Deep Reinforcement Learning and StarCraft II

Daniel Geschwender, Yanfeng Liu, Jeevan Rajagopal

July 31, 2019

1 Introduction

Deep reinforcement learning has gained tremendous popularity in recent years due to the significant advancement in both the capability of computing hardware and amount of available training data. It trains an agent to achieve a specific set of goals with only the overall performance score as the reward feedback. A deep neural network is used as a generic function approximator to model the environment, policy, and other non-linear relationships. This relieves human beings from the burden of coming up with a heuristic model that often oversimplifies the real situation. In this project we train a deep neural network to play StarCraft II minigames using a reinforcement learning technique called Asynchronous Advantage Actor-Critic (A3C). The results show the excellent ability of a neural network to adapt to arbitrary tasks with different levels of difficulty. The effectiveness of reinforcement learning is also explored and analyzed.

2 StarCraft II Learning Environment

StarCraft II Learning Environment (SC2LE) [3] is jointly released by DeepMind and Blizzard Entertainment for the purpose of encouraging next generation artificial intelligence research. It is written in Python and provides a well-rounded API designed to mimic the environment a human player faces. The API provides a set of feature maps as part of the input, such as height, unit health, mini map, etc. Note that this is slightly different from what a human player sees: realistic RGB graphics will render far away objects smaller than those closer to the camera. The API provides a orthographic view instead to eliminate this extra bit of confusion. A sample of the input is shown in Figure 1.

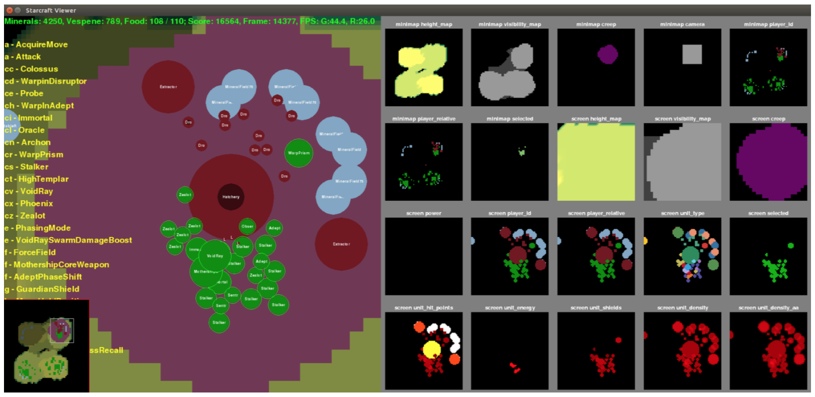

Minerals: 4250, Vespene: 789, Food: 108 / 110; Score: 16564, Frame: 14377, FPS: G:44.4, R:26.0

Figure 1: SC2LE provides a set of feature maps as input to the agent

The full edition game is currently assessed as a grand challenge for AI for the following reasons.

First, the game is multi-agent. Two or more players can play the game in continuous time steps. It is also multi-agent within each player. A player has tens or even hundreds of units under command at any given time during a game, so arranging their positions and actions becomes a challenge on its own.

Second, the fog of war introduces imperfect information. An agent must learn to send units out periodically in order to obtain the latest map information. The agent also has to be able to make decisions when faced with uncertainty. For example, one gamer might not know what strategy the other gamer is using, but it has to produce units ahead of time to prepare for possible conflicts. This requires both experience from previous games and improvisation based on the partial information it gathered during the game.

Third, the action space is huge. A location-based action counts different combinations of coordinates as different actions, so the location-based action alone takes up a huge part of the action space already. For example, on a 64 by 64 screen, possible upper left and bottom right coordinate combinations can be as many as $(64 \times 64)^2 = 2^{24} \approx 1.68 \times 10^8$ already. If a more realistic resolution is used (such as 1920 by 1080) then the action space becomes astronomical. A common way to deal with it is discretization at the cost of inaccurate selection.

Fourth, the reward is sparse and severely delayed. The only real feedback is win/loss at the very end. However, many things can happen between a disadvantaged early game and the final result. A human player is able to make long term plans and deal with short term loss on things such as technology progress or unit death. It is unknown if a computer can come up with coherent long term strategies.

We decided that the full game will be too challenging to train on, so we focus on mini games instead. Specifically, we train our network primarily on MoveToBeacon but also explore its possibilities on other more challenging mini games later.

The player controls one marine in MoveToBeacon, and the goal is to reach as many beacon locations as possible within the 2 minute time constraint. Every time a beacon is reached, 1 point is rewarded to the player. A sample frame is shown in Figure 2. This is significantly simpler than the full game because camera movement is unnecessary, fog of war is disabled, and a lot of unrelated actions are removed from the action space by the API, such as constructing a building. The end goal is clearly defined, and there is no conflict between multiple goals. A greedy approach should be able to achieve optimal results.

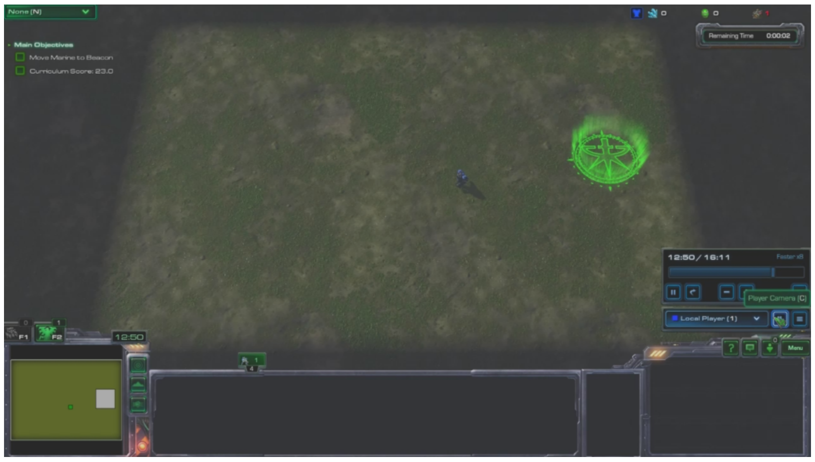


Figure 2: Sample scene from MoveToBeacon mini game

3 A3C and Network Implementation

Our training method of choice for experimentation was the Asynchronous Advantage Actor-Critic (A3C). Given the size of the action space in StarCraft II and the need to reduce the training time as much as possible, A3C seemed to be a good fit for experimentation. Furthermore, the method was used not only in this domain, but in others with success.

3.1 Description of A3C

The Asynchronous Advantage Actor-Critic method can be broken down into its major components. The 'Asynchronous' portion refers to the use of multiple actor-learners which update a global set of parameters as well as their own local set of parameters independent of other actor-learners. Advantage is a term used to describe the relationship between the "real" reward, or the reward obtained at the end of an episode minus the *expected reward*, which is derived from the baseline, or value function. Finally, the advantage term affects the probability of the action being selected in the future [2]. Simply put, the better an actor-learner performs in comparison to the expected performance, the higher the advantage term is. Conversely, if the actor-learner performs worse than expected, then the advantage term is lower. Finally, the Actor-Critic portion refers to the actor-critic architecture, which refers to the actor (using a policy π) and a critic (using the baseline value function mentioned earlier, $b_t \approx V^\pi(t)$) [1]. Using this architecture helps reduce variance in the policy estimate without introducing bias, leading to faster convergence.

As described by Mnih et al. [1], the general structure of a given actor-learner in A3C is as follows.

1. Reset both the value and policy gradients to 0
2. Get initial state s_t
3. Perform action a_t given policy $\pi(a_t|s_t;\theta')$ until either a terminal state is reached or the max number of steps t_{max} is reached
4. Set reward R to zero if at a terminal state, otherwise set it equal to $V(s_t, \theta'_v)$
5. perform n-step return to accumulate both the value and policy gradients, starting from the last state.
6. Update the global policy and value functions θ and θ_v with the local gradients $d\theta$ and $d\theta_v$ respectively.

This process is repeated until some global step limit is reached. Since this process relies on multiple agents acting independently of one another, it is assumed that each agent will have different experiences, thus reducing, if not eliminating, the need for experience replay and allowing on-policy methods (e.g., actor-critic) to be used [1].

3.2 Network Implementation

While Mnih et al. [1] used a convolutional neural network that split into one softmax output for the policy π and one linear output for the value $V(s_t;\theta_v)$, we decided to implement the architecture described by Vinyals et al. [3] due to their network having been used on the StarCraft II problem space. Specifically, we implemented a variant of their "FullyConv" architecture, which used a convolutional layer with 16 filters and a kernel of 5x5, followed by another convolutional layer with 32 filters and a kernel of size 3x3 on both the screen and minimap. To preserve resolution, no stride was used and inputs were padded. These were then concatenated with a reshaped version of the non-spatial features (e.g., number of actions available, in-game statistics, etc.) to form the state representation. From there, the spatial actions were determined using a 1x1 convolutional layer with a single output, and the non-spatial actions were determined using a fully connected layer with 256 nodes and ReLu activation and more fully connected linear layers afterwards. Vinyals et al. did propose another network that incorporated an LSTM to address more complex mini-games, but we decided not to use it as it would not have been useful for the *Move to Beacon* minigame. Furthermore, the loss functions their network uses is split between three components: the value loss, policy loss, and entropy loss. The loss functions is described in Vinyals et al. [3] as:

$$(R_t - v_\theta(s_t))\nabla_\theta \log \pi_\theta(a_t|s_t) + \beta(R_t - v_\theta(s_t))\nabla_\theta v_\theta(s_t) + \eta \sum_a \pi_\theta(a|s_t) log \pi_\theta(a|s_t)$$

The first part refers to the policy loss, where R_t refers to the result from the n-step return in the A3C algorithm, and $(R_t - v_\theta(s_t))$ refers to the advantage term. The next term refers to the value loss, which is controlled by the hyperparameter β. Finally, the entropy regularization, which is controlled by the hyperparameter η, promotes exploration in the actor-learner.

While we follow the general structure of the aforementioned network and loss functions, we had to trim down the input space due to the amount of time it took to train the architecture. Specifically, our network uses the 'screen_player_relative', 'single_select', and 'game_loop' observations. These show the marine and beacon locations, which unit, if any, is selected, and the number of elapsed episode steps, respectively. By removing unnecessary components for the *Move to Beacon* minigame, we reduced the number of trainable network parameters from approximately 4 million to around 70 thousand, which decreased our training time drastically.

4 Experimental Results

4.1 Hyperparameter Experiments

As a first experiment, we perform a hyperparameter search, testing three values for each of three hyperparameters:

- *learning_rate*: {0.01, 0.1, 1.0}
- β (value loss coefficient): {0.1, 1.0, 10.0}
- η (entropy loss coefficient): {0.0, 0.01, 0.1}

We test all 27 combinations of the hyperparameter values. Each run is allocated 4 CPUs and 12 GB of memory. Each run uses a single agent (i.e., no parallelism) that updates the network parameters every 40 game steps. Each run trains for the shorter of 3 hours or 200,000 game steps. Figure 3 shows the agents' episode scores over the course of training. The horizontal axis indicates the number of game steps the agent has trained on. The vertical axis indicates the episode score achieved by the agent. The dark line corresponds to the best performing run, with hyperparameters set as follows: *learning_rate* = 0.01, $\beta = 0.1$, $\eta = 0.0$. Surprisingly, the best performance comes when removing the entropy regularization. This is likely due to simplicity of the MoveToBeacon task.; it does not require a great deal of exploration of the action space, rather a refinement of a basic policy that is largely invariant across states.

As a follow-up experiment, we perform 30 runs using the best hyperparameter settings from the first experiment. The runs only differ by the random initialization of the network weights. Figure 4 show the results of the 30 agents' scores over the course of training. We see that performance varies dramatically between the runs, indicating that the weight initialization has a significant impact on performance. Thus, without further runs for each hyperparameter configuration, we cannot conclude that the selected hyperparameter setting is significantly better than the others tested. However, the results indicate that the selected hyperparameter setting is sufficient to enable our network to learn a good policy.

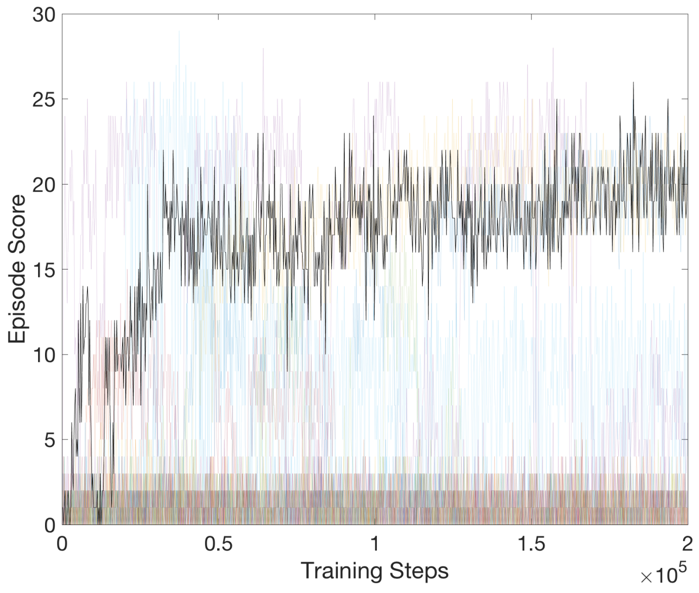


Figure 3: Agents' episode scores over the course of training. Each line shows a different hyperparameter configuration. The dark line corresponds to the run with best mean episode score

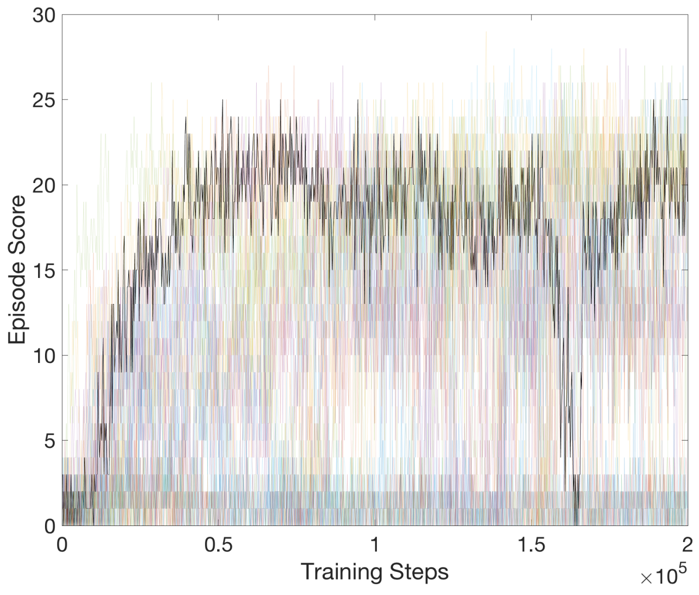


Figure 4: Agents' episode scores over the course of training. Each line shows a different random variable initialization using the same hyperparameter configuration. The dark line corresponds to the run with best mean episode score

4.2 Learned Policy and Value Functions

Below, we discuss the policy and value function learned by our best performing agent. Table 1 shows the action policy in the initial episode compared to the learned action policy in the final episode. We distinguish between the policy used when the marine character is not selected and the policy used when the marine is selected. When the marine is not selected, many of the actions are invalid, and thus masked out by the environment. Further, the agent is aware that the marine is not selected and may learn to behave differently in this situation. In both states, the initial policy is nearly uniformly random. By the final episode, the policy becomes much more focused around particular actions. With no marine selected, the agent chooses 'select_army' with 98% probability. This action selects all units, which, in this mini-game, selects the single marine unit. With the marine selected, the agent chooses either 'select_army' or 'attack_screen', each with 38% probability, or any of the remaining actions, each with 2% probability. The 'attack_screen' action is a form of move action that allows the selected unit to move to the specified screen coordinates. This policy is effective, though suboptimal. Through further learning, we expect the agent would learn to avoid wasteful use of 'select_army' when the marine is already selected.

Table 1: Initial versus final action policy. With no marine selected, invalid actions are masked out and marked with a '-'. This policy comes from the best performing run of Figure 4 .

	no_op	move_camera	select_point	select_rect	select_control_group	Stop_quick	select_army	Move_minimap	Move_screen	Attack_screen	Attack_minimap	Patrol_minimap	Patrol_screen	HoldPosition_quick	Smart_screen	Smart_minimap
With no marine selected																
initial	17%	17%	17%	17%	17%	-	17%	-	-	-	-	-	-	-	-	-
final	0%	0%	0%	0%	0%	-	98%	-	-	-	-	-	-	-	-	-
With marine selected																
initial	6%	6%	6%	6%	6%	6%	6%	6%	6%	6%	6%	6%	6%	6%	6%	7%
final	2%	2%	2%	2%	2%	2%	38%	2%	2%	38%	2%	2%	2%	2%	2%	2%

Figure 5 shows the evolution of the value function over the course of training. The horizontal axis indicates the number of game steps the agent has trained on. The vertical axis indicates the step within the given episode, ranging from step 0 at the top to step 240 at the bottom. The darkness of the shading indicates the value function prediction, with the darkest green corresponding to a value of about 20 and white corresponding to 0. Initial predictions are nearly homogeneous 0 values across the episode. As the agent begins to earn rewards, the value function increases its prediction, though the predictions still remain mostly homogeneous throughout an episode. Gradually, the value function correctly learns that early states have a higher reward potential than later states because later in the episode there is less remaining time to score. The dip in the value function prediction near the end corresponds to the dip in agent score seen in Figure 4.

4.3 A3C Parameter Experiments

We also experiment with several parameters of the A3C learning procedure:

- *#_asynchronous_agents*: {1, 2, 4, 8, 16}
- t_{max}: {40, 240}
- *advantage*: {true, false}

The *#_asynchronous_agents* parameter specifies the number of parallel agents we launch. The t_{max} parameter specifies the number of game steps between running updates. An update is always run

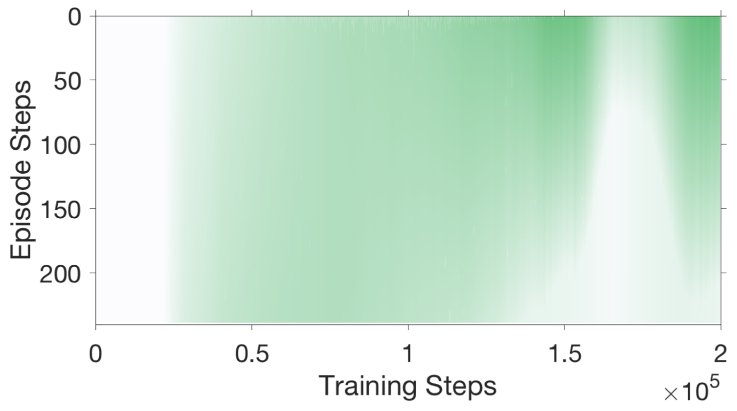


Figure 5: Value function predictions throughout an episode over the course of training. Darker shades indicate a larger value prediction (about 20) while lighter shades indicate a smaller value prediction (0). These predictions come from the best performing run of Figure 4

at the end of an episode. Setting this to 240 (the length of an episode) will only update the network at the end of an episode. The *advantage* parameter specifies whether the agent scales the policy gradient by the advantage (calculated using the critic's value function prediction) or by the raw n-step rewards.

We test all 20 combinations of the A3C parameter values. Each run is allocated 8 CPUs and 24 GB of memory. Each run trains for the shorter of 6 hours or 200,000 game steps per agent. Figure 6 shows the agents' episode scores over the course of training. Note that the number of training steps reported on the horizontal axis accounts for the total training steps pooled between all asynchronous agents. Thus, some lines extend beyond the others. Further, the scores plotted all come from the first asynchronous agent of each configuration. The dark line corresponds to the best performing run, with A3C parameters set as follows: $\#\_asynchronous\_agents = 8$, $t_{max} = 40$, $advantage =$ false.

We find that agents are able to achieve good scores (average of 15+) using any number of parallel agents and using advantage or raw rewards. However, the highest score achieved when using $t_{max} = 240$ is 12, with an average of 5.6. This may indicate that more frequent network updates are beneficial to learning, though additional runs would be required to verify this.

4.4 Results Comparison

Beyond the MoveToBeacon mini-game, we perform simple experiments on the remaining six mini-games provided with the SC2LE. We again use our simplified network, considering only a small subset of the available environment observations. We train the network using the same parameters used in the experiments shown in Figure 4. For each mini-game, we perform four runs with different random weight initializations. Figure 7 shows the results of the four agents' scores over the course of training on each of the six mini-games. In each figure, the dark line corresponds to the run with best mean episode score.

In Table 2, we compare our results to those presented in the SC2LE release paper [3]. We reproduce their results table and add an additional row with our results on each of the seven mini-games. On all seven mini-games, our network is able to outperform the random policy, but cannot achieve human level performance or beat the three networks proposed by DeepMind. This may largely be due to lack of training time. All three DeepMind networks where trained with 64 asynchronous agents for 600 million game steps using 100 hyperparameter configurations for each

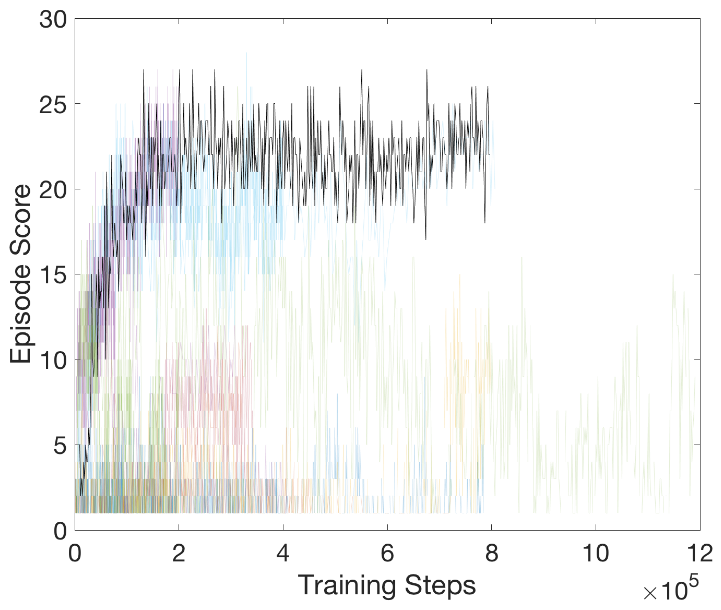


Figure 6: Agents' episode scores over the course of training. Each line shows a different configuration of the A3C parameters. The dark line corresponds to the run with best mean episode score

mini-game. Our network was trained using a single agent run for 200 thousand game steps using a handful of hyperparameter configurations. In many of our training results, we note a slight upward trend in the episode score, indicating further improvement could be made through continued training.

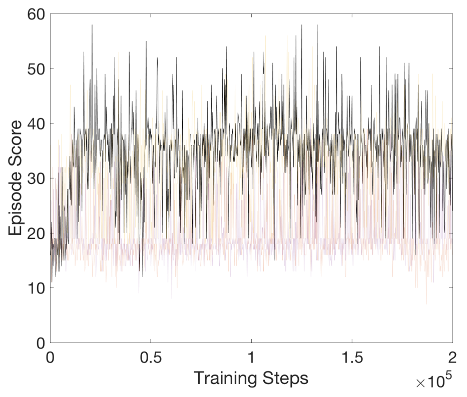
5 Conclusion

In this project, we successfully developed a neural network to interface with the StarCraft II Learning Environment through the use of the PySC2 library. We trained this network using the Asynchronous Advantage Actor-Critic reinforcement learning algorithm. Our network learned to play StarCraft II mini-games at a level between random and human performance. State-of-the-art networks on this learning task are trained for substantially longer and we expect our network could approach human-level performance on the simpler mini-games given additional training time.

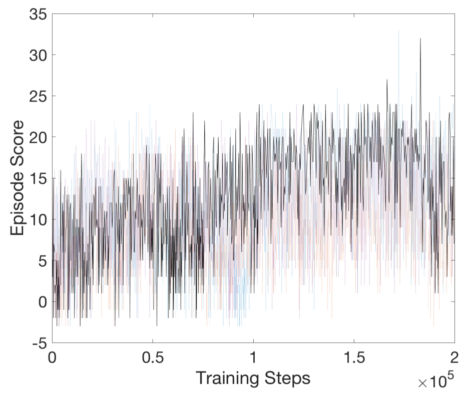
In order to perform successfully on the more complex mini-games as well as the full game, we would need to add back in the full set of environment observations as well as consider adding additional complexity to the network. Of the environment observations, we expect that 'hit_points' screen feature layer would be vital to performing well on the combat oriented mini-games. The network would likely benefit from the inclusion of LSTM layers to enable long-term planning. The network could also benefit from increasing the receptive field of the convolutional layers. Currently, the spatial outputs have only a 7×7 receptive field in the spatial inputs. Thus, the spatial policy cannot account for complex spatial states. Further, it may be useful to explore variations of the A3C algorithm, as suggested by Mnih et al. [1], such as integrating experience replay and considering alternative estimates of the advantage function.

Table 2: Comparison of our network's results (last row) against the results presented in [3]

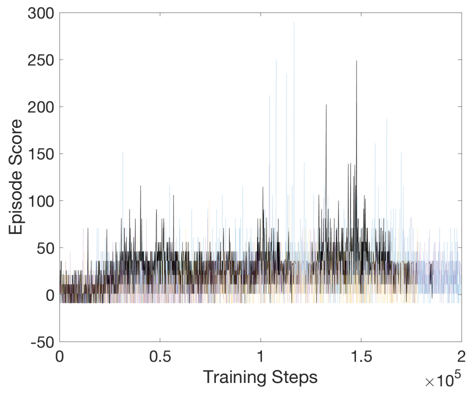
Agent	**Metric**	MoveToBeacon	CollectMineralShards	FindAndDefeatZerglings	DefeatRoaches	DefeatZerglingsAndBanelings	CollectMineralsAndGas	BuildMarines
Random Policy	mean	1	17	4	1	23	12	<1
	max	6	35	19	46	118	750	5
Random Search	mean	25	32	21	51	55	2318	8
	max	29	57	33	241	159	3940	46
DeepMind Human Player	mean	26	133	46	41	729	6880	138
	max	28	142	49	81	757	6952	142
StarCraft GrandMaster	mean	28	177	61	215	727	7566	133
	max	28	179	61	363	848	7566	133
Atari-net	best mean	25	96	49	101	81	3356	<1
	max	33	131	59	351	352	3505	20
FullyConv	best mean	26	103	45	100	62	3978	3
	max	45	134	56	355	251	4130	42
FullyConv LSTM	best mean	26	104	44	98	96	3351	6
	max	35	137	57	373	444	3995	62
Our Network	best mean	21	35	11	25	38	73	<1
	max	29	58	33	290	128	995	6



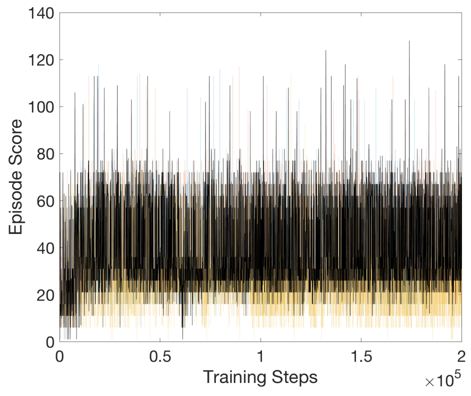
(a) CollectMineralShards



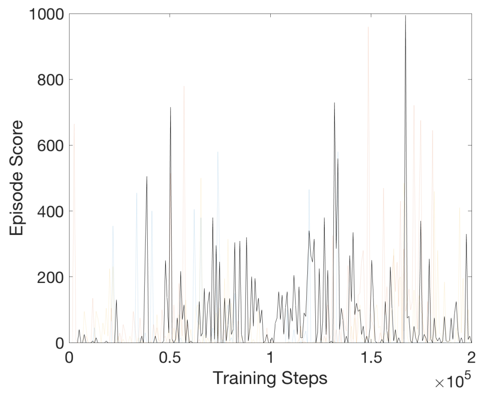
(b) FindAndDefeatZerglings



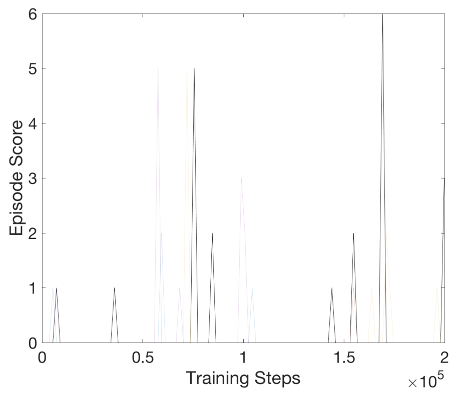
(c) DefeatRoaches



(d) DefeatZerglingsAndBanelings



(e) CollectMineralsAndGas



(f) BuildMarines

Figure 7: Agents' episode scores over the course of training for the six other mini-games

References

[1] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International Conference on Machine Learning*, pages 1928–1937, 2016.

[2] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.

[3] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, et al. Starcraft ii: a new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*, 2017.