-value for a given state/state-action-pair. Unfortunately, this did not help our predictions and gave slightly worse performance.

5.2.4 Multi-Step Direct V Prediction

As one can see in section 3.3 and fig. A.1 the pure MCS bot (random rollout) performs only slightly worse than the PUCTAgent, this lead us to the conclusion that the strength of the PUCTAgent comes primarily from the roll-outs and not necessarily from the guidance by the neural network. Thus, we wanted to use our value predicting version of the PUCTAgent which lead to significant speed up and combine it with the strength of rolling-out for some steps until the prediction is easier for the neural network. The idea would have been to play a few steps guided by the neural network like the PUCTAgent and only then predict a state value. Unfortunately, because of time constraints we were not able to implement this idea anymore.

5.3 Design

5.3.1 Learning the Value function

The value function is learnt by a neural network that aims to decrease the mean squared error (MSE) loss between the value predicted for an intermediate game state and the actual return at the end of the game. In fig. 5.2, one can observe how the value function loss evolves during training. The setup for the training is a PUCTAgentModDirV (section 5.2.2) with 50 Monte Carlo simulations against 3 PUCTAgentModDirV with 50 simulations as well. The bots train their neural networks only when playing games and not during the internal simulations. Thus they have 11'000 games with many different intermediate game states to learn from. We can see that the MSE loss stagnates at around 16, thus the difference between real encountered game return and the neural network prediction is 4. As in a 4 player game, one can expect to take around 9 points on average, this seems to be quite a high loss to stagnate on, but is could still be useful to differentiate between good and bad moves. However, it is difficult to interpret how this correlates to the overall performance of the value-predicting bot.

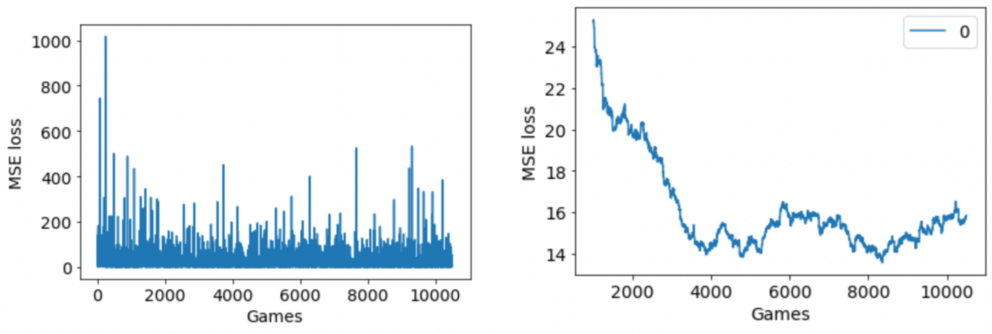


Figure 5.1: Value function MSE loss. The right graph uses a rolling average of 1'000 while the left is not averaged

5.3.2 Learning the Move Probabilities

In contrast to learning the value function, where we designed the neural network architecture, the learning objective and the loss metric, in this section, we are just evaluating the training of the move probabilities with the previously available architecture. The move probabilities are learnt by incentivizing the neural network to give deterministic probabilities. This is achieved by giving the neural network the objective to reduce the loss term $-\sum log($`predicted_probability`$)$, i.e. the loss is zero for a move probability equal to 1 meaning absolute certainty. The predicted move probabilities after training for 11'000 games where quite deterministic with $-\sum log($`move_probabilities`$)$ being at about 12.5 (see fig. 5.2. This seems to mean that the move probabilities guide the simulations well.

With the combined actor-critic network infrastructure (both bots in section 5.2.1 and section 5.2.2) we have to give a combined objective for the neural network to minimise, thus we decided to add the two losses and minimise the sum of them. For the separate network actor-critic we can learn the two objectives separately.

5.3.3 Influence of the discount factor on learning

Like in any machine learning project, we also had to experiment with the hyperparameters. An important one for the learning of the value function, was the discount factor. To learn the value function, we store all the intermediate game states we pass through while completing a game and back-propagate the loss after the game. In reinforcement learning, often older game states are depreciated to

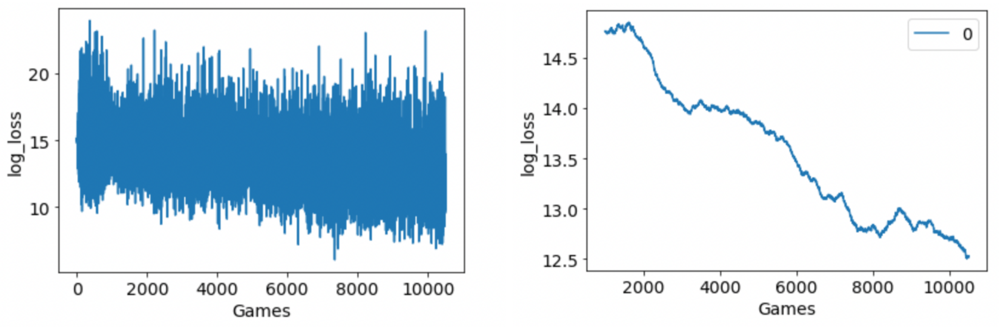


Figure 5.2: Move probabilities $-\sum log(prob)$ loss. The right graph uses a rolling average of 1000 while the left is not averaged

discount factor:	1	0.99	0.98
PUCTAgent	40.9	40.9	40.9
Predict Q	39	36.6	36.5
Predict V	39.3	38.1	38
Seperate NN	34.7	38.5	37.7

Table 5.1: Win rate against three Random bots for 1400 games initialised without training

give the new ones more value, the same seemed to be of some benefit here. When back-propagating the return, we multiply the value with some $\gamma < 1$ such that the states at the beginning of the game are less strongly weighted. We tried out the discount factors γ 1 (not depreciating),0.99 and 0.98. The results were not totally conclusive, discount factor equal to one actually lead in some cases to better win rate, but the training loss was worse and with a lot more variance meaning that this some how disturbs the learning. Discount factor 0.98 lead to more smoother training (less variance in the losses), but lower win rate. Thus, we chose to go with a trade-off of 0.99. One can see the win rate for the bots with different discount factor in table 5.1.

5.4 Evaluation of the Approaches

In order to evaluate the performance of our bots, we created two setups in which we measured the average scores and win rates. In both cases we used random bots to simulate the multiplayer scenario.

In the first setup, we played each initially untrained bot against 3 random bots for 1400 games. As random agents play extremely rapidly, this lead to fast evaluation of whether our bot might be able to learn something about playing the game well.

On the other hand, for the second setup, we played each bot directly against the currently reigning champion the PUCTAgent and two random bots, in order to have a direct comparison.

From table 5.2, which is the results of the first setup, we see that Predict Q and Predict V have similar levels against random bots. Nonetheless they both struggle in direct comparison with the PUCTAgent, as it can be seen in table 5.3, summarising the results of the second setup.

	Winrate	Mean Score
PUCTAgent	40.9	-8.575
Predict Q	36.6	-9.717
Predict V	38.1	-9.3535
Seperate NN	38.5	-9.5

Table 5.2: Win rate against three Random bots for 1400 games initialised without training

	Winrate	Mean Score
PUCTAgent	33.5	-7.69
Predict Q	26.3	-9.30
Predict V	27.0	-9.55
Seperate NN	27.1	-9.57

Table 5.3: Win rate against one PUCTAgent and two Random bots for 2000 games initialised without training

Again, we suspect the reason to be that learning is difficult in this game, due to the extreme amount of hidden information. The fact that we already saw in previous chapters that the MCSAgent, which doesn't even try to learn anything, is performing as well as the RL agents (see section 3.3), shows us that our approach of basing ourselves more on learning rather than on the Monte Carlo simulations is difficult.

Nevertheless, we noticed that our agents are all about 4-5 times faster than the PUCTAgent. So when taking time as a relevant factor of the game, we might be able to improve our bots relative performance. Before testing this hypothesis in section 5.6, we will show the drastic time difference between the agents.

5.5 Speed Comparison

From table 5.4 and table 5.5, we can see that the PUCTAgentModDirV(section 5.2.2) is close to 5 times faster in training and also in evaluation than the PUCTAgent. Because training and evaluation take nearly the same time, we presume the simulations are much more time costly in comparison to the backpropagation while training. The expensive time cost of the policy guided simulations in the PUCTAgent is made even more clear when seeing how it compares to the random simulations of the MCSAgent. Thanks to shorter simulations, our bots are on average far quicker than the PUCTAgent and about as quick as the random simulations of the MCS Agent. In fig. 5.3, one can see how the thinking time of the MCS based agents decreases over the number of cards that are left to play. This is clearly due to the fact that at the beginning of the game, simulations are 10 moves long, and then it decreases as we get rid of our cards.

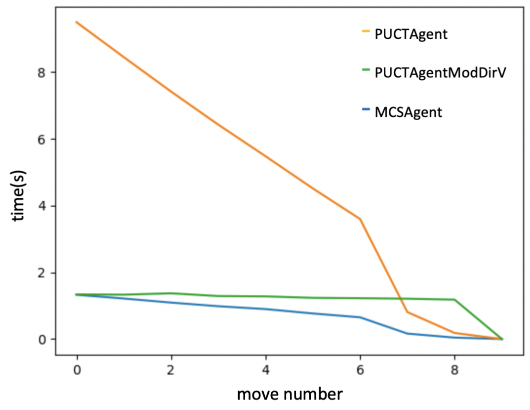


Figure 5.3: For 200 mc the time used for each move, corresponding to the 10 rounds.

mc	50	100	200	300	400
MCS Agent	0.196	0.376	0.731	1.068	1.390
PUCTAgent	1.241	2.413	4.730	6.946	9.089
PUCTAgentModDirV	0.287	0.577	1.166	1.714	2.215

Table 5.4: Average time for one move in a full game, with the agent being in evaluation mode.

mc	50	100	200	300	400
MCS Agent	0.191	0.369	0.715	1.051	1.370
PUCTAgent	1.217	2.372	4.639	6.829	8.938
PUCTAgentModDirV	0.281	0.567	1.147	1.683	2.175

Table 5.5: Average time for one move in a full game, with the agent being in training mode.

Agent	Mean Score
PUCT_{200}	-9.407
MCS_{1200}	-8.476
MCS_Mod_{1000}	-10.717
MCS_Mod_{200}	-12.347
HIGHEST	-19.87

Table 5.6: Results of a tournament with 5 Agents on 300 games. The subscript means the maximum number of simulations per move `mc_max`. HIGHEST plays always the highest card.

5.6 Showdown

Although some attempts at building a more skilful agent than PUCTAgent (section 3.2.3) weren't as successful as we wished they where. We are now discussing 3 different ways that we looked at to create "the best bot out there".

First we recall that our PUCT with a scattering rig $\tau \approx -5$ did already beat the PUCT Agent on a set of 300 games, by a small margin (table 4.3).

But motivated by the fact that the PUCT agent wasn't better than the pure MCS agent (see Figure A.1), and much slower, we first simply tuned up the number of simulations done by MCS so that it would have a thinking time less or equal to the PUCT agent. In addition to this, we used our game knowledge gathered in the rigging part (see chapter 4) to direct the search of the Pure MCS agent. This was motivated in parts by a paper on how to improve MCTS by adding expert knowledge (11). We simply directed the moves in the simulations towards strategies that were known to be positive on the playing level of the agents: playing high cards, playing close cards and playing isolated cards. This means that the simulations should be a better representation of how other good players play. Inexplicably however, this new modified MCS bot performed quite poorly w.r.t. pure MCS and PUCT. Table 5.6 summarises the scores of a tournament between these agents on 300 games.

Following the same principle of taking into account speed in the evaluation and recalling that the PUCTAgentModDirV(section 5.2.2) is close to 5 times faster in training and evaluation modes than the previously best bot, the PUCTAgent (see section 5.5), we compared both agents in the case they have similar training time in self-play and the same thinking time during an evaluation tournament.

Therefore, we first setup two independent training environments where both bots self-played against 3 other instances of their kind and PUCTAgentModDirV had 5 times as many games (10000 against 2000). We trained both bots with 50 Monte Carlo Simulations (mc) at each move. The opposition's tournament included two random agents (for faster overall play time, while still holding a playing environment) along with the two trained agents and comprised 400 games. We gave the PUCTAgentModDirV 200 simulations against 50 for the trained PUCTAgent so that, according to table 5.4, both agents will approximately take the same time to play a move. The results are summarised in fig. A.2.

Although with unlimited play time the fully rolling out bot PUCTAgent seems to perform better, when time is taken into account (as it is in blitz *6nimmt!*, similar to blitzchess) the bot which we created clearly outperforms PUCTAgent. Indeed, PUCTAgentModDirV wins more than 40% of the games while the PUCTAgent less than 30. This means our bot clearly beats the PUCTAgent and therefore, under the assumption that we give both agents the same time for training and playing, we created, to our knowledge, the best *6nimmt!* bot out there!

	Winrate	Mean Score
PUCTAgentModDirV$_{200}$	0.40	-8.88
PUCTAgent$_{50}$	0.29	-10.03
Random1	0.15	-13.36
Random2	0.15	-13.45

Table 5.7: Win rate and mean score of PUCTAgentDirV against PUCTAgent with same time for training and playing a move over 400 games.

Chapter 6

Conclusion

In the first part of the project, we tried to explain the behaviour of an existing RL agent through so-called rigging. We interpreted the performance of different hard-wired strategies against different opponent. This gave us some insight as to which strategies yield good results in this game.

In the second part, we were able to develop multiple new architectures on top of the few existing bots available for this game and outperformed the previously best bot. Amongst the proposed architectures, the most promising was based on learning the values of different game states and thus providing a policy for the *6nimmt!* game. As this neural network based policy is much faster then the Monte-Carlo method based approximation used previously, we leveraged on this speedup and were able to clearly beat the previously best bot when given equal time for training and moving.

Working on these implementations, we questioned the learning abilities of the initial agents as it seems the high stochasticity of the game prevented them to learn relevant strategies. However we showed that something could in fact be learnt as our new implementations, which rely very heavily on predictions, performed quite well.

As a final note, we wish to express how this project was full of novelty for us and how interesting it was to work on such a project. We started with little knowledge about reinforcement learning and had to read through books and many papers to understand the needed theory. But most importantly, the project taught use how to apply reinforcement learning techniques in practice. This project also gave us a taste of the work that arises with machine learning projects. Indeed, the amount of plotting, parameter tuning, debugging, etc. was enormous and the results where often very surprising. This gives us the motivation to further work on similar projects and use our learnings. Also, we hope to publish this report along side our code, such that other students can have an objective to beat and thus also learn about the interesting topic that is reinforcement learning.

Bibliography

[1] P. Müller, "Tichu bot," Aug. 2020.

[2] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, Y. Chen, T. Lillicrap, F. Hui, L. Sifre, G. van den Driessche, T. Graepel, and D. Hassabis, "Mastering the game of go without human knowledge," Nature, vol. 550, pp. 354–, Oct. 2017. [Online]. Available: http://dx.doi.org/10.1038/nature24270

[3] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, T. Lillicrap, and D. Silver, "Mastering atari, go, chess and shogi by planning with a learned model," Nature, vol. 588, no. 7839, pp. 604–609, 2020. [Online]. Available: https://doi.org/10.1038/s41586-020-03051-4

[4] N. Bard, J. N. Foerster, S. Chandar, N. Burch, M. Lanctot, H. F. Song, E. Parisotto, V. Dumoulin, S. Moitra, E. Hughes, I. Dunning, S. Mourad, H. Larochelle, M. G. Bellemare, and M. Bowling, "The hanabi challenge: A new frontier for ai research," 2019.

[5] W. Kramer, 6nimmt! - Set of instructions, version 2.4 ed., AMIGO Spiel + Freizeit GmbH, 63128 Dietzenbach.

[6] R. S. Sutton and A. G. Barto, Reinforcement Learning: An Introduction, 2nd ed. The MIT Press, 2018. [Online]. Available: http://incompleteideas.net/book/the-book-2nd.html

[7] J. Brehmer, "rl-6-nimmt," https://github.com/johannbrehmer/rl-6-nimmt, 2020.

[8] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.

[9] E. Puiutta and E. M. S. P. Veith, "Explainable reinforcement learning: A survey," CoRR, vol. abs/2005.06247, 2020. [Online]. Available: https://arxiv.org/abs/2005.06247

[10] M. Jaatinen, K. Fjeld, and Clionerd, "6 nimmt strategy tips," https://boardgamegeek.com/thread/436792/6-nimmt-strategy-tips, 2009.

[11] G. Chaslot, C. Fiter, J.-B. Hoock, A. Rimmel, and O. Teytaud, "Adding expert knowledge and exploration in monte-carlo tree search," 05 2009.

Appendix A

Figures

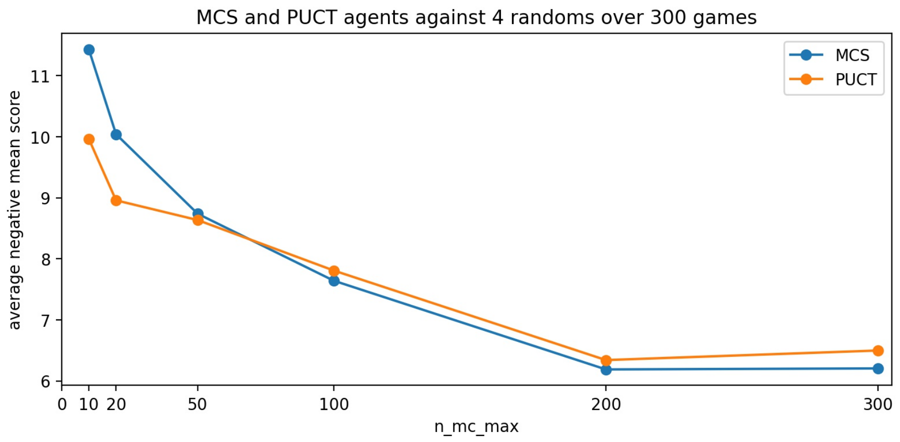


Figure A.1: Average mean score of the pure MCS and the PUCT agent against 4 random agents, depending on the number of Monte Carlo simulations done by the agent.

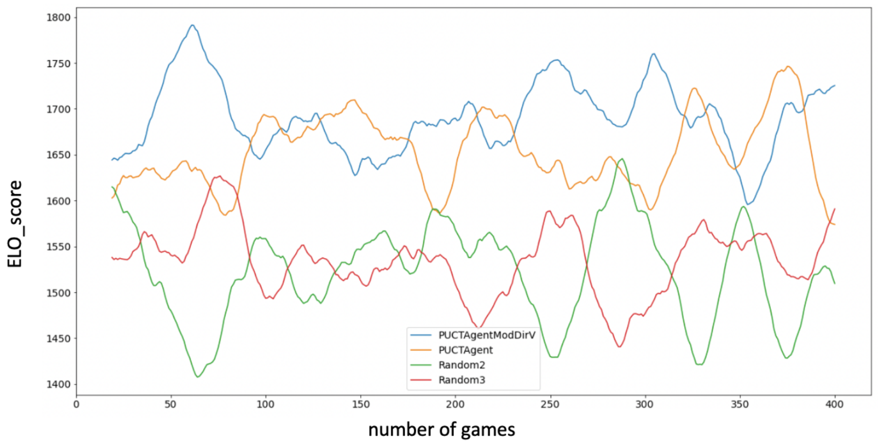


Figure A.2: ELO evolution of PUCTAgentDirV against PUCTAgent with same time for training and playing a move over 400 games.