university of groningen / faculty of science and engineering

Reinforcement Learning in the Game of Tron using Vision Grids and Opponent Modelling

Bachelor's Project Thesis

Stefan Knegt, s2221543, s.j.l.knegt@student.rug.nl
Supervisor: M.A. Wiering

Abstract: In this thesis we propose the use of vision grids as state representation to learn the game Tron. This approach speeds up learning by significantly reducing the number of unique states. Secondly, 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 rollouts, 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 of the agent with two activation functions, namely the sigmoid and exponential linear unit (Elu). The results show that the Elu activation function outperforms the sigmoid activation function in most cases. Secondly, vision grids significantly increase learning speed and in most cases it 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 model of the opponent, which in combination with Monte-Carlo rollouts significantly increases the agent's performance.

1 Introduction

Reinforcement learning algorithms allow an agent to learn from its environment and thereby optimise its behaviour [1]. Such environments can be modelled as a Markov Decision Process (MDP) [2] [3], 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 [4], where reinforcement learning has led to great successes. This paper examines the effectiveness of reinforcement learning for the game of Tron. In order to deal with the relatively large state space in which only a small part is relevant, we propose the use of vision grids in order to speed up learning. In addition, this thesis describes an opponent modelling strategy with which performance can be significantly improved.
In this research the agent is constructed using a multi-layer perceptron (MLP) [5]. 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 Starcraft [6] and Ms. Pac-Man [7]. To build upon this work we will be using an MLP with two different activation functions and compare their performance. Apart from the well-known sigmoid activation function, we will be using the exponential linear unit (Elu). The exponential linear unit has three advantages compared to the sigmoid function [8]. 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 [8] and we are interested to see whether it can improve performance when using an MLP in combination with reinforcement learning.
One of the main challenges of using reinforcement learning in the game of Tron is the size of the environment. If we look at how humans play this game we see that they mainly focus their attention around the current position of the agent. Therefore, we propose the use of vision grids [6]. 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 these vision grids of variable sizes, the agent can acquire 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 informa-

tion, which can speed up the learning process of the agent. We will compare the performance of the agent when using different sized vision grids as opposed to using the entire game state as input.
Lastly, this thesis examines the effectiveness of opponent modelling in the game of Tron. Every advanced player uses some form of opponent modelling while playing games [9]. From games as tic-tac-toe to chess, we try to anticipate the next move of the opponent or detect a policy the opponent is following. This thesis proposes a novel opponent modelling technique in which the agent learns the opponent's behaviour by predicting the next move of the opponent and observing the result. As the agent learns to correctly forecast its opponent's action, we will apply Monte-Carlo rollouts [10] similar to [11]. In such a rollout 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. These rollouts are performed multiple times and the results are averaged. These averages have shown to substantially increase performance in games such as Backgammon [10], Go [12], and Scrabble [13].
Contributions In this thesis 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 relatively large state spaces. Furthermore, we confirm the benefit of the Elu activation function when the number of inputs increases. Finally, this thesis introduces a novel opponent modelling technique. With this technique the agent can form a model of the opponent during the learning phase, which can be subsequently used in prediction-based methods such as Monte-Carlo rollouts.
With this thesis we will try to answer the following questions, where we define performance to be the number of games the agent has won or tied:

1. Does the use of vision grids as input to the multi-layer perceptron increase the performance of the agent compared to using the full game state as input?

2. Can performance be increased by using the exponential linear unit as activation function instead of the sigmoid activation function?

3. Can the agent's performance be increased by modelling the opponent's behaviour and subsequently using this model in Monte-Carlo rollouts?

In the next section we explain the theoretic background and lay out the framework that was built to simulate the game and agent. Section 3 will explain the opponent modelling technique used. Then in section 4, we will outline the performed experiments and their results. Finally, in section 5 we present our conclusions and possible future work.

2 Framework

2.1 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 2.1 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 at the beginning of each new game state. In this research the game is played with two players. The environment is represented by a 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 itlsef 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. For the opponent we used two different implementations. Both 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. When we let these opponents play against each other, we observe 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.

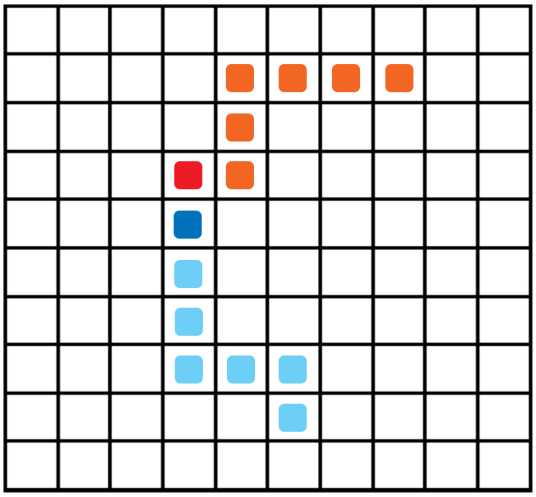

Figure 2.1: Tron game environment with two agents, where their heads or current location are in a darker colour.

2.2 Reinforcement learning

When the agent starts playing the game, it will randomly choose an action 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 learns to select the action that leads to the highest possible 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) [3]. 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 can't 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 $0 \leq \gamma \leq 1$.

In addition to 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:

$$V^{\pi}(s) = E\Big(\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s, \pi\Big) \tag{2.1}$$

Where r_t is the reward received at time t. So the value function gives the expected outcome of the game if both players select the actions given by their policy. This implies that 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 2.1 in terms

of the components of an MDP:

$$V^{\pi}(s) = \sum_{s'} T(s, \pi(s), s') \\ (R(s, \pi(s), s') + \gamma V^{\pi}(s')) \tag{2.2}$$

From equation 2.2 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. This is done for all possible next states and the result is summed. Together the definition of the MDP, policy and value function allow for the use of reinforcement learning. Next, we will look at the particular reinforcement learning algorithm employed in this research: Q-learning.

2.3 Q-learning

The previously defined value function gives the value of a state following a given policy. In this research we will be using Q-learning [14]. Therefore, 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 2.3.

$$Q^{\pi}(s, a) = E\Big(\sum_{t=0}^{\infty} \gamma^t r_t | s_0 = s, a_0 = a, \pi\Big) \tag{2.3}$$

So 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:

$$Q(s_t, a_t) = E(r_t) + \gamma \sum_{s_{t+1}} \\ T(s_t, a_t, s_{t+1}) \max_a Q(s_{t+1}, a) \tag{2.4}$$

Which states that the Q-value of a state-action pair depends on the expected reward, next state s_{t+1}, and the highest Q-value in the next state. However, we don't 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. This allows us to value a certain state-action pair independent of the opponent's move. The Q-learning algorithm is given by:

$$\widehat{Q}(s_t, a_t) \leftarrow \widehat{Q}(s_t, a_t) + \alpha(r_t + \\ \gamma \max_a \widehat{Q}(s_{t+1}, a) - \widehat{Q}(s_t, a_t)) \tag{2.5}$$

Where $0 \leq \alpha \leq 1$ defines 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 [15].

2.4 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 an environment with 100 different positions and two agents, the number of unique states is approximately equal to 2^{100}. 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 $Q(s, a)$ [16]. 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 similar to Bom et al. [7] there was a small advantage of using a single action neural network. The neural network is trained using back-propagation [17], where the target Q-value is calculated using equation 2.5. 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 and quality of learning. The target Q-value then becomes:

$$Q^{target}(s_t, a_t) \leftarrow r_t + \gamma \max_a \widehat{Q}(s_{t+1}, a) \tag{2.6}$$

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 righthand side of equation 2.6, the reward received in the final game:

$$Q^{target}(s_t, a_t) \leftarrow r_t \tag{2.7}$$

2.4.1 Activation function

In order to allow for the neural network's decision boundary to be non-linear we make use of an activation function in the hidden layer. One of the most eminent activation functions is the sigmoid function:

$$O(a) = \frac{1}{1 + e^{-a}} \tag{2.8}$$

This function transforms the output to a value between 0 and 1. Recently, it has been proposed that the exponential linear unit functions better in some domains [8]. We will compare the performance of the agent using the sigmoid function and the exponential linear unit (Elu). The exponential linear unit is given by the following equation:

$$O(a) = \begin{cases} a & \text{if } a \geq 0 \\ \beta(e^a - 1) & \text{if } a < 0 \end{cases} \tag{2.9}$$

Where we set β equal to 0.01. 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 the problem at hand.

2.5 Vision grids

The first state representation used as input to the MLP is the entire game grid (10x10). 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. As mentioned previously, 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. There are three different types of vision grids used (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 2.2.

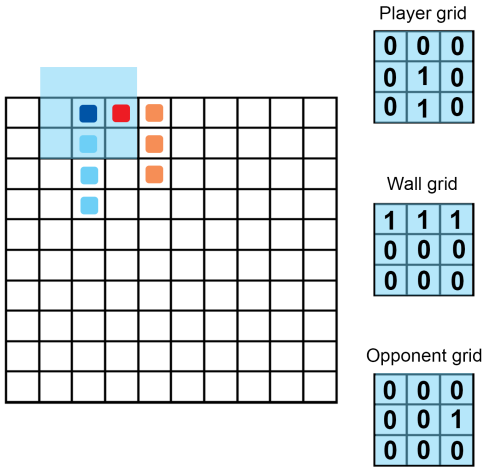

Figure 2.2: Vision grid example with the current location of both players in a darker color.

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 agent for the three different state representations.

3 Opponent modelling

Planning is one of the key challenges of artificial intelligence [18]. This thesis 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 rollouts. Many opponent modelling techniques focus on probabilistic models and imperfect-information games [19] [20], which makes them very problem specific. Our opponent modelling technique is widely applicable as it works by predicting the opponent's action and learning

from the results using the back-propagation algorithm [17]. Over time the agent learns a model of the opponent, which can be seen as the probability distribution of the opponent's next move. Therefore, 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 model of the opponent, which means that no extra phase is added to the learning process. In addition, the opponent modelling happens with the same 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.

As mentioned earlier, the opponent is modelled with the same neural network that calculates the Q-values for the agent. Four output nodes are appended to the network, which represent the probability distribution over the opponent's possible moves. 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.

$$P(s_t, o_i) = \frac{e^{o_i}}{\sum_{k=1}^{K} e^{o_k}} \tag{3.1}$$

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 mentioned earlier, 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 often 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 rollouts [10]. Such a rollout 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 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. At every action selection moment in the game m rollouts 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 rollout i:

$$\widehat{Q}_{sim}(s_t, a_t) = \gamma^i r_{t+i} \tag{3.2}$$

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 action performed:

$$\widehat{Q}_{sim}(s_t, a_t) = \gamma^n \widehat{Q}(s_{t+n}, a_{t+n}) \tag{3.3}$$

See algorithm 2.1 for a detailed description.

This kind of rollout is also called a truncated rollout as the game is not necessarily played to its conclusion [10]. In order to determine the importance of the number of rollouts m, we will compare the performance of the agent with one rollout and ten rollouts.

Algorithm 3.1 Monte-Carlo Rollout

Input: Current game state s_t, starting action a_t, horizon N, number of rollouts M

Output: Average reward 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 = γ 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
```

4 Experiments and Results

In order to answer our research questions, several experiments have been conducted. In all experiments the agent is trained for 1.5 million games against two different opponents. After that, 10.000 test games are played. In these test games, the agent makes no explorative moves. 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.

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.

4.1 State representation

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 repre-

sentation 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 4.1, 4.2, and 4.3 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 moves 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 4.1, 4.2, and 4.3. These results are also gathered from ten independent trials, for which also the standard error is reported.

Table 4.1: Performance score and standard errors with small vision grids as state representation.

Agent	Sigmoid	Elu
Random	0.56 (0.037)	0.62 (0.019)
Deterministic	0.35 (0.044)	0.39 (0.016)

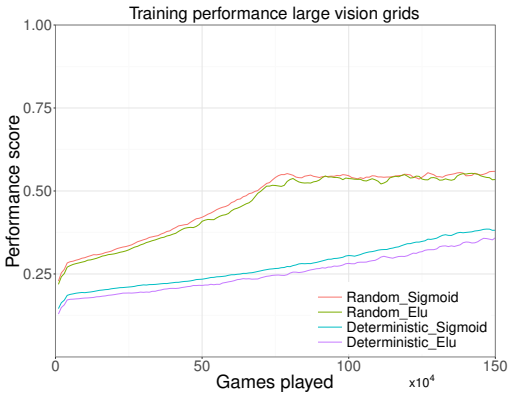

Figure 4.2: Performance score for large vision grids as state representation over 1.5 million training games.

Table 4.2: Performance score and standard errors with large vision grids as state representation.

Agent	Sigmoid	Elu
Random	0.54 (0.036)	0.53 (0.022)
Deterministic	0.37 (0.034)	0.39 (0.025)

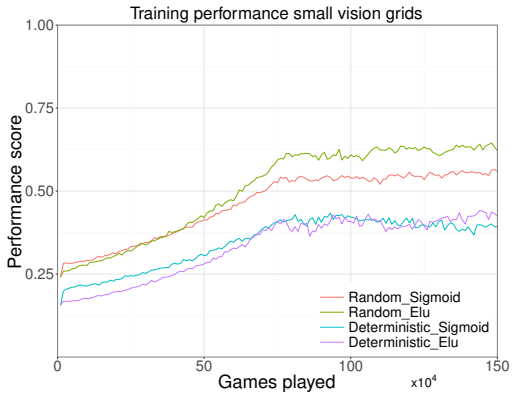

Figure 4.1: Performance score for small vision grids as state representation over 1.5 million training games.

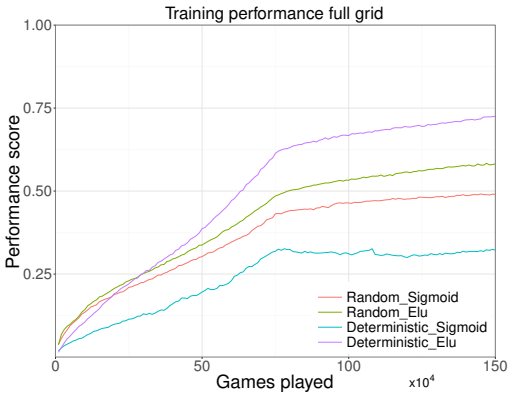

Figure 4.3: Performance score for the full grid as state representation over 1.5 million training games.

Table 4.3: Performance score and standard errors with the full grid as state representation.

Agent	Sigmoid	Elu
Random	0.49 (0.017)	0.58 (0.025)
Deterministic	0.31 (0.023)	0.72 (0.007)

From tables 4.1, 4.2, and 4.3 we can conclude that with the sigmoid activation function, the use of vision grids increases the performance of the agent when compared to using the full grid. However, the opposite holds when the Elu activation function is used. If we compare the performance of the agent with either small or large vision grids, we observe that with small vision grids the performance is better against the random opponent while there is no notable difference against the semi-deterministic opponent. 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 4.3. The agent reaches a performance score of 0.72 in this case, which is the highest performance score obtained. This is the only case in which the agent obtains a higher score against the semi-deterministic opponent than the random opponent. 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.

4.2 Opponent modelling and Monte-Carlo rollouts

Opponent modelling requires information not only about the agent's current position, but also about the opponent's position. As explained in section 3, this increases the number of vision grids used and therefore affects the number of input and 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.

Also for these experiments preliminary experiments showed that decreasing the exploration from 10% to 0% over the first 750.000 games led to optimal 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 optimal 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 network are randomly initialised between -0.5 and 0.5.
For the opponent modelling experiments we again trained the agent against both opponents and with both activation functions. In figure 4.4, 4.5, and 4.6 we find the training performance for the three different state representations. Table 4.4, 4.5, and 4.6 show the performance during the 10.000 test games after training the agent with opponent modelling.

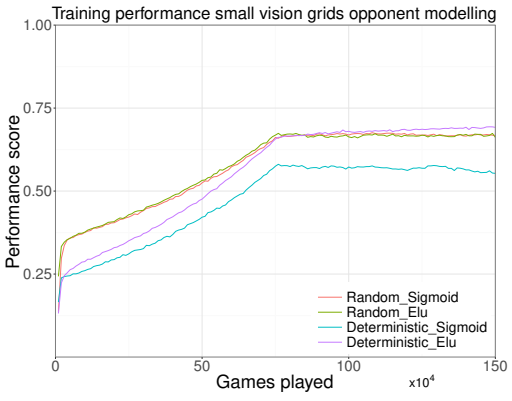

Figure 4.4: Performance score for small vision grids as state representation over 1.5 million training games with opponent modelling.

Table 4.4: Performance score and standard errors with opponent modelling and small vision grids as state representation.

Agent	Sigmoid	Elu
Random	0.67 (0.004)	0.67 (0.009)
Deterministic	0.57 (0.015)	0.69 (0.005)

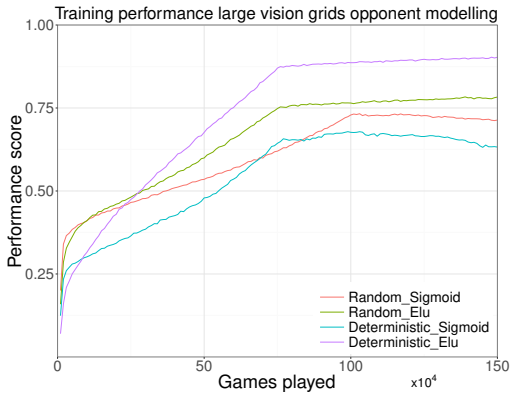

Figure 4.5: Performance score for large vision grids as state representation over 1.5 million training games with opponent modelling.

Table 4.5: Performance score and standard errors with opponent modelling and large vision grids as state representation.

Agent	Sigmoid	Elu
Random	0.72 (0.005)	0.79 (0.003)
Deterministic	0.63 (0.019)	0.90 (0.003)

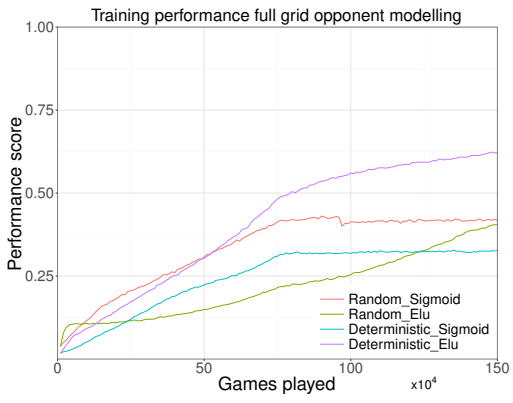

Figure 4.6: Performance score for the full grid as state representation over 1.5 million training games with opponent modelling.

Table 4.6: Performance score and standard errors with opponent modelling and the full grid as state representation.

Agent	Sigmoid	Elu
Random	0.42 (0.016)	0.40 (0.025)
Deterministic	0.32 (0.023)	0.62 (0.015)

When we compare these results with the results obtained without opponent modelling, we observe several differences. First of all, when the full grid is used as state representation the performance drops with opponent modelling, as can be seen when comparing table 4.3 and 4.6. 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. This confirms our expectations that our opponent modelling framework is especially helpful against an opponent that follows a more deterministic policy.
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 4.7.

Table 4.7: Performance score and standard errors with the Elu activation function and opponent vision grids, but without opponent modelling.

Agent	Small VG	Large VG
Random	0.69 (0.008)	0.82 (0.009)
Deterministic	0.69 (0.003)	0.89 (0.003)

From this table 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 the MLP.
After the agent is trained using opponent modelling, we applied rollouts in order to try to increase the performance of the agent even further.

The number of actions in a rollout 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 rollout 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 rollouts. 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 agents in figure 4.7 and 4.8. 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.

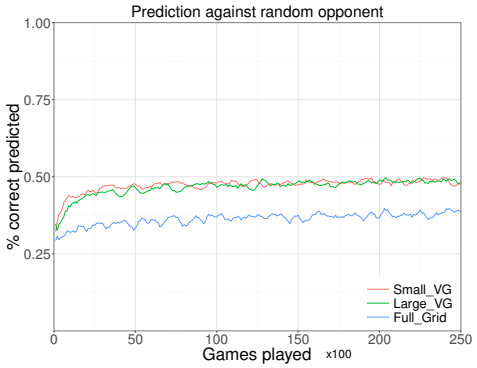

Figure 4.7: Percentage of moves correctly predicted against the random opponent.

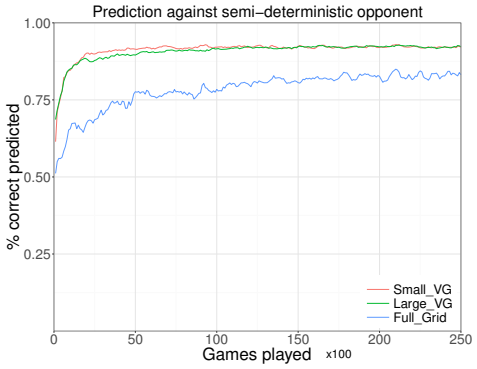

Figure 4.8: Percentage of moves correctly predicted against the semi-deterministic opponent.

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 4.8, 4.9 and 4.10.

Table 4.8: Performance score and standard errors with one rollout and a depth of ten actions with small vision grids.

Agent	Sigmoid	Elu
Random	0.83 (0.002)	0.84 (0.003)
Deterministic	0.93 (0.002)	0.96 (0.001)

Table 4.9: Performance score and standard errors with one rollout and a depth of ten actions with large vision grids.

Agent	Sigmoid	Elu
Random	0.66 (0.008)	0.66 (0.004)
Deterministic	0.95 (0.002)	0.98 (0.001)

Table 4.10: Performance score and standard errors with one rollout and a depth of ten actions with full grid.

Agent	Sigmoid	Elu
Random	0.65 (0.004)	0.72 (0.007)
Deterministic	0.54 (0.010)	0.75 (0.010)

The Monte-Carlo rollouts 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 rollouts. 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 rollouts, performance can be increased to very high levels. From table 4.7 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 rollouts. 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 4.4 and 4.7, we see that rollouts

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 rollout for each action at any state, we also tested whether increasing the number of rollouts to ten would affect the agent's performance. The results are displayed in table 4.11, 4.12, and 4.13.

Table 4.11: Performance score and standard errors with ten rollouts and a depth of ten actions with small vision grids.

Agent	Sigmoid	Elu
Random	0.84 (0.016)	0.88 (0.001)
Deterministic	0.93 (0.002)	0.96 (0.001)

Table 4.12: Performance score and standard errors with ten rollouts and a depth of ten actions with large vision grids.

Agent	Sigmoid	Elu
Random	0.90 (0.001)	0.91 (0.001)
Deterministic	0.96 (0.002)	0.98 (0.001)

Table 4.13: Performance score and standard errors with ten rollouts and a depth of ten actions with full grid.

Agent	Sigmoid	Elu
Random	0.72 (0.008)	0.74 (0.009)
Deterministic	0.55 (0.008)	0.78 (0.010)

When comparing the agent's performance with one and ten rollouts, we detect one noteworthy difference. The agent's performance against the random opponent considerably increases when we use ten instead of one rollout. This increase is especially large when we use large vision grids. 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.

5 Conclusion

This thesis 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. This thesis 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. 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 rollouts, it can be used to significantly increase performance against a wide variety of 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. Another interesting possibility for future research is to examine whether performance can be increased by learning the Q-values from the rollouts similar to [21]. The increase in performance with rollouts suggests that learning the Q-values with more than one-step look-ahead could lead to better performance. Finally, the proposed opponent modelling technique is widely applicable and we are interested to see whether it also proves useful in other problems.

References

[1] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*. Cambridge, MA, USA: MIT Press, 1st ed., 1998.

[2] M. van Otterlo and M. Wiering, *Reinforcement Learning and Markov Decision Processes*, pp. 3–42. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012.

[3] R. Bellman, "A markovian decision process," *Indiana Univ. Math. J.*, vol. 6 No. 4, pp. 679–684, 1957.

[4] G. Tesauro, "Temporal difference learning and TD-gammon," *Commun. ACM*, vol. 38, no. 3, pp. 58–68, 1995.

[5] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Neurocomputing: Foundations of Research*, ch. Learning Internal Representations by Error Propagation, pp. 673–695. Cambridge, MA, USA: MIT Press, 1988.

[6] A. Shantia, E. Begue, and M. Wiering, "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.

[7] L. Bom, R. Henken, and M. Wiering, "Reinforcement learning to train Ms. Pac-Man using higher-order action-relative inputs," *In Proceedings of IEEE International Symposium on Adaptive Dynamic Programming and Reinforcement Learning : ADPRL*, 2013.

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

[9] H. He, J. L. Boyd-Graber, K. Kwok, and H. D. III, "Opponent modeling in deep reinforcement learning," *CoRR*, vol. abs/1609.05559, 2016.

[10] G. Tesauro and G. R. Galperin, *On-line Policy Improvement using Monte-Carlo Search*, pp. 1068–1074. MIT Press, 1997.

[11] M. G. Lagoudakis and R. Parr, "Reinforcement learning as classification: Leveraging modern classifiers," in *Proceedings of the Twentieth International Conference on Machine Learning*, pp. 424–431, 2003.

[12] B. Bouzy and B. Helmstetter, *Monte-Carlo Go Developments*, pp. 159–174. Boston, MA: Springer US, 2004.

[13] B. Sheppard, "World-championship-caliber scrabble," *Artificial Intelligence*, vol. 134, no. 12, pp. 241 – 275, 2002.

[14] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, no. 3, pp. 279–292, 1992.

[15] R. S. Sutton, "Learning to predict by the methods of temporal differences," *Machine Learning*, vol. 3, no. 1, pp. 9–44, 1988.

[16] G. A. Rummery and M. Niranjan, "On-line Q-learning using connectionist systems," CUED/F-INFENG/TR 166, Cambridge University Engineering Department, September 1994.

[17] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, *Neurocomputing: Foundations of Research*, ch. Learning Representations by Back-propagating Errors, pp. 696–699. Cambridge, MA, USA: MIT Press, 1988.

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

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

[20] S. Ganzfried and T. Sandholm, "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.

[21] J. Baxter, A. Tridgell, and L. Weaver, "Knightcap: A chess program that learns by combining TD(lambda) with game-tree search," *CoRR*, vol. cs.LG/9901002, 1999.