University of Groningen

Learning from Monte Carlo Rollouts with Opponent Models for Playing Tron

Knegt, Stefan; Drugan, Madalina M.; Wiering, Marco

Published in:
ICAART 2018

DOI:
10.1007/978-3-030-05453-3_6

IMPORTANT NOTE: You are advised to consult the publisher's version (publisher's PDF) if you wish to cite from it. Please check the document version below.

Document Version
Final author's version (accepted by publisher, after peer review)

Publication date:
2018

Link to publication in University of Groningen/UMCG research database

Citation for published version (APA):
Knegt, S., Drugan, M. M., & Wiering, M. (2018). Learning from Monte Carlo Rollouts with Opponent Models for Playing Tron. In J. van den Herik , & A. Rocha (Eds.), *ICAART 2018: Agents and Artificial Intelligence* (pp. 105-129). (Lecture Notes in Computer Science book series ; Vol. 11352). Springer. https://doi.org/10.1007/978-3-030-05453-3_6

Copyright
Other than for strictly personal use, it is not permitted to download or to forward/distribute the text or part of it without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license (like Creative Commons).

The publication may also be distributed here under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license. More information can be found on the University of Groningen website: https://www.rug.nl/library/open-access/self-archiving-pure/taverne-amendment.

Take-down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Downloaded from the University of Groningen/UMCG research database (Pure): http://www.rug.nl/research/portal. For technical reasons the number of authors shown on this cover page is limited to 10 maximum.

Download date: 19-11-2022

Learning from Monte Carlo Rollouts with Opponent Models for Playing Tron

Stefan J.L. Knegt[1], Madalina M. Drugan[2], and Marco A. Wiering[1]

[1] Institute of Artificial Intelligence and Cognitive Engineering
University of Groningen, The Netherlands
stefanknegt@gmail.com, m.a.wiering@rug.nl
[2] ITLearns.Online, The Netherlands
madalina.drugan@gmail.com

Abstract. This paper describes a novel reinforcement learning system for learning to play the game of Tron. The system combines Q-learning, multi-layer perceptrons, vision grids, opponent modelling, and Monte Carlo rollouts in a novel way. By learning an opponent model, Monte Carlo rollouts can be effectively applied to generate state trajectories for all possible actions from which improved action estimates can be computed. This allows to extend experience replay by making it possible to update the state-action values of all actions in a given game state simultaneously. The results show that the use of experience replay that updates the Q-values of all actions simultaneously strongly outperforms the conventional experience replay that only updates the Q-value of the performed action. The results also show that using short or long rollout horizons during training lead to similar good performances against two fixed opponents. [3]

Keywords: Reinforcement Learning · Opponent Modelling · Games · Monte Carlo Rollouts · Multi-layer Perceptrons

1 Introduction

Reinforcement learning (RL) algorithms [21] allow an agent to learn to play a game from trial and error by observing the result of each game. Often the result of a single game provides little information to learn from, as in many cases the game rules only return a value 1, 0, or -1 depending whether the game was won, ended in a draw, or was lost by the agent. Therefore, many games need to be played in order to learn which moves are optimal in each game

[3] Cite this paper as: S.J.L. Knegt, M.M. Drugan and M.A. Wiering. Learning from Monte Carlo Rollouts with Opponent Models for Playing Tron. Agents and Artificial Intelligence. Springer International Publishing. edited by: J. van den Herik and A.P. Rocha. 2019. Pages 105-129. DOI: 10.1007/978-3-030-05453. https://www.springerprofessional.de/en/learning-from-monte-carlo-rollouts-with-opponent-models-for-play/16365998

state. Furthermore, games usually consist of very large state spaces and therefore appropriate function approximation techniques need to be used to generalize over the state space. The oldest self-learning program that learned to play a game is Samuel's checkers playing program [13]. It combined several machine learning methods and reached a decent amateur level in playing checkers. A very successful attempt to using reinforcement learning to play games is TD-Gammon [22], that learned to play the game of Backgammon at human expert level using temporal difference learning [20] and multi-layer perceptrons. Although in the '90s, learning from 1.5 million games took multiple months, with the current computing power this can be done within several hours. A more recent and even more impressive system is AlphaGo Zero [18] which learned to play the complex game of Go from scratch and was able to beat its predecessor AlphaGo [16], which first learned from games played by human players and was able to beat the human Grandmaster Lee Sedol in 2016. AlphaGo Zero combines several techniques in a novel and effective manner: reinforcement learning, Monte Carlo tree search (MCTS) [8] and deep neural networks [14] by training a value function to predict the result of a game and a policy network on the frequency with which moves were selected in the MCTS rollout phase. AlphaGo Zero was later followed by AlphaZero [17] that learned to play chess and shogi from scratch according to the same principles as AlphaGo Zero and was able to strongly outperform the best previous computer programs for these games. This research has shown that learning from Monte Carlo tree search results is a very effective method for mastering different kinds of games.

Although in most game playing programs, no opponent model is learned, the optimal move in a game state can also depend on the opponent's playing style. This holds especially if a fixed opponent is used for playing the game. Therefore, for such games it would be useful to learn a model of the opponent in order to predict its moves. In most research on opponent modelling [5,19,6] the algorithm to learn the opponent model is problem specific and does not learn quickly. Therefore, in our previous work we developed a novel opponent modelling technique that learns to play the game of Tron and models the playing style of the fixed opponents simultaneously [7]. This technique was then combined with Monte Carlo rollouts, and the results of this system were much better than without using the opponent models.

In this paper, we extend our previous research on using reinforcement learning to play the game of Tron against two fixed opponents. We are primarily interested if learning from the outcomes of the Monte Carlo rollouts can increase the performances obtained in [7] even further. Although learning from the results of lookahead planning has been successfully applied in chess [1,17], Go [18], and other types of problems, this has not been integrated with learning a model of the opponent. To deal with the large state spaces of Tron, the learning algorithm combines Q-learning [24] with a multi-layer perceptron (MLP) [12]. This technique has already been successfully applied in games such as Backgammon [22], Ms. PacMan [2] and Starcraft [15]. Because the field of play in Tron is a 10×10 grid, there is no need to use deep reinforcement learning [10], however, as shown

in [7], the use of vision grids to give a partial agent-centered representation of the game state was very effective and will also be used in this paper. Another extension is that in this paper experience replay [9] will be used in two different ways to learn from the estimates obtained through the Monte Carlo rollouts.

Contributions: We developed a novel system for learning to play the game of Tron that combines reinforcement learning, opponent models, and learning from lookahead planning with Monte Carlo rollouts. To speed-up learning, we examine two extensions compared to our work described in [7]: 1) Learning from lookahead planning, where the estimates obtained with Monte Carlo rollouts are used to train the Q-values of the actions, and 2) Using experience replay with a replay memory to learn from less games. Furthermore, we created two different methods in which the agent can learn from the Monte Carlo rollouts using experience replay: by only learning from the estimates obtained through the rollouts of the performed action, or learning from the estimates of all possible actions in a specific game state. By using the learned model of the opponent and the game rules, estimates are obtained for all actions in each game state that are used for selecting an action and that can be used for training the system. Therefore, learning from the rollout estimates of all actions does not require any computational overhead. Different experiments have been performed with different lookahead horizons and numbers of rollouts in order to examine if learning from the rollouts improves performance. The experiments are performed against two different fixed opponents and using two different game-state representation with different sizes of the vision grids. The results show that experience replay is very effective when training on the Monte Carlo rollout estimates of all actions. This leads to our new method attaining similar high win rates against the fixed opponents when learning from 150,000 games instead of the 1.5 million games used in our previous paper.

Outline: In the next section we explain the previous research we have done, including a description of the game of Tron and the RL system that was combined with opponent models and Monte Carlo rollouts. In Section 3, the novel method of learning from Monte Carlo Rollouts with experience replay is described. Section 4 describes the experiments and the results. In Section 5, the conclusions are presented together with possible future work directions.

2 Reinforcement Learning with Opponent Models for Playing Tron

In this section, we describe our previous approach [7] for learning to play Tron by self-play. Our RL-Tron system achieved remarkable successes against two different fixed opponent agents and consists of 3 elements: (1) Q-learning with multi-layer perceptrons are used to learn an approximation of the state-action value function, (2) The used multi-layer perceptron is combined with a novel algorithm for predicting which action the opponent selects in a game state (the opponent model), (3) The state-action value function and the opponent model

are used in Monte Carlo rollouts to select an action in a game state based on future state trajectories during the final test games.

We will first describe the game of Tron. Then we describe the combination of Q-learning and multi-layer perceptrons. In subsection 2.3, we will describe three different state representations that were used in [7] for learning to play Tron. Finally, we will describe the opponent-model learning technique and how it was used in the Monte Carlo rollouts.

2.1 The Game of Tron

Tron is an arcade video game released in 1982 inspired by the Walt Disney motion picture Tron. In this game the player guides a light cycle in an arena against an opponent. The player has to do this, while avoiding the walls and the trails of light left behind by the opponent and the player itself. Figure 1 depicts an example game state played by two agents. For this research we developed a framework in order to use reinforcement learning in this game. This framework implements the game as a sequential decision problem, where two agents can play against each other and the environment is represented by a 10×10 grid.

At the beginning of a game the agents are randomly placed in either the top half or bottom half of the grid. At every game state there are four possible actions: moving up, down, right, or left. It is important to note that one of the four moves will always lead to the agent hitting its own trail of light. When both agents have selected a move, the new game state is determined. Whenever two agents move to the same location in the grid the game ends in a draw as well as when both agents hit a trail of light or the wall at the same time. In all other cases, the game continues until one of the two agents hits a trail of light or the wall. When looking at the possible amount of different game states, we estimate this to be in the order of 10^{20}, which is similar to the game Othello that consists of a board of 7×7 cells.

In this research the agent always plays against one fixed opponent at a time and the opponents employ either a semi-random or semi-deterministic strategy. The semi-random opponent always randomly selects one of the four possible moves, unless that move results in an immediate collision. The semi-deterministic agent always tries to select its previous action and if that would result in a collision, it selects a random possible action. Therefore, both opponent strategies are constructed such that the opponent will never move to a location that is already visited or is a wall, unless there is no other possibility. Although both strategies seem quite basic, the semi-deterministic strategy can be a relatively good strategy for the game of Tron. By always selecting the previous action, if this is possible, the agent makes long trails of light and thereby easily closes the other agent in, which will eventually lead to this agent losing the game. We tested the performance of both opponent strategies by letting them play against each other for many games. The results of these matches showed that the semi-deterministic strategy (going straight as long as possible) wins in 55% of the games and loses in 25% of the games, while 20% of the games end in a draw. From here on we will refer to the agent employing the collision-avoiding

Fig. 1. Tron game environment with two agents, where their heads or current locations are in a darker colour (taken from [7]).

random policy as the random opponent and the semi-deterministic opponent will be referred to as the deterministic opponent.

2.2 Reinforcement Learning

Reinforcement learning (RL) algorithms allow an agent to learn to optimize its behavior from its interaction with an environment. Each time-step, an agent observes the state of the environment and uses this information to select an action. Then the state is changed to a next state and the agent receives a scalar reward signal for this state transition. The aim of the agent is to learn an action-selection policy that maximizes the obtained future discounted sum of rewards. Reinforcement learning algorithms can be used to solve different kinds of sequential decision problems. Because the game of Tron is fully observable and the agent plays against a fixed (i.e. non-adaptive) opponent, the game can be modelled as a Markov Decision Process [11], which is defined by the following components:

- A set of states S, where $s_t \in S$ denotes the state at time t.
- A set of actions A, where $a_t \in A$ denotes the action selected at time t.
- A transition function $T(s, a, s')$, which specifies the probability of moving to state s' after selecting action a in state s.
- A reward function $R(s, a, s')$, which sends a reward signal to the agent for executing action a in state s and subsequently moving to state s'. r_t denotes the reward obtained at time-step t.
- A discount factor γ that makes rewards received further in the future less important, where $0 \leq \gamma \leq 1$.

In the game of Tron, the reward for winning a game is 1, for a draw the reward is 0, and if the RL agent loses it receives a reward of -1. There are no other rewards emitted while the game is not over. The environment is stochastic for the agent, because the agent selects an action and at the same time the opponent selects an action, after which the state of the game is updated. Because the agent learns to play against two different opponents that use a stochastic policy, the game is non-deterministic.

The agent needs to optimize its policy $\pi(s)$ which outputs an action a given the current state s. Instead of directly optimizing this policy, value function based RL algorithms use state-value functions or state-action value functions. The state-action value function $Q^{\pi}(s,a)$ denotes the expected sum of discounted rewards obtained when the agent selects action a in state s and follows policy π afterwards:

$$Q^{\pi}(s,a) = E\Big(\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s, a_0 = a, \pi\Big) \tag{1}$$

Where E denotes the expectancy operator. This previous value is almost impossible to compute, because it involves an expectancy over all possible future state sequences which can be arbitrarily long. Instead, by using Bellman's equation the computation can be broken up into parts:

$$Q(s_t,a_t) = E(r_t) + \gamma \sum_{s_{t+1}} T(s_t,a_t,s_{t+1}) \max_a Q(s_{t+1},a) \tag{2}$$

For general game-playing programs the transition model is not known, extremely large, or complex to combine with function approximation techniques. Furthermore, it is much more effective to use a reinforcement learning algorithm that learns to focus on parts of the state-space which are most rewarding for the agent. In many papers about learning to play games with reinforcement learning, the Q-learning algorithm [24] is used. Q-learning updates the approximation of the Q-value of a state-action pair denoted as $\widehat{Q}(s_t,a_t)$ after an experience (s_t,a_t,r_t,s_{t+1}) by:

$$\widehat{Q}(s_t,a_t) \leftarrow \widehat{Q}(s_t,a_t) + \alpha(r_t + \gamma \max_a \widehat{Q}(s_{t+1},a) - \widehat{Q}(s_t,a_t)) \tag{3}$$

Where $0 \leq \alpha \leq 1$ denotes the learning rate. If state s_{t+1} is a terminal state (i.e. the game is over), the following update is used:

$$\widehat{Q}(s_t,a_t) \leftarrow \widehat{Q}(s_t,a_t) + \alpha(r_t - \widehat{Q}(s_t,a_t)) \tag{4}$$

Value-function Approximation Because the state space in Tron is very large (around 10^{20} different states), we need to combine Q-learning with a function approximator. For this, we use a multi-layer perceptron (MLP) that receives as input the game-state representation and outputs the Q-values of the four different actions. The multi-layer perceptron consists of a single hidden layer and

is trained with online backpropagation. After each experience (s_t, a_t, r_t, s_{t+1}) the target value for the MLP when executing action a_t in state s_t is:

$$Q^{target}(s_t, a_t) \leftarrow r_t + \gamma \max_a \widehat{Q}(s_{t+1}, a) \tag{5}$$

When a terminal state is reached, the target value is computed by only using the final reward of the game:

$$Q^{target}(s_t, a_t) \leftarrow r_t \tag{6}$$

These target values are then used by backpropagation to update the Q-value output of the selected action in the given state. Different activation functions can be used in the hidden layer of the multi-layer perceptron, while we use a linear activation unit for the output units representing the Q-values of the different actions. A commonly used activation function in the hidden layer of RL systems is the sigmoid function that transforms its weighted sum of inputs a to a value between 0 and 1:

$$O(a) = \frac{1}{1 + e^{-a}} \tag{7}$$

Another possible activation function is the exponential linear unit, which has been shown to perform better when training deep neural networks on image recognition problems [4]. We therefore compared the performance of the agent using the sigmoid function and the exponential linear unit (Elu) in the hidden layer [7]. The exponential linear unit computes the activation of the hidden units with the following equation:

$$O(a) = \begin{cases} a & \text{if } a \geq 0 \\ \beta(e^a - 1) & \text{if } a < 0 \end{cases} \tag{8}$$

Where we set β to 0.01 after performing some preliminary experiments.

2.3 State Representation

When applying Q-learning to the game of Tron, it is possible to use the entire game grid (10×10) as input for the MLP. This would translate to 100 input variables, which have a value of one whenever a position has been visited by one of the agents and zero otherwise. In order to also assure that the agent is aware of its current position in the game, we supply another 10×10 grid in which the current position of the agent is equal to one. Finally, a 10×10 grid is used in which the head of the opponent has a value of one. Therefore, this representation consists of 300 input units.

In our previous work [7], we compared the use of this full-grid information method to the use of local vision grids. The results indicated that using vision grids is a very useful method to obtain information about the relevant parts of the environment and attain high performance scores. A vision grid can be seen as

a local view of the environment taken from the position of the agent. This vision grid is a square rectangle with an uneven dimension. We used two different grid sizes: a small vision grid with an area of 3×3 and a large vision grid with a size of 5×5. To get all important information from the player and the opponent, 7 different vision grids are combined (in all these grids the standard value is zero):

- The player trail grid contains information about the locations visited by the agent itself: whenever the agent has visited the location it will have a value of one instead of zero.
- The player's opponent-trail grid contains information about the locations visited by the opponent: if the opponent or its tail is in the 'visual field' of the agent these locations are encoded with a one.
- The player's wall grid represents the walls: whenever the agent is close to a wall the wall locations will get a value of one.
- The player's opponent-head grid contains information about the current location of the head of the opponent. If the opponent's head is in the agent's visual field, this location will be encoded with a one.
- The opponent wall grid represents the walls for the opponent: whenever the opponent is close to a wall the wall locations will get a value of one.
- The opponent-trail grid contains information about the locations visited by the opponent: whenever the opponent has visited the location it will have a value of one instead of zero.
- The opponent's player grid encodes the locations visited by the player seen from the perspective of the opponent: if the player or its trail is in the 'visual field' of the opponent these locations are encoded with a one.

Because 7 vision grids are used, the total number of inputs for the small vision grid is 63 and for the large vision grid it is 175. This shows that the dimensionality of the input space is significantly reduced when using vision grids when compared to the full grid that used 300 inputs. An example game state and the seven associated (small) vision grids can be found in Figure 2.

2.4 Opponent Modelling and Monte Carlo Rollouts

The main contribution of our previous paper [7] was a novel opponent modelling technique, which allows an agent to learn a model of the opponent while at the same time learning to play the game. This learned opponent model was then used in Monte Carlo rollouts to select moves during the final test games.

In opponent modelling the task is to learn the opponent's behaviour in different states to predict what the opponent's next action will be. Opponent modelling techniques have mainly focused on imperfect information games [19,5] and are relatively problem specific. Our technique focuses on games in which the opponent's behaviour is fully observable. In our opponent modelling technique the agent learns a model of the opponent by learning to predict the opponent's next move using the same MLP as is used to learn to play the game. A possible benefit of incorporating opponent modelling in the same neural network is

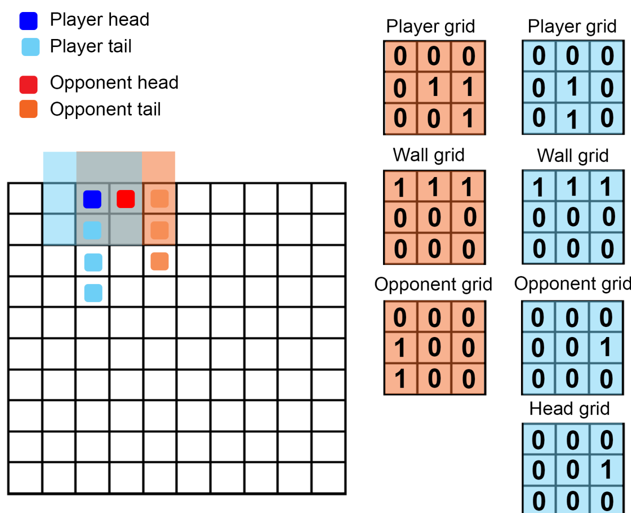

Fig. 2. An illustration of a game state and the associated values in the 7 vision grids.

that the agent might learn hidden features regarding the opponent's behaviour, which could increase performance. The fact that the model can be learned using the same network and the backpropagation algorithm [12] is a reason that the method is widely applicable, as one only needs to slightly change the structure of the neural network. In fact, this opponent modelling technique can be used in many games in which the opponent's actions are observable. Another benefit of this technique is that the agent simultaneously learns a policy and a model of the opponent and no extra training phase is needed.

After the training phase has finished the agent's model of the opponent is reflected as a probability distribution over the opponent's next moves given the current state. This probability distribution can then be used in planning algorithms such as Monte Carlo rollouts.

In order to incorporate the opponent modelling technique we need to alter the structure of the neural network used. For this, we add four (as the action space consists of four actions) output units to the network, in which we will use a softmax activation function. By doing so, the output of these added nodes represent the probability distribution over the opponent's next action given the current game state. The softmax function is used to transform the vector o containing the output modelling values for the next $K = 4$ possible actions of the opponent to values in the range $[0, 1]$ that add up to one:

$$P(s_t, o_i) = \frac{e^{o_i}}{\sum_{k=1}^{K} e^{o_k}} \tag{9}$$

After training these four output nodes with backpropagation, the output of each of these nodes represents the probability of the opponent conducting action o_i

in state s_t. When training the additional output nodes with backpropagation, we compute a target vector for the four output nodes. The target is one for the action conducted by the opponent and zero for all other actions given the previous game state. Whenever an agent follows a fully deterministic policy, this technique would allow the agent to learn to correctly predict all of the opponent's moves. Although in reality a policy is seldom entirely deterministic, players use certain rules to play a game and the agent can learn these rules with this opponent modelling technique.

So far we have explained how this opponent modelling technique can be incorporated in the neural network that is combined with Q-learning. In order to increase the performance of the agent, we can use the learned opponent model in a planning algorithm. In this research we will use the model in so-called Monte Carlo rollouts [23]. Such a rollout is used to estimate the value $\widehat{Q}_{sim}(s, a)$, the expected Q-value of performing action a in state s and subsequently performing the action suggested by the current policy for n steps. This simulated Q-value is estimated by simulating the game ahead using the opponent model to determine the opponent's moves in this simulation. The number of rollouts to determine $\widehat{Q}_{sim}(s, a)$ can vary and this determines how the opponent's action is selected. If one rollout is used the opponent's move with the highest probability is carried out. When more than one rollout is performed, the opponent's action is selected based on the probability distribution. For every game state we can use a variable amount of rollouts m with a horizon or length of n actions to determine the expected or simulated Q-value of performing all actions in the given game state. Whenever more than one rollout is used ($m > 1$) we average the obtained simulated Q-values per action.

If a game ends before the horizon is reached, the simulated Q-value $\widehat{Q}_{sim}(s_t, a_t)$ for a single rollout equals the reward obtained in the simulated game (1 for winning, 0 for a draw, and -1 for losing) properly discounted by the number of moves i until an end state is reached:

$$\widehat{Q}_{sim}(s_t, a_t) = \gamma^i r_{t+i} \tag{10}$$

If the game is not finished before reaching the rollout horizon the simulated Q-value is equal to the discounted Q-value of the last (greedy) action performed:

$$\widehat{Q}_{sim}(s_t, a_t) = \gamma^n \widehat{Q}(s_{t+n}, a_{t+n}) \tag{11}$$

See Algorithm 1 for a detailed description. This kind of rollout is also called a truncated rollout as the game is not necessarily played to its conclusion [23]. In order to determine the importance of the number of rollouts m and the length of the horizon n, we will perform different experiments with different amounts of rollouts and lengths of the horizon.

3 Learning from Monte Carlo Rollouts

In our previous paper [7], Monte Carlo rollouts using the model of the opponent were used to determine the optimal move given the current game state but they

Algorithm 1 Monte Carlo rollout with Opponent Model (taken from [7])

Input: Current game state s_t, starting action a_t, horizon N, number of rollouts M
Output: Average utility of performing action a_t at time t and subsequently following the policy over M rollouts

```
for m = 1, 2, ..M do
  i = 0
  Perform starting action a_t
  if M = 1 then
    o_t ← argmax_o P(s_t, o)
  else if M > 1 then
    o_t ← sample P(s_t, o)
  end if
  Perform opponent action o_t
  Determine reward r_{t+i}
  rolloutReward_m = r_{t+i}
  while not game over do
    i = i + 1
    a_{t+i} ← argmax_a Q(s_{t+i}, a)
    Perform action a_{t+i}
    if M = 1 then
      o_{t+i} ← argmax_o P(s_{t+i}, o)
    else if M > 1 then
      o_{t+i} ← sample P(s_{t+i}, o)
    end if
    Perform opponent action o_{t+i}
    Determine reward r_{t+i}
    if Game over then
      rolloutReward_m = γ^i r_{t+i}
    end if
    if not Game over and i = N then
      game over ← True
      rolloutReward_m = γ^N Q(s_N, a_N)
    end if
  end while
  rewardSum = rewardSum + rolloutReward_m
  m = m + 1
end for
return rewardSum/M
```

were only used in testing the final performances of the different systems. In this paper, we investigate whether it is beneficial to learn from the action estimates that are gathered in the rollouts while playing the game. In our novel approach, Monte Carlo rollouts will be used while training the RL system in two ways: for selecting actions in a game state while trainingand for computing target Q-values in a game-state. Because using rollouts to select an action requires significantly more computations, we will combine this method with experience replay [9]. This could allow the agent to use significantly fewer training games to attain the same performances as in our previous paper. For learning from the Monte Carlo rollouts, the target values of the Q-values have to be adapted. With Q-learning the target is determined by the following formula:

$$\widehat{Q}^{target}(s_t, a_t) \leftarrow r_t + \gamma \max_a \widehat{Q}(s_{t+1}, a) \tag{12}$$

Now, we will use the action estimates obtained with the rollouts as targets, as they arguably reflect the expected future reward of performing an action more accurately. If a game finishes during a rollout, the target is equal to the reward obtained in the simulated game (1 for winning, 0 for a draw, and -1 for losing) with appropriate discounting using the length i of the current rollout:

$$\widehat{Q}^{target}(s_t, a_t) = \gamma^i r_{t+i} \tag{13}$$

If the game is not finished before reaching the rollout horizon, the target Q-value is equal to the discounted Q-value of the last (greedy) action performed:

$$\widehat{Q}^{target}(s_t, a_t) = \gamma^n \widehat{Q}(s_{t+n}, a_{t+n}) \tag{14}$$

where n is the length of the horizon as before. Note that these targets are related to n-step backups [21], but here we use simulated experiences instead of real experiences. If multiple rollouts are used for selecting an action, these target values are averaged over all rollouts.

An interesting advantage of using the model of the game and the learned model of the opponent, is that estimates are collected for all actions. Therefore, instead of only training on the target of the selected action, it is also possible to train the multi-layer perceptron on estimates of all four actions obtained through the rollouts. Note that this is only possible because we have a model of the game and learn the model of the opponent, and this is quite different from most RL algorithms that only learn from the experience obtained with a single action: (s_t, a_t, r_t, s_{t+1}).

Experience replay does not only increase sample efficiency, in some cases it is needed to ensure stability (convergence to an optimal policy) in the learning process [3]. At first we implemented rollout learning without experience replay and experienced extreme difficulties when training the agents online in the roll-outs, as the Q-values often exploded. This was caused by the fact that the back-propagation algorithm [25] assumes that training samples are independent. However, with online learning in the rollouts a lot of experiences are collected

which were not independent as they were gathered sequentially. This is another reason why we used experience replay.

The implementation works as follows: at every state s_t we store the current game state s_t, the action performed a_t, the opponent's action o_t and the target Q-value computed with equations 13 or 14, in the replay memory. We will let the system play 50 games (collecting around 1000 experiences in the replay memory), and we randomly select experiences from this pool to train on. The target values of the experiences are given by the multi-layer perceptron or the rewards during the rollouts as explained before. In case multiple rollouts are used, the targets are averages of the different rollout estimates. In total, we use 10 times the size of the replay memory to draw random experiences and train the MLP. Afterwards, the replay memory is emptied and the new MLP is used to play again 50 games and so on.

4 Experiments and Results

Previous results [7] showed that vision grids increase performance in most cases, as compared to using the full grid as state representation. Furthermore, that research showed that the Elu activation function outperforms the Sigmoid activation function. In the experiments conducted in our previous research the number of training games was set to 1.5 million. Their results are shown in subsections 4.1 to 4.3. In the new experiments, shown in subsections 4.4 to 4.6, 150,000 training games are used. The agent plays against the random and semi-deterministic opponent and the number of test games is equal to 10,000. In these test games, the agent makes no exploration actions. In order to obtain meaningful results, all experiments are conducted ten times. The performance is measured as the number of games won plus 0.5 times the number of games tied. This number is divided by the number of games to get a score between 0 and 1. In all experiments we use an MLP with the weights initialised between -0.5 and 0.5. In all experiments we use one hidden layer, as preliminary experiments indicated that this led to the best results. The state representation technique determines the number of input nodes and the number of hidden nodes varies from 200 to 400.

4.1 Learning without Monte Carlo Rollouts

We performed many preliminary experiments to tune all hyperparameters. In these experiments concerning Q-learning without Monte Carlo rollouts, the number of input/hidden nodes are equal to 300/300, 175/400, and 63/200 for the full grid, large vision grids, and small vision grids respectively. In all experiments the MLP has eight output nodes, which represent the four Q-values for the different actions and the four outputs to model the opponent's probability of selecting that action. In all experiments ϵ-greedy exploration is used. In most experiments the exploration rate ϵ decreases over the first 750,000 games from 10% to 0%.

The exception to this rule is that with large vision grids and the sigmoid activation function against the random opponent, the exploration rate decreases from 10% to 0% over the first million games. The learning rate α and discount factor γ are 0.005 and 0.95 respectively. The learning rate is 0.001 when using the full grid as state representation with the sigmoid activation function and set to 0.0025 with large vision grids and the sigmoid activation function against the random opponent. These changes were made to assure stable results. We note that in this experiment no rollouts are performed. Figures 3, 4, and 5 show the training performance for the three different state representations. Table 1 shows the performance over the final 10,000 test games.

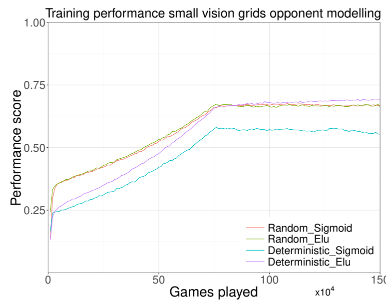

Fig. 3. Final performance score for small vision grids as state representation over 1.5 million training games with opponent modelling but without rollouts (taken from [7]).

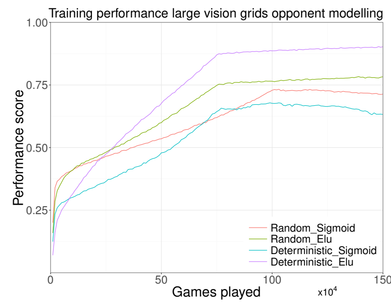

Fig. 4. Final performance score for large vision grids as state representation over 1.5 million training games with opponent modelling but without rollouts (taken from [7]).

The results show that the large vision grids with the Elu activation function obtain the best results against the two opponents. The worst results are obtained with the full-grid representation, which may need even more than 1.5 million training games to obtain good performances.

Table 1. Final performance score and standard errors with opponent modelling without rollouts.

State representation	Opponent	Sigmoid	Elu
Small vision grids	Random	0.67 (0.004)	0.67 (0.009)
Large vision grids	Random	0.72 (0.005)	**0.79 (0.003)**
Full grid	Random	0.42 (0.016)	0.40 (0.025)
Small vision grids	Deterministic	0.57 (0.015)	0.69 (0.005)
Large vision grids	Deterministic	0.63 (0.019)	**0.90 (0.003)**
Full grid	Deterministic	0.32 (0.023)	0.62 (0.015)

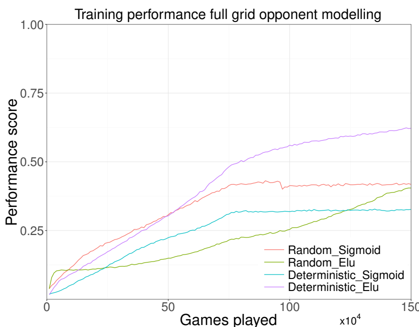

Fig. 5. Final performance score for the full grid as state representation over 1.5 million training games with opponent modelling but without rollouts (taken from [7]).

Although in these experiments, a model of the opponent is learned by the MLP at the same time as the Q-value function, the opponent model is not used during game play. Therefore, the only reason this might be beneficial is as having an auxiliary task while training the MLP. We therefore performed another experiment, where the MLP is not updated on the actions of the opponent and therefore does not learn an opponent model. Using the same setup as before, results of 10 simulations with 10,000 final test games were generated that can be found in Table 2.

Table 2. Final performance score and standard errors with the Elu activation function and opponent vision grids, but without opponent modelling (taken from [7]).

State representation	Random	Deterministic
Small vision grids	0.69 (0.008)	0.69 (0.003)
Large vision grids	**0.82 (0.009)**	**0.89 (0.003)**

From these results we can conclude that the agent did not profit from learning an opponent model as there is no clear improvement when we compare Table 1 to Table 2. The benefit of learning the opponent model is therefore researched in the next subsections, in which Monte Carlo rollouts are used that make use of the model of the game and the opponent model to generate simulated experiences before selecting an action.

4.2 Learning with Opponent Modelling and Monte Carlo Rollouts

By letting the agent learn a model of the opponent, this model can be used in Monte Carlo rollouts. The rollouts in this subsection are only used for selecting actions during the final test games. Here, the game is simulated ten steps into the future, as this was found to be the optimal amount of actions in the trade-off

between looking far into the future and assuring that the predicted actions are correct in this rollout. In order to test the effect of the amount of rollouts per state action pair, we compare the agent's performance when one and ten rollouts are conducted. Since the opponent's actions within the rollouts are determined by the learned probability distribution, we plot the prediction accuracy of the agent against both opponents in Figure 6 and 7 for the first 25,000 training games. These figures show that with the Elu activation function, which learns slightly faster than the sigmoid activation function, the agent correctly predicts 50% of the random opponent's moves and 90% of the semi-deterministic opponent's moves when we use vision grids. When the full grid is used, this accuracy is 40% and 80% respectively.

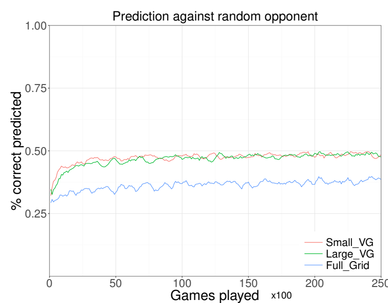

Fig. 6. Percentage of moves correctly predicted against the random opponent (taken from [7]).

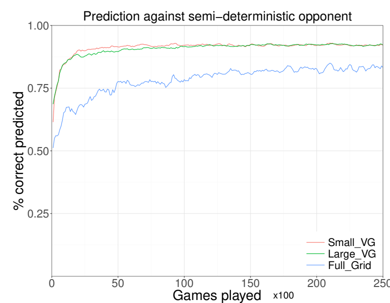

Fig. 7. Percentage of moves correctly predicted against the semi-deterministic opponent (taken from [7])

Table 3. Final performance score and standard errors with one rollout and a horizon of ten actions.

State representation	Opponent	Sigmoid	Elu
Small vision grids	Random	0.83 (0.002)	**0.84 (0.003)**
Large vision grids	Random	0.66 (0.008)	0.66 (0.004)
Full grid	Random	0.65 (0.004)	0.72 (0.007)
Small vision grids	Deterministic	0.93 (0.002)	0.96 (0.001)
Large vision grids	Deterministic	0.95 (0.002)	**0.98 (0.001)**
Full grid	Deterministic	0.54 (0.010)	0.75 (0.010)

Now we turn to the agent's performance in 10,000 test games when using the rollouts after the agent was trained for 1.5 million games as before. The performance score and standard error using one rollout with a horizon of ten steps during 10,000 test games can be found in Table 3. From the results we can con-

clude that Monte Carlo rollouts increase the agent's performance in all scenarios, except for when large vision grids are used against the random opponent. A very high performance score of 0.98 is obtained using large vision grids and the Elu activation function against the semi-deterministic opponent, which shows that using this technique performance can be increased significantly. This increase is especially large when we use vision grids and play against the semi-deterministic opponent. When the opponent employs the collision-avoiding random policy, small vision grids lead to the highest performance and when comparing these results to the previous experiments we see that Monte Carlo rollouts also increase performance against the random opponent. This shows that although the policy of the opponent is far from deterministic, opponent modelling still significantly increases performance from 0.67 to 0.83 with the sigmoid activation function and from 0.67 to 0.84 with the Elu activation function when small vision grids are used as state representation.

After applying one rollout for each action in a given state, we also tested whether increasing the number of rollouts to ten would affect the agent's performance. The results are displayed in Table 4. We find one noteworthy difference in the agent's performance when using one or ten rollouts. The agent's performance against the random opponent considerably increases when we use ten instead of one rollout. Against the semi-deterministic opponent, increasing the number of rollouts has no noticeable effect. This is because the agent predicts the semi-deterministic opponent correctly in over 90% of the cases, causing the advantage of action sampling and multiple rollouts to be absent.

Table 4. Final performance score and standard errors with ten rollouts and a horizon of ten actions.

State representation	Opponent	Sigmoid	Elu
Small vision grids	Random	0.84 (0.016)	0.88 (0.001)
Large vision grids	Random	0.90 (0.001)	**0.91 (0.001)**
Full grid	Random	0.72 (0.008)	0.74 (0.009)
Small vision grids	Deterministic	0.93 (0.002)	0.96 (0.001)
Large vision grids	Deterministic	0.96 (0.002)	**0.98 (0.001)**
Full grid	Deterministic	0.55 (0.008)	0.78 (0.010)

4.3 Monte Carlo Rollouts without using the Learned Opponent Model

We observed that the agent's performance significantly increases when we use Monte Carlo rollouts. It is however not certain that without the opponent model but with Monte Carlo rollouts this performance would be lower. Therefore, we test the agent's performance when the moves in the rollouts are determined randomly instead of by the opponent model. The results of these experiments can be found in Table 5. The relatively low performance scores indicate that it is indeed the model of the opponent that increases the agent's performance when

rollouts are used. Without a good opponent model, the results of Monte Carlo rollouts cannot be trusted for selecting an action, because the generated rollouts are not similar to how the game would actually be played.

Table 5. Final performance score and standard errors with one rollout and a horizon of ten actions without using the learned model of the opponent.

State representation	Opponent	Sigmoid	Elu
Small vision grids	Random	0.46 (0.007)	0.50 (0.001)
Large vision grids	Random	0.50 (0.001)	0.51 (0.002)
Full grid	Random	0.37 (0.010)	0.35 (0.006)
Small vision grids	Deterministic	0.34 (0.003)	0.35 (0.001)
Large vision grids	Deterministic	0.35 (0.001)	0.36 (0.001)
Full grid	Deterministic	0.21 (0.007)	0.21 (0.005)

4.4 Learning from Monte Carlo Rollouts

In this subsection, we will experiment with learning from the rollout estimates and lower the amount of training games to only 150,000. This can be done by either learning the Q-value of a single performed action using its rollout estimate or by updating the Q-values of all four actions simultaneously.

For this experiment the agent first plays 100,000 games without Monte Carlo rollouts and learns in an online manner as before. Afterwards, Monte Carlo rollouts are used for 50,000 training games in which the Q-values of the MLP are trained on the rollouts estimates (while the MLP is still trained on actions of the opponent). We want to note that during these 50,000 games experience replay is used, as explained in Section 3. Exploration decreases linearly from 10% to 0% over the first 100,000 games and the MLP now uses 400 hidden units for both the small and large vision grids. The learning rate is still set to 0.005 both for the initial 100,000 training games with online learning and for the later 50,000 training games where experience replay on the rollout estimates is used.

We again trained and tested against two opponents, but this time only using the Elu activation function and vision grids. We also compare the use of one rollout with a horizon of ten actions and five rollouts with a horizon of five actions. The previous results have shown that the first works better against the deterministic opponent, while the latter works better against the random opponent. We did not use 10 rollouts with a horizon of 10 actions, since such experiments are computationally very expensive. An experiment with the large vision grids and 5 rollouts with a horizon of 5 actions already took around 10 hours to complete on our CPUs. Note that all experiments have been conducted ten times to obtain meaningful results. The training performance against both opponents when learning only on the estimates of one action (the selected action) can be found in Figures 8 and 9 and the performance score over the 10,000 final test games can be found in Table 6.

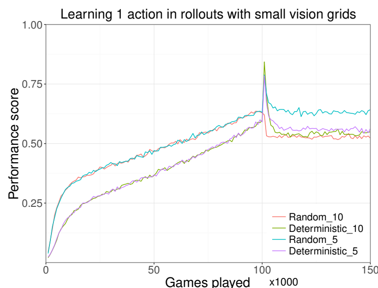

Fig. 8. Training performance with small vision grids when updating the Q-value of a single action. Random and deterministic indicate the opponent and 5 and 10 represent the horizon.

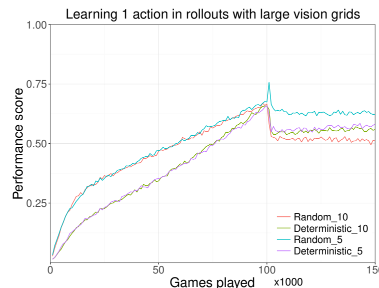

Fig. 9. Training performance with large vision grids when updating the Q-value of a single action. Random and deterministic indicate the opponent and 5 and 10 represent the horizon.

Table 6. Final test performance for 10,000 games while training the Q-value of the single performed action on its rollout estimate.

State representation	Opponent	1 Rollout 10 Actions	5 Rollouts 5 Actions
Small vision grids	Random	0.45 (0.005)	0.58 (0.006)
Large vision grids	Random	0.43 (0.006)	0.58 (0.010)
Small vision grids	Deterministic	0.55 (0.006)	0.57 (0.007)
Large vision grids	Deterministic	0.54 (0.005)	0.59 (0.008)

From the results we can conclude that when we update the Q-value of the single performed action, learning from the rollout estimates hinders performance. The peaks in figures 8 and 9 are caused by the onset of using Monte Carlo rollouts for selecting actions after 100,000 games. The system's learning dynamics seem unstable and therefore the performance immediately drops with a large amount.

As mentioned in Section 3, the targets are determined by the simulated Q-values computed with the rollouts. However, since we determine a target for every possible action, it is possible to apply back-propagation on all four Q-values rather than only for the Q-value of the performed action. We believe that by updating the Q-values of all four actions, the agent can faster learn the correct Q-values and that this therefore should increase performance. We have conducted the same experiment as above, but now by updating the Q-values of all four actions using the rollout estimates. The results of training using small

and large vision grids can be found in Figures 10 and 11. The performance over the test games can be found in Table 7.

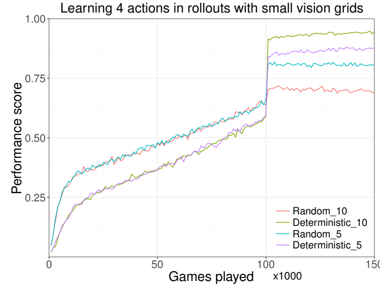

Fig. 10. Training performance with small vision grids when updating the Q-values of all actions. Random and deterministic indicate the opponent and 5 and 10 represent the horizon.

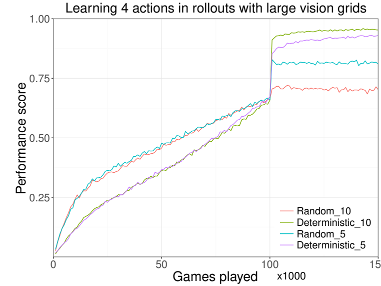

Fig. 11. Training performance with large vision grids when updating the Q-values of all actions. Random and deterministic indicate the opponent and 5 and 10 represent the horizon.

Table 7. Final test performance for 10,000 games while training Q-values of 4 actions using the rollout estimates.

State representation	Opponent	1 Rollout 10 Actions	5 Rollouts 5 Actions
Small vision grids	Random	0.74 (0.002)	0.82 (0.001)
Large vision grids	Random	0.74 (0.002)	**0.84 (0.002)**
Small vision grids	Deterministic	0.94 (0.001)	0.88 (0.002)
Large vision grids	Deterministic	**0.96 (0.001)**	0.93 (0.001)

If we compare the results from training one action and training four actions, we can conclude that it is very beneficial to train four actions simultaneously, rather than only learning the Q-value of the performed action. Now we also see in the figures that the agent's performance increases over the last 50,000 games but only against the deterministic opponent. This is most likely caused by the fact that the opponent model against this opponent is more accurate and therefore leads to more accurate targets in the rollouts. In addition, when we compare these results with the results after 1.5 million games in Table 3, where we also used one rollout with ten actions, we see that the results are slightly worse against the deterministic opponent. Against the random opponent the performance increases with large vision grids, but decreases with small vision grids. These results show that with the new proposed system good performances can already be obtained with 150,000 training games. From the results shown

in Table 7 we can also conclude that it is better to use a horizon of 10 actions against the semi-deterministic opponent.

4.5 Learning from Monte Carlo Rollouts with Exploration

In the previous experiments, during the rollout learning phase no exploration was used. In this subsection, we want to explore if it would not be better to sometimes use an exploration action instead of the action proposed by the Monte Carlo rollouts. Therefore, we conducted the same experiments as in the previous section, however, now exploration decreases from 10% to 0% over the first 100,000 games and goes from 10% to 0% over the 50,000 games played using the rollouts. The training performances can be found in Figures 12, 13, 14, and 15. The performance over the test games can be found in Tables 8 and 9.

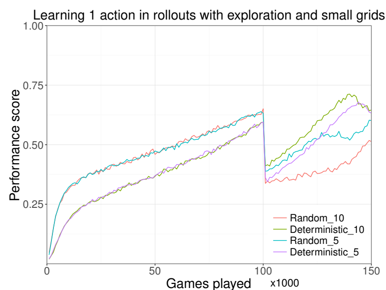

Fig. 12. Training performance with small vision grids and exploration in the rollouts when updating only the Q-value of 1 action. Random and deterministic indicate the opponent and 5 and 10 represent the horizon.

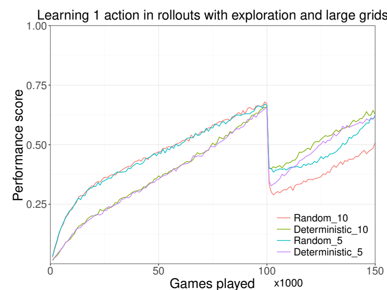

Fig. 13. Training performance with large vision grids and exploration in the rollouts when updating only the Q-value of 1 action. Random and deterministic indicate the opponent and 5 and 10 represent the horizon.

Table 8. Final test performance for 10,000 games while learning the Q-value of 1 action from the rollout estimates with exploration during the rollout training phase.

State representation	Opponent	1 Rollout 10 Actions	5 Rollouts 5 Actions
Small vision grids	Random	0.47 (0.003)	0.59 (0.011)
Large vision grids	Random	0.43 (0.006)	0.59 (0.006)
Small vision grids	Deterministic	0.63 (0.020)	0.60 (0.014)
Large vision grids	Deterministic	0.62 (0.011)	0.65 (0.009)

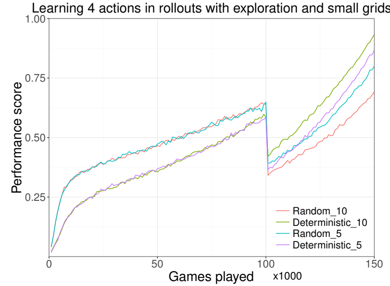

Fig. 14. Training performance with small vision grids and exploration in the rollouts when updating the Q-values of 4 actions. Random and deterministic indicate the opponent and 5 and 10 represent the horizon.

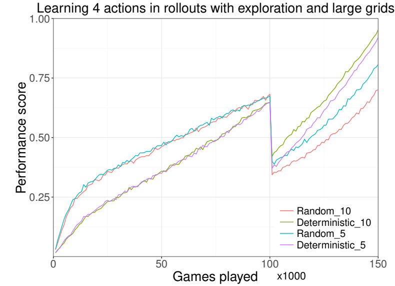

Fig. 15. Training performance with large vision grids and exploration in the rollouts when updating the Q-values of 4 actions. Random and deterministic indicate the opponent and 5 and 10 represent the horizon.

Table 9. Final test performance for 10,000 games while learning the Q-values of 4 actions from the rollout estimates with exploration during the rollout training phase.

State representation	Opponent	1 Rollout 10 Actions	5 Rollouts 5 Actions
Small vision grids	Random	0.73 (0.002)	0.83 (0.002)
Large vision grids	Random	0.74 (0.002)	**0.84 (0.002)**
Small vision grids	Deterministic	0.94 (0.001)	0.87 (0.004)
Large vision grids	Deterministic	**0.96 (0.001)**	0.93 (0.002)

From the figures it becomes clear that exploration over the last 50,000 games first leads to a drop in performance, while afterwards the performance increases strongly. When we compare the results from learning in rollouts with exploration and without exploration, we see that exploration increases performance when we train one action while using the large vision grid. When learning the Q-values of four actions simultaneously, there is no benefit of exploration. This is most likely due to the fact that with learning four actions the agent learns better Q-values for the actions it does not conduct, which reduces the benefit of exploration.

4.6 Using Experience Replay with 1-step Rollouts

In order to determine the importance of using long horizons in the rollouts, we conducted experiments in which the agent uses experience replay but only used a rollout with 1 action. Therefore the horizon in the rollouts is reduced to 1 action,

where the agent tries all possible actions and the model of the opponent is used to predict the next state. In these experiments we only perform one rollout per state during training. When playing the final test games, the agent still uses 1 rollout with a horizon of 10 actions. The exploration strategy and all other hyperparameters are the same as before. The results can be found in Table 10.

Table 10. Final test performance for 10,000 games with 1-step rollouts

State representation	Opponent	4 actions	1 action
Small vision grids	Random	0.73 (0.003)	0.47 (0.008)
Large vision grids	Random	**0.73 (0.001)**	0.46 (0.012)
Small vision grids	Deterministic	0.94 (0.001)	0.65 (0.007)
Large vision grids	Deterministic	**0.95 (0.001**)	0.65 (0.013)

When we compare Table 10 to Tables 8 and 9 with 1 rollout, we notice no significant differences. Therefore, we can conclude that learning from the estimates of long rollouts is not necessary and actually costs more computing power. This is similar to learning from n-step backups [21] which can lead to a lower bias but suffers from a higher variance and is therefore not always fruitful.

Learning on the estimates of all actions using the Monte Carlo rollouts is however always much better than learning only on the estimate of a single action. Experience replay is general is always used to update the Q-value of a single action, because that action was the only one that was experienced in a state. However, with the model of the game and the opponent model, we have shown that experience replay can be improved by updating the Q-values of all actions simultaneously. This led to similar results as in our previous paper [7], but now by only learning from 150,000 games instead of from 1.5 million games.

5 Conclusion

This paper described a novel approach to learning to play the game of Tron. The proposed method combines reinforcement learning, multi-layer perceptrons, vision grids, opponent modelling and Monte Carlo rollouts in a novel way. Instead of only using Monte Carlo rollouts while playing test games, the new system uses Monte Carlo rollouts to select moves while training and learns from the estimates obtained with the Monte Carlo rollouts. We have extended the use of experience replay to make it possible to update the state-action values of all actions in a state simultaneously. This is possible due to the Monte Carlo rollouts that generate action estimates for all actions.

The results showed that the use of the novel experience replay method that updates all action values simultaneously strongly outperforms experience replay where only the performed action gets its action value updated. One reason is that experience replay on a single selected action requires a lot of exploration to compare the utilities of different actions in the same state. Furthermore, with

the new method all action values are updated in the same state. The proposed system is able to perform similarly after training on 150,000 games to our previous system [7], which needed 1.5 million training games to obtain very good performances. The results also showed that while training the system, longer horizons in the Monte Carlo rollouts were not necessary to obtain good results, as with a horizon of a single action similar performances were obtained. Learning from longer horizons may reduce the bias due to bootstrapping, but also suffers from a higher variance. Against the semi-deterministic opponent the RL system profits from Monte Carlo rollouts with longer horizons, whereas against the random opponent the system profits from multiple rollouts.

This research opens up several interesting possibilities for future research. Instead of Monte Carlo rollouts, it would be interesting to also combine Monte Carlo tree search with opponent-model learning. Furthermore, from the different rollouts much more information is obtained than currently used for updating the state-action value function. It would be possible to update action values of each game state that was visited during one of the rollouts. Another direction is to combine the power of the vision grids with convolutional neural networks (CNNs). The vision grids summarize the most important local information, but lack more global information. By combining the vision grids with CNNs it should be possible to profit from the faster learning process using the local information, while still being able to integrate important global information. Finally, it would be interesting to examine if our extended experience replay algorithm that trains on simulated experiences of all actions in a given state would also be useful for learning to play other games.

References

1. Baxter, J., Tridgell, A., Weaver, L.: Learning to play chess using temporal differences. Machine Learning **40**(3), 243–263 (2000)
2. Bom, L., Henken, R., Wiering, M.: Reinforcement learning to train Ms. Pac-Man using higher-order action-relative inputs. In: 2013 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL). pp. 156–163 (2013)
3. de Bruin, T., Kober, J., Tuyls, K., Babuška, R.: The importance of experience replay database composition in deep reinforcement learning. In: Deep Reinforcement Learning Workshop, NIPS (2015)
4. Clevert, D., Unterthiner, T., Hochreiter, S.: Fast and accurate deep network learning by exponential linear units (elus). CoRR **abs/1511.07289** (2015)
5. Ganzfried, S., Sandholm, T.: Game theory-based opponent modeling in large imperfect-information games. In: the 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 2. pp. 533–540. International Foundation for Autonomous Agents and Multiagent Systems (2011)
6. He, H., Boyd-Graber, J.L., Kwok, K., III, H.D.: Opponent modeling in deep reinforcement learning. CoRR **abs/1609.05559** (2016)
7. Knegt, S., Drugan, M., Wiering, M.: Opponent modelling in the game of Tron using reinforcement learning. In: ICAART 2018: 10th International Conference on Agents and Artificial Intelligence. pp. 29–40 (2018)

8. Kocsis, L., Szepesvári, C.: Bandit based Monte-Carlo planning. In: Fürnkranz, J., Scheffer, T., Spiliopoulou, M. (eds.) Machine Learning: ECML 2006. pp. 282–293. Springer Berlin Heidelberg (2006)
9. Lin, L.J.: Reinforcement Learning for Robots Using Neural Networks. Ph.D. thesis, Carnegie Mellon University, Pittsburgh (January 1993)
10. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M.: Playing atari with deep reinforcement learning. arXiv preprint arXiv:1312.5602 (2013)
11. van Otterlo, M., Wiering, M.: Reinforcement learning and Markov decision processes. In: Wiering, M., van Otterlo, M. (eds.) Reinforcement Learning: State-of-the-Art, pp. 3–42. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
12. Rumelhart, D.E., Hinton, G.E., Williams, R.J.: Learning internal representations by error propagation. In: Parallel Distributed Processing, vol. 1, pp. 318–362. MIT Press (1986)
13. Samuel, A.L.: Some studies in machine learning using the game of checkers. IBM Journal on Research and Development **3**, 210–229 (1959)
14. Schmidhuber, J.: Deep learning in neural networks: An overview. Neural Networks **61**, 85–117 (2015)
15. Shantia, A., Begue, E., Wiering, M.: Connectionist reinforcement learning for intelligent unit micro management in Starcraft. In: Neural Networks (IJCNN), The 2011 International Joint Conference on. pp. 1794–1801. IEEE (2011)
16. Silver, D., Huang, A., Maddison, C.J., Guez, A., Sifre, L., van den Driessche, G., Schrittwieser, J., Antonoglou, I., Panneershelvam, V., Lanctot, M., Dieleman, S., Grewe, D., Nham, J., Kalchbrenner, N., Sutskever, I., Lillicrap, T., Leach, M., Kavukcuoglu, K., Graepel, T., Hassabis, D.: Mastering the game of Go with deep neural networks and tree search. Nature **529**(7587), 484–489 (2016)
17. Silver, D., Hubert, T., Schrittwieser, J., Antonoglou, I., Lai, M., Guez, A., Lanctot, M., Sifre, L., Kumaran, D., Graepel, T., Lillicrap, T., Simonyan, K., Hassabis, D.: Mastering chess and shogi by self-play with a general reinforcement learning algorithm. arXiv preprint arXiv:1712.01815 (2017)
18. Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., van den Driessche, G., Graepel, T., Hassabis, D.: Mastering the game of go without human knowledge. Nature **550**, 354 (Oct 2017)
19. Southey, F., Bowling, M., Larson, B., Piccione, C., Burch, N., Billings, D., Rayner, C.: Bayes bluff: Opponent modelling in poker. In: Proceedings of the 21st Annual Conference on Uncertainty in Artificial Intelligence (UAI). pp. 550–558 (2005)
20. Sutton, R.S.: Learning to predict by the methods of temporal differences. Machine Learning **3**(1), 9–44 (1988)
21. Sutton, R.S., Barto, A.G.: Introduction to Reinforcement Learning. MIT Press, Cambridge, MA, USA, 1st edn. (1998)
22. Tesauro, G.: Temporal difference learning and TD-gammon. Commun. ACM **38**(3), 58–68 (1995)
23. Tesauro, G., Galperin, G.R.: On-line policy improvement using Monte-Carlo search. In: Jordan, M.I., Petsche, T. (eds.) Advances in Neural Information Processing Systems 9, pp. 1068–1074. MIT Press (1997)
24. Watkins, C.J., Dayan, P.: Q-learning. Machine learning **8**(3), 279–292 (1992)
25. Werbos, P.J.: Beyond Regression: New Tools for Prediction and Analysis in the Behavioral Sciences. Ph.D. thesis, Harvard University (1974)