Curriculum and Transfer Learning for DQNs in Starcraft II

Alan Zhou

Abstract

Starcraft II is a popular real time strategy video game in which players collect resources, build buildings, and create armies of units. The game presents an interesting combination of challenges for reinforcement learning not commonly found in other reinforcement learning environments. While very capable StarCraft II agents have been created, the most successful of these require vast computational resources, which are only available to the wealthiest companies and labs. In this report we present a deep Q-learning model trained on small mini games in the StarCraft II environment, utilizing a number of transfer learning and curriculum learning techniques to converge to good solutions using only modest amounts of computing power. We conclude by offering a number of ways in which these methods might be used to improve on the model given slightly more time and resources.

1 Introduction

Starcraft II is a popular real time strategy video game in which players collect resources, build buildings, and create armies of units. In a 1v1 setting the game is won when one player has destroyed all of the other players' buildings. One might think of Starcraft II as a form of high-speed chess, where actions are performed in real time and players must utilize both strategy and speed of action to win.

As a reinforcement learning problem, Starcraft II presents a combination of challenges that are not typically found in other traditional game-based reinforcement learning problems:

- Continuous Environment: Starcraft II represents a nearly continuous environment in both time and space, where actions must be made continuously as the game clock progresses and can be made on any pixel on the screen.
- Massive state-action space: Starcraft II contains a massive action and state space. This state action space changes vastly as the player progresses through the game and constructs more units and buildings.
- Multi-agent interaction: The optimal strategy for a given agent is highly dependent on the actions of the opposing player. On a lower level, we

might consider Starcraft II to be a multi-agent problem in terms of the units it controls - the units within an army must each individually collaborate to achieve a common goal[1]

- Imperfect knowledge: An agent can only observe a fraction of the overall map.
- Delayed credit assignment: It is not clear how rewards should be assigned to actions.

While very capable Starcraft II agents have been created, specifically by Google's DeepMind, there still exists many flaws with those agents. In particular, training agents requires vast quantities of data and great computational power. By applying transfer and curriculum learning we hope to demonstrate a pathway to increased training efficiency of such agents.

2 Related Work

2.1 Games and Deep Reinforcement Learning

Deep reinforcement learning on raw video input was shown as early as 2013 with Atari games in The Arcade Learning Environment[1]. The techniques shown there involve using Deep Q-Learning with experience replay. While we hope to emulate something similar with Starcraft II, we note that those Atari games lack multi-agent interaction, imperfect knowledge, and massively delayed credit assignment.

In 2015, Hasselt, Guez, and Silver showed that a new technique known as double Q-learning could improve upon previous DQN techniques, by reducing a common issue in Q-learning where action values can become overestimated under certain conditions[6]. This is the algorithm we use to train our model.

2.2 Transfer Learning and Curriculum Learning

Transfer learning to pass on knowledge between agents trained on different but similar tasks, has existed for many years [5], even before the proliferation of deep learning and transfer learning in deep learning. More recently, more advanced curriculum learning techniques have arisen such as task dimension simplification, promising initializations, mistake-driven substasks, option-based subgoals, task-based subgoals, and composite subtasks [2]. These techniques allow for significant speedups in training speed. In some cases, curriculum learning has been applied to Starcraft, although in very limited subtasks of the game [3].

[1] we note here of course that an agent can choose to simultaneously select multiple units and issue commands, although such simultaneous instruction means that all units perform the same action.

2.3 State of the Art Starcraft Agents

Most of the pioneering work in training RL agents to play Starcraft II has been done by Google's DeepMind lab. The Starcraft II Learning Environment (SC2LE) as well as PySC2, a python environment that wraps the Starcraft II API, was developed by DeepMind and is currently the best way to train and test RL agents with the game [9].

Using the SC2LE, Deepmind created AlphaStar, an agent capable of playing Starcraft II at a high enough level to beat professional Starcraft II players. Using a complex deep learning architecture, AlphaStar was first trained by supervised learning / transfer learning from human replays, and then trained in a multi-agent reinforcement learning process involving population-based and multi-agent based methods [7]. While these results were impressive, it is worth noting that there were a number of unfair advantages given to the agent, in particular how it was able to interact with its environment in a way that human agents never could.

Very recently DeepMind announced that their newest version of Alphastar was able to achieve 'Grandmaster' level in Starcraft II. This means it achieved an MMR ranking higher than 99.8% of human players[2]. How different this new iteration of Alphastar is from its previous form is not entirely clear, as DeepMind did not publish a full paper with its initial unveiling of Alphastar. However, at its core, Alphastar is represented by a deep neural network that represents a policy $\pi_\theta(a_t|s_t, z) = \mathbb{P}[a_t|s_t, z]$ where s_t represents all observations from the start of the game to time t and z is a strategy sampled from a pool of prelearned strategies. In terms of RL techniques, Alphastar utilizes off-policy learning, temporal difference learning, clipped importance sampling, and a self-imitation algorithm (UPGO). It also utilizes league training [8][3].

3 Technical Background

3.1 Deep Q-Learning

Our agents are trained with deep Q-networks (DQN). Q-learning is given by Sutton and Barlo [4] as

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$$

Because our environment is so complex and our state-action space is so large, we need to do value-function approximation, for which deep learning models are perfectly suited. This give us an objective function such as mean square error

[2]MMR or *match making rating*, is a score assigned to each player based on their win loss record, taking into account the strength of their opponents.

[3]It is worth noting that even this advanced agent has a number of issues that prevent it from achieving the level of domination over its human counterparts that AlphaGo or AlphaZero have done in their respective domains.

$\overline{VE}$:

$$\overline{VE}(w) = \sum_{s \in S} \mu(s)[v_\pi(s) - \hat{v}(s, w)]^2$$

To update our weight vector w, we apply stochastic gradient descent (SGD) methods:

$$\begin{aligned} w_{t+1} &= w_t - \frac{1}{2}\alpha\nabla[v_\pi(S_t) - \hat{v}(S_t, w_t)]^2 \\ &= w_t + \alpha[v_\pi(S_t) - \hat{v}(S_t, w_t)]\nabla\hat{v}(S_t, w_t) \end{aligned}$$

In the case of deep learning, each $w_i \in w$ simply represents a weight between two given nodes in our neural network. The update to our Q values at each training step for our DQN is

$$Q(s, a) = r(s, a) + \gamma \max_a Q(s', a)$$

3.2 Experience Replay

Experience replay is a technique developed as part of the original Atari DQN algorithm[1]. While interacting with the environment, our agent stores transition tuples of (s, a, r, s'), in a replay memory D with capacity N. During training the agent randomly samples a minibatch of transitions to train the network on. By randomizing the sampling process, we avoid the problem of correlation between consecutive samples causing inefficient training.

3.3 Double DQns

Q-learning sometimes leads to overestimation of action values, so in order to correct for this and lead to faster performance, we utilize double DQNs[6]. For this method, our model contains two DQN networks. After an action is selected, one network (the *online* network) is updated based on the Q value of the *target* network. The idea here is to allow the online network to converge to a stationary target rather than attempt to update a network towards a target that is also changing as a result of that update. After a fixed number of time steps the target network overwrites its own weights with the online network's. The update rule for this algorithm be expressed as follows:

$$Q(s, a) = r(s, a) + \gamma Q'(s', \text{argmax}_a Q'(s', a))$$

where Q is the online network and Q' is the target network. The error is MSE as in the standard DQN algorithm[4].

3.4 Transfer Learning

Because of our choice to use deep learning for our models, transfer learning is a very natural follow up step to the training of a network. To do transfer learning we simply assign the weights of a network trained on one task to another network with the same architecture.

[4]For a full rundown of the DQN algorithm, check out [1]

4 Experiment Design

The result of our work on this project is sc2bot, a python package built to train deep reinforcement learning agents on custom maps in StarCraft II[5]. For computation we used 3 machines: one personal desktop with an Nvidia Geforce RTX 2070 graphics card and two google cloud virtual machine instances with Tesla P100 graphics cards attached.

4.1 Environment

For our environment we are using StarCraft II in conjunction with DeepMind's PySC2 learning environment [7]. PySC2 exposes Blizzard's StarCraft II Machine Learning API as a Python RL environment[10]. It provides an interface that allows RL agents to interact with the game, getting observations and sending actions. Figure 1 shows the visualization of PySC2.

One of the key features of PySC2 is that it exposes a number of 'feature layers' to the agent, which are datastreams not normally available to a person playing the game. These are predefined features like hit points or unit type intended to make reinforcement learning easier - normally such representations would need to be inferred from the raw pixel output of the game.

For our experiment, we primarily work with three feature layers: the 'player relative' layer, which shows the points on the map occupied by a specific owner; the 'unit hit 'points layer, which give the hitpoints of the unit occupying a given space on the map; and the 'unit type' layer, which gives the unit type at any given space on the map. All of these layers are discretized to the same size that the game screen is discretized. Any spot that a given feature layer is not relevant is a assigned a value of 0 for that spot in the feature layer.

4.2 Mini Games

For the experiment we are using two mini games: **Move To Beacon** and **Defeat Roaches**. For each mini game we discretize the screen to a 32 x 32 grid.

4.2.1 Move To Beacon

In the Move To Beacon task (figure 2), a flat empty map the size of the screen is initialized, with one marine placed at a random location on the map and a green beacon placed at another location on the map. Whenever the marine manages to touch the beacon, the agent is rewarded 1 point and the beacon spawns randomly at another point on the map. The agent is restricted to only making move actions with the marine. Each episode lasts for 90 seconds.

[5]The package and all developed sourcecode for this project is available at `https://github.com/alanxzhou/sc2bot`.

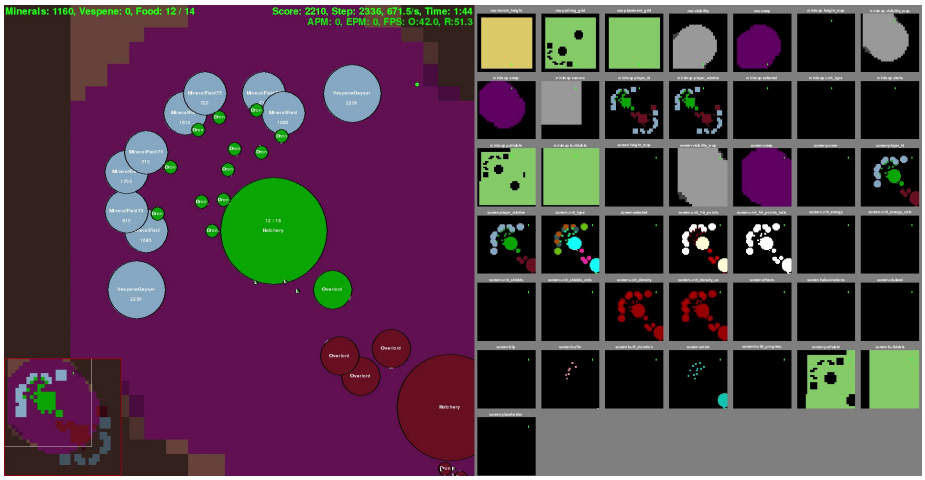

Figure 1: The features that pysc2 sees. ON the left half, each distinct shape is a different unit. On the right, each square represents a feature layer that a machine learning algorithm might use.

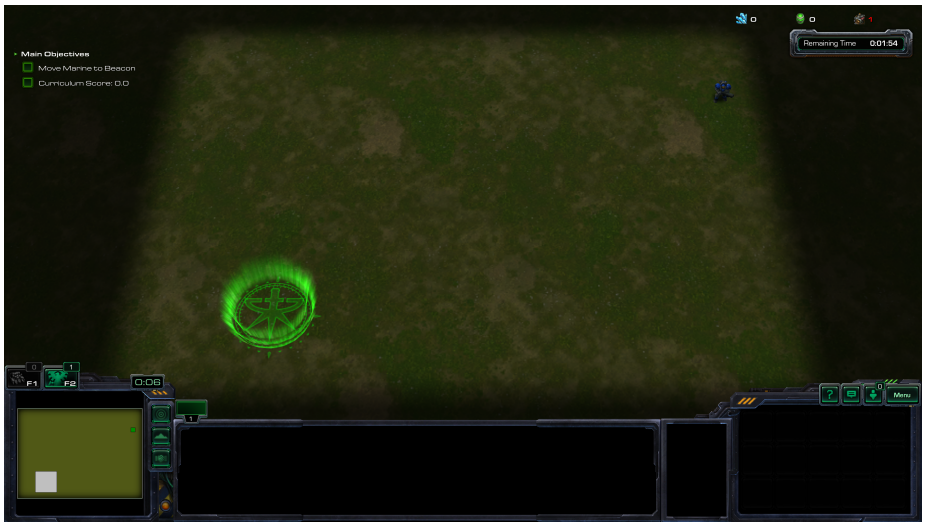

Figure 2: The starting screen for the Move To Beacon mini game. In the top right is the marine that the agent controls. In the bottom left of the screen we see the green beacon.

4.2.2 Defeat Roaches

In the Defeat Roaches task, a flat empty map the size of the screen is initialized, with 8 marines placed randomly on the left or right side of the map, arranged in a vertical line. 4 roaches are placed randomly on the opposite side of the map, also arranged vertically. If all roaches are defeated then 5 new marines appear adjacent to the current marines and 4 new roaches spawn on the opposite side of the map. The episode ends when either of the following happens:

- All marines are defeated.
- 90 seconds have elapsed.

For this mini game we restrict the agent's actions to only attack-moving[6]. For rewards, the reward breakdown is as follows:

- 500 points for defeating a roach
- -10 points per time step
- -1000 points for the agent killing one of its own marines

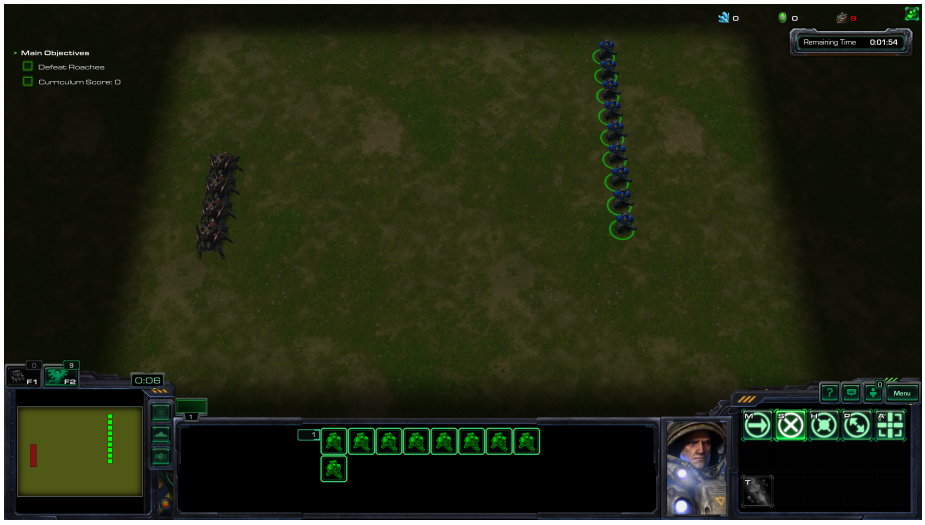

Figure 3: The Defeat Roaches mini game. On the right is the army of marines controlled by the agent. On the left are the four roaches.

[6]The attack-move action is the following: if the point chosen to attack-move to is empty, then all selected units move towards that point and attack any enemy units that enter their line of sight along the way. If the point chosen to attack-move to is contains a unit, then all selected units will attack that unit.

4.3 Network Architecture

Our network architecture is a fairly basic convolutional neural network (CNN) without any fully connected layers at the end. The reason for this is that even though fully connected (FC) layers allow us to create much more complex representations of the state space, they increase the complexity of the network by orders of magnitude. In particular, for a 32 x 32 grid, with an FC layer of connections we would need $32^2 \times 32^2 \approx 10^6$ individual network weights, which we found to be too many to update within our state goals of using modest computing power and time. Our CNN architecture is shown in figure 4.

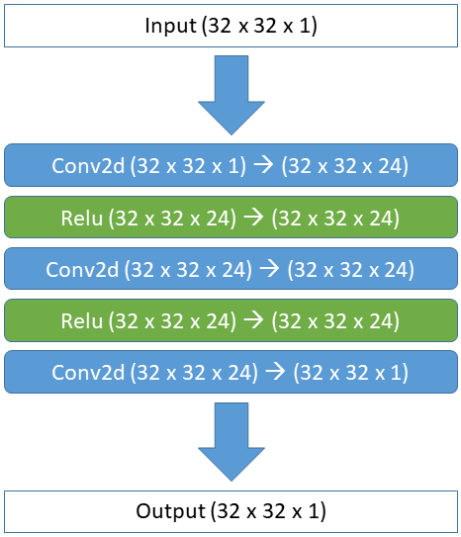

Figure 4: The CNN architecture for our agent. For each layer the inputs are to the left of the arrow and the outputs are to the right. The dimensions are (height x width x number of channels). For each layer we use a 3 x 3 convolution.

4.4 Transfer Learning

After training the beacon agent to convergence, we then transfer over the weights from the beacon agent to the roach agent in order to initialize a network that we believe should perform well in the new environment.

4.5 Training Procedure

For each of our mini games, we actually know a priori what behavior we should be looking for. As a result, we can use scripted agents to train our DQNs by demonstration. The way we chose to do this is simply to do so through preloading experience replay. That is, for each mini game, we let a scripted agent fill a list of (state, action, reward, next state) tuples and then trained

a DQN on this list of transition tuples before our agent interacted with the environment at all.

For the Move To Beacon mini game we pretrained an agent with the above experience replay technique and then let the agent train on 10000 episodes and compared it to another agent that trained on 10000 episodes in the StarCraft II environment. For this agent we only exposed the agents to the 'player relative' features.

For the Defeat Roaches mini game, pretraining seemed to have no effect on the agent's proficiency. However, we transferred over the DQN weights from one of the beacon agents to one of the roach agents and then let it train on 10000 episodes of the mini game. We then compared that to an agent that didn't benefit from transfer learning but also trained on 10000 episodes of the roach mini game.

For each DQN network we tried exposing the networks to different combinations of feature layers but ultimately our most effective DQNs were the ones that used just the 'player relative' feature layer.

5 Experimental Results

We manage to achieve moderate success on both the Move To Beacon task and the Defeat roaches task[7]. We were able to achieve the greatest success when we used only the 'player relative' features from PySC2. For the sake of brevity, we won't discuss numerical results of the other agents we trained in this report any further[8].

5.1 Move to a Beacon

For the Move to Beacon task we found that the pretraining routine[9] greatly decreased the number of episodes required to converge to an optimal policy, achieving that feat at around 200 episodes, whereas the non-pretrained agent took about 300 episodes to reach an optimal policy. For this task, since the task was so simple, both agents managed to find the same optimal policy. One strange result is that around the 900 episode mark, the agent that received pretraining experienced a temporary sudden drop in the reward it received but

[7] All technical demonstration videos are available for viewing on Google Drive at `https://drive.google.com/open?id=1sUV3-lQr2v7ntuNb67e33-b3hr67DR9x`

[8] To see some demonstrations of how these agents performed, feel free to view the following videos on Google Drive:

- Agent with fully connected layers (roaches task): battle_agent_full.mp4
- Agent with fully connected layers that can only do actions that within a certain radius of the selected army (roaches task): local_agent.mp4
- Agent with a CNN with slightly different convolutional layer parameters (beacon task): weird_beacon.mp4

[9] To see the scripted agents involved in the pretraining routine see beacon_oracle.mp4 for the beacon scripted agent and roaches_oracle.mp4 for the roach scripted agent.

then recovered - the cause of this is as of yet unknown. The reward curves for both agents are shown in figure 5[10].

Figure 5: Number of episodes vs loss for the Beacon Agents. The blue line represents the pretrained agent and the orange line represents the non-pretrained agent. We can see that the pretrained agent converges more quickly to an optimal policy. We also note a strange drop in reward around the 900 episode mark. We are not sure what caused this.

5.2 Defeat Roaches

For the Defeat Roaches task we found that the pretraining routine did not affect the overall performance of the network. However, we did find that transfer learning from the beacon task greatly improved performance and led to the agent converging to a different policy than the agent that didn't get transfer learning. The reward curves for both agents are shown in figure 6[11]

[10] You can view these agents on the Google Drive as beacon_pretrain.mp4 and beacon_no_pretrain.mp4. The videos are not very different since both agents seem to follow the same optimal policy.

[11] You can view these agents on the Google Drive as transferred_battleagentbeacon.mp4 for the agent with transfer learning and nottransferred_battleagentbeacon.mp4 for the agent without transfer learning.

Figure 6: Number of episodes vs loss for the Roach Agents. The blue line represents the agent that used transfer learning and the orange line represents the agent that did not use transfer learning. We can see that the agent with transfer learning converges almost immediately to an effective policy, while the agent without transfer learning converges to a suboptimal policy.

6 Conclusion and Future Work

We were successfully able to use transfer learning to improve performance for our agents using DQNs in the StarCraft II environment. By using rudimentary CNN models with limited numbers of features we were able to achieve good performance with limited computational resources and time. This offers a means by which individuals without the resources of a large scale company or lab might be able to create their own RL agents in the StarCraft II environment.

Given more time and resources there are a number of immediate next steps that could be conducted. An immediate improvement would be to slightly alter the input to our CNNS to take in 3 different feature arrays: the 'player relative' array (the feature we were using for our experiments), the 'unit hit points' array, and the 'unit type' array. The reason for this is that even though our agent converged to a relatively good policy for the Roach task, it didn't reach an optimal policy. This was to be expected since an optimal policy would likely involve telling marines to attack the roach with the lowest hit points, which our agent that was only exposed to the player relative array would know nothing about [12].

[12]The reason attacking roaches with lower hit points is more effective is that if a roach is defeated it can no longer do any damage, so decreasing the amount of damage done to the marines while maintaining their damage output should be the most effective strategy.

Another relatively low hanging fruit would be to use slightly greater computation power and time. This would allow us to train a model using fully connected layers, since 10000 episodes was insufficient to converge to any good policy given that many parameters. The increased computation power / time would also allow us to train on raw image data, which would certainly take much longer to train.

In terms of longer term avenues for future work, by training on various mini-games (by default there are 7), we could eventually develop multiple agents proficient in different mini games and combine them to do hierarchical machine learning. We might also try multi-agent training whereby we train agents against each other so that we might learn more dynamic and effective policies.

References

[1] 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.

[2] Sanmit Narvekar, Jivko Sinapov, Matteo Leonetti, and Peter Stone. Source task creation for curriculum learning. In *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*, AAMAS '16, pages 566–574, Richland, SC, 2016. International Foundation for Autonomous Agents and Multiagent Systems.

[3] Kun Shao, Yuanheng Zhu, and Dongbin Zhao. Starcraft micromanagement with reinforcement learning and curriculum transfer learning. *CoRR*, abs/1804.00810, 2018.

[4] Richard S Sutton and Andrew G Barto. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

[5] Matthew E. Taylor, Peter Stone, and Yaxin Liu. Transfer learning via inter-task mappings for temporal difference learning. *Journal of Machine Learning Research*, 8(1):2125–2167, 2007.

[6] Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. 2015. cite arxiv:1509.06461Comment: AAAI 2016.

[7] Oriol Vinyals, Igor Babuschkin, Junyoung Chung, Michael Mathieu, Max Jaderberg, Wojtek 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, 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.

[8] Oriol Vinyals, Igor Babuschkin, Wojciech M. Czarnecki, Michaël Mathieu, Andrew Dudzik, Junyoung Chung, David H. Choi, Richard Powell, Timo Ewalds, Petko Georgiev, Junhyuk Oh, Dan Horgan, Manuel Kroiss, Ivo Danihelka, Aja Huang, T andrevor Sifre, Land Cai, John P. Agapiou, Max Jaderberg, Alexander S. Vezhnevets, Rémi Leblond, Tobias Pohlen, Valentin Dalibard, David Budden, Yury Sulsky, James Molloy, Tom L. Paine, Caglar Gulcehre, Ziyu Wang, Tobias Pfaff, Yuhuai Wu, Roman Ring, Dani Yogatama, Dario Wünsch, Katrina McKinney, Oliver Smith, Sch Tom, Timothy Lillicrap, Koray Kavukcuoglu, Demis Hassabis, Chris Apps, and David Silver. Grandmaster level in starcraft ii using multi-agent reinforcement learning. *Nature*, 2019.

[9] 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.

[10] 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.