Opponent Modelling in the Game of Tron using Reinforcement Learning

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*
[2] *ITLearns.Online, The Netherlands*
stefanknegt@gmail.com, madalina.drugan@gmail.com, m.a.wiering@rug.nl

Keywords: Reinforcement Learning, Opponent Modelling, Q-Learning, Computer Games

Abstract: In this paper we propose the use of vision grids as state representation to learn to play the game Tron using neural networks and reinforcement learning. This approach speeds up learning by significantly reducing the number of unique states. Furthermore, we introduce a novel opponent modelling technique, which is used to predict the opponent's next move. The learned model of the opponent is subsequently used in Monte-Carlo roll-outs, in which the game is simulated n-steps ahead in order to determine the expected value of conducting a certain action. Finally, we compare the performance using two different activation functions in the multi-layer perceptron, namely the sigmoid and exponential linear unit (Elu). The results show that the Elu activation function outperforms the sigmoid activation function in most cases. Furthermore, vision grids significantly increase learning speed and in most cases this also increases the agent's performance compared to when the full grid is used as state representation. Finally, the opponent modelling technique allows the agent to learn a predictive model of the opponent's actions, which in combination with Monte-Carlo roll-outs significantly increases the agent's performance.

1 INTRODUCTION

Reinforcement learning algorithms allow an agent to learn from its environment and thereby optimise its behaviour (Sutton and Barto, 1998). Such environments can be modelled as a Markov Decision Process (MDP) (van Otterlo and Wiering, 2012; Bellman, 1957), where an agent tries to learn an optimal policy from trial and error. Reinforcement learning algorithms have been widely applied in the area of games. A well-known example is backgammon (Tesauro, 1995), where reinforcement learning has led to great success. This paper examines the effectiveness of reinforcement learning for the game of Tron. One of the main challenges of using reinforcement learning in games is the large size of the state space. Another challenge is how an agent can learn to model its opponent effectively and use this opponent's model to significantly increase its performance.

To deal with large state spaces, in many cases the agent is constructed using a multi-layer perceptron (MLP) (Rumelhart et al., 1988). The MLP will receive the current game state as its input and has to determine the move that will result in the highest reward in the long term. The combination of an MLP and reinforcement learning has showed promising results, for instance in Backgammon (Tesauro, 1995), Ms. PacMan (Bom et al., 2013) and Starcraft (Shantia et al., 2011). Furthermore, deep reinforcement learning using neural networks with many layers have also obtained impressive results on a variety of games (Mnih et al., 2013).

In most research on learning to play games with connectionist reinforcement learning, the MLP uses only the well-known sigmoid activation function. However, there are other choices such as the exponential linear unit (Elu). The exponential linear unit has three advantages compared to the sigmoid function (Clevert et al., 2015). It alleviates the vanishing gradient problem by its identity for positive values, it can return negative values which might improve learning, and it is better able to deal with a large number of inputs. This activation function has shown to outperform the ReLU in a convolutional neural network on the ImageNet dataset (Clevert et al., 2015). Another way to deal with large state spaces is to give the agent a partial view of the environment. If we look at how humans play the game Tron we see that they mainly focus their attention around the current position of the agent. Therefore, vision grids (Shantia et al., 2011) can be useful. A vision grid can be seen as a snapshot of the environment from the agent's point of view. An example could be a three by three square around the 'head' of the agent. By using a

vision grid of an appropriate size, the agent can acquire the most important information about the dynamic state of the environment. Not only does this dramatically decrease the number of unique states, it also reduces the amount of irrelevant information, which can speed up the learning process of the agent.

For most game research, the agent does not learn an explicit opponent model. In most cases, roll-outs or lookahead strategies are used that select opponent's actions according to how the agent itself would select actions or according to simple rules. Although roll-outs have shown to substantially increase performance in games such as Backgammon (Tesauro and Galperin, 1997), Go (Bouzy and Helmstetter, 2004; Silver et al., 2016a), and Scrabble (Sheppard, 2002), the disadvantage of this approach is that particular weaknesses of the opponent cannot be exploited, as no true model of how the opponent selects actions is used. Opponent modelling has been studied for imperfect-information games such as poker (Ganzfried and Sandholm, 2011; Southey et al., 2005). Furthermore, in combination with Q-learning (Watkins and Dayan, 1992) it has proven to lead to better performances (He et al., 2016). However, as noted by (Collins, 2007), the learned models are often environment specific and take considerable effort to learn. As a solution to this problem, Mealing (Mealing, 2015) proposed a dynamic opponent modelling variant, which uses sequence prediction to learn high rewarding strategies.

Contributions: In this paper, we developed different state representations for the game of Tron. We show that with vision grids we can reduce the number of unique states, which helps overcoming the challenge of using reinforcement learning in problems with large state spaces. We use the information from the vision grids as input for a multi-layer perceptron that is trained using a reinforcement learning algorithm. Next to using the common sigmoid function in the hidden layer of the MLP, we will also use the Elu activation function and compare the results of both activation functions. The most important contribution of this paper is a novel opponent modelling technique. In our proposed algorithm, the agent learns the opponent's behaviour by predicting the next move of the opponent, observing the result, and adjusting the neural network's parameters based on this observation. If the opponent is following a policy, the agent should be able to learn this policy over time. This model of the opponent is subsequently used in Monte-Carlo roll-outs. In such a roll-out the game is simulated n steps ahead in order to determine the expected value of performing action a in state s and subsequently executing the action that is associated with the highest Q-value in each state. In these roll-outs, the learned opponent model is used to select actions for the opponent. The roll-outs are performed multiple times and the results are averaged. We performed many different experiments to compare all methods (3 state representations, sigmoid / Elu, opponent model / no opponent model, different numbers of roll-outs). From the results we can conclude that vision grids are effective for faster training and better final performances. Furthermore, when we combine the vision grids with opponent modelling and roll-outs, the performances are very good, reaching very high scores against 2 different fixed opponents.

Outline: In the next section we explain the framework that was built to simulate the game and agent. Section 3 describes reinforcement learning combined with multi-layer perceptrons. In Section 4, we explain the use of vision grids for Tron and the novel opponent modelling technique. Then in section 5 we describe the experiments and show their results. Finally, in section 6 we present our conclusions and possible future work.

2 THE GAME OF TRON

Tron is an arcade video game released in 1982 and was 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 player itself. See Figure 1 for a graphical depiction of the game. We developed a framework that implements the game of Tron as a sequential decision problem where each agent selects an action for each new game state. In this research the game is played with two players. The environment is represented by a 10 by 10 grid in which the player starts at a random location in the top half of the grid and the opponent in the bottom half. After that, both players decide on an action to carry out. The action space consists of the four directions the agents can move in. When the action selection phase is completed, both actions get carried out and the new game state is evaluated. In case both agents move to the same location, the game ends in a draw. A player loses if it moves to a location that is previously visited by either itself or the opponent or when the agent wants to move to a location outside of the grid. If it happens that both agents lose at the same moment, the game counts as a draw. We estimate the number of possible different states in the game to be of the order 10^{20}, which is similar to the game Othello that consists of a board of 7×7 cells.

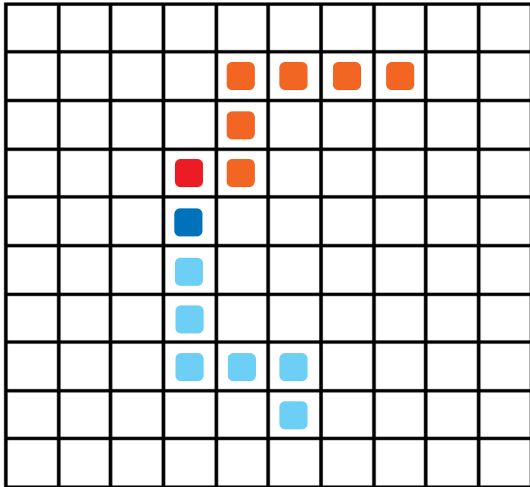

Figure 1: Tron game environment with two agents, where their heads or current location are in a darker colour.

For the opponent we used two different implementations. Both fixed opponents always first check whether their intended move is possible and therefore will never lose unless they are fully enclosed. The first agent randomly chooses an action from the possible actions, while the second agent always tries to execute its previous action again. If this is not possible, the opponent randomly chooses an action that is possible and keeps repeating that action. This implies that this opponent only changes its action when it encounters a wall, the opponent or its own tail. This strategy is very effective in the game of Tron, because it is very efficient in the use of free space and it makes the agent less likely to enclose itself. We tested these opponents by letting them play against each other, and observed that the opponent employing the strategy of going straight as long as possible only loses 25% of the games and 20% of the games end in a draw. From here on we will refer to the agent employing the collision-avoiding random policy as the random opponent and the other opponent will be referred to as the semi-deterministic opponent.

3 REINFORCEMENT LEARNING

When the agent starts playing the game, it will randomly choose actions from its action space. In order to improve its performance, the agent has to learn the best action in a given game state and therefore we train the agent using reinforcement learning. Reinforcement learning is a learning method in which the agent learns to select the optimal action based on in-game rewards. Whenever the agent loses a game it receives a negative reward or punishment and if it wins it will receive a positive reward. As it plays a large number of games, the agent should learn to select the action that leads to the highest possible expected reward given the current game state. Reinforcement learning techniques are often applied to environments that can be modelled as a so-called Markov Decision Process (MDP) (Bellman, 1957). An MDP is defined by the following components:

- A finite set of states S, where $s_t \in S$ is the state at time t.
- A finite set of actions A, where $a_t \in A$ is the action executed at time t.
- A transition function $T(s,a,s')$. This function specifies the probability of ending up in state s' after executing action a in state s. Whenever the environment is fully deterministic, we can ignore the transition probability. This is not the case in the game of Tron, since it is played against an opponent for which we cannot perfectly anticipate its next move.
- A reward function $R(s,a,s')$, which specifies the reward for executing action a in state s and subsequently going to state s'. In our framework, the reward is equal to 1 for a win, 0 for a draw, and -1 in case the agent loses. Note that there are no intermediate rewards.
- A discount factor γ to discount future rewards, where $0 \leq \gamma \leq 1$.

To let the agent act in this MDP, we need a mapping from states to actions. This is given by the policy $\pi(s)$, which returns for any state s the action to perform. The value of a policy is given by the sum of the discounted future rewards starting in a state s following the policy π:

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

Where r_t is the reward received at time t. The value function gives the expected outcome of the game if both players select the actions given by their policy. The value of a state is the long-term reward the agent will receive, while the reward of a state is only short-term. Therefore, the agent has to choose the state with the highest possible value. We can rewrite equation 1 in terms of the components of an MDP:

$$V^{\pi}(s) = \sum_{s'} T(s,\pi(s),s')(R(s,\pi(s),s') + \gamma V^{\pi}(s')) \tag{2}$$

From equation 2 we see that the value of a particular state s depends on the transition function, the

probability of going to state s' times the reward obtained in this new state s' and the value of the next state times the discount factor. In practice, the transition function is often unknown and therefore we have to use a reinforcement learning algorithm. Next, we will look at the particular reinforcement learning algorithm employed in this research: Q-learning.

3.1 Q-learning

In this research we will be using Q-learning (Watkins and Dayan, 1992), for which the value of a state becomes a Q-value of a state-action pair, $Q(s,a)$, which gives the value of performing action a in state s. This Q-value for a given policy is given by equation 3.

$$Q^{\pi}(s,a) = E\Big(\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s, a_0 = a, \pi\Big) \quad (3)$$

The value of performing action a in state s is the expected sum of the discounted future rewards following policy π. The Q-value of an individual state-action pair is given by:

$$Q(s_t,a_t) = E(r_t) + \gamma \sum_{s_{t+1}} T(s_t,a_t,s_{t+1}) \max_a Q(s_{t+1},a) \quad (4)$$

The Q-value of a state-action pair depends on the expected reward and the highest Q-value in the next state. However, we do not know s_{t+1} as it depends on the action of the opponent. Therefore, Q-learning keeps a running average of the Q-value of a certain state-action pair. The Q-learning algorithm is given by:

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

Where $0 \leq \alpha \leq 1$ denotes the learning rate. As we encounter the same state-action pair multiple times, we update the Q-value to find the average Q-value of this state-action pair. This kind of learning is called temporal-difference learning (Sutton, 1988).

3.2 Function Approximator

Whenever the state space is relatively small, one can easily store the Q-values for all state-action pairs in a lookup table. However, since the state space in the game of Tron is far from small the use of a lookup table is not feasible in this research. In addition, since there are many different states it could happen that even after training, some states have not been encountered before. When a state has not been encountered before, action selection happens without information from experience. Therefore, we use a neural network as function approximator. To be more precise, we will be using a multi-layer perceptron (MLP) to estimate $Q(s,a)$. This MLP will receive as input the current game state s and its output will be the Q-value for each action given the input state. One could also choose to use four different MLPs, which output one Q-value each (one for every action). We have tested both set-ups and there was a small advantage of using a single action neural network. The neural network is trained using back-propagation (Rumelhart et al., 1988), where the target Q-value is calculated using equation 5. As a simplification we set the learning rate α in this equation equal to 1, because the back-propagation algorithm of the neural network already contains a learning rate, which controls for the speed of learning. The target Q-value for action a_t in state s_t is therefore:

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

This target is valid as long as the action taken in the state-action pair does not result in the end of the game. Whenever that is the case, the target Q-value is equal to the first term of the right-hand side of equation 5, the reward received in the final game:

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

3.2.1 Activation function

In order to allow the neural network's value function approximation to be non-linear we use an activation function in the hidden layer. One of the most often used activation functions is the sigmoid function:

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

This function transforms the weighted sum of inputs for a hidden unit to a value between 0 and 1. Recently, it has been proposed that the exponential linear unit performs better in some domains (Clevert et al., 2015). We will compare the performance of the agent using the sigmoid function and the exponential linear unit (Elu) in the hidden layer. The exponential linear unit is given by the following equation:

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

We set β equal to 0.01 after some preliminary experiments. This function transforms negative activations to a small negative value, while positive activation is unaffected. We will compare the performance of the agent with both activation functions to determine which performs better for learning to play Tron.

4 STATE REPRESENTATION AND OPPONENT MODELS

In this section, we will first describe the different state representations that will be used by the agent. Then, we will describe how a model of the opponent can be learned and used for selecting actions using roll-outs.

4.1 Vision Grids

The first state representation used as input to the MLP is the entire game grid (10×10). This translates to 100 input nodes, which have a value of one whenever it is visited by one of the agents and zero otherwise. Another 10 by 10 grid is fed into the network, but this time only the current position of the agent has a value of one. This input allows the agent to know its own current position within the environment. The second type of state representation and input to the MLP that will be tested are vision grids. A vision grid can be seen as a snapshot of the environment taken from the point of view of the agent. This translates to a square grid with an uneven dimension centred around the head of the agent. To receive the most relevant information from the state of the game, three different types of vision grids are combined (in all these grids the standard value is zero):

- The player 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 opponent grid contains information about the locations visited by the opponent: if the opponent is in the 'visual field' of the agent these locations are encoded with a one.
- The wall grid represents the walls: whenever the agent is close to a wall the wall locations will get a value of one.

An example game state and the three associated vision grids can be found in Figure 2. We will test vision grids with a size of three by three (small vision grids) and five by five (large vision grids) and compare the performance of the agents with these small and large vision grids to an agent that receives all information from the game state.

4.2 Opponent Modelling

This paper introduces an opponent modelling technique with which a model of the opponent is learned from observations. This model can subsequently be used in planning algorithms such as Monte-Carlo roll-outs. Planning is one of the key challenges of artificial intelligence (Silver et al., 2016b). Many opponent modelling techniques focus on probabilistic models and imperfect-information games (Southey et al., 2005; Ganzfried and Sandholm, 2011), which makes them very problem specific. Our novel opponent modelling technique predicts the opponent's action using the multi-layer perceptron and learns from the observed actions using the back-propagation algorithm (Rumelhart et al., 1988). Over time the agent learns to model which action the opponent will likely select when it is in a specific state. The model is a probability distribution of the opponent's next move given the state representation. Because of its simplicity, this technique can be generalised to any setting 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, which means that no extra phase is needed for the learning process. In addition, the opponent modelling happens with the same neural network that calculates the Q-values for the agent. This might allow the agent to learn hidden features regarding the opponent's behaviour, which could further increase performance.

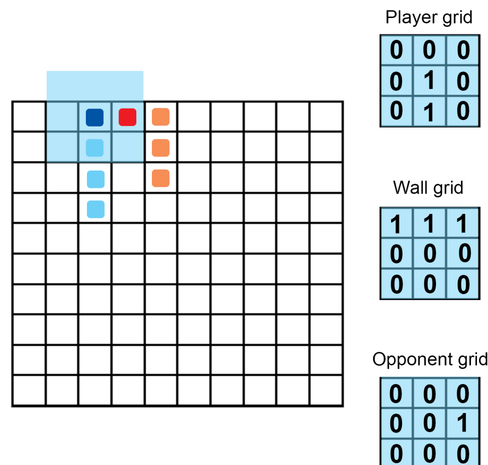

Figure 2: Vision grid example with the current location of both players in a darker color.

For modelling the opponent, four output nodes are appended to the network, which represent the probability distribution over the opponent's possible actions. The output can be interpreted as a probability distribution, because we use a softmax layer over the four appended output nodes. The softmax function transforms 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:

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

This transforms the output values to the probability of the opponent conducting action o_i in state s_t. In addition to these four extra output nodes, the state representation for the neural network changes when modelling the opponent. In the case of the standard input representation by the full grid, an extra grid is added where the head of the opponent has a value of one. In the case of vision grids, an extra 4 vision grids are constructed. The first three are the same as before, but then from the opponent's point of view. In addition, an opponent-head grid is constructed which 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.

In order to learn the opponent's policy, the network is trained using back-propagation where the target vector is one for the action taken by the opponent and zero for all other actions. If the opponent is following a deterministic policy, this allows the agent to perfectly forecast the opponent's next move after sufficient training. Although in reality a policy is seldom entirely deterministic, players use certain rules to play a game. Therefore, our semi-deterministic agent is a perfect example to test opponent modelling against.

Once the agent has learned the opponent's policy, its prediction about the opponent's next move will be used in so-called Monte Carlo roll-outs (Tesauro and Galperin, 1997). Such a roll-out is used to estimate the value $Q_{sim}(s,a)$, the expected Q-value of performing action a in state s and subsequently performing the action suggested by the current policy for $n-1$ steps. The opponent's actions are selected on the basis of the agent's model of the agent. If one roll-out is used the opponent's move with the highest probability is carried out. When more than one roll-out is performed, the opponent's action is selected based on the probability distribution. At every action selection moment in the game m roll-outs of length n are performed and the results are averaged. The expected Q-value is equal to the reward obtained in the simulated game (1 for winning, 0 for a draw, and -1 for losing) times the discount factor to the power of the number of moves conducted in this roll-out i:

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

If the game is not finished before reaching the roll-out horizon the simulated Q-value is equal to the discounted Q-value of the last action performed:

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

See algorithm 1 for a detailed description.

This kind of roll-out is also called a truncated roll-out as the game is not necessarily played to its conclusion (Tesauro and Galperin, 1997). In order to determine the importance of the number of roll-outs m, we will compare the performance of the agent with one roll-out and ten roll-outs.

Algorithm 1 Monte-Carlo Roll-out with Opponent Model

Input: Current game state s_t, starting action a_t, horizon N, number of roll-outs M
Output: Average reward of performing action a_t at time t and subsequently following the policy over M roll-outs
for $m = 1, 2, ..M$ **do**
  $i = 0$
  Perform starting action a_t
  if $M = 1$ **then**
    $o_t \leftarrow argmax_o P(s_t, o)$
  else if $M > 1$ **then**
    $o_t \leftarrow sample\ P(s_t, o)$
  end if
  Perform opponent action o_t
  Determine reward r_{t+i}
  $rolloutReward_m = \gamma r_{t+i}$
  while not game over **do**
    $i = i + 1$
    $a_{t+i} \leftarrow argmax_a Q(s_{t+i}, a)$
    Perform action a_{t+i}
    if $M = 1$ **then**
      $o_{t+i} \leftarrow argmax_o P(s_{t+i}, o)$
    else if $M > 1$ **then**
      $o_{t+i} \leftarrow sample\ P(s_{t+i}, o)$
    end if
    Perform opponent action o_{t+i}
    Determine reward r_{t+i}
    if Game over **then**
      $rolloutReward_m = \gamma^{i} r_{t+i}$
    end if
    if not Game over and $i = N$ **then**
      game over $\leftarrow$ True
      $rolloutReward_m = \gamma^{N} Q(s_N, a_N)$
    end if
  end while
  $rewardSum = rewardSum + rolloutReward_m$
  $m = m + 1$
end for
return $rewardSum/M$

5 EXPERIMENTS AND RESULTS

To compare the different state representations, the use of different activation functions in the MLP and the usefulness of the opponent modelling technique and roll-outs, many different experiments have been conducted. In all experiments the agent is trained for 1.5 million games against two different opponents, which lasts for around one day for one simulation. After that, 10,000 test games are played. In these test games, the agent makes no explorative actions. In order to obtain meaningful results, all experiments are conducted ten times and the results are averaged. 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. This is a common performance score for games.

With the use of different game state representations as input to the MLP, the number of input nodes varies. The number of hidden nodes varies from 100 to 300 and is chosen such that the number of hidden nodes is at least equal but preferably larger than the number of input nodes. This was found to be optimal in the trade-off between representation power and complexity. Also, the use of several hidden layers has been tested, but this did not significantly improve performance and we therefore chose to use only one hidden layer.

5.1 State Representations

For setting all hyper-parameters of the different algorithms, we ran many preliminary experiments. In the first part of this research, without opponent modelling, the number of input nodes for the full grid is equal to 200 and the number of hidden nodes is 300. When vision grids are used, the number of input nodes decreases to 27 and 75 for vision grids with a dimension of three by three and five by five respectively. The number of hidden nodes when using small vision grids is equal to 100, while for large vision grids 200 hidden nodes are used. In all these cases the number of output nodes is four.

During training, exploration decreases linearly from 10% to 0% over the first 750,000 games after which the agent always performs the action with the highest Q-value. This exploration strategy has been selected after performing preliminary experiments with several different exploration strategies. There is one exception to this exploration strategy. When large vision grids are used against the semi-deterministic opponent, exploration decreases from 10% to 0% over the 1.5 million training games. In this condition the exploration policy is different, because the standard exploration settings led to unstable results. The learning rate α and discount factor γ are 0.005 and 0.95 respectively and are equal across all conditions except for one. These values have been selected after conducting preliminary experiments with different learning rates and discount factors. When the full grid is used as state representation and the agent plays against the random opponent, the learning rate α is set to 0.001. The learning rate is lowered for this condition, because a learning rate of 0.005 led to unstable results. All weights and biases of the network are randomly initialised between -0.5 and 0.5.

In Figure 3, 4, and 5 the performance score during training is displayed for the three different state representations. In every figure we see the performance of the agent against the random and semi-deterministic opponent with both the sigmoid and Elu activation function. For every 10,000 games played we plot the performance score, which ranges from 0 to 1. We see that for all three state representations performance increases strongly as long as some explorative actions are made. When exploration stops at 750,000 games, performance stays approximately the same, except for the full grid state representation with the Elu activation function against the semi-deterministic opponent. We have also experimented with a constant exploration of 10% and with exploration gradually falling to 0% over all training games, however this did not lead to better performances. After training the agent, we tested the agent's performance on 10,000 test games. The results are displayed in Table 1 and 2. These results are gathered from ten independent trials, for which also the standard error is reported.

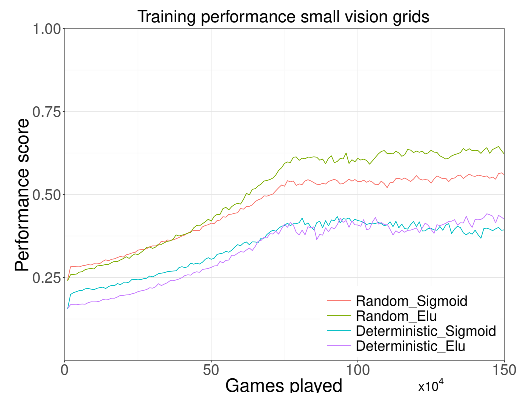

Figure 3: Performance score for small vision grids as state representation over 1.5 million training games. Note that after 750,000 games the agent stops performing exploration moves.

From Table 1 and 2 we can conclude that with the

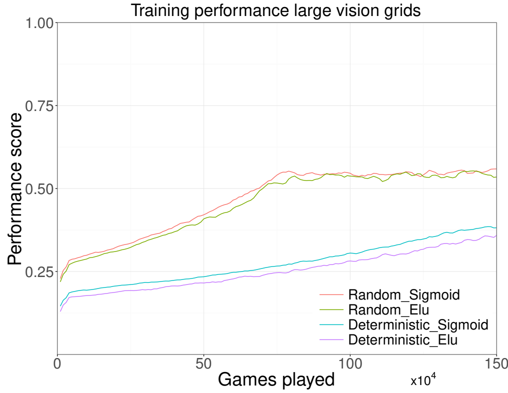

Figure 4: Performance score for large vision grids as state representation over 1.5 million training games.

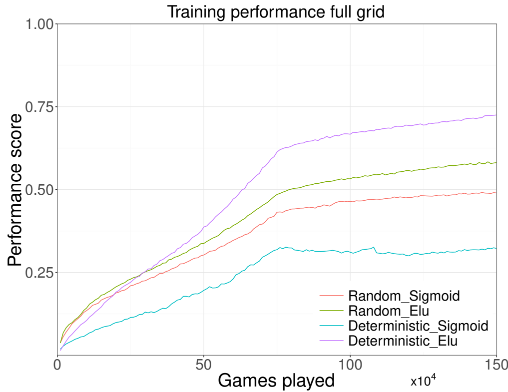

Figure 5: Performance score for the full grid as state representation over 1.5 million training games.

sigmoid activation function, the use of vision grids increases the performance of the agent when compared to using the full grid. Against the random opponent, the small vision grid with the Elu activation function performs best. Striking is the performance of the agent using the full grid against the semi-deterministic opponent using the Elu activation function, which can be found in Table 2. The agent reaches a performance score of 0.72 in this case, which is the highest performance score obtained. This finding might be caused by the fact that the agent can actually profit from the semi-deterministic policy the opponent is following, which it detects when the full grid is used as state representation because it provides more information about the past moves of the opponent. Against both opponents, the use of the Elu activation function with the full-grid representation performs significantly better than the sigmoid function.

Table 1: Performance score and standard errors against the random opponent.

State representation	Sigmoid	Elu
Small vision grids	0.56 (0.037)	**0.62 (0.019)**
Large vision grids	0.54 (0.036)	0.53 (0.022)
Full grid	0.49 (0.017)	0.58 (0.025)

Table 2: Performance score and standard errors against the deterministic opponent.

State representation	Sigmoid	Elu
Small vision grids	0.35 (0.044)	0.39 (0.016)
Large vision grids	0.37 (0.034)	0.39 (0.025)
Full grid	0.31 (0.023)	**0.72 (0.007)**

5.2 Opponent Modelling without Monte-Carlo Roll-outs

Opponent modelling requires information not only about the agent's current position, but also about the opponent's position. As explained in section 4, this increases the number of vision grids used and therefore affects the number of inputs and best found number of hidden nodes of the MLP. In the basic case where the full grid is used, the number of input nodes increases to 300 and the number of hidden nodes stays 300. For the large vision grids the number of input nodes increases to 175 and the number of hidden nodes increases to 300. Finally, when using the small vision grids the number of input nodes becomes 63 and the number of hidden nodes increases to 200. In all networks with opponent modelling the number of output nodes is eight (the 4 Q-values for the different actions and the 4 outputs to model the opponent's probability of selecting that action).

For these experiments preliminary experiments showed that decreasing the exploration from 10% to 0% over the first 750,000 games led to the best results in most cases. However, with large vision grids and the sigmoid activation function against the random opponent, exploration decreases from 10% to 0% over 1 million training games. The learning rate α and discount factor γ are for the opponent modelling experiments also 0.005 and 0.95 respectively. These values have been found to lead to the best results, however there are some exceptions. When the full grid is used as state representation in combination with the sigmoid activation function, the learning rate is lowered to 0.001. This lower learning rate is also used with small vision grids and the sigmoid activation function against the random opponent. Finally, when large vision grids are used in combination with the sigmoid activation function against the random opponent, a learning rate of 0.0025 is used. Similar to the previous experiments, all weights and biases of the neural

networks are randomly initialised between -0.5 and 0.5.

For the opponent modelling experiments we trained the agent against both opponents and with both activation functions. We note that in this experiment, no roll-outs are performed. Therefore any possible performance improvement is caused by the additional state information or the use of the additional outputs that learn to model the opponent. The latter could be helpful to learn better features in the hidden layer. Figures 6, 7, and 8 show the training performance for the three different state representations. Table 3 and 4 show the performance during the 10,000 test games after training the agent with opponent modelling.

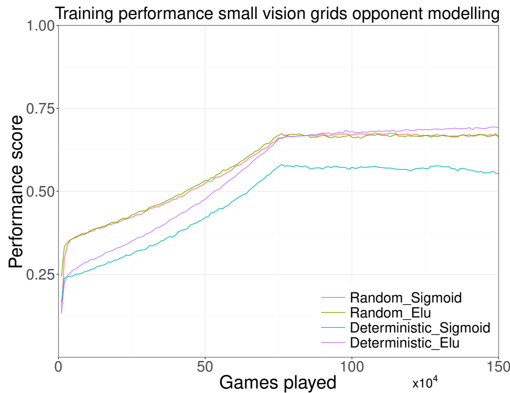

Figure 6: Performance score for small vision grids as state representation over 1.5 million training games with opponent modelling but without rollouts.

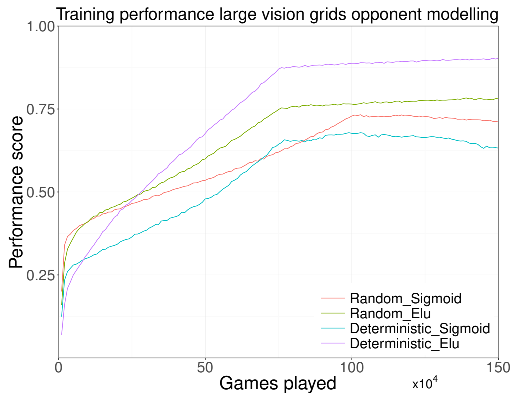

Figure 7: Performance score for large vision grids as state representation over 1.5 million training games with opponent modelling but without rollouts.

When we compare these results with the results obtained without opponent modelling, we observe several differences. First of all, when the full grid

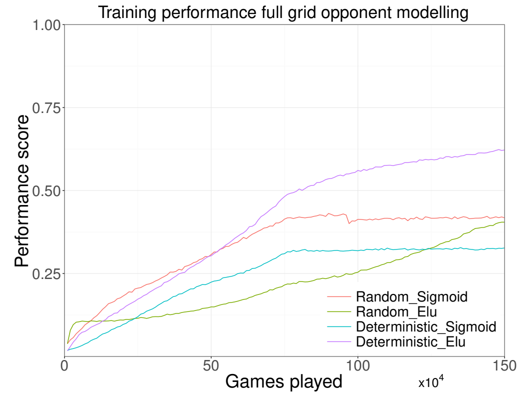

Figure 8: Performance score for the full grid as state representation over 1.5 million training games with opponent modelling but without rollouts.

Table 3: Performance score and standard errors with opponent modelling without rollouts against the random opponent.

State representation	Sigmoid	Elu
Small vision grids	0.67 (0.004)	0.67 (0.009)
Large vision grids	0.72 (0.005)	**0.79 (0.003)**
Full grid	0.42 (0.016)	0.40 (0.025)

Table 4: Performance score and standard errors with opponent modelling without rollouts against the deterministic opponent.

State representation	Sigmoid	Elu
Small vision grids	0.57 (0.015)	0.69 (0.005)
Large vision grids	0.63 (0.019)	**0.90 (0.003)**
Full grid	0.32 (0.023)	0.62 (0.015)

is used as state representation the performance drops with opponent modelling. The opposite holds for both small and large vision grids, where performance increases with opponent modelling. The most significant increase in performance appears with large vision grids against the semi-deterministic opponent, where a performance score of 0.90 is obtained.

In order to test whether this increase in performance with vision grids arises due to the opponent modelling technique, we conducted another experiment. In this experiment the set-up is exactly the same as in the opponent modelling experiment, but now the agent does not learn to model the opponent. The average results of ten test games with the Elu activation function can be found in Table 5.

From Table 5 we can conclude that the agent's increase in performance with opponent modelling is due to the extra vision grids generated. This is the case since there is not much difference in performance with and without opponent modelling when the extra vision grids for opponent modelling are also fed into

Table 5: Performance score and standard errors with the Elu activation function and opponent vision grids, but without opponent modelling.

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)

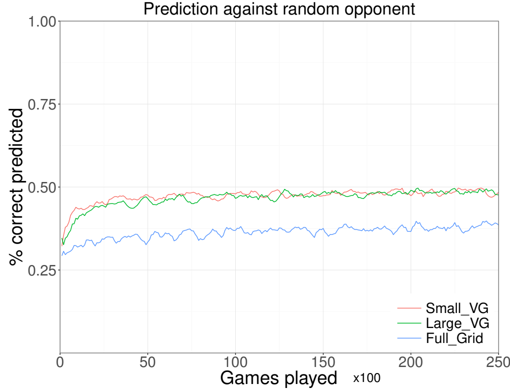

Figure 9: Percentage of moves correctly predicted against the random opponent.

the MLP.

5.3 Opponent Modelling with Monte-Carlo Roll-outs

After the agent is trained using opponent modelling, we applied roll-outs in order to try to increase the performance of the agent even further. The number of actions in a roll-out is set to ten, as this gives the agent the opportunity to look far enough in the future to choose the optimal action. Further increasing the number of actions of a roll-out will often not benefit the agent, as the average amount of actions in a game is twenty. We compare the performance of the agent with one and ten roll-outs. Since the opponent's actions within the roll-outs are determined by the learned probability distribution, we plot the prediction accuracy of the agent against both agents in Figure 9 and 10. These results are for the Elu activation function, which learns slightly faster than the sigmoid activation function. We observe that within 25,000 games 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 equal to 40% and 80% respectively.

The performance score and standard error using one roll-out with a horizon of ten steps during 10,000 test games can be found in Table 6 and 7. The Monte-Carlo roll-outs further increase the agent's performance in most cases. However, performance decreases when large vision grids are used against the random opponent. In all other cases, performance considerably increases with the use of roll-outs. The highest performance score obtained is 0.98, which is obtained with large vision grids and the Elu activation function against the semi-deterministic opponent. This shows that by applying opponent modelling and Monte-Carlo roll-outs, performance can be increased to very high levels. From Table 7 we observe that also with small vision grids, performance scores of over 0.90 are obtained against the semi-deterministic opponent. If we compare the results with vision grids and the full grid as state representation, we observe that vision grids significantly increase performance with opponent modelling and Monte-Carlo roll-outs. This increase is most evident against the semi-deterministic opponent. When the opponent employs the collision-avoiding random policy, small vision grids lead to the highest performance. When comparing Table 3 and 6, we see that

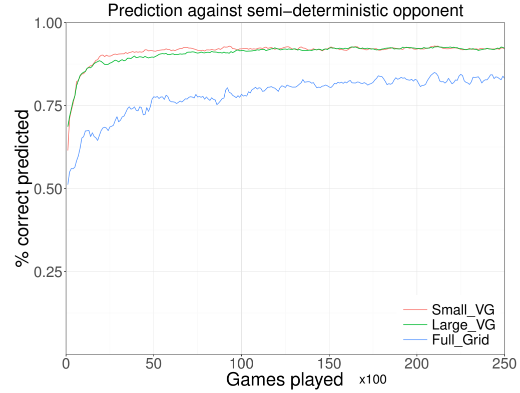

Figure 10: Percentage of moves correctly predicted against the semi-deterministic opponent.

Table 6: Performance score and standard errors with one roll-out and a depth of ten actions against the random opponent.

State representation	Sigmoid	Elu
Small vision grids	0.83 (0.002)	**0.84 (0.003)**
Large vision grids	0.66 (0.008)	0.66 (0.004)
Full grid	0.65 (0.004)	0.72 (0.007)

Table 7: Performance score and standard errors with one roll-out and a depth of ten actions against the deterministic opponent.

State representation	Sigmoid	Elu
Small vision grids	0.93 (0.002)	0.96 (0.001)
Large vision grids	0.95 (0.002)	**0.98 (0.001)**
Full grid	0.54 (0.010)	0.75 (0.010)

Table 8: Performance score and standard errors with ten roll-outs and a depth of ten actions against the random opponent.

State representation	Sigmoid	Elu
Small vision grids	0.84 (0.016)	0.88 (0.001)
Large vision grids	0.90 (0.001)	**0.91 (0.001)**
Full grid	0.72 (0.008)	0.74 (0.009)

Table 9: Performance score and standard errors with ten roll-outs and a depth of ten actions against the deterministic opponent.

State representation	Sigmoid	Elu
Small vision grids	0.93 (0.002)	0.96 (0.001)
Large vision grids	0.96 (0.002)	**0.98 (0.001)**
Full grid	0.55 (0.008)	0.78 (0.010)

roll-outs also increase performance against this 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 roll-out for each action at any state, we also tested whether increasing the number of roll-outs to ten would affect the agent's performance. The results are displayed in Table 8 and 9. When comparing the agent's performance with one and ten roll-outs, we detect one noteworthy difference. The agent's performance against the random opponent considerably increases when we use ten instead of one roll-out. This increase is especially large when we use large vision grids. Against the semi-deterministic opponent, increasing the number of roll-outs 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 roll-outs to be absent.

In order to determine whether it is the model of the opponent that allows the agent to attain very high performance levels using roll-outs, we also investigated the performance of the agent when the moves of the opponent in the roll-outs are determined randomly rather than from the learned model of the opponent. The results can be found in Table 10 and 11. From the results, we can conclude that it is indeed the model of the opponent that increases the agent's performance when roll-outs are used, because a bad opponent model results in much worse performances in combination with roll-outs.

Table 10: Performance score and standard errors with one roll-outs and a depth of ten actions against the random opponent and without using the learned model of the opponent.

State representation	Sigmoid	Elu
Small vision grids	0.46 (0.007)	0.50 (0.001)
Large vision grids	0.50 (0.001)	0.51 (0.002)
Full grid	0.37 (0.010)	0.35 (0.006)

Table 11: Performance score and standard errors with one roll-outs and a depth of ten actions against the deterministic opponent and without using the learned model of the opponent.

State representation	Sigmoid	Elu
Small vision grids	0.34 (0.003)	0.35 (0.001)
Large vision grids	0.35 (0.001)	0.36 (0.001)
Full grid	0.21 (0.007)	0.21 (0.005)

6 CONCLUSION

This paper has shown that vision grids can be used to overcome the problems associated with applying reinforcement learning in problems with large state spaces. Using vision grids as state representation not only increased the learning speed, it also increased the agent's performance in most cases. From all state representations, the large vision grids obtain the best performances. They reduce the number of different possible inputs compared to full grids, but contain more information that the small vision grids.

This paper also confirms the benefits of the Elu activation function over the sigmoid activation function. Against the semi-deterministic opponent, the Elu activation function increased the agent's performance in eleven of the twelve conducted experiments and against the random opponent performance increased in eight of the twelve experiments. From this it seems that the Elu activation function performs especially much better than the sigmoid function in case of less noisy updates due to the more deterministic opponent.

Finally, the introduced opponent modelling technique allows the agent to concurrently learn and model the opponent and in combination with planning algorithms, such as Monte-Carlo roll-outs, it can be used to significantly increase performance against two widely different opponents.

An interesting possibility for future research is to test whether the use of vision grids causes the agent to form a better generalised policy. We believe that this is the case, since vision grids are less dependent on the dimensions of the environment and possible obstacles the agent might encounter. Therefore, the learned policy will better generalise to other environments. Finally, the proposed opponent modelling

technique is widely applicable and we are interested to see whether it also proves useful in other problems.

REFERENCES

Bellman, R. (1957). A markovian decision process. *Indiana Univ. Math. J.*, 6 No. 4:679–684.

Bom, L., Henken, R., and Wiering, M. (2013). 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)*, pages 156–163.

Bouzy, B. and Helmstetter, B. (2004). *Monte-Carlo Go Developments*, pages 159–174. Springer US, Boston, MA.

Clevert, D., Unterthiner, T., and Hochreiter, S. (2015). Fast and accurate deep network learning by exponential linear units (elus). *CoRR*, abs/1511.07289.

Collins, B. (2007). Combining opponent modeling and model-based reinforcement learning in a two-player competitive game. *Master's thesis, School of Informatics, University of Edinburgh*.

Ganzfried, S. and Sandholm, T. (2011). Game theory-based opponent modeling in large imperfect-information games. In *the 10th International Conference on Autonomous Agents and Multiagent Systems-Volume 2*, pages 533–540. International Foundation for Autonomous Agents and Multiagent Systems.

He, H., Boyd-Graber, J. L., Kwok, K., and III, H. D. (2016). Opponent modeling in deep reinforcement learning. *CoRR*, abs/1609.05559.

Mealing, R. A. (2015). *Dynamic opponent modelling in two-player games*. PhD thesis, University of Manchester, UK.

Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1988). *Neurocomputing: Foundations of Research*, chapter Learning Internal Representations by Error Propagation, pages 673–695. MIT Press, Cambridge, MA, USA.

Shantia, A., Begue, E., and Wiering, M. (2011). Connectionist reinforcement learning for intelligent unit micro management in Starcraft. In *Neural Networks (IJCNN), The 2011 International Joint Conference on*, pages 1794–1801. IEEE.

Sheppard, B. (2002). World-championship-caliber scrabble. *Artificial Intelligence*, 134(12):241 – 275.

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., and Hassabis, D. (2016a). Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489.

Silver, D., van Hasselt, H., Hessel, M., Schaul, T., Guez, A., Harley, T., Dulac-Arnold, G., Reichert, D., Rabinowitz, N., Barreto, A., and Degris, T. (2016b). The predictron: End-to-end learning and planning. *CoRR*, abs/1612.08810.

Southey, F., Bowling, M., Larson, B., Piccione, C., Burch, N., Billings, D., and Rayner, C. (2005). Bayes bluff: Opponent modelling in poker. In *Proceedings of the 21st Annual Conference on Uncertainty in Artificial Intelligence (UAI*, pages 550–558.

Sutton, R. S. (1988). Learning to predict by the methods of temporal differences. *Machine Learning*, 3(1):9–44.

Sutton, R. S. and Barto, A. G. (1998). *Introduction to Reinforcement Learning*. MIT Press, Cambridge, MA, USA, 1st edition.

Tesauro, G. (1995). Temporal difference learning and TD-gammon. *Commun. ACM*, 38(3):58–68.

Tesauro, G. and Galperin, G. R. (1997). *On-line Policy Improvement using Monte-Carlo Search*, pages 1068–1074. MIT Press.

van Otterlo, M. and Wiering, M. (2012). *Reinforcement Learning and Markov Decision Processes*, pages 3–42. Springer Berlin Heidelberg, Berlin, Heidelberg.

Watkins, C. J. and Dayan, P. (1992). Q-learning. *Machine learning*, 8(3):279–292.