Serpentine StarCraft II Agent - Danger Noodle

Team Serpentine, M.R.M. Beurskens, W.J.G.M. van den Bemd

June 2019

1 Abstract

In June of 2019 team Serpentine submitted a StarCraft II minigame agent to the FruitPunch AI Competition, which was the debut competition of our team. In this competition we designed and implemented a machine learning based agent that was able to play a variation on a StarCraft II game. We managed to end in first place using a reinforcement learning setup with a neural network, implementing Q-learning and an epsilon greedy training policy. We managed to create coherent behaviour that allowed us to win. However scripted tactics still appeared superior and the approach did not generalize well against stationary opponents. A more structured implementation approach, modular enough to test different parts of the algorithm automatically and to debug much more efficiently, is required to solve problems like this in the future. Going forward we believe that separating concerns during implementation for these kinds of algorithms is essential in order for them to be usable.

2 Introduction

In June of 2019 team Serpentine submitted a StarCraft II minigame agent to the FruitPunch AI Competition, which was the debut competition of our team. In this competition we designed and implemented a machine learning based agent that was able to play a variation on a StarCraft II game [14]. StarCraft II is a real time strategy game where economy management and unit control play a central role. This challenge focused on the unit management side and abstracted away the economic problem. The challenge thus became much simpler, and we were able to design, implement and train an agent in six weeks time. We competed against four other teams and ended up in first place.

This document is structured as follows. In the related works we will provide overview of other machine learning based agents and suitable game based artificial intelligence environments. The section on the challenge environment provides a description of the StarCraft II minigame we used to compete and corresponding specifications. Following this we introduce the design and implementation of the code and conclude with our takeaways and future intentions.

3 Related Work

Video games are a popular platform to test and train artificial intelligence algorithms [12] [3] [16]. StarCraft II is an example of such a game in which the problem is mostly framed as a sparse reward problem with partial state knowledge [14]. Game developer Blizzard released an API in order to easily interface with the game using Python and supports research using their platform. Using the level editor it is possible to make custom games as well. DeepMind used this editor to benchmark several agents playing subsets of the real game [5]. The full game has been played by agents to some extent by Tencent [10] and DeepMinds' AlphaStar [13].

Other gaming platforms that are extensively used for AI research include OpenAI Gym [4] [11] for ATARI games and ViZDoom [7] [15] for visual based reinforcement learning in Doom. We used the DeepMind papers on Deep Q-learning Networks for ATARI games [8] [9] as an inspiration for our StarCraft II agent. Some recent work in Quake 3 Arena [6] highlights algorithms geared towards multiple players as well. Finally, Google recently published an open source platform to evaluate reinforcement learning algorithms in particular using a soccer game environment [1].

4 Challenge Environment

The competition presented a modified version of the game StarCraft II as a platform for teams to compete. The challenge consists of two opposing players controlling five identical units each as seen in Figure 1. The goal of the game is to get a score of 200, or alternatively secure the highest score after 4 minutes. Points are awarded for capturing a flag in the middle of the screen or for defeating enemy units, rewarding 10 points and 5 points respectively. Units spawn after they are defeated as soon as their individual cool down timers expire. Figure 2 shows an overview of the map used in the competition.

Units can be moved individually or in groups, are able to attack normally and are able to use a ranged attack once every 2 seconds which cover an area of attack. The game is controlled through the PySC2 API [2] using interactions such as "select all units", "move camera" and "activate special attack". Interactions in this framework are limited to 2 interaction per second for artificial agents. As a consequence it does not seem viable to micromanage individual units, as performing actions could take anywhere between 1 to several seconds. Such a time window is slow relative to the pace of the game. Through experimentation we extracted more details about the environment which are listed in Appendix A.

5 Software Architecture

The serpentine Starcraft II bot has a hierarchical structure based on hand built policies which correspond to actions an agent is able to perform. A deep neu-

Figure 1: Units on the map from the two opposing teams. One team is colored red and the other team is colored blue.

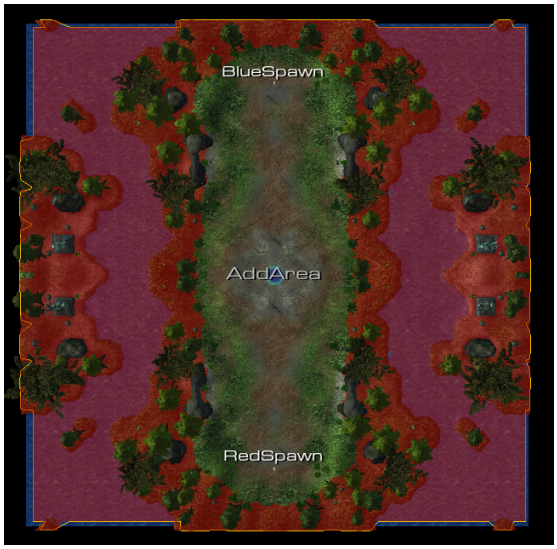

Figure 2: Overview of the competition map. Two spawn points at the top and bottom of the map, "BlueSpawn" and "RedSpawn" respectively, produce units when they are defeated. The capture point "AddArea" in the middle contains the flag.

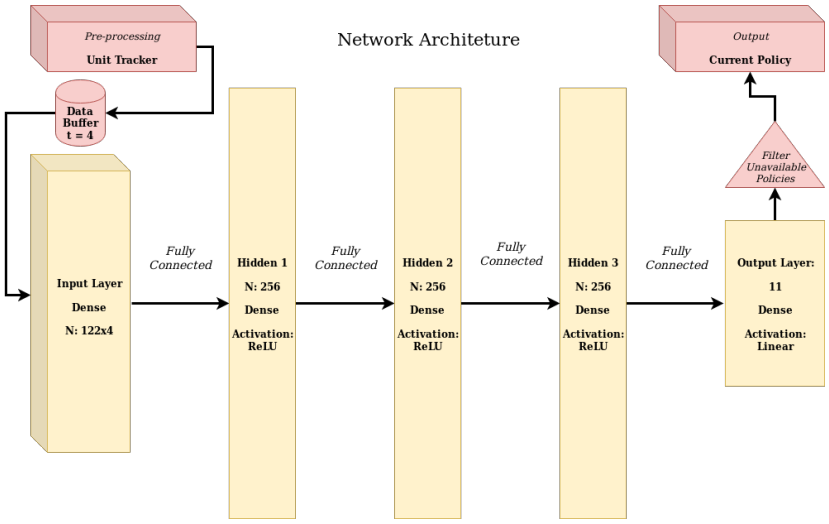

Figure 3: Architecture of the neural network used to model agent behaviour. The network consists of fully connected layers. Input comes from the unit tracker using multiple time steps to capture temporal information. Output is the policy the agent should execute. In this case a subset (11) of the total number available (21).

ral network is used to map observations to policies. The agent trains against scripted agents and is rewarded for increasing it's own score. By iterating training cycles, the network maps inputs to outputs while maximizing the score.

5.1 The Network

The network is built up of several fully connected layers as represented in Figure 3. It takes input from a pre-processing system called the unit tracker, and outputs expected rewards for the policies that the agent can perform. In essence the network is trying to learn the mapping from observation to a policies expected reward as accurately as possible, and we simply instruct it to always choose the best option available to it. We always try to perform the policy with the highest expected reward, however when this is not possible we simply default to another policy until the agent finds one that does work.

Because the network models the game, it is possible for it to learn without actually winning in each training session. The more accurate the model, the better we expect our performance to be.

The input to the network does not only include the output of the unit tracker for the current time step but also for a number of previous time steps. This way we try to encode temporal information into the network.

5.2 Pre-Processing

The pre-processing unit is called the unit tracker. It reads the PySC2 API and constructs a number of useful metrics for the game including but not limited to: Friendly and enemy unit coordinates, death flags, spawn timers and pineapple spawn timers. It is a compact representation of the information that can be gained from the API. This pre-processing step allows us to have smaller networks, and we are able to do away with any convolutional layers that might be necessary when processing inputs as image data.

5.3 Policies

Policies are low level actions that agents can perform. They often correspond to API action calls to the Starcraft interface with hard coded parameters. They can also be a simple combination of action calls. Examples include "Move camera to pineapple", "Attack nearest enemy" and "Walk to left flank". The total number of policies is 21, but we do not always include all policies during training. The neural network outputs a predicted reward for each of these policies, and we choose the one that is expected to yield the highest score.

However, the StarCraft environment does not allow for certain actions to be performed at certain points in the game. Instead of changing the architecture of the network at each time step to accommodate availability of actions, the output is filtered. If the chosen policy cannot be executed the net most likely one is passed to the filter until an available policy is found until finally a "No-Op" policy is executed, which does nothing but is always available. This allows the networks structure to be static throughout simulation and training.

6 Implementation and Training

The network used for this agent should model the expected reward as close as possible to the actual reward gained. Therefore it should have explored a large amount of states, for which it should all learn to predict the expected reward accurately.

To ensure that the agent does not fall into repetitive behaviour in a small subset of the total states the epsilon greedy algorithm is used. This forces the agent to select random actions during training. Only random actions are selected at first, but as the training continues, the percentage of forced random actions is also reduced. This allows the agent to explore the state space similar to its normal behaviour, thus making it better at generalizing at new situations.

The input to the network is an array storing the output of the unit tracker over 4 previous time steps. The output is equal to the amount of policies available to us at the game. If a policy is unavailable, we choose the next best policy until once can be executed.

Training was done by running approximately 6000 games before the performance capped. We trained the network alternating between mirrors of the network and a scripted agent set to immediately capture the flag with 5 units.

7 Results

Our agent played in the tournament against three other teams. Two of those teams had agents that did nothing at all. Against these agents we appeared to also score nothing. Our agent has not generalized well against stationary opponents, which is to be expected as these kinds of opponents were not included in the training set. Even when training against ourselves, at least part of the move set is executed stochastically because of our implementation of the epsilon greedy policy.

Against the last opponent we did manage to win. This opponent moved rather stochastically against which our agent was equipped by at least moving to the center of the stage to capture the flag and attacking opponents. However, we were never able to beat our own scripted agent, indicating that the performance of our agent was lacking despite winning the competition.

8 Conclusion

One of the core problems this competition has been the huge problem space and lack of modularity in our implementation. This made testing small pieces of code impossible, and hindered troubleshooting. A more structured approach to testing the implementation of neural networks is necessary. These black box models are tough to understand, so every logistic piece of code that can be untangled and tested separately will provide an advantage in troubleshooting and improve understanding. We believe that separating concerns during implementation for these kinds of algorithms is essential in order for them to be usable.

Implementing AI algorithms is not a trivial exercise. It requires both understanding of theory and experience in implementation. Other approaches should be considered seriously in the future as well. The problem space of the competition was not sufficiently complex to void other more easily understandable approaches to the problem. We would like to conclude that the simplest approach to solving the problem, while still addressing the core requirements of the solution, is probably the best one. Therefore it is safe to say that this competition broadened both our understanding and our definition of AI. We will keep exploring more algorithms and approaches, both simple and complex, as our expertise and understanding grows. we hope to steadily increase the performance of our systems going forward and challenge ourselves again with larger competitions in the future.

References

[1] Introducing a new framework for flexible and reproducible reinforcement learning research. `https://ai.googleblog.com/2018/08/introducing-new-framework-for-flexible.html`. Accessed: 2019-06-25.

[2] Pysc2 - starcraft ii learning environment. `https://github.com/deepmind/pysc2`. Accessed: 2019-06-25.

[3] Christopher Amato and Guy Shani. High-level reinforcement learning in strategy games. In *Proceedings of the 9th International Conference on Autonomous Agents and Multiagent Systems: Volume 1 - Volume 1*, AAMAS '10, pages 75–82, Richland, SC, 2010. International Foundation for Autonomous Agents and Multiagent Systems.

[4] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016.

[5] Deepmind. Starcraft ii agents. `http://starcraftgym.com/`.

[6] Max Jaderberg, Wojciech M. Czarnecki, Iain Dunning, Luke Marris, Guy Lever, Antonio García Castañeda, Charles Beattie, Neil C. Rabinowitz, Ari S. Morcos, Avraham Ruderman, Nicolas Sonnerat, Tim Green, Louise Deason, Joel Z. Leibo, David Silver, Demis Hassabis, Koray Kavukcuoglu, and Thore Graepel. Human-level performance in first-person multiplayer games with population-based deep reinforcement learning. *CoRR*, abs/1807.01281, 2018.

[7] Michal Kempka, Marek Wydmuch, Grzegorz Runc, Jakub Toczek, and Wojciech Jaskowski. Vizdoom: A doom-based AI research platform for visual reinforcement learning. *CoRR*, abs/1605.02097, 2016.

[8] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.

[9] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Fidjeland, Georg Ostrovski, Stig Petersen, Charles Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015.

[10] Peng Sun, Xinghai Sun, Lei Han, Jiechao Xiong, Qing Wang, Bo Li, Yang Zheng, Ji Liu, Yongsheng Liu, Han Liu, and Tong Zhang. Tstarbots: Defeating the cheating level builtin AI in starcraft II in the full game. *CoRR*, abs/1809.07193, 2018.

[11] Ruben Rodriguez Torrado, Philip Bontrager, Julian Togelius, Jialin Liu, and Diego Pérez-Liébana. Deep reinforcement learning for general video game AI. *CoRR*, abs/1806.02448, 2018.

[12] Aaron Tucker, Adam Gleave, and Stuart Russell. Inverse reinforcement learning for video games. *CoRR*, abs/1810.10593, 2018.

[13] Oriol Vinyals, Igor Babuschkin, Junyoung Chung, Michael Mathieu, Max Jaderberg, Wojciech M. Czarnecki, Andrew Dudzik, Aja Huang, Petko Georgiev, Richard Powell, Timo Ewalds, Dan Horgan, Manuel Kroiss, Ivo Danihelka, John Agapiou, Junhyuk Oh, Valentin Dalibard, David Choi, Laurent Sifre, Yury Sulsky, Sasha Vezhnevets, James Molloy, Trevor Cai, David Budden, Tom Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Toby Pohlen, Yuhuai Wu, Dani Yogatama, Julia Cohen, Katrina McKinney, Oliver Smith, Tom Schaul, Timothy Lillicrap, Chris Apps, Koray Kavukcuoglu, Demis Hassabis, and David Silver. AlphaStar: Mastering the Real-Time Strategy Game StarCraft II. `https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/`, 2019.

[14] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, John Quan, Stephen Gaffney, Stig Petersen, Karen Simonyan, Tom Schaul, Hado van Hasselt, David Silver, Timothy P. Lillicrap, Kevin Calderone, Paul Keet, Anthony Brunasso, David Lawrence, Anders Ekermo, Jacob Repp, and Rodney Tsing. Starcraft II: A new challenge for reinforcement learning. *CoRR*, abs/1708.04782, 2017.

[15] Marek Wydmuch, Michal Kempka, and Wojciech Jaskowski. Vizdoom competitions: Playing doom from pixels. *CoRR*, abs/1809.03470, 2018.

[16] Georgios N. Yannakakis and Julian Togelius. *Artificial Intelligence and Games*. Springer Publishing Company, Incorporated, 1st edition, 2018.

Appendices

A Challenge Environment Details

Through experimentation we extracted the details of the challenge environment. Time related values are always relative to the game speed, as it is possible to accelerate matches according to hardware capacity. The values listed in Table 1 are relative to the game played at the speed of a regular StarCraft II match.

Units have a shield and hit points. The shield is always damaged first until gone. Only afterwards will hit points be affected. A unit is defeated when hit points reach zero. Shield can regenerate after some time has passed between taking damage.

Some additional facts about the map:

- Turning does not affect movement speed of units.
- Units can start performing actions the AI tick after executing their previous action.
- The ranged attack can also damage friendly units.

Table 1: Overview of measured game constants

Map Measurements	
Distance between spawn and flag:	30 meters
Total map size:	80x80 meters
Playing field:	Approx. 70x23 (non rectangular)
Minimal distance between units:	1.25 meters
Ranged Attack Measurements	
Explosion distance:	8 meters
Total damage per unit:	60 damage (does not stack)
Tick damage:	20 damage/AI tick
Explosion radius:	2.13 meters
Explosion duration:	1.5 seconds (3 AI ticks)
Explosion cool down:	2.0 seconds (4 AI ticks)
Timing Measurements	
AI action rate	2/second
Flag measurements	
Flag points:	10 points
Flag spawn timer:	10.1 seconds
If units are in flag area when flag is spawned, a force field pushes them away	
Flag spawn force field duration:	0.1 seconds
Flag spawn pushing distance:	3.3 meters
Flag force field trigger radius:	3.5 meters
Unit Measurements	
Points gained on defeat:	5 points
Unit spawn timer:	12 seconds
Auto attack range:	4.1 meters
Auto attack damage:	14 damage
Auto attack speed:	1 per 2 seconds
Movement speed:	2.95 meters/second
Shield regeneration starts after:	10 seconds of no damage
Shield regeneration rate:	2 shield points per second
Hit points / Shield:	80/80