Effectiveness of Connectionist Q-learning Strategies on Agent Performance in Asteroids

Bachelor's Project Thesis

Sjors Mallon, s2369087, sjorsmallon@gmail.com,
Niels Meima, s2980185, nielsmeima1@gmail.com
Supervisor: dr. M.A. Wiering

Abstract: This research proposes a higher-order state extraction algorithm serving as input for neural networks to learn to play the Atari game Asteroids. Asteroids is a 1980's space shooter, and poses a challenging environment due to its continuous and stochastic nature. Learning capabilities of the reinforcement learning algorithms Q-learning, Q-learning combined with a target network, Double Q-learning, QV-learning and QVMAX-learning are compared at a constant difficulty level, both using online learning and experience replay. Q-learning combined with a target network achieved the highest win rate of 0.76, in both the online and experience replay setting. Furthermore, the influence of incremental learning on agent performance is compared to learning at a constant difficulty. Incremental learning did not show a significant improvement in performance. Finally, state modeling in combination with Monte Carlo rollouts is used to learn from predictions about the future. Results show that learning from predictions is ineffective in its current implementation. The agent effectively learns to play the game Asteroids using the higher-order state extraction algorithm in combination with the described reinforcement learning algorithms.

1 Introduction

Reinforcement learning (RL) is an area in the field of machine learning in which an agent autonomously learns by interacting with an environment. Previous work has shown the viability of reinforcement learning techniques applied to video games (Kormelink et al., 2018; Bom et al., 2013; Leuenberger and Wiering, 2018). However, the application of such techniques is not always successful. Asteroids is among such games, performing significantly worse than an expert human player when deep reinforcement learning algorithms which learn from pixel-input is used (Mnih et al., 2013). This paper examines an approach in which a higher-order state extraction algorithm is combined with various reinforcement learning algorithms and analyzes their effectiveness in the context of the game of Asteroids. Asteroids is an arcade game developed by Atari in 1979. The player controls a spaceship, which can be moved forward, left and right and also has the ability to shoot projectiles. The goal is to achieve the highest possible score by destroying asteroids in a space environment. The environment is stochastic: asteroids spawn and move in random directions and have random velocities. The stochastic environment, combined with the limited action space, clear goal and previous unsatisfactory agent performance make Asteroids an interesting subject for reinforcement learning research.

Video games often present a challenge in reinforcement learning due to the size of their state space. In Asteroids, the asteroids can be at any location, moving in a random direction with a random velocity all relative to the position of the spaceship, resulting in a huge state space. In many cases multilayer perceptrons (MLPs) are used to deal with large state spaces. Using the current game state as an input, the MLP is used to select the action that will provide the highest reward in the long run.

Another approach to reduce the state space is by reducing the number of variables needed to represent a game state. We study how a higher-order state extraction algorithm can be used to effectively reduce the state space, whilst still maintaining crucial game features. The reduced description of the game state is then used as input to the MLP, aiming for faster learning and higher agent performance compared to a naive description of the game state.

The higher-order state extraction algorithm is combined with several reinforcement learning algorithms. The agent is trained in both an online manner, where sequential states are prsented to each RL algorithm and using a technique called experience replay (Lin, 1992).

The impact of incremental learning is also studied. In Asteroids the game gets progressively harder, by an increasing number of asteroids on the playing field. Higher difficulties might pose too difficult a challenge to achieve good agent performance. With incremental learning the difficulty slowly increases during training until a certain difficulty is reached, compared to normal learning where the difficulty is constant.

Previous research shows how learning from opponent modelling in combination with Monte Carlo rollouts, can significantly increase agent performance (Knegt et al., 2018a). Such modelling could also be applied to the environment. The influence of learning from modelled states in combination with Monte Carlo rollouts on agent performance is therefore also examined.

Contributions. In this paper we propose a higher-order state extraction algorithm and combine it with various reinforcement learning algorithms, such as Q-learning, double-Q learning, Q-learning combined with a target network, QV-learning and QVMAX-learning. The state extraction algorithm proves to effectively reduce the game state, whilst maintaining crucial game features. Online training and training using experience replay are compared in their effectiveness. Furthermore, the influence of incremental learning compared to learning at a constant difficulty will be more closely examined. Lastly, state modeling in combination with Monte Carlo rollouts is studied as a way of possible agent performance improvement.

Outline. In section 2 we explain the game of Asteroids in more detail, as well as the AsteRLoids framework. The higher-order state extraction algorithm and the used hyper-parameters are described in this section as well. In section 3 the used reinforcement learning algorithms are described, along with why a function approximator is needed. Then in section 4 the experiments and their results are described, including incremental learning and the influence of state modelling combined with Monte Carlo rollouts. Finally, in section 5 we discuss the results and possible future research.

2 Asteroids

Asteroids is a game in which the player controls a spaceship in space. The game is 2D and is viewed from a top-down position. The ship is surrounded by asteroids and UFOs. The ship can be controlled by either rotating left, rotating right, moving forward, and shooting. The ship has inertia: it will continue flying in the same direction for a set amount of time after releasing the “moving forward” button, slowly losing velocity.

Asteroids are spawned in random locations at the start of the level. The ship can destroy Asteroids and UFOs by shooting them with a bullet. Upon destroying an Asteroid or an UFO, the player is rewarded with points. An asteroid can exist in three sizes: small, medium or large. The number of points gained for destroying asteroids depends on the size of the Asteroid. Smaller asteroids reward more points. The different sizes of asteroids are represented as circles with increasing radii. An asteroid that is of middle or large size will split into smaller asteroids upon destruction. Asteroids with smaller sizes have higher velocities. A player can also enter hyperspace. Upon pressing the hyperspace button, the ship disappears from the map and randomly reappears on the map. This can instantly kill the player, and can thus be seen as a high-risk, high reward manoeuvre. The borders of the playing field are not solid borders. If an object (the ship, an asteroid or a bullet) connects with a border of the playing field, the object is moved to the opposite border of the playing field, depending on which border the object initially collides with. The goal of Asteroids is to clear the screen of asteroids and UFOs. After the entire screen has been cleared, a new level with more asteroids on screen starts. The ultimate goal of the game is to reach the highest score. The player starts off with three lives. If the ship touches either an asteroid or a UFO, the player loses a life. If all three lives are lost, the game is over and the score is final.

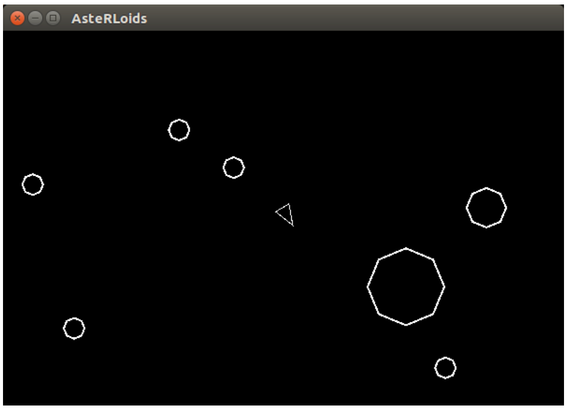

Figure 2.1: Game simulation from the AsteRLoids framework. Asteroids of different sizes can be seen, with the spaceship in the middle of the playing field.

2.1 AsterRLoids

We have developed the AsteRLoids framework which is a recreation of the original Asteroids game. The parameters for velocities, rotational speeds etc. were set by closely inspecting and comparing them to the original game. The UFOs are not implemented, as we are mostly interested in seeing basic functionality in an already complex stochastic continuous environment. The same reasoning applies to the hyperspace mechanic, introducing extra stochasticity. In a stochastic environment, there is a weaker relation between two states and an agent's action, as changes in the state do not necessarily result from an action that the agent performs. This could impact the learning performance negatively, which is why the current framework does not support it. The Asteroids spawn at random locations with a random heading. Their heading will never change, thus continually moving in their starting direction. When a large or medium-sized asteroid is destroyed, it spawns two asteroids one size smaller. These Asteroids start in the location of their parent asteroid, but are also given random headings.

State representation. In order for the agent to learn using the described reinforcement learning algorithms, the state of the environment must be conveyed to the agent through a representation. We carefully crafted a higher-order state extraction algorithm to extract the essential features from the environment and convert them to a representation the MLP can use as input. From the position of the ship, we divide the playing area in n vision segments. This can be visualized by each segment accounting for $360/36 = 10$ degrees. For each segment, we determine a danger level $\omega \in [0.0, 1.0]$, based on the distance between the ship and the closest asteroid in a segment. A danger level of 1.0 represents the most danger, where an asteroid will be touching the agent. A danger level of 0.0 represents no danger, which means there is no asteroid in the respective segment. This means that the system does not provide information about the other asteroids in this segment. By virtue of the wrapping borders, the maximum distance any object can be away from the ship is given by:

$$D_{max} = \sqrt{\left(\frac{w}{2}\right)^2 + \left(\frac{h}{2}\right)^2}. \tag{2.1}$$

Where w is the width of the environment and h is the height of the environment. We define the value ω of a segment to be equal to the inverse ratio between the euclidian distance from the ship to the closest asteroid D_{SA} in that segment and the maximum distance D_{max}:

$$\omega = 1 - \frac{D_{SA}}{D_{max}}. \tag{2.2}$$

Asteroids that are further away than D_{max} are through the border wrapping property of the game per definition closer in another segment. The wrapping of the borders can be represented by surrounding the playing field by exact copies of the state of the playing field, excluding the ship itself. This ensures the algorithm always correctly determines the danger level of the closest asteroid in every segment. The calculated danger levels are part of the input vector for the MLP. We add the danger levels of the previous state to the current danger levels, to serve as the complete input vector. This allows the agent to gain some notion of movement in the environment, as opposed to seeing static snapshots of the environment. The MLP thus has a total of $36 * 2 = 72$ inputs.

Reward function. Actions leading to certain events are rewarded a scalar reward r_t. The rewards are used in the reinforcement learning algorithms to learn optimal policy. The game events

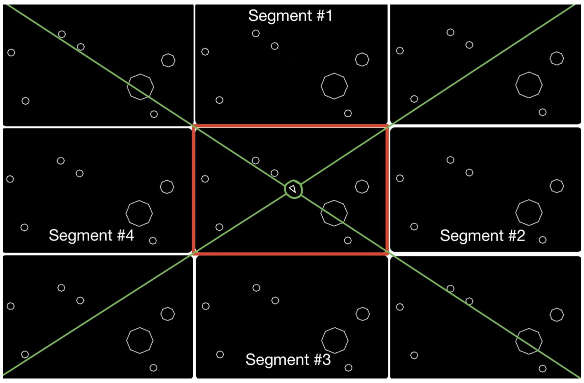

Figure 2.2: Wrapping borders property represented by the original playing field (red border) surrounded by copies of itself excluding the space ship. Here, 4 vision segments are used to divide up the playing field.

and their associated rewards are listed in table 2.1. The agent is rewarded upon clearing all the asteroids in the current level, the agent has then won the level. Hitting an asteroid by shooting is a necessary intermediate goal towards level completion. This imitates the actual reward structure of the game, where the player gets rewarded points for shooting an asteroid. Shooting smaller asteroids does not reward more points, as the agent is unaware of the size of the Asteroids as it is not part of the representation. If the agent dies, it receives a negative reward. This reward has the largest magnitude, as in order to accomplish the goal of completing a level, the agent at least needs to stay alive.

Table 2.1: Events and their scalar rewards in the AsteRLoids framework

Event	Reward
Wins	200
Hits asteroid	20
Dies	-500

3 Reinforcement learning

Reinforcement learning is an area in machine learning in which the agent tries to learn optimal behavior by interacting with its environment (Sutton and Barto, 1998). At each time-step t the agent receives a state s_t from the environment and executes an action a_t from its action space. The environment receives the chosen action a_t and emits a next state s_{t+1}. A reward function $R(s_t, a_t, s_{t+1})$ assigns a scalar reward r_t. The obtained scalar reward r_t indicates the quality of the executed action a_t in state s_t. The goal of reinforcement learning methods is to maximize the received rewards in the long run. A value function defines the expected cumulative future discounted reward. By optimizing the value function, the agent learns to predict the value of a state, and thus learns to reason about which actions would be optimal in a certain state. In this paper multiple reinforcement learning algorithms are implemented and compared, which will be explained now.

3.1 Q-learning

A policy is a mapping of states to actions. An optimal policy can be learned by using Q-learning (Watkins, 1989). In Q-learning, each state-action pair is associated with a Q-value $Q(s_t, a_t)$, representing the cumulative expected future reward. By optimizing the Q-function using the Q-learning rule, an optimal policy can be learned. The Q-learning rule is given by:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\delta_t \tag{3.1}$$

With δ_t given by:

$$\delta_t = r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t) \tag{3.2}$$

The learning rate $\alpha \in [0, 1]$ represents how much the Q-values are altered for each iteration of the learning algorithm. The discount factor $\gamma \in [0, 1]$ represents how future rewards should be valued compared to immediate rewards. By interacting with the environment and using the learning rule, the agent is able to optimize its policy based on the rewards it receives.

Function Approximator. The Q-value function can either be stored in tabular form or by using a function approximator such as a multi-Layer perceptron (MLP). When the number of state-action pairs is relatively small, the tabular form could be used to store all associated Q-values. In Asteroids, however, the number of state-action pairs is enormous. If a tabular representation would be used, there would be no guarantee that after training, all state-action pairs would have been seen. If an unseen state would then occur during playing, there is no previous experience to base the action selection on. Therefore, a multi-layer perceptron (MLP) is used to estimate $Q(s, a)$. The MLP has the ability to deal with unseen states, giving an estimation based on states it has actually seen. The input of the MLP is the current game state s, and in turn the MLP will produce an output which consists of Q-values for each action in the action space. The MLP is trained using back-propagation based on an adaptation of Eq. 3.2. The adaptation excludes the learning rate α from the equation, since it is already incorporated in the back-propagation algorithm of the MLP. The adaptation specifies the calculation of target values needed for back-propagation. The target Q-value for a selected action a_t in state s_t is then:

$$Q^{target}(s_t, a_t) = r_t + \gamma \max_a Q(s_{t+1}, a) \quad (3.3)$$

However, if the executed action a_t results in a terminal state, the following equation is used to calculate the target Q-value:

$$Q^{target}(s_t, a_t) = r_t \quad (3.4)$$

3.2 Q-learning with target network

Q-learning has been enhanced for deep RL by using a target network (Mnih et al., 2013) for selecting and evaluating the action with the highest expected cumulative future reward, instead of using the original network for this task. The technique can be used to stabilize the learning process. There are now two networks with their respective weights θ and θ'. The target network is initialized with the weights of the original network: $\theta' = \theta$. The introduction of a target network uses an adaption of Eq. 3.3 to calculate the target Q-value:

$$Q^{target}(s_t, a_t) = r_t + \gamma \max_a Q(s_{t+1}, a; \theta'_t) \quad (3.5)$$

In terminal states, the target Q-value is given by Eq. 3.4. The target network is now used to choose and evaluate the action with the highest expected cumulative future reward. The target Q-value is back-propagated in the same way as previously described in Q-learning. Additionally, every N_t time-steps the weights θ' of the target network are replaced with the weights θ of the original network.

3.3 Double Q-learning

In some environments Q-learning might perform very poorly. This poor performance is caused by over-estimations of Q-values, possibly steering away from the optimal policy. The over-estimations are caused by the fact that in Q-learning the max operator is used to both select and evaluate an action. By decoupling the selection and evaluation of an action, the over-estimations of Q-values can be reduced (Van Hasselt, 2010; van Hasselt et al., 2016).

This decoupling is realized by learning two Q-functions with their respective sets of weights A and B, resulting in Double Q-learning. Every time-step, one of the MLPs is randomly selected. This MLP is used to select an action according to the Q-value and the other MLP will determine the value of the selected action. Double Q-learning leads to the following change in the equation for calculating the target value in non-terminal states, where the target value calculation in case the network with the set of weights A is given by:

$$Q^A(s_t, a_t) \quad \leftarrow \quad r_t \; + \; \gamma Q^B(s_{t+1}, a^*) \quad (3.6)$$

In which $a^* = \max_a Q^A(s_{t+1}, a)$.

3.4 QV-learning

QV-learning (Wiering, 2005) works by keeping track of two value functions: the state-value function and the Q-function. The Q-function is learned by learning from the V-values using the Q-learning rule. The V-function, reflecting the value of a state, might converge faster to optimal values when compared to the Q-function. The V-function input only considers the current state as opposed to the Q-function which considers state-action pairs, which leads to a higher update frequency of the V-function compared to the Q-function. Using a V-function to learn the Q-function might therefore be a more effective way to learn the Q-function.

The V-learning rule is defined as:

$$V(s) \leftarrow V(s) + \alpha \delta_t \quad (3.7)$$

Where δ_t is defined as:

$$\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t) \tag{3.8}$$

The Q-learning rule is adapted to learn from V-values and is defined as:

$$Q^{target}(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\delta_t \tag{3.9}$$

Where δ_t is defined as:

$$\delta_t = r_t + \gamma V(s_{t+1}) - Q(s_t, a_t) \tag{3.10}$$

The target Q-value is then given by:

$$Q^{target}(s_t, a_t) = r_t + \gamma V(s_{t+1}) \tag{3.11}$$

3.5 QVMAX-learning

QVMAX-learning is very similar to QV-learning: it also learns the V-function and the Q-function (Wiering and van Hasselt, 2009). However, as opposed to QV-learning, QVMAX-learning is an off-policy algorithm. In QVMAX-learning the V-function is learned using the following rule:

$$V(s_t) \leftarrow V(s_t) + \alpha\delta_t, \tag{3.12}$$

where δ_t is defined as:

$$\delta_t = r_t + \max_a Q(s_{t+1}, a) - V(s_t) \tag{3.13}$$

The Q-learning rule as described in Eq. 3.9 and the target Q-value target calculation as described in Eq. 3.11 stay the same.

3.6 Experience Replay

The described reinforcement learning algorithms are used in online learning, applying Q-learning updates to consecutive states. However, the described algorithms can be enhanced using experience replay (Lin, 1992). Each time-step t an experience $e_t = (s_t, a_t, r_t, s_{t+1})$ is stored in a replay memory $M = \{e_1, e_2, \ldots, e_n\}$ with a maximum size of n (Zhang and S. Sutton, 2017). The new experience will replace the oldest experience in the memory if the memory is at its maximum capacity. The memory is initialized by letting the untrained agent play in the simulation. Every time-step a sample of size s is drawn randomly and uniformly $\{e_1, e_2, \ldots, e_s\} \sim U(M)$ from the replay memory. The sample containing s experiences is then used to apply learning updates according the learning rules of the described algorithms. Experience replay has two main advantages compared to online learning. First, each experience might be used multiple times, allowing the agent to learn multiple times from the same experience, in contrast to online learning where each experience is thrown away after training on it. Second, learning from consecutive examples as is the case in online learning can be inefficient, due to strong correlations between experiences and a violation of the i.i.d assumption made by most reinforcement learning algorithms. Experience replay restores this assumption (Mnih et al., 2015).

3.7 Frame-skipping

Previous approaches to playing Atari games (Mnih et al., 2013), showed frame-skipping to be an effective way of playing more games in the same computational time as well as reducing high similarity of sequential states, both helping with the learning ability of the agent. Instead of selecting an action at every time-step in the simulation, the agent only receives a state and selects an action every kth time-step. Every other time-step, the previous selected action is repeated, except for the shooting action. Repeated shooting would lead to a very high, non-realistic shooting frequency. Simulating one time-step without action selection is much less computationally intensive than both simulating and selecting an action, explaining the increase in the number of games that can be played in the same computational time.

4 Experiments and Results

To compare the reinforcement learning algorithms using online learning and experience replay, incremental learning and the influence of state modelling combined with Monte Carlo rollouts, several experiments have been conducted. In every experiment the agent is trained for 100 epochs, where an epoch equals 200 games, resulting in 20,000 training games in total. The performance of the agent is tracked during training. The training requires around one day of CPU computation time for one experiment. After training, the agent's final performance is determined by playing 1000 test games. This benchmark is performed 10 times. During these test games the exploration rate is 1%. The performance is defined as the average number of games won, also called the win rate. The performance of the agent in all conditions is compared. The MLPs representing the various value functions share their architecture: 72 input nodes, 36 for the current danger levels and 36 for the previous dangers levels, 200 hidden nodes, and 5 output nodes. Adding extra hidden layers, or decreasing the number of hidden nodes decreased the performance of the agent. Increasing the number of hidden nodes, only added computational time, but showed no performance benefit. We remove all games that last less than 30 frames from the benchmark, as no move that the ship can make will prevent it from dying, i.e. the resulting state of the game is not dependent on the action of the agent.

Hyper-parameters. The hyper-parameters were set by conducting preliminary experiments and are the same for all conditions. The learning rate α is set to 0.0005 and the discount factor γ is set to 0.95. The weights of the MLPs are initialized between -0.5 and 0.5. The exploration rate for ϵ-greedy decreases linearly from 10% to 1% during training and is equal to 1% during testing (Thrun, 1992). Experience replay uses a sample size of 10 and a memory size of 500,000. The update frequency of the target network is 200 network iterations. The number of frames that is skipped before selecting an action is 4. Table A.1 in appendix A provides a more elaborate overview of all hyper-parameters.

4.1 Constant Difficulty

The algorithms are first trained on a constant difficulty, where every game starts with six asteroids. Difficulty six was chosen because it requires the agent to develop not only a shooting strategy, but also an evasive strategy to be able to achieve high win rates. Figure 4.1 shows the win rates of the algorithms using online learning during training. Every 200 games, the win rate is recorded. Figure 4.2 shows the win rates of the algorithms using experience replay during training. The final win rates obtained by running 10,000 test games are displayed in Table 4.1. The highest obtained win rate is achieved by Q-learning combined with a target network with and without experience replay. From the results in Table 4.1, we see very little difference in final performance between the online algorihms and algorithms using experience replay. From Figure 4.2 we can see that all experience replay algorithms have a slightly higher peak around epoch 10, and afterwards only slowly increases. There is still a noticeable upward trend, especially for Q-learning with a target network. Double-Q learning and online learning with a target network shows to improve performance slightly in both online learning and experience replay.

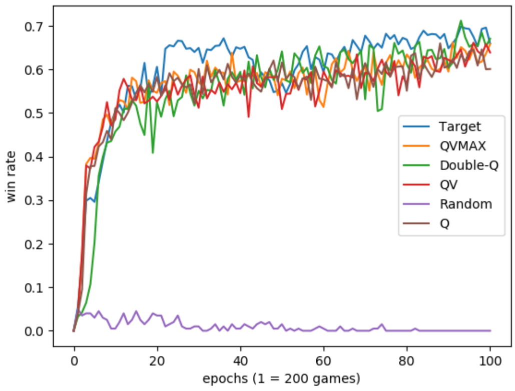

Figure 4.1: Comparison of the win rate of the algorithms using online learning at constant difficulty six during training

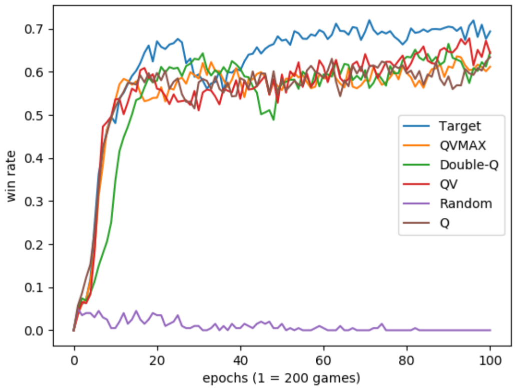

Figure 4.2: Comparison of the win rate of the algorithms using experience replay at constant difficulty six during training

Table 4.1: Final win rate of each algorithm in the constant difficulty condition

Learning algorithm	Win rate	SE
Online - Q	0.71	0.005
Online - Double-Q	0.73	0.005
Online - Target	**0.76**	0.005
Online - QV	0.70	0.005
Online - QVMAX	0.68	0.005
Exp. Replay - Q	0.68	0.005
Exp. Replay - Double-Q	0.73	0.005
Exp. Replay - Target	**0.76**	0.005
Exp. Replay - QV	0.69	0.005
Exp. Replay - QVMAX	0.67	0.005
Random walk	0.02	0.001

4.2 Incremental Learning

In Asteroids, whenever a level is completed, a new level starts at a higher difficulty. Training the agent at a constant difficulty does not reflect this property. Furthermore, a certain difficulty might be too complex and difficult for the agent to progress in without prior training. This would lead to the agent not progressing further into the level, which in turn might hinder learning. By starting at a lower difficulty and increasing the difficulty gradually whilst training, the final performance on higher difficulties might be better compared to non-incremental training. In incremental learning, the agent starts at difficulty three and ends at difficulty six. The agent plays $20000/3 \approx 6666$ games on each difficulty. Both non-incremental and incremental learning thus train for the same number of games.

Figure 4.3 shows the win rates of the algorithms using online learning during training. Figure 4.4 shows the win rates of the learning algorithms using experience replay during training. The final win rates obtained by running 10,000 test games are displayed in Table 4.2. The highest obtained win rate is achieved by online Q-learning combined with a target network.

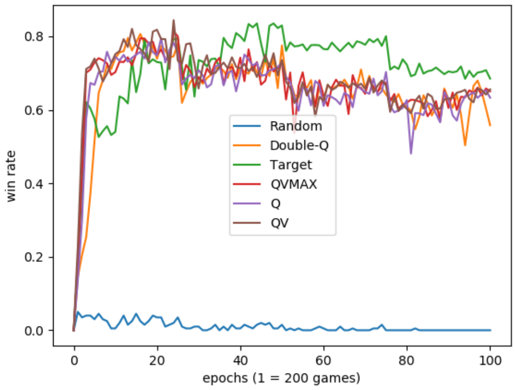

Figure 4.3: Comparison of the win rate of the algorithms using online learning in incremental learning during training

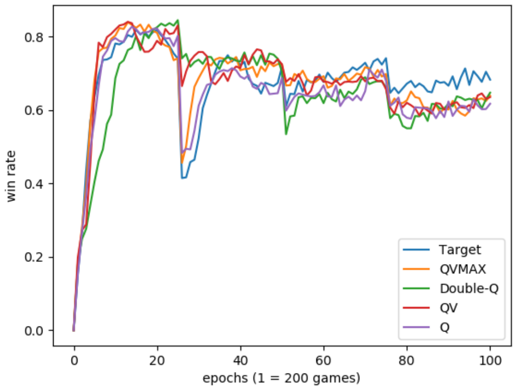

Figure 4.4: Comparison of the win rate of the algorithms using experience replay in incremental learning during training

Both Figure 4.3 and 4.4 shows a sharp rise in performance, slowly decreasing over the number of games played and increasing difficulty. This can be attributed to the difficulty increasing over the number of games. Additionally, in Figure 4.4 shows a dip in winrate at 25 epochs. This is the moment that the difficulty increases from 3 to 4. However, we also see that the win rate swiftly recovers after. The performance difference between online learning and experience replay can be seen in Table 4.2: almost all algorithms using experience replay outperform their online counterparts, except for QV-learning. The peak win rate of Online - Target is not significantly higher than any result in Table 4.1. Online-Target learning and online QV-learning show a beneficial or benign effect of incremental learning, with the rest of the algorithms showing significantly lower win rates. For experience replay, QV-learning and QVMAX-learning benefit from in-

Table 4.2: Final win rate of each algorithm in the incremental learning condition

Learning algorithm	Win rate	SE
Online - Q	0.63	0.005
Online - Double-Q	0.66	0.005
Online - Target	**0.77**	0.005
Online - QV	0.74	0.005
Online - QVMAX	0.66	0.005
Exp. Replay - Q	0.65	0.004
Exp. Replay - Double-Q	0.70	0.004
Exp. Replay - Target	0.76	0.005
Exp. Replay - QV	0.71	0.005
Exp. Replay - QVMAX	0.69	0.005
Random walk	0.02	0.001

cremental learning. It seems in these cases that the policy that the MLP tries to optimize is highly generalizable, not only from higher to lower difficulties as expected, but from lower to higher difficulties as well. Online Q-learning suffers the most, dropping 8% from constant difficulty to incremental learning.

4.3 Transition Modeling and Monte Carlo Rollouts

In transition modeling, the task is to learn the influence of an action on a state in order to predict what the next state of the environment will be. We use a technique based on opponent modeling in the game of Tron (Knegt et al., 2018a) (Knegt et al., 2018b). We introduce a transition prediction network which predicts the next state based on the same input that we supply to the regular network. In order to allow the transition modeling network to learn the relation between the state of the environment and the action that the agent takes, we add n binary input nodes to the network, with n equal to the size of the action space. An input value of 1.0 indicates the respective action was selected, whereas an input value of 0.0 indicates the action was not selected. Only one binary input can have the value of 1.0, the rest of the binary inputs have value 0.0. This allows the state prediction network to account for the action that the agent takes. A rollout has a horizon i, which is the maximum rollout depth, e.g. the maximum number of steps we predict the future in.

The MLP performs rollouts using the result from the state prediction network, and not with the actual environment simulation. In turn, there is no access to any of the measures that we can use in determining the Q-values in the other algorithms. The original game state can be converted to a state representation, but in this conversion, some information is lost, like the number of asteroids, the size of the Asteroids, their direction, the ship's direction and velocity. It is not possible to reproduce an actual game state from the predicted state representation. In order to remedy this, the network has additional outputs that predict whether the agent is alive or not, whether the agent won or not, and what the reward r_{sim} was associated with the state transition. The algorithm uses these predicted values in order to establish whether it is in a terminal state or not. The predicted state and the previous (predicted) state are then converted to an input vector to the state prediction network, in order to progress further into the future. If the rollout has reached the horizon depth and a_t results in a non-terminal state, the target Q-value is given by:

$$Q^{target}(s_t, a_t) = \sum_{i=0}^{T} \gamma^i (r_{t+i}) + \gamma^T \max_a Q(s_{t+T}, a) \quad (4.1)$$

The limit T defines the reached horizon depth. If the predicted state is a terminal state, the target Q-value is given by:

$$Q^{target}(s_t, a_t) = \sum_{i=0}^{T} \gamma^i (r_{t+i}) \quad (4.2)$$

From previous research, it seems beneficial to perform a rollout for every action at state s_t (Knegt et al., 2018a). Therefore, instead of having one target Q-value for the best action, we end up with a target Q-value for every action in the action space. After the first prediction the agent selects actions greedily according to its current policy. Rollouts are performed on a constant difficulty using online Q-learning, as the performance compared to the other tested algorithms did not differ much.

Initially, the state prediction network is trained alongside the regular network. After 5000 games, the regular network will perform rollouts with horizon $i = 1$, using the state prediction network for 15,000 games. Figure 4.5 shows the win rate when

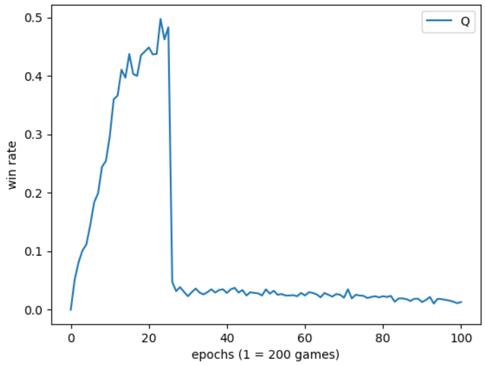

Figure 4.5: Win rate of Q-learning using online learning combined with state modeling and Monte Carlo rollouts during training

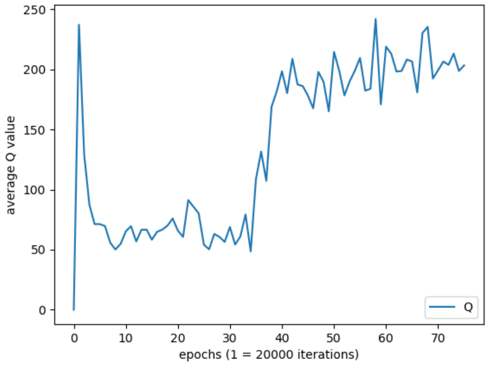

Figure 4.6: Average Q-values using online learning combined with state modeling and Monte Carlo rollouts during training

Table 4.3: Final win rate Q-learning using online learning combined with state modeling and Monte Carlo rollouts

Learning algorithm	Win rate	SE
Online - Default	0.01	0.001

state modeling and Monte Carlo rollouts are used. A drop in performance is seen at epoch 25, when the MLP starts using the state prediction network to perform rollouts. Figure 4.6 shows the average Q-value during training. Although, the average Q-value increases during training, this is not correlated with an increase in win rate, as can be seen in Table 4.3. Although the loss of the state prediction network was relatively low (0.03), there is no correlation between the rise in the average Q-value as seen in Figure 4.6 and the win rate during training. No explanation has yet been found for this lacking correlation, which should be there based on the relatively low loss of the prediction network. However, this loss might pose to be too high in the highly stochastic environment the agent has to deal with.

5 Discussion

In this paper, it is shown that the proposed higher-order state extraction algorithm can be used to effectively extract a state from the game Asteroids and serve it as usable input to a function approximator. Even though there is a loss of environment information by virtue of the state extraction algorithm (number of asteroids in each segment, their direction, heading, and size), the agent still performs admirably. Even though the agent is unaware of its own position, heading and velocity, it can navigate through the environment and complete levels. Furthermore, in this paper various reinforcement learning algorithms were compared using online learning and experience replay. When training at a constant difficulty Q-learning combined with a target network achieved the highest win rate, with and without experience replay. The differences between online learning and experience replay are small in this setting. The algorithms using experience replay have a higher peak win rate at the start of the training session, but the win rate only slowly increases afterwards. Training with incremental difficulty has a mixed effect on win rate on all algorithms. Online Q-learning and online double Q-learning suffered significant decreases in win rate, with online Q-learning suffering a drop of 8% win rate in the incremental learning setting. Online QV-learning got the highest increase in the transition between constant difficulty and incremental difficulty, with the benchmark yielding a 4% increase in the incremental difficulty setting.

The policy that the MLPs try to optimize in lower difficulties seems generalizable for higher difficulties. In general, the difference between experience replay and online algorithms does not seem significant. However, both Figure 4.2 and 4.4

show that the experience replay algorithms reach a higher win rate quicker than their online counterparts. An explanation for this lack of expected difference in win rate between online and experience replay could be that the algorithms using experience replay do reach their optimal policy more quickly than the online algorithms, but this optimal policy for maximizing the reward does not necessarily reflect a higher win rate. This could be attributed to our reward structure, where the reward for destroying an asteroid is relatively high. Further research is necessary to see whether the difference is more pronounced when the reward for shooting is lower, or removed altogether.

For future research, it would be interesting to see whether directly shifting from difficulty three to difficulty six during training will yield similar results. The generalizability of the policy can also be further explored by training at difficulty three and benchmarking at difficulty six. Additionally, increasing difficulty non-linearly is something that could be explored.

Using Monte Carlo rollouts to learn from future state predictions does not seem to be effective in its current implementation, as the agent is not able to achieve similar or better performance. The error in the state prediction only increases when the horizon increases. Because a terminal state is determined by the values that the state prediction network produces, the further into the future we roll out, the less reliable these measures become. It could therefore be that the state prediction network predicts that the agent is in a terminal state, while if the sequence of actions would occur in the "real world", this would not be the case. Therefore, the Q-value MLP might learn a policy which reflects the state prediction "world" more so than the actual "world" it is performing these actions in, which would lead to worse performance. More research would be needed to determine whether such an approach could be successful.
For future research it would be interesting to see the application of the higher-order state extraction algorithm to other problems, since it is a very general way of extracting states. Furthermore, examining whether agent performance can reach similar levels when re-introducing the now left out extra stochastic elements, would be interesting.

Acknowledgements. Our thanks go out to the the Center for Information Technology of the University of Groningen for providing access to the Peregrine high performance computing cluster and providing meaningful support in the setup of our application.

References

Bom, L., Henken, R., and Wiering, M. (2013). Reinforcement learning to train Ms. Pac-Man using higher-order action-relative inputs. *2013 IEEE Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL)*, pages 156–163.

Knegt, S., Drugan, M., and Wiering, M. (2018a). Learning from Monte Carlo Rollouts with Opponent Models for Playing Tron.

Knegt, S., Drugan, M., and Wiering, M. (2018b). Opponent modelling in the game of tron using reinforcement learning. *Proceedings of the 10th International Conference on Agents and Artificial Intelligence (ICAART), 2018.*

Kormelink, J., Drugan, M., and Wiering, M. (2018). Exploration Methods for Connectionist Q-learning in Bomberman. *Proceedings of the 10th International Conference on Agents and Artificial Intelligence (ICAART), 2018.*

Leuenberger, G. and Wiering, M. (2018). Actor-critic reinforcement learning with neural networks in continuous games. *Proceedings of the 10th International Conference on Agents and Artificial Intelligence (ICAART),2018.*

Lin, L.-J. (1992). Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine Learning*, 8(3):293–321.

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

Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A., Veness, J., BelleMare, M., Graves, A., Riedmiller, M., Fidjeland, A., Ostrovski, G., Petersen, S., Beattie, C., Sadik, A., Antonoglou, I., King, H., Kumaran, D., Wiestra, D., Legg, S., and Hassabis, D. (2015). Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533.

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

Thrun, S. B. (1992). Efficient exploration in reinforcement learning. Technical report, Pittsburgh, PA, USA.

Van Hasselt, H. (2010). Double Q-learning. In Lafferty, J. D., Williams, C. K. I., Shawe-Taylor, J., Zemel, R. S., and Culotta, A., editors, *Advances in Neural Information Processing Systems 23*, pages 2613–2621. Curran Associates, Inc.

van Hasselt, H., Guez, A., and Silver, D. (2016). Deep reinforcement learning with double q-learning. *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, pages 2094–2100.

Watkins, C. (1989). Learning from delayed rewards. PhD Thesis.

Wiering, M. A. (2005). Qv-learning: A new on-policy reinforcement learning algrithm. *Proceedings of the 7th European Workshop on Reinforcement Learning, D. Leone (editor).*

Wiering, M. A. and van Hasselt, H. (2009). The QV family compared to other reinforcement learning algorithms. *Proceedings of IEEE International Symposium on Adaptive Dynamic Programming and Reinforcement Learning (ADPRL), Nashville, USA*, pages 101–108.

Zhang, S. and S. Sutton, R. (2017). A deeper look at experience replay. *ArXiv e-prints.*

A Appendix: Hyperparameters

Table A.1: Hyperparameters

Hyperparameter	**Value**	**Description**
Number of segments	36	Number of vision segments, indicating dangerous levels around agent
Number of input nodes	72	Number of input nodes of the MLP: $2 \cdot \#segments = 72$. Exists of previous and current dangerous levels
Number of hidden nodes	200	Number of nodes in the hidden layer of the MLP
Number of output nodes	5	Number of the nodes in the output layer, every node represents the value of an action in the action space
Activation function $input \rightarrow hidden$	sigmoid	Activation function between the input layer and the hidden layer
Activation function $hidden \rightarrow output$	linear	Activation function between the hidden layer and the output layer
Learning rate α	0.0005	Learning rate of the MLPs
Discount factor γ	0.95	Discount factor of the learning algorithms
Initial exploration	0.1	Initial value of ϵ in ϵ-greedy exploration
Final exploration	0.01	Final value of ϵ in ϵ-greedy exploration
Skip frame k	4	Number of frames that an the last chosen action is repeated. Every 5th action, the agent sees chooses an new action and the MLPs are updated
Replay sample size s	10	number of experiences sampled from the memory and used for updating the MLPs
Replay memory size n	200000	experiences for updating the MLPs are samples from this number of most recent experiences
Target network update frequency N_t	200	the number of network iterations with which the target network is updated

B Appendix: Division of work

The implementation of the game and algorithms were done together in pair programming style. We routinely switched between “driving” and “navigating”. From experience, we learned that this prevented either of us from making mistakes that are hard to spot. Sjors performed the experiments and benchmarking for the constant difficulty setting, and Niels performed the experiments and benchmarking for the incremental difficulty setting. The introduction was a joint effort. Niels wrote most of the Reinforcement learning section. Sjors wrote most of the Asteroids / framework section. The results section was a joint effort, as well as the Monte Carlo rollout section. The conclusion was also a joint effort, as we had to compare both of our results. In general, we worked in close cooperation, as we were both reliant on the same framework.