university of groningen / faculty of science and engineering

Learning to Play an Aerial Combat Game with Reinforcement Learning

Bachelor's Project Thesis

Jan Willem de Wit, s2616602, j.w.de.wit@student.rug.nl
Supervisor: Dr M.A. Wiering

Abstract: Reinforcement learning has been used previously to let agents learn to play games. We have created a game environment with planes, that have to learn to fly from rewards. They were successful in learning this using the Continuous Actor-Critic Learning Automaton (CACLA). We have extended the planes with the ability to shoot each other. The agents are unable to learn this very well, because the reward from shooting is delayed which makes it difficult to credit the actions leading to the reward correctly. Furthermore, we have explored different configurations of CACLA in a multiagent environment. We have tested configurations that we have named *Personal*, *Shared* and *Team*. They are all successful in learning to fly in the busier environment, learning approximately the same strategy for flying.

1 Introduction

Reinforcement Learning (RL) has been used to learn to play many different games. It has been applied to arcade games like Ms. Pac-Man (Bom, Henken, and Wiering, 2013), Atari games (Mnih, Kavukcuoglu, Silver, Graves, Antonoglou, Wierstra, and Riedmiller, 2013), board games like Othello (Van Der Ree and Wiering, 2013), but also first person shooters (McPartland and Gallagher, 2011), Capture the Flag (Ivanovic, Zambetta, Li, and Rivera-Villicana, 2014), the Prisoner's Dilemma (Gao, 2012), and it has been used to aid humans in playing games (Taylor, Carboni, Fachantidis, Vlahavas, and Torrey, 2014). However, one of the challenges that still remains is the application of RL in continuous state-action game environments. When states and possible actions are limited, it is relatively easy for an agent to find the optimal policy. But when the state space and action space become larger or even infinite, it becomes impossible to try out all possibilities, so we need different ways to deal with this.

In this thesis we will apply reinforcement learning in a game with continuous states and actions. We have created an aerial combat game environment with planes, where throttle and steering are controlled by continuous values. In reinforcement learning, an agent is placed in an environment, where it has to take actions in order to receive the highest rewards. In our game, the difficulty lies in the large amount of possible states and actions.

We will use the Continuous Actor Critic Learning Automaton (CACLA) algorithm (van Hasselt and Wiering, 2007) with a multi-layer perceptron to see if it can be used to teach planes how to fly in the game environment. We then extend this to a multiagent setup, to see if planes can learn to fly in a more busy environment, without crashing into each other. We will experiment with different setups of the algorithm to see if we can improve learning when multiple planes are part of a team.

Additionally, we will explore shooting. We have given the planes the ability to shoot, and they will receive a reward for a bullet hitting a plane. However, this reward is delayed, making it very difficult for the action leading to the reward being reinforced properly. This is known as the temporal credit assignment problem (Kaelbling, Littman, and Moore, 1996). The continuous nature of our game makes this extra difficult, because of the large amount of states leading up to a reward in the case of shooting.

Our research questions are: (1) Can agents learn to fly and shoot in a continuous game environment using the CACLA algorithm? (2) How can we configure CACLA to maximise performance in a multiagent setup?

Outline of this thesis. In section 2, we describe the game environment we developed. In section 3, we discuss the relevant reinforcement learning algo-

rithms. Section 4 explains the experiments we have run and shows the results. We discuss the results in section 5.

2 Game environment

We developed a 2D game environment in which multiple planes can fly. The planes can control throttle, steering and firing of bullets. Both throttle and steering are continuous actions. We have implemented physics that simulate gravity and air pressure. This is set up in such a way that the height planes can reach is limited, due to the increased air pressure. When a plane flies out of the environment on the left, it reappears on the right. When a plane reaches the bottom of the environment, it counts as crashing into the ground, and the plane is destroyed.

The environment has a size of 1000×1000 pixels. The planes are initialised at a random position in the environment, with a random angle. As their initial velocity is zero, they are falling down at first, and have to recover from this. The planes have a starting health of 5, of which one is subtracted every time it is hit by a bullet.

The game is updated 60 times per second in normal speed. The game is reset when all the planes are destroyed. The game is also reset after a certain amount of frames, even if one or more planes are still alive. This is done to save time, and to let planes learn from more different starting positions to increase the amount of exposure they get to different situations.

Figure 2.1 shows a screenshot of part of the game environment, with two planes flying in it. Bullets that have been shot are also visible. The numbers (5) on the planes indicate the amount of health left.

In the game environment gravity is simulated, so that a plane will fall down if no or little throttle is applied. Additionally, we simulate the effect of altitude-dependent air pressure. This prevents the planes from flying too high, and confines them to a limited amount of space. This is important because we want to teach the planes how to avoid other planes, which would be too easy if the environment was infinitely large. In practice, we have seen that the planes were sometimes able to fly higher than the originally intended environment height of 1000, getting up to around 1200 as well. This makes them

Figure 2.1: Two planes flying in the environment

disappear from the visible environment, but it has no consequences for the planes and their learning. When planes fly out of the left or right side of the game environment, they reappear on the other side. This makes it easier for the planes to fly, while keeping the environment space managable.

3 Reinforcement Learning

In reinforcement learning, an agent has to take actions in an environment in order to maximise the sum of received rewards (Sutton and Barto, 1998). In each time step, an agent can use a policy to map the current state to an action. The policy describes for each state what the best action is. Alternatively, a value function can be used. The value function approximates the expected sum of future rewards from a certain action in a certain state. In the next time step, the actual received reward and the value of the next state are used to adjust the policy or the value function, so that the policy or value function becomes closer to the optimal one.

One important aspect of reinforcement learning is exploration. If an agent always executes the actions that it thinks are best, it can never discover different actions that might have an even better outcome. For successful learning, the agent needs to select actions for which the exact outcome is un-

certain every now and then.

3.1 Actor-Critic Methods

Actor-critic methods differ from other reinforcement learning algorithms, by using both a policy and a value function (Sutton and Barto, 1998). In actor-critic methods, the actor determines what action to take in a certain state, making it equivalent to the policy. The critic evaluates the action taken in a specific state, making it equivalent to the value function.

The Temporal-Difference error (TD-error) is used in actor-critic algorithms to determine whether the action selected at time t improved the situation. It is defined as follows:

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

where r_{t+1} is the received reward in the state after selecting the action, γ is the discount factor, V is the value function, and s_t is the state at time step t.

The TD-error contrasts the expected change in value going from state s_t to state $s_{(t+1)}$ to the actual received reward, and therefore describes the discrepancy between the expected value and the actual received reward. If the TD-error is positive, the outcome was better than expected, so the selected action will be strengthened in the actor, so that it will be selected in this situation more often. If the TD-error is negative, the outcome was worse than expected, so the action should be made less likely to be selected in the future.

Actor-critic methods aim to combine the strong points of algorithms that only use an actor or only use a critic (Konda and Tsitsiklis, 2000). They have been used in, for example, playing games (Sundar and Ravikumar, 2014) and robot navigation tasks (Muse and Wermter, 2009), and it can produce behaviour similar to humans and animals in certain tasks (Sakai and Fukai, 2007).

3.2 Continuous Actor Critic Learning Automaton

The Actor Critic Learning Automaton (ACLA) is a type of actor-critic algorithm (van Hasselt and Wiering, 2007). It only uses the sign of the TD-error for determining the update to the actor, and not the actual value. This algorithm is the basis for the Continuous Actor Critic Learning Automaton (CACLA), which is an adaptation to ACLA that allows for continuous actions. This is done by using function approximators.

In our game, the game state includes inputs like position, velocity, and rotation, which are continuous. To map these inputs to a value, we need to use a function approximator. For this, we use a standard multilayer perceptron (MLP) with one hidden layer. The critic MLP approximates the value function. It uses a linear activation function as its outputs values can become large. Another policy MLP is used to output the actions. It uses a sigmoid activation function as the actions are numbers between 0 and 1. Using the value output by the critic, we can calculate the TD-error. We do this using the temporal difference (δ_t):

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

At the end of the game, this becomes:

$$\delta_t = r_t - V_t(s_t)$$

The value of the critic is then updated by backpropagating the following target value through the MLP:

$$T(s_t) = V_t(s_t) + \delta_t$$

For this we use $(s_t, T(s_t))$ as a training example and we backpropagate with a learning rate α to update the weights.

The actor is only updated if the TD-error is positive. Then the MLP is trained to output the target:

$$T(s_t) = a_t$$

If the executed action a_t led to a better outcome for state s_t than expected, the action output by the actor is moved in the direction of this action.

3.3 Action Selection

In actor-critic methods, the actor sometimes needs to select actions other than the preferred action in the current situation, in order to explore. This can be done using ϵ-greedy exploration. In ϵ-greedy exploration, the agent takes a random action with a probability of ϵ, otherwise it selects the best expected action.

An alternative is to use Gaussian exploration. In Gaussian exploration, a random value is added to the action value proposed by the actor. Instead of selecting a completely random action, like in ϵ-greedy, an action is selected that is slightly different than the expected best action. This way, the learnt actions are not discarded completely in exploration, but used as a basis for improvement.

We use Gaussian exploration, with a standard deviation that slowly decreases. The high standard deviation at the beginning allows for actions that are very different from the action that the actor selects. When the standard deviation of the noise becomes low, only slight changes are made to the action selected by the actor, which allows for the actions to be refined.

3.3.1 Smoothing

We have constructed an exploration method that smoothens the noise over multiple time steps. In our game, a single action has relatively little effect on the state. Because the average noise added to the action over time is zero, the small impact of the noise will result in very little overall change. By smoothening the noise, about the same amount of noise is effectively used in a number of frames, allowing flying behaviour to change in a more consistent way. We use the following formula to smoothen the noise:

$$z_t = z_{t-1} * s + y_t * (1 - s)$$

where z is the noise added to the action, s is the smoothing parameter, and y_t is noise sampled from a Gaussian distribution. Effectively, it adds some new noise on top of the old noise, instead of using random noise in every time step.

4 Experiments and Results

The game is set up in such a way that it automatically restarts after all planes have been destroyed. To prevent games from running forever when planes have learnt how to fly perfectly, we also reset the game after a while regardless of how many planes are still alive. We do this after 1200 frames, which equals 20 seconds when run in real-time. When the program is running, the amount of noise added to the actions for exploration is slowly reduced. This way, after a certain amount of games is played, no significant amount of noise is added anymore. For getting our test results, we alternate between training and testing modes. In training mode, noise is added to the action for exploration. In testing mode, no noise is added to the actions. This way, the planes can demonstrate what they have learnt. During testing, the total scores for each game are saved. The random initialisation of the positions of the planes has a lot of influence on performance; if planes are placed low to the ground or too close to a different plane, it might simply be impossible for it to recover. To not let these outliers affect the data too much, we average over these games to give a performance score at this time in training. We run this entire simulation until the amount of noise gets low; we vary the exact amount of games played in a simulation per setup. Then, the neural networks for CACLA are initialised again with weights between -0.5 and 0.5, and the simulation is run again. In total, the results are averaged over 20 simulations.

4.1 Single Agent Experiment: Flying

To evaluate the way planes learn to fly, we use a setup with two teams, both teams having one plane. So in total, there are two planes in the environment. The planes receive a small positive reward if they fly high, and a small negative reward when they fly low, according to the following formula:

$$r_t = 1 - (Y/1000) - 0.2$$

where r_t is the height-based reward at time t and Y is the y-position of the plane (note that $Y = 0$ is at the top of the environment and $Y = 1000$ at the bottom).

Additionally, the agents receive a strong negative reward (-100) when they crash, either into the ground or into the other plane. The planes do not have an action for shooting in this experiment, and therefore do not get any rewards related to shooting.

4.1.1 Network inputs and outputs

We use the following inputs for the networks in the single agent setup.

From the plane's own information:

- rotation

- x-position
- y-position
- angular velocity
- x-velocity
- y-velocity
- absolute speed

The following information is added from other planes in the environment:

- rotation
- x-position
- y-position
- x-velocity
- y-velocity
- team identifier
- health status (alive/dead)

To limit the scale of the inputs, the position values are divided by 1000 (the size of the environment), and the velocities are divided by 10. The rotation is a value between $-\pi$ and π.

The actor outputs two values: throttle and rotation. Both are values between 0 and 1. The rotation is then scaled so that an output close to 0 results in a sharp turn to the left, an output close to 1 results in a sharp turn to the right, and at a value of 0.5, the plane does not change direction.

4.1.2 Parameters

We performed preliminary experiments to tune the hyperparameters. We start the standard deviation of the noise at a value of 0.7. Every update, this value is multiplied by 0.999999. This way, it takes about 3000 games for the standard deviation to become less than 0.1. The noise smoothing parameter is 0.9. We used a discount factor of 0.98. For the actor, we used 25 neurons in the hidden layer. For the critic, we used 15 hidden units. We will compare different learning rates, but keep these the same between the actor and the critic. For testing, we test performance every 100 training games. We then test for 25 games without noise added, and average this score to calculate the average performance at this time in learning. We reset the game if all the planes have died, or if 1200 frames have passed.

The learning rates we will compare are:

- 0.0001
- 0.0005
- 0.001
- 0.005

4.1.3 Results

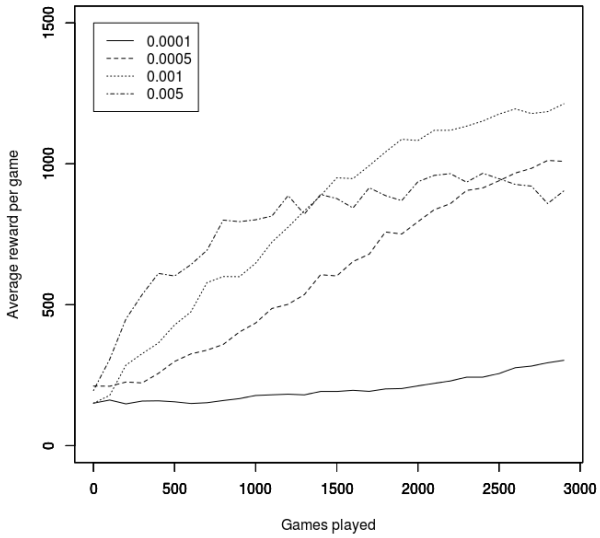

Figure 4.1: Different learning rates compared for single agent flying experiment

Figure 4.1 shows the performance of a single agent learning to fly with different learning rates. With a very low learning rate of 0.0001, there is very little improvement after 3000 games. Setting the learning rate higher at 0.0005, the average reward increases a lot more. Increasing the learning rate to 0.001 yields another improvement. The agent learns quicker, and ends up at a higher average reward after 3000 games. When we increase the learning rate even further to 0.005, the performance increases even faster at first, but stagnates later, yielding less average reward than learning rates of 0.001 and 0.0005 after about 3000 games played.

4.2 Single Agent Experiment: Shooting

To evaluate shooting performance, we compare two CACLA agents. They have been trained to fly according to section 4.1. We then turn off height-based rewards for the agents. For one plane, we enable rewards from bullets hitting the other plane. We will let the agent train using this reward, to see if it can learn to consistently hit the other plane by shooting. We do not give an agent that is hit by a bullet a negative reward, as there is little a plane could do to avoid being hit by a bullet. The amount of bullets that can be shot in succession is limited. After each shot, the next shot can only occur 4 frames later. Also, a maximum of 8 bullets can be shot right after each other. Every bullet shot is added to the counter, but every time the plane does not decide to shoot, the counter is decremented. If the maximum of 8 is reached, 30 frames (half a second in real-time) need to pass before the plane can fire again. This is done to prevent the planes from firing continuously to increase possible rewards. This counter is not part of the input.

4.2.1 Network inputs and outputs

We use the same inputs as in the non-shooting setup, as described in section 4.1.1. We have added an output to the actor for shooting. It outputs a continuous value just like the throttle and steering, that we interpret as the willingness to shoot. If the value is above 0.5, a bullet will be shot, and if it is below 0.5, the plane will not shoot.

4.2.2 Parameters

For shooting, we decrease the speed at which the standard deviation of the noise changes. We use a multiplier of 0.9999995. This way, there is more time for the planes to learn. We set the learning rate to 0.0005. This value is on the low side to avoid negative forgetting effects from having a learning rate that is too high. For testing, we vary the discount factor.

The discount factors we will compare are:

- 0.9
- 0.99
- 0.999

4.2.3 Results

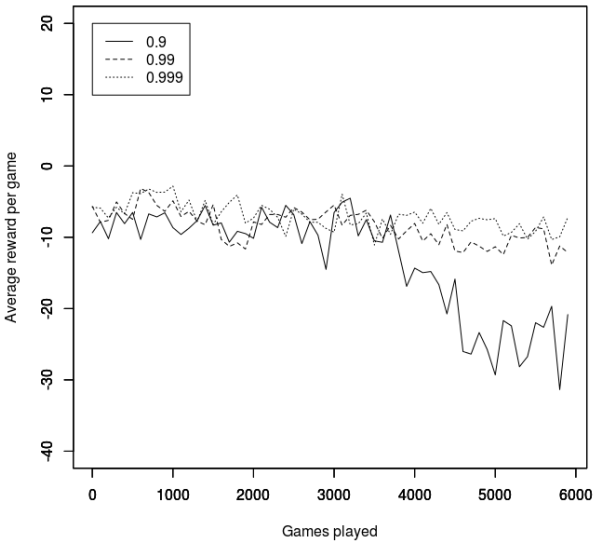

Figure 4.2: Different values for discount factor compared for single agent shooting experiment

Figure 4.2 shows the performance of an agent that was trained to fly previously, but now only receives rewards for bullets hitting an opposing plane, tested for different values of the discount factor. It shows that after 6000 games played, with none of the discount factors the agent was able to improve its performance. Looking at the data, agents sometimes manage to get some successful bullet hits, as indicated by the total reward received in a game being above 0. But the average score is always below zero, caused by the negative reward (-100) the agents receive from crashing. The performance for a discount factor of 0.9 decreases a lot after about 3000 episodes, to the point where the agent crashes much more often than at the beginning of training, and more than for the other two discount factor values. The difference in performance between discount factors of 0.99 and 0.999 is very small.

4.3 Multi-Agent Experiment

The game allows for multiple planes to be part of one team. This makes the game more complicated, as there are more planes that one plane needs to avoid hitting. We add a second plane to each team, for a total of 4 planes in the environment. We will

compare flying performance for three different configurations. Because of the poor results from the shooting experiments with one plane per team, we have disabled shooting for this experiment.

4.3.1 Network inputs and outputs

We use the same network inputs as in the single agent setup, as described in section 4.1.1. Additionally, we have added the team identifier and health status to the information about the plane being controlled. This means that there are now 9 inputs for the plane being controlled, and 7 inputs for every other plane, for a total of $9 + 7 + 7 + 7 = 30$ inputs. In one of our configurations, two planes are controlled by the same actor and critic; this configuration has $9 + 9$ inputs for the two planes that it controls, and $7 + 7$ inputs for the two planes on the other team, for a total of 32 inputs. The outputs by the actor are the throttle and steering. But for the configuration where two planes are controlled by the same actor and critic, actions for both planes are output at the same time; this means that there is a total of 4 outputs from the actor.

4.3.2 Parameters

For the multi-agent setup, we increase the amount of neurons to 50 for the actor, and 30 for the critic. We also decrease the speed at which the standard deviation of the noise changes, to a multiplier of 0.9999999. This way, it takes about 25,000 games for the standard deviation to drop below 0.1.

We will compare the following architectures for CACLA:

- *Personal CACLA* Each plane has its own actor and own critic, and therefore its own policy and value function. This data is not shared between planes of the same team. Planes get a personal reward, which is used to update their own actor and critic.

- *Shared CACLA* Each team has one actor and one critic. Both planes update the same actor and the same critic, and they use their personal reward for the update. This means that the actor and critic are updated twice for every frame. Because two planes on the same team have the same interests, the increased amount of updates per game update might cause this configuration to learn quicker.

- *Team CACLA* Each team has one actor and one critic. Instead of having inputs and outputs for one plane at a time, the actor and critic in this configuration have inputs and outputs for all planes on a team at the same time. This means that for the actor, the input consists of the full information for the planes on the team, and basic inputs for all planes on opposing teams. The output consists of controls for each plane on a team. For the critic, the input is the same as for the actor: full information for the planes on the team, and basic inputs for all planes on opposing teams. It only outputs a value for the entire team, as there is no way to credit rewards to individual planes. This also means that for updating CACLA, the summed time-step reward for all the planes is used.

4.3.3 Results

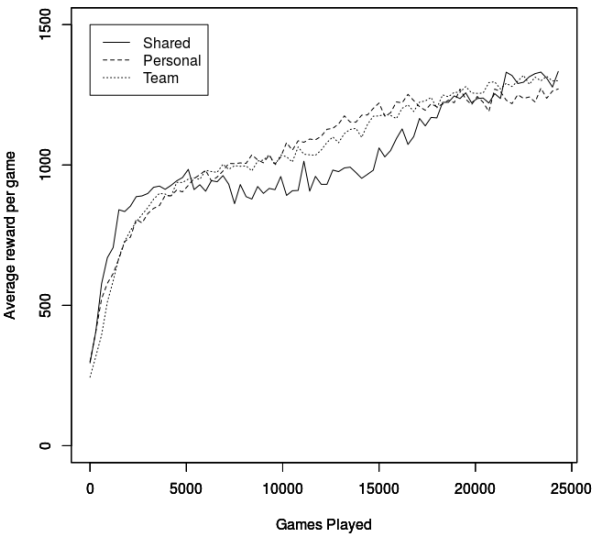

Figure 4.3: Performance of multi-agent architectures

Figure 4.3 shows the performance of the three different multi-agent configurations. After about 25,000 games, all three configurations have very similar performance; none of the three configurations is better than the rest. From observing

the planes in normal speed after they have been trained, they all show rather good evasive behaviours. In fact, the flying behaviour seems very similar overall between the three configurations; they have likely found the same strategy for flying. However, they do not always succeed in avoiding other planes; especially in difficult starting conditions, where two planes are initialised close to each other, they are not always able to avoid each other.

5 Discussion

We have shown that CACLA can be applied to a 2D game in which planes need to learn how to fly. Agents can quickly learn to adjust their actions to prevent crashing into the ground. They also learn to fly as high as possible, gaining what is practically the maximum reward. While avoiding other planes is not always successful, they learn evasive behaviours as well.

We have experimented with learning rates of 0.0001, 0.0005, 0.001, and 0.005. The results were best with 0.001, but when we increased it further to 0.005, performance in the end is worse. Because it takes relatively long to teach planes how to fly in this environment, especially in the multi-agent setup, we would like to use a learning rate that is as high as possible, but there is the risk that the agent will not be able to learn the optimal policy.

We have tested flying with different learning rates. However, we use the same learning rate for the actor as we use for the critic. It is possible that having different learning rates for the actor and critic could yield even better performances.

When we expand the game with shooting, the agents do not manage to consistently hit planes from the opposing team. We have experimented with different discount factors, but none of them results in consistently good performances. A discount factor that is too low actually decreases gained reward over time, which indicates that planes are destroyed more often than they used to. In our case, that happened with a discount factor of 0.9. Because the reward of a bullet hit is delayed, it may be the case that the agent associates possible bullet hits with flying close to the other plane, increasing the likelihood of crashing into this plane. The reason why shooting is so difficult, is due to the complexity of the game and the fact that the reward is delayed. A bullet hitting the other plane is a relatively rare occurence in the game, as a limited number of bullets can be shot in quick succession. This is done to prevent planes from just shooting all the time in the hope of hitting something. But this also means that the amount of successful bullet hits to learn from is limited. Besides that, a bullet hit is dependent on many different factors. For example, the position of the opponent is important, but out of control for the attacking plane. This makes it hard for the plane to learn a generalisable strategy for shooting from a limited number of bullet hits.

Shooting is the most challenging part of the game. Considering the time it takes for planes to learn how to fly, it is plausible that simply much more training time is needed to increase the amount of bullet hits the plane can learn from. Alternatively, we could increase the amount of bullets a plane can shoot in a limited amount of time, but this will make it even more difficult to assign credit to the right actions.

Because the events and states before shooting are crucial for a bullet hit, there may be a need for putting more focus on previous states. Perhaps it would work to save all previous states, and put more focus on the events leading up to successful bullet hits, putting the plane in a more favourable situation for learning.

In a multi-agent setup, we have seen that the configuration of the network does not matter too much. Having a shared actor and critic, where the actor and critic are updated by both planes sequentially, slightly increases learning speed in early games, but falls behind in the middle of training, to catch up again in the end. In the case of having one network with inputs and outputs for two planes at the same time (the *Team* configuration), the fact that the reward of both planes combined is used for updating CACLA does not seem to hinder performance. Because there is only one update per team in every time step, instead of one update per plane, this method requires less computational power.

However, the *Team* configuration has more inputs and outputs for the actor and the critic. It is likely that it needs more hidden neurons, while the *Personal* and *Shared* configurations could do with fewer. We have kept the amount of hidden neurons the same for each configuration. We also chose a relatively large amount of hidden neurons. Because a larger amount of neurons increases the compu-

tation time needed, experiments could be done to determine the lowest amount of neurons possible without performance decrease. Additionally, testing could be done with more hidden layers. This can improve performance in certain problems (Sontag, 1992). This could determine if having multiple layers improves performance in our case or allows for a lower total amount of neurons.

The fact that the *Team* configuration performs very well, despite the fact that rewards are combined for all the planes, is interesting. This configuration means that when one plane flies exceptionally well while the other performs badly, the actions for both planes are reinforced. While this does not prevent the agents in our game from keeping up with the other configurations, it is very likely that this approach will cause much slower learning in other tasks, or that it will simply not work at all. It might also be less successful in a configuration with more than two planes.

We have observed that flying in the multiagent setup is not completely optimal yet, but the planes do show evasive behaviour. From observing the planes, it seems like the strategy they learn is very similar. It might be that this is a local optimum, that each of the configurations learns successfully. It is possible that the relatively low learning rate simply limits the speed at which either of the configurations can learn. Perhaps the *Personal* and *Shared* configurations can benefit from a higher learning rate more than the *Team* configuration can.

In future research, experiments could be done to optimise parameter settings. Besides that, different approaches need to be developed to improve performance in tasks in continuous environments in which the reward is delayed.

References

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

Y. Gao. A reinforcement learning based strategy for the double-game prisoner's dilemma. *CEUR Workshop Proceedings*, 918:317–331, 2012.

J. Ivanovic, F. Zambetta, X. Li, and J. Rivera-Villicana. Reinforcement learning to control a commander for capture the flag. In *2014 IEEE Conference on Computational Intelligence and Games*, pages 1–8, August 2014.

Leslie Pack Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4: 237–285, 1996.

Vijay R. Konda and John N. Tsitsiklis. Actor-critic algorithms. In *Advances in neural information processing systems*, pages 1008–1014, 2000.

M. McPartland and M. Gallagher. Reinforcement Learning in First Person Shooter Games. *IEEE Transactions on Computational Intelligence and AI in Games*, 3(1):43–56, March 2011.

Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

David Muse and Stefan Wermter. Actor-Critic Learning for Platform-Independent Robot Navigation. *Cognitive Computation*, 1(3):203–220, September 2009.

Yutaka Sakai and Tomoki Fukai. The Actor-Critic Learning is Behind the Matching Law: Matching Versus Optimal Behaviors. *Neural Computation*, 20(1):227–251, November 2007.

E. D. Sontag. Feedback stabilization using two-hidden-layer nets. *IEEE Transactions on Neural Networks*, 3(6):981–990, November 1992.

D. Krishna Sundar and K. Ravikumar. An actor–critic algorithm for multi-agent learning in queue-based stochastic games. *Neurocomputing*, 127:258–265, 2014.

Richard S. Sutton and Andrew G. Barto. *Reinforcement learning: an introduction*. Adaptive computation and machine learning. The MIT press, Cambridge, MA, 1998.

Matthew E. Taylor, Nicholas Carboni, Anestis Fachantidis, Ioannis Vlahavas, and Lisa Torrey. Reinforcement learning agents providing advice in complex video games. *Connection Science*, 26 (1):45–63, March 2014.

Michiel Van Der Ree and Marco Wiering. Reinforcement learning in the game of othello: learning against a fixed opponent and learning from self-play. In *Adaptive Dynamic Programming and Reinforcement Learning (ADPRL), 2013 IEEE Symposium on*, pages 108–115, 2013.

Hado van Hasselt and Marco A. Wiering. Reinforcement learning in continuous action spaces. In *Approximate Dynamic Programming and Reinforcement Learning, 2007. ADPRL 2007. IEEE International Symposium on*, pages 272–279, 2007.