university of groningen / faculty of science and engineering

QVA-learning: Testing a Novel Reinforcement Learning Algorithm Using Other-Play in the Helenix Environment

Bachelor's Project Thesis

Job Heeres, k.j.heeres@student.rug.nl,
Supervisor: Dr M.A. Wiering

Abstract: In this paper we use the Helenix environment and other-play, to test a new reinforcement learning algorithm called QVA-learning. This algorithm builds upon QV-learning by adding a function which should improve the learning behaviour. Within the game of Helenix, 5 different algorithms will train against each other using other-play. We found that when comparing the final models, Q-learning performed best under these conditions and that QVA-learning managed to outperform both double Q-learning and its predecessor QV-learning. However, when looking at the entire learning process, only SARSA manages to outperform QVA-learning. We conclude that QVA-learning shows potential and improves upon its predecessor QV-learning.

1 Introduction

Artificial Intelligence(AI) research often uses competitive games to test out hypothesis, compare different algorithms or show how well an AI can compete against a human. In 1978 the program Chess 4.7 managed to beat the chess master David Levy in a game of chess(Douglas, 1978). This was the first victory of a computer against a human chess master. 40 years later, in 2019, AlphaStar would use multi-agent reinforcement learning to play the incredibly complex game of Starcraft 2. Using both human and agent matches to train on, it managed to rank among the top 0.2% of players, putting it on the level of a Starcraft 2 grandmaster (Vinyals et al., 2019). Competitive games provide an environment with strict rules and simple logic that makes them excellent for AI research. Rather than trying to implement AI directly into complex situations, we first want to see how effective they are in solving these simpler problems. This is why we decided to use the game of Helenix (Louwers, 2019) to test the QVA-learning algorithm against other reinforcement learning algorithms.

Reinforcement learning is a form of machine learning where the agent tries to find when to perform which actions to maximise the reward that it receives. It is not told which actions are good or bad, but instead has to discover this by trying them out. Because of the way this works, it also allows the agent to consider the whole problem and make decisions that might cause it to lose points in the short term, but will give a much bigger reward in the long term(Sutton and Barto, 2018). This means that reinforcement learning is suited for solving more general problems, like playing a game, with much less need to divide the problem into very small sub-problems. This can be seen in AlphaZero which managed to master the three different games of chess, shogi and Go.(Silver et al., 2018)

There are multiple ways to train an agent in a game environment. One such method is using reinforcement learning with self-play. With self-play the agents playing the game are all powered by the same algorithm. This is a very effective method for training a reinforcement learning algorithm as can be seen in (Silver et al., 2018) whith AlphaZero learning to play chess, shogi and Go, and in (Vinyals et al., 2019) for starcraft, which we also mentioned before. A big advantage of this method is that it allows the algorithm to very quickly play a huge number of games against itself. The trial and error nature of reinforcement learning means that this is necessary for the algorithm to properly

learn how to play. We could compare different algorithms by first having them train using self play and then letting them play against each other, as can be seen in (Dries and Wiering, 2012). However, in (Louwers, 2019) we see a different method used in Helenix, the same environment we will be using. We believe it would be interesting to try out QVA-learning in a similar setup. This method is called other-play. As the name already suggests, this method involves the different agents being controlled by different algorithms during training. By doing this the training process will be quite different from self-play, with the agents being trained against specific algorithms.

In this paper we will be using the game of Helenix as seen in (Louwers, 2019). The game of Helenix is very similar to a two player version of the game splix.io, which in turn has mechanics similar to the game Pac-Xon. Pac-Xon was also used in (Schilperoort et al., 2018), where a Q-learning and two double Q-learning algorithms are trained to play it. In Helenix, two players each control a head starting in their own territory. The goal is to capture more territory and eliminate the other player. There will be a more in depth description of the game later on. In this game we will be comparing five different algorithms. These algorithms are Q-learning (Watkins, 1989; Watkins and Dayan, 1992), SARSA (Rummery and Niranjan, 1994), double Q-learning (van Hasselt, 2010), QV-learning (Wiering, 2005) and QVA-learning. We want to find how well QVA-learning performs against established reinforcement learning algorithms, and whether it significantly improves upon its predecessor QV-learning.

2 Reinforcement Learning

In this section we will shortly explain how reinforcement learning problems are formulated. We then go over different part of reinforcement learning, and explain the different hyper parameters that we will be using along the way. We will also explain the different algorithms that we will be comparing in our experiment.

2.1 Reinforcement Learning Problems

When formulating a reinforcement learning problem with an agent and an environment there are a few elements that are important. The states S that the problem can be in, the actions A that the agent can perform in these states, and the rewards $R_a(s, s')$ for transitioning from one state to another using an action. This kind of problem is called a Markov Decision Process. At each point in time t the agent gets to see what state it is in and has to determine what action to perform. Once the action has been performed the environment will tell the agent the reward it has received, and the new state it is in. This continues until a termination condition has been reached.

Another important aspect are the transition probabilities$P_a(s, s')$. This is the probability that an action a in a state s will lead to a new state s'. These transition probabilities exist for every action in every state in a Markov Decision Process, but are often not known to the agent.

2.2 Value Functions

When using reinforcement learning, an agent tries to maximise the reward that it receives from its environment. It does this by estimating the value of every it visits. Whenever it experience a certain action in a certain state, it calculates the value of that state. The estimation of the reward of a state is done through a value function$V(s)$. It is also possible to use a value function for the estimated value of an action in a state, in which case we use $Q(s, t)$. So simply put, the value function dictates how the reward is used in determining the value of a state or action.

For example the function:

$$V(S_t) = R_{t+1} + R_{t+2}....R_T \tag{2.1}$$

With $V(S_t)$ being the value of state S at point t and R_{t+n} being the reward at the nth points after t. So in this case, the value of a state is based on the rewards that it can get in the future. Where every future rewards is equally valuable.

Let's simplify equation (2.1) to the following:

$$V(S_t) = R_{t+1} + V(S_{t+1}) \tag{2.2}$$

This function might work in very simple environments where the agent knows exactly what the future holds. In a situation where the far future is much harder to estimate than the near future, this function is not sufficient. To solve this we introduce the discount factor γ. When an agent doesn't know exactly what the odds are of getting into future states, it is useful for early rewards to have a bigger impact than later rewards, seeing as these earlier rewards are easier to estimate. γ is a value between 0 and 1 that is used for this. If we change equation (2.2) to include this we get:

$$V(S_t) = R_{t+1} + \gamma V(S_{t+1}) \tag{2.3}$$

The result is that now the rewards are multiplied by γ^n where n is the number of steps after t. So now that we have introduced the concepts of value functions and discount factors, how exactly are the values updated?

2.3 Update Rules

There is still a problem with equation (2.3). If the agent is not certain of what the future holds, it can't always make an accurate estimation immediately. In competitive games, agents do not know the transition probabilities of their environment. This means that what worked once might not work again, which is when the actual reinforcement learning comes into plate. If we were to use equation (2.3) we would be recalculating the estimated value of a state every time we arrive at that state. Instead we want the agent to update the estimation that it already has, with the new experiences it gets whenever it reaches that state.

However, this raises a question. How heavily should new experiences influence the already established estimation? To determine this we need to introduce another variable, the learning rate α. α is a value between 1 and 0 that can be used to do this. These points would change equation (2.3) as follows:

$$V(S_t) = (1-\alpha)V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1})) \tag{2.4}$$

The result is that we use both the current estimated value as the new estimation. The larger α is the heavier new experiences will influence the estimated value of a state. Determining what α to use for a problem is a matter of experimentation.

The exact equations used to update the value estimations differ per algorithm. Though most algorithms we use in this paper use $Q(S_t, A_t)$ instead of $V(S_t)$, equation (2.4) can still be seen as the basis for most of the equations used in these algorithms.

2.4 Action Quality

The action that an agent should take in a state is that agent's policy π. The goal of the algorithms is to find the optimal policy. The way the policy can be seen is as the collection of all value estimates.

In simpler problems, with not too many states and where the transition probabilities are know, it is possible to calculate the estimates for every action and every state of the problem. These estimates can then be put into a table which can then be used by the agent to look up which actions to take.

For more complex problems, like Helenix, this is not really feasible. The biggest problem is the sheer number of possible states that can occur. Having to visit them all to be able to properly calculate the ideal policy would significantly decrease learning speed. It would also require a really big table. Another problem is that the agent does not know what their opponent will do, and thus does not know the transition probabilities, which means that it needs to perform actions multiple times for an accurate estimation. What we need instead is some way to approximate the value function.

A function approximator is some method with an adjustable weight that can be used to approximate a function. In this case the value function. The approximator we will be using is a multilayer perceptron (MLP) (Rumelhart et al., 1986). MLPs are feedforward artificial neural networks. The way we will be trying to approximate the value function is by mapping the state to the MLP's input nodes. The MLP will have a single hidden layer that is fully connected to both the input and output layers. Depending on whether we are using the MLP to approximate $V(s)$ or $Q(s, a)$, we map one node for the state, or one node for each action that the agent can perform in that state, to the output layer. When given a state as input, this means that MLPs used to approximate $V(s)$ will output an approximation of the value of that state. MLPs used to approximate $Q(s, a)$ will instead output an approximation of the value of every action that can

be performed within that state.

This approximation will be based on the weights of the MLP, which will be adjusted during training. The MLPs are trained using stochastic gradient descent combined with back-propagation, which is fairly standard for artificial neural networks.(LeCun et al., 1998)

2.5 Action Selection

There is still one factor that needs to be mentioned. How should the algorithm use the estimations to decide what to do? The naive solution would be to use the policy of always choosing the action with the highest estimate. This would result in the algorithm very quickly settling on a policy that is suboptimal. The algorithm will keep using this policy that it thinks is the best one, for the sole reason that any action that would deviate from this policy is considered worse. To deal with this we will be using an action selection algorithm.

During training we will be using the ϵ-greedy action selection algorithm. The way this algorithm works is quite simple. When an agent has to select an action it will not automatically choose the action with the highest estimate. Instead the policy will be that there is a probability of $1 - \epsilon$ that the agent chooses the action with the highest estimate and a probability of ϵ that instead a random action is chosen. This simple algorithm prevents the agent from getting stuck in a sub optimal policy, and facilitates a much larger exploration of options than would otherwise be the case.

The further along the training gets, the higher the chances are that the estimates are accurate and the less desirable it is for a random action to be selected. To make use of this ϵ decreases linearly during training, setting both a starting ϵ and a final ϵ. This way the agent has the advantage of being able to explore a lot in the early parts of training, and being able to exploit what it has discovered in the later parts of training.

Another action selection algorithm we will be using is Boltzmann exploration. This algorithm uses the following policy:

$$\pi(s, a) = \frac{e^{Q(s,a)/T}}{\sum_{i=1}^{m} e^{Q(s,a^i)/T}} \tag{2.5}$$

In which m is the number of actions. Instead of ϵ we use T here. Where a high T correlates with a lot of exploration and a low T with little exploration.

2.6 Algorithms

In the following section we will use what we just discussed to describe the different algorithms that we are going to compare to each other. To prevent the equations from becoming cluttered we will be using the following notation:

$$V(S_t) \leftarrow R_{t+1} + \gamma V(S_{t+1}) \tag{2.6}$$

instead of:

$$V(S_t) = (1-\alpha)V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1})) \tag{2.7}$$

For each of the following algorithm it also holds that if the next state $S + 1$ is the final state, then:

$$Q(S_t, A_t) \leftarrow R_{t+1} \tag{2.8}$$

2.6.1 Q-Learning

Q-learning (Watkins, 1989; Watkins and Dayan, 1992) is the oldest algorithm on this list and uses the following equation to update the state-action value estimation:

$$Q(S_t, A_t) \leftarrow R_{t+1} + \gamma \max_a Q(S_{t+1}, a) \tag{2.9}$$

This equation means that the state-action value estimate of action A_t in state S_t, is updated using the reward plus γ times the estimated state-action value of the next state $S_t + 1$ if the agent were to take the estimated best action a. This is used to train a single MLP, which is then used in the action selection process.

2.6.2 SARSA

SARSA (Rummery and Niranjan, 1994) is very similar to Q-learning but uses the next action that will be taken rather than just the action with the highest expected value to update the state-action value estimate:

$$Q(S_t, A_t) \leftarrow R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) \tag{2.10}$$

The idea is that this would improve learning by not overestimating, like Q-learning might do.

2.6.3 Double Q-Learning

Double Q-learning (van Hasselt, 2010) updates two different state-action value estimates rather than one:

$$Q_1(S_t, A_t) \leftarrow R_{t+1} + \gamma Q_2(S_{t+1}, \arg\max_a Q_1(S_{t+1}, a)) \tag{2.11}$$

$$Q_2(S_t, A_t) \leftarrow R_{t+1} + \gamma Q_1(S_{t+1}, \arg\max_a Q_2(S_{t+1}, a)) \tag{2.12}$$

Double Q-learning uses two MLPs, on for each estimate. During training the algorithm randomly updates one of the two estimates, which then trains the associated MLP. When it has to select an action it randomly chooses which MLP to use.

2.6.4 QV-learning

QV-learning (Wiering, 2005) also uses two estimates to determine which actions to take. The first one is a state-action value estimate. The second one is a state value estimate.

$$V(S_t) \leftarrow R_{t+1} + \gamma V(S_{t+1}) \tag{2.13}$$

As we can see here, the state value is updated using γ times the expected value of the next state. The state value is then used to train an MLP which is used to calculate the state-action-value estimate.

$$Q(S_t, A_t) \leftarrow R_{t+1} + \gamma V(S_{t+1}) \tag{2.14}$$

The state action value estimate is calculated using the state value, and then used to train a different MLP. This MLP is also the one used in the action selection process.

2.6.5 QVA-learning

QVA-learning seeks to improve upon QV-learnig with a third estimate. The state-action value estimate and the state value estimate are the same as they were for QV-learning.

$$V(S_t) \leftarrow R_{t+1} + \gamma V(S_{t+1}) \tag{2.15}$$

$$Q(S_t, A_t) \leftarrow R_{t+1} + \gamma V(S_{t+1}) \tag{2.16}$$

However, rather than using the state-action MLP to choose an action, a third advantage estimate is made using the other two MLPs.

$$A(S_t, a) \leftarrow Q(S_t, a) - V(S_t) \tag{2.17}$$

Where $A(S_t, a)$ is calculated for every action a in state S_t.
This advantage estimate is then used to train a third MLP which is used to determine which action to take. The idea is that this could lead to significantly improved learning behaviour over QV-learning.

3 Helenix

In this section we will describe the game of Helenix. We will first explain the rules and mechanics of the game. After that we will explain how to get points, and how the states of Helenix are represented using vision grids, which are then used as input for the different MLPs.

3.1 Rules and Mechanics

As mentioned before, Helenix is mechanically very similar to a two player version of (splix.io). The mechanics of the game are quite simple. The players control a unit which takes up one tile of the board. This unit will be called the "head" of a player. Every time unit the head has to move forward, left or right. When it moves it changes its orientation towards the direction it has moved. Each player starts with some squares marked as their territory. Within this territory they can move safely. When the head leaves its territory it will leave a trail which will be called the "tail" of the player that controls that head. An agent in their territory and an agent leaving their territory can be seen in figures 3.1 and 3.2. When the head reenters its own territory, the area that is enclosed by its tail, including the tail itself, will become parts of that player's territory. If any of this territory is owned by another player, it is transferred to the player that just encircled it. This process can be seen in figures 3.3 and 3.4. However, when the head is outside of its own territory, and a second player's head intersects the first player's tail, the first player dies and the game ends. This is demonstrated in figures 3.5 and 3.6.

This about covers the basic mechanics of the game. There are a few specific situations that are worth mentioning. Firstly, when two heads collide in a neutral square, both players are eliminated. If they collide in a square controlled by one of the players, only the player that does not control the

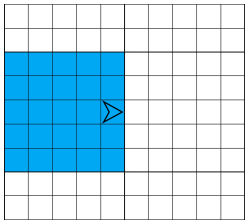

Figure 3.1: A player at the edge of its own territory.

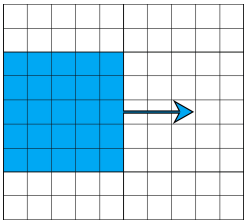

Figure 3.2: A player that has left its own territory.

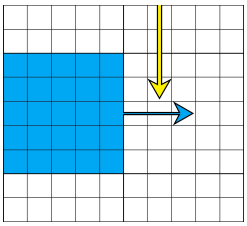

Figure 3.5: A player that is about to be intercepted by another player.

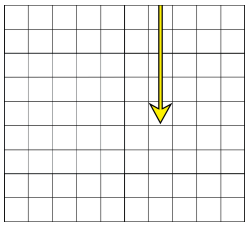

Figure 3.6: A player That has just intercepted another player.

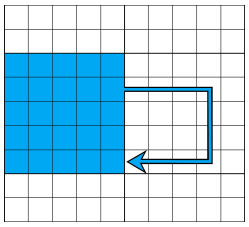

Figure 3.3: A player in neutral territory, about to reconnect with its own territory.

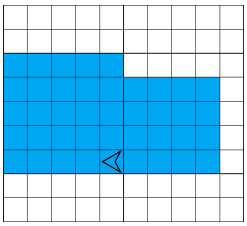

Figure 3.4: A player That has just reconnected with its territory.

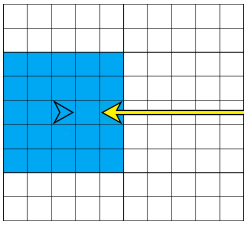

Figure 3.7: A player about to collide with another player on its own territory.

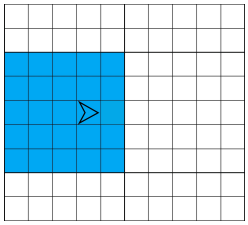

Figure 3.8: A player having eliminated another player through head collision.

square is eliminated. This can be seen in figures 3.7, 3.8, 3.9 and 3.10. Secondly, though the snake like nature might raise the idea that it would be the case, nothing happens when a player crosses their own tail. Finally, if a player goes out of bounds that player is eliminated.

The starting setup for every game is the same. We have a 21x21 grid where the outer most squares are marked as the edge, leaving a 19x19 grid for the agents to play on. The players spawn at the coordinates (4,11) and (18,11), facing towards each other. Both players are also given a 5x5 piece of territory centered around their head at the start of the game. This starting setup can be seen in figure 3.11.

3.2 Points

For the algorithms it is important to know which actions are desirable and which actions are not. The goal of the game is getting as much territory as possible or eliminating your opponent. To achieve this we give the agents feedback based on their actions. For every square that an agent captures they gain 1 point. For every square that an agent loses

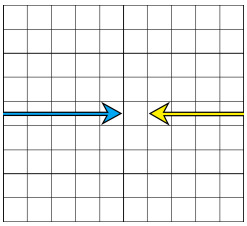

Figure 3.9: Two players about to collide in neutral territory.

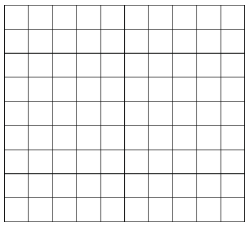

Figure 3.10: Both players being eliminated by the collision.

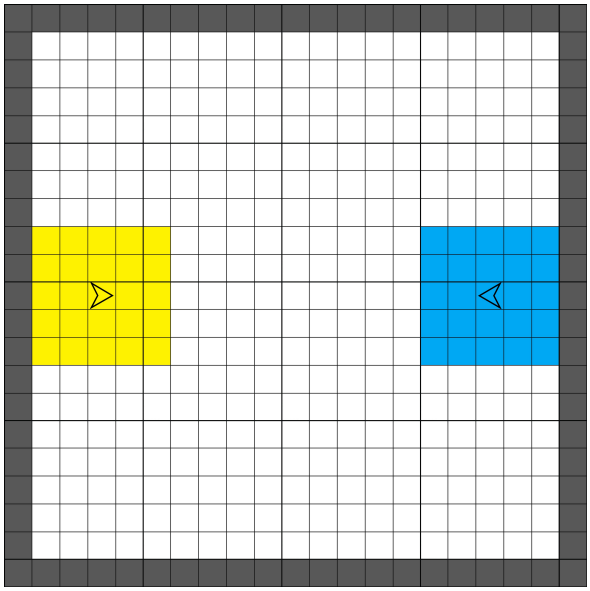

Figure 3.11: The starting setup for every match of Helenix

they lose one point. If an agent eliminates another agent, that agent gains 50 point. If an agent is eliminated they lose 50 points. If both agents eliminate each other by colliding in neutral territory, neither agent gains or loses any points. These points are used as the reward for the reinforcement learning algorithms.

3.3 States

To train the MLPs used in these algorithms we must be able to tell it what state the agent is in, and which actions it can take. The actions part is easy, as this is always the same. The agent can go left, right or forward. The states is slightly more complicated. Using the full 21x21 area as input would be inefficient. Due to all of the information that would have to be conveyed this would result in a very large number of input nodes, which would significantly slow down learning speed. Instead it uses a 5x5 vision grid to limit the number of input nodes, a technique which has been used successfully in other experiments (Shantia et al., 2011; Knegt. et al., 2018).

Rather than use the entire board, these vision grids map the area around the agent. These vision grids are matrices representing the area around the agent using ones and zeros. There are 6 different features that have to be conveyed, meaning that each of these will have its own vision grid, where a 1 marks the presence of that feature and a 0 the absence. These features are the edge of the board, the agent's own tail segments, the agent's own territory, enemy tail segments, enemy territory and enemy heads.

To further reduce the number of different states that have to be considered, it also uses a rotation invariant vision grid, as seen in Knegt. et al., 2018. This means that for each state, all rotation of that state are considered the same state

4 Experimental Setup

The MLPs for the experiment take the previously mentioned vision grid as input, resulting in an input layer of $25 * 6 = 150$ input nodes. They have 500 hidden nodes, and have an output layer of three nodes if estimating actions, or one node if estimating states. The nodes within the MLPs use a sigmoid non-linearity function for activation.

To be able to make a more accurate comparison we use different learning rates for certain algorithms. We used a learning rate of 0.001 as our base. We found that QV-learning performed better with a learning rate of 0.0005 for it's Q and V mlp, and that QVA-learning performed better with a learning rate of 0.002 for its A mlp. The other algorithms performed best with the base learning rate of 0.001.

For the experiment we need to consider ten different algorithm combinations. We run each of these combinations ten times, with 1 million episodes per run. With each episode taking 100 actions or until a player is eliminated. The discount factor that is used is 0.99. Each action has a small penalty of 0.01 to encourage exploration. The exploration that we will be using is ϵ-greedy with ϵ decreasing linearly from 0.4 to 0.01. These parameters are based on Louwers, 2019 and testing to confirm that they work.

Once the networks have been trained we will let them play games against each other to see how well they perform. Per algorithm combination we will have each network of the first algorithm play against each network of the second algorithm for 1000 episodes. This means that for each algorithm

combination we will receive the average points per episode over 10*10*1000 episodes of those algorithms against each other. During these runs the networks will not be learning anymore and we will be using Boltzmann exploration with T = 0.5, instead of ϵ-greedy.

5 Results

The learning curves of the different algorithms can be seen in the appendix. The ten graphs show the ten algorithm combinations. Each line consists of average number of point of the ten different runs with the standard deviation shown in a transparent sleeve around the line. The graphs have the episodes on the x-axis and the average number of points on the y-axis.

Table 5.1 shows the mean number of points of each algorithm over the entire learning process. Per algorithm pair, we calculated these by taking the means of the ten runs, and taking the mean and standard deviation of those. Because the data is not normally distributed, we used the Wilcoxon signed rank test to determine if the differences were significant. We found that in all ten combinations the p-value was smaller than 0.05, which means that the differences between the means are significant.

The results of the final models can be seen in table 5.2. This table shows the average number of points for each algorithm in each algorithm combination. This was generated by having the trained models of each combination play against each other for 1000 episodes. We found that this data is also not normally distributed so we will again be using the Wilcoxon signed rank test to determine significance. We found that the p-value for each of the algorithm combinations is smaller than 0.05. Therefor, the differences between the means are significant.

6 Conclusion

When looking at the final models, we see that Q-learning performs significantly better against every other algorithm. We also see that the final QVA-learning models manage to outperform both double Q-learning, and QV-learning.

When we look at the learning curves and compare the algorithms in that context, QVA-learning performs much better. Beating every other algorithm except for SARSA.

The final conclusion we can take away from this is that QVA-learning shows potential. It manages to outperform established reinforcement learning algorithms. It also performed better in both situations than its predecessor QV-learning, showing that the additions made with QVA-learning appear to have improved learning behaviour.

7 Discussion

QVA-learning performed fairly well, however to determine the true potential of QVA-learning more research will be required. There are a few different experiments that come to mind. First would be testing QVA-learning using self-play rather than other-play. If also done in helenix this could also provide interesting results for comparing the effectiveness of self-play vs other-play. Testing QVA-learning in other games and environments would also be useful for determining its effectiveness. Further fine tuning variables might also show how the AMLP in QVA-learning could best be utilised. Finally, as also suggested in Louwers (2019), adapting the environment to allow more than two players could be interesting. Especially if looking at the performance of the algorithms at different player counts. It would be interesting to see how QVA-learning performs in an environment with a lot of other agents.

References

J. R. Douglas. Chess 4.7 versus david levy. *BYTE*, page 84, 1978.

S. van den Dries and M. A. Wiering. Neural-fitted td-leaf learning for playing othello with structured neural networks. *IEEE Transactions on Neural Networks and Learning Systems*, 23(11): 1701–1713, 2012.

Stefan J. L. Knegt., Madalina M. Drugan., and Marco A. Wiering. Opponent modelling in the game of tron using reinforcement learning. In *Proceedings of the 10th International Conference*

	Q	SARSA	DQ	QV	QVA
Q	-	35.76(1.10)	43.6(0.86)	36.71(0.78)	35.44(0.96)
SARSA	63.93(0.93)	-	67.52(0.68)	70.54(1.09)	68.85(1.12)
DQ	40.23(0.60)	34.80(0.70)	-	36.16(1.00)	35.48(0.56)
QV	62.21(1.14)	65.02(1.18)	65.93(1.06)	-	65.64(1.67)
QVA	63.95(0.80)	66.90(1.26)	67.25(0.72)	69.72(1.43)	-

Table 5.1: Mean points per episode of the left algorithm against the top algorithm calculated over the entire learning process. The standard deviation is shown in brackets.

	Q	SARSA	DQ	QV	QVA
Q	-	185.89(35.85)	200.41(28.95)	200.46(41.75)	195.09(31.67)
SARSA	105.57(28.10)	-	171.01(26.79)	150.91(18.89)	164.47(22.87)
DQ	81.17(21.57)	111.05(18.78)	-	135.69(22.79)	134.30(21.42)
QV	91.86(30.42)	131.68(17.81)	150.07(28.37)	-	125.18(27.04)
QVA	98.25(21.73)	127.68(19.00)	155.09(20.56)	148.44(22.88)	-

Table 5.2: Mean points per episode of the left algorithm against the top algorithm calculated after 1000 episodes of the final models playing against each other. The standard deviation is shown in brackets.

on Agents and Artificial Intelligence - Volume 1: ICAART, pages 29–40, 2018.

Yann LeCun, Leon Bottou, Genevieve B. Orr, and Klaus Robert Müller. *Efficient BackProp*, pages 9–50. Springer Berlin Heidelberg, 1998.

B. Louwers. From self-play to other-play: learning by playing against different algorithms in the helenix environment, bachelor's thesis, university of groningen, 2019.

D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, October 1986.

G. A. Rummery and M. Niranjan. *On-Line Q-Learning Using Connectionist systems.* Cambridge University Engineering Department, 1994.

J. Schilperoort, I. Mak, M. M. Drugan, and M. A. Wiering. Learning to play pac-xon with q-learning and two double q-learning variants. In *2018 IEEE Symposium Series on Computational Intelligence (SSCI)*, pages 1151–1158, 2018.

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

D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lancot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

splix.io. *https://splix.io/about.*

R. S. Sutton and A.G. Barto. *Reinforcement learning: An introduction.* MIT press, 2018.

H. P. van Hasselt. Double q-learning. *Advances in neural information processing systems 23*, 2010.

O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, J. Oh, D. Horgan, M. Kroiss, I. Danihelka, A. Huang, L. Sifre, T. Cai, J. P. Agapiou, M. Jaderberg, A. S. Vezhnevets, R. Leblond, T. Pohlen, V. Dalibard, D. Budden, Y. Sulsky, J. Molloy, T. L. Paine, C. Gulcehre, Z. Wang, Pfaff T., Y. Wu, R. Ring, D. Yogatama, D. Wünsch, K. McKinney, O. Smith, T. Schaul, T. Lillicrap, K. Kavukcuoglu, D. Hassabis, C. Apps, and D. Silver. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature 575*, pages 350–354, 2019.

C. J. Watkins. *Learning from Delayed Rewards, PhD thesis, Cambridge University.* PhD thesis, 1989.

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

M. A. Wiering. Qv(λ)-learning: A new on-policy reinforcement learning algorithm. *Proceedings of the 7th European Workshop on Reinforcement Learning*, pages 17–18, 2005.

Appendix

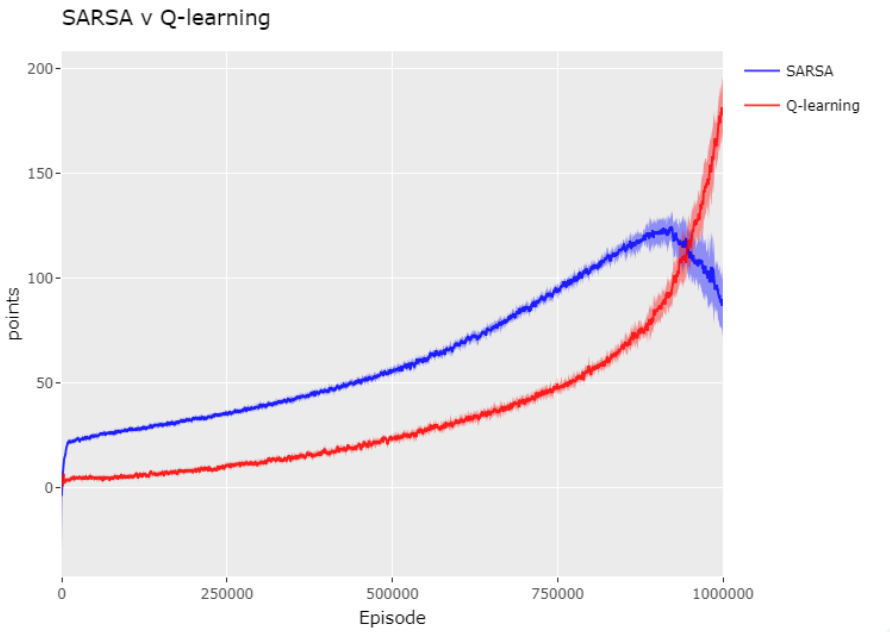

Figure .1: Learning curve of Q-learning versus SARSA

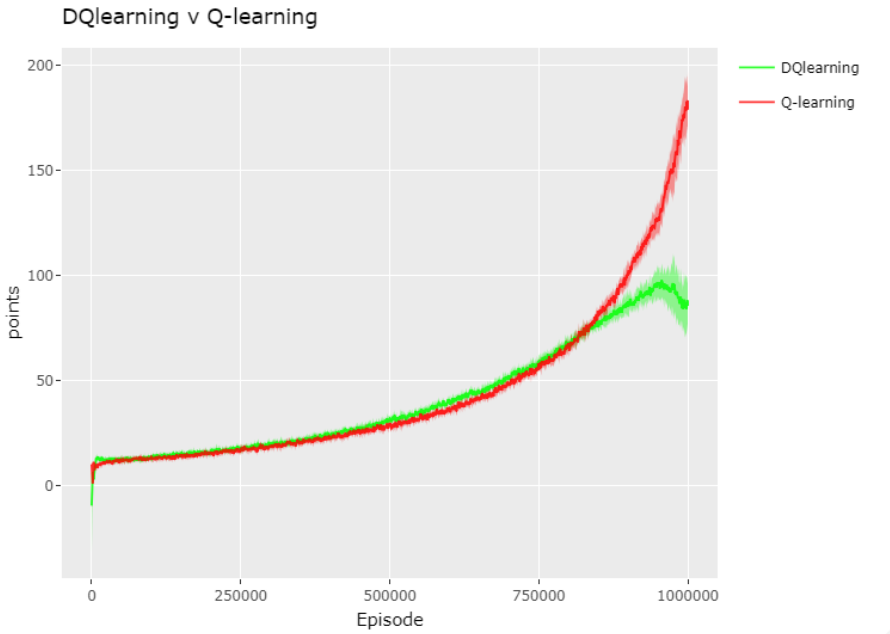

Figure .2: Learning curve of Q-learning versus double Q-learning

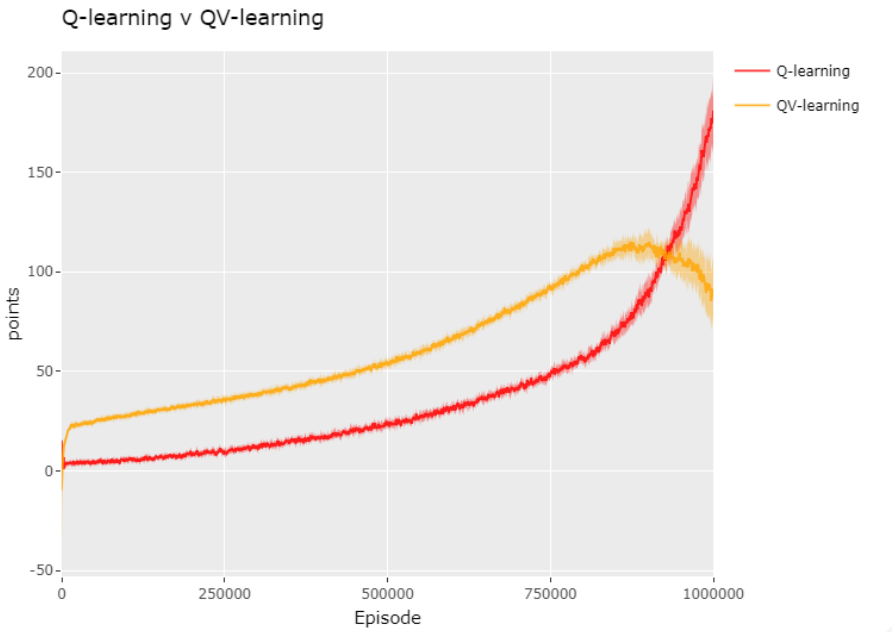

Figure .3: Learning curve of Q-learning versus QV-learning

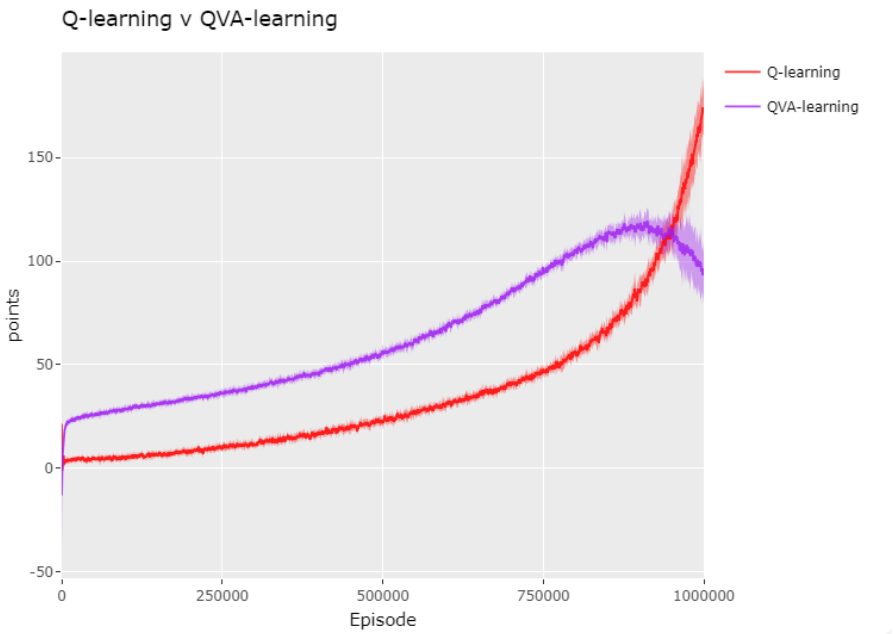

Figure .4: Learning curve of Q-learning versus QVA-learning

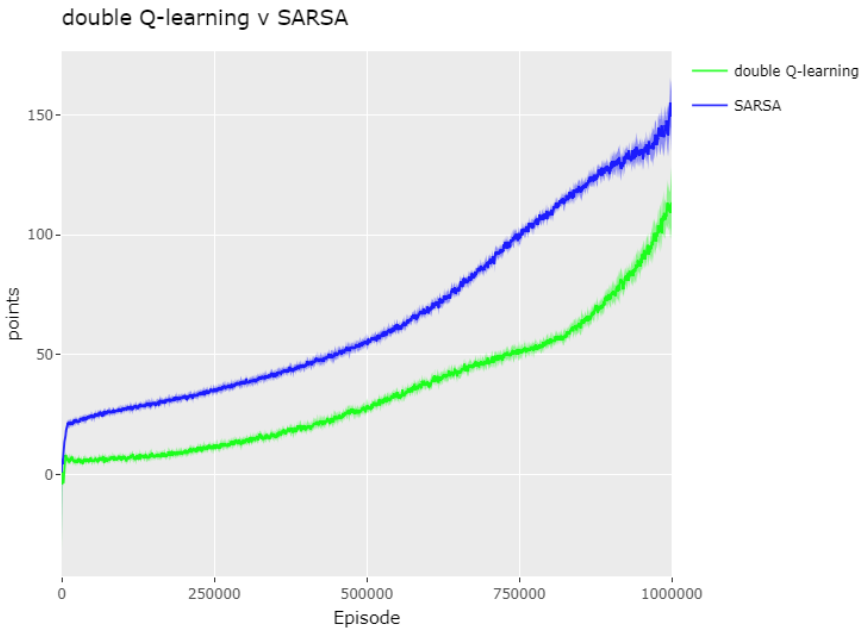

Figure .5: Learning curve of SARSA versus double Q-learning

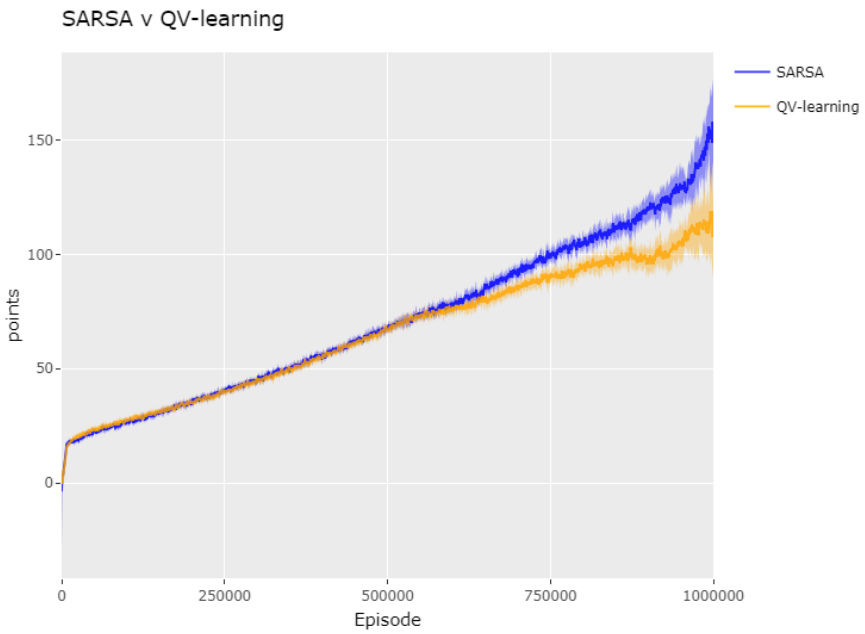

Figure .6: Learning curve of SARSA versus QV-learning

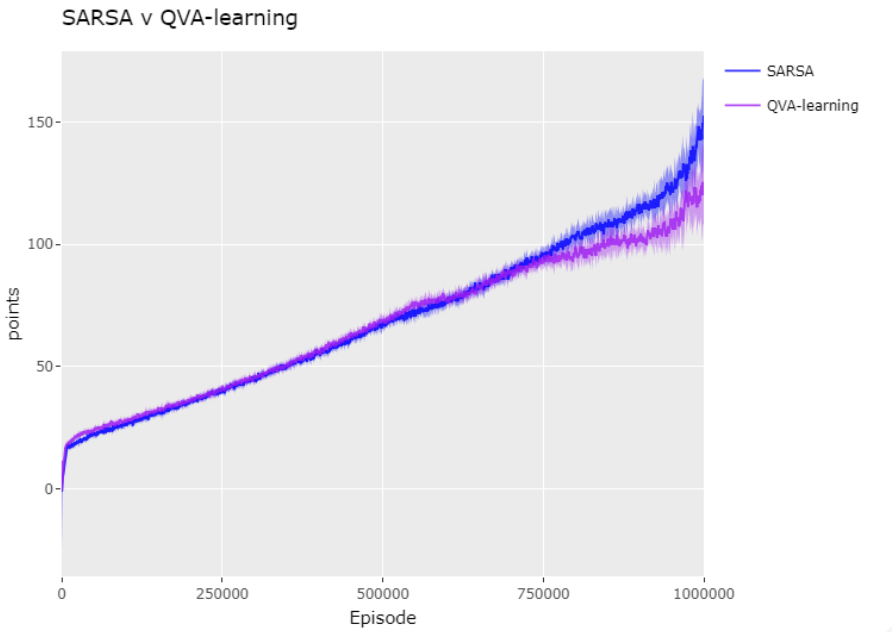

Figure .7: Learning curve of SARSA versus QVA-learning

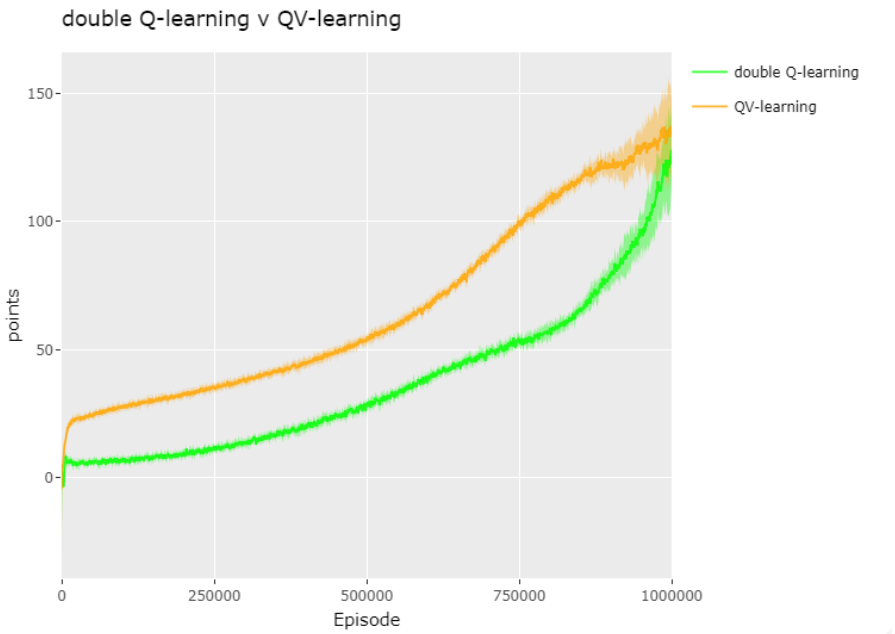

Figure .8: Learning curve of double Q-learning versus QV-learning

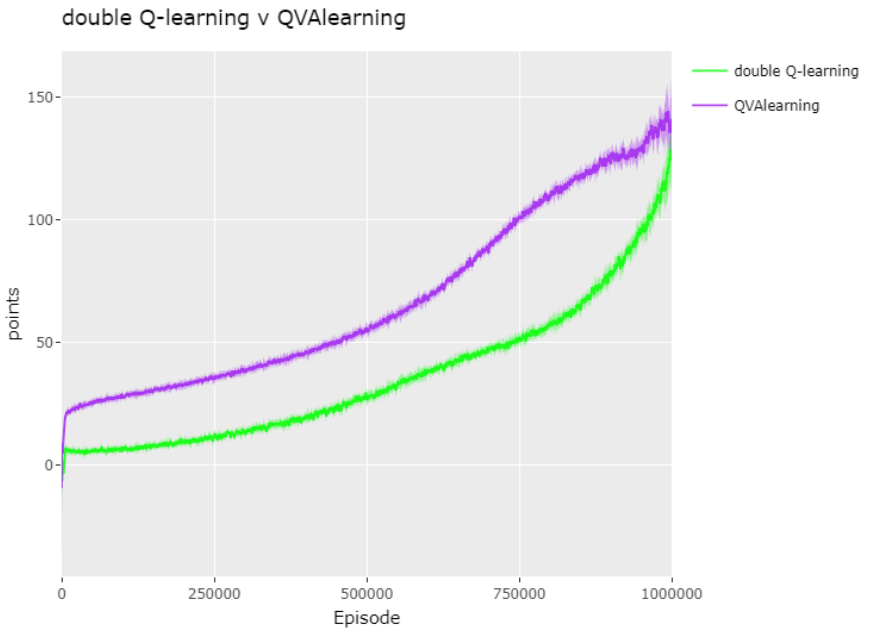

Figure .9: Learning curve of double Q-learning versus QVA-learning

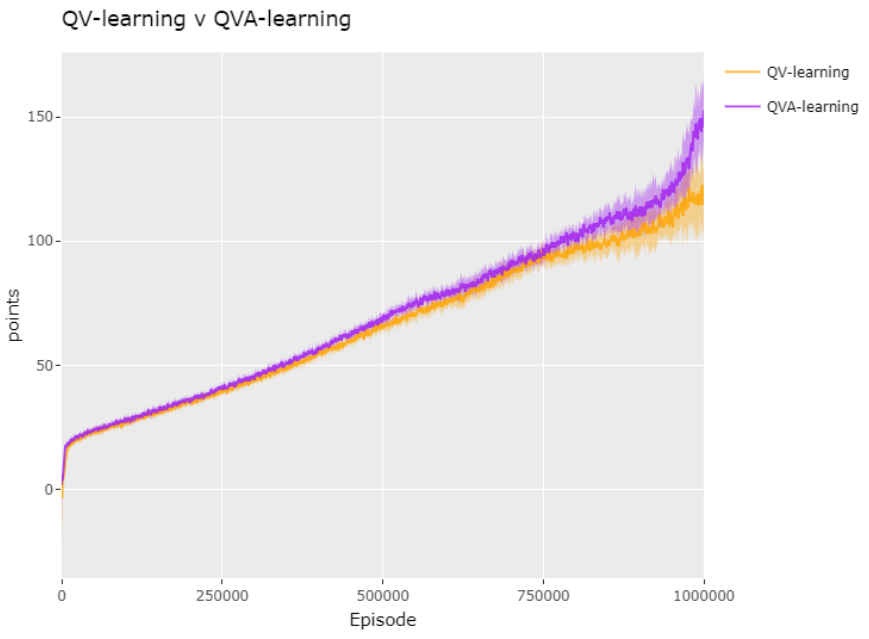

Figure .10: Learning curve of QV-learning versus QVA-learning