Meta-Learning for StarCraft II Minigame Strategy

Jerome Kafrouni jk4100
Roop Pal rmp2191
Connor Hargus cjh2209

Abstract

We begin by recreating Google DeepMind's results for 5 minigames provided with the August 2017 release of the SC2LE library for StarCraft II. To this effect, we produce baseline results with random, scripted agents, and simple Q-learning. We also reproduce results using the A3C training algorithm using both the full action and observation space in addition to A3C models with action and observation spaces reduced by incorporating domain knowledge. Finally, we discuss our attempts to apply the algorithm of Meta-Learning Shared Hierarchies by Frans et al. to the problem of strategy generalizability across minigames.

1. Introduction

Deep reinforcement learning has made significant strides in recent years, with superhuman results achieved in board games such as Go. However, there are a number of obstacles preventing such methods from being applied to challenges in the real world. For instance, more realistic strategic situations often involve much larger spaces of possible states and actions, an environment state which is only partially observed, multiple actors to control, and a necessity for long-term strategies involving not hundreds but thousands or tens of thousands of steps [6]. Additionally, a diverse set of skills and adaptability to new situations are required for agents to perform well. It has thus been suggested that creating learning algorithms which outperform humans in playing real-time strategy (RTS) video games would signal a more generalizable result about the ability of a computer to make decisions in the real world.

Of the current RTS games on the market, StarCraft II is one of the most popular. The recent release by Google's DeepMind of SC2LE (StarCraft II Learning Environment) presents an interface with which to train deep reinforcement learners to compete in the game, both in smaller "minigames" and on full matches [8]. While current unsupervised reinforcement learning algorithms are no where near matching human performance in playing full matches, they are able to make some progress on the minigames. The 7 minigames included in SC2LE set up environments for achieving various essential components of SC2 gameplay such as moving units to certain locations, fighting small battles, and building new units. For our project, we worked on minigames involving moving to target locations as well as combat minigames.

Our work focuses on five of the seven minigames released with the SC2LE environment: MoveToBeacon, CollectMineralShards, FindAndDefeatZerglings, DefeatRoaches and DefeatZerglingsAndBanelings. The first two correspond to the simple task of moving units towards simple goals. In DefeatRoaches and DefeatZerglings, the agent has a full view of the map, and needs to defeat uniform enemies in the former and two different types of enemies in the latter, which requires a better strategy. Finally, in FindAndDefeatZerglings the agent also needs to explore the map by moving the camera and exploring unknown areas. The goal of our work is then to first be able to have different agents that perform well on each of these games, i.e. training agents that can solve simple tasks of the full StarCraft II game, and then try to have a more general single agent that can play these different games, as it would in the full game.

2. Related Work

With its release of SC2LE, Google's DeepMind also tested a parallelized policy gradient method called Asynchronous Advantage Actor-Critic (A3C) in combination with a set of modern deep learning architectures to establish baselines for performance on the included minigames. The neural network architectures they used included Atari-Net [1], a fully convolutional network (predicting actions without reducing resolution at each layer), and a fully convolutional network with long short-term memory. While the latter of these models performed slightly better across the set of minigames, they all fell dramatically short of StarCraft GrandMaster performance on the more complicated tasks. In "DefeatRoaches", the highest mean episodic reward achieved by any of Google's architectures was 101, while that of a GrandMaster was 215. "DefeatZerglingsAndBanelings" proved significantly harder, with the highest mean score of the RL algorithm at 96 while GrandMaster players averaged 727. The difference in performance capa-

bilities between these two tasks likely stems from the fact that the second, in which the player must defeat two different types of units, demands the use of different strategies and abilities depending on the enemy unit being faced.

In the October 2017 paper Deep Learning for Video Game Playing, Justesen et al. describe the unique challenges of RTS games and varying approaches utilized for controlling multiple agents [6]. Some algorithms, such as Intrinsic Curiosity Module (ICM) and Independent Q-Learning (IQL), treat all units as independent individuals. This assumption allows for dramatic simplifications in the necessary complexity of the model for taking actions, but it clearly lacks much of an ability to utilize the advantages which come from group agent strategy. They also describe how Multiagent Bidirectionally-Coordinated Networks (BiCNet) have shown promise for controlling large numbers of units, and that for small skirmishes of fewer than 10 units (similar to the minigames we intend to tackle) Counterfactual Multi-Agent (COMA) methods using policy gradient have shown state of the art results in StarCraft I by treating units as cooperative agents. In the SC2LE library, however, the emphasis is on centralized agents which are trained to behave using the set of actions which are available to a human players, rather than treating each unit as separate agents. The available actions include clicking and dragging to select units and clicking to send units, and the screen is viewed in a simplified version of how a human player would see it (with the minimap, etc). For StarCraft II, then, the gold standard for performance remains Google DeepMind's A3C implementations, for which only the results were released [8].

3. Problem Setting

We consider a mini-game G, where the player controls an initial army of one or more homogeneous "marines". Treating the player as a centralized agent controlling these homogeneous units, an ideal strategy uses these units in a cohesive group strategy to either move to certain locations on the map (MoveToBeacon and CollectMineralShards minigames), defeat an army of homogeneous agents ("roaches" in DefeatRoaches or "zerglings" in FindAndDefeatZerglings), or defeat an army of heterogeneous agents ("zerglings" and "banelings" in DefeatZerglingsAndBanelings). During the course of an episode, the number of units controlled by the agent may vary, some agents being killed and others being awarded after defeating enemy armies.

We denote by s the true current state of the game. At each time step, the agent chooses a single action $a \in A$. These simultaneous actions induce a transition to a new state s', determined by a Markov Decision Process (MDP) M defined by the probability distribution $P(s'|s,a) : S \times A \times S \to [0,1]$.

Consider that our agent makes an observation o of the state s, which in our SC2LE implementation setting relies on a simplified version of the player's view. In the "FindAndDefeatZerglings" minigame, this observation is only a partial observation. In the full version of StarCraft, a player's view also represents only a small area of the map and thus the observation is significantly occluded. The agent follows a policy π, and receives rewards through the reward function $r(s,a) : S \times A \to \mathbb{R}$, that depends on the action taken at a given time step, and we denote by γ the factor which discounts distant rewards.

The total discounted return for the agent is $R_t = \sum_{l=0}^{\infty} \gamma^l r_{t+l}$. In the ideal scenario, we would have access to a value function $V\pi(s_t) = \mathbb{E}_{s_{t+1:\infty}, a_{t:\infty}}[R_t|s_t]$ and/or an action-value function $Q(s_t, a_t) = \mathbb{E}_{s_{t+1:\infty}, a_{t+1:\infty}}[R_t|s_t, a_t]$. Modeling these functions based on observations lets us select a better policy to maximize expected discounted rewards.

In the Q-Learning approach, we create an approximator function $Q(s,a,\theta)$ which attempts to model the value of an action given a state $Q(s,a)$ by optimizing parameters θ at each step of the game minimizing the loss function $r_t + \gamma \max_{a'} Q(s_{t+1}, a', \theta_t) - Q(s_t, a_t, \theta_t)$ by adding the gradient of this loss to the parameters of our model θ. Modeling the Q function allows us to choose the action at each step which the model believes will maximize the expected overall reward (though we may introduce exploration through an ϵ-greedy choice of action). In a model which will be described in the Preliminary Results section below, we create a simple look-up table $Q(s,a,\theta)$ which progressively better approximates the value of a state-action, but which dramatically simplifies the state and action spaces in order to make doing so tractable.

Another form of reinforcement learning, of which the Asynchronous Advantage Actor-Critic (A3C) model is a popular variant, is the policy gradient approach. In this method, we create a model which simply maps an input state to an action with the policy function $\pi(a|s,\theta)$. Using a baseline function $b_t(s_t)$ to reduce variance, our optimization step in this case uses the gradient $\nabla_\theta \log \pi(a_t|s_t,\theta)(R_t - b_t(s_t))$. In the case of A3C, we use a baseline which is a centralized "critic" or value function $V(s_t, \theta_v)$ with some parameters θ_v. We can share parameters across a neural network to compute both $V(s_t,\theta_v)$ as well as $\pi(a|s,\theta)$, using a single fully connected layer to an output for the former and a separate softmax layer also coming off the final hidden layer to yield $\pi(a|s,\theta)$. In A3C, the term "asynchronous" refers to the fact that we can run a number of agents in parallel, each of which makes updates to the parameters of a single centralized policy and value function using the gradient $\nabla_\theta \log \pi(a_t|s_t,\theta)(R_t - V(s_t))$ where R_t is approximated using the value function along with observed rewards. Furthermore, by giving each of the agents running in parallel different randomization parame-

ters for selecting actions via the policy $\pi(a_t|s_t, \theta)$, A3C has been shown to provide for more robust exploration of the action space [7].

4. Replicating State of the Art

4.1. Baselines

We have successfully trained several models to replicate DeepMind's results andcompare against our own work. In addition to the randomized agent which is included in the PySC2 package (which simply selects an action at random at each time step), we have implemented simple strategies in the form of scripted agents for each minigame (in the `scripted_agents` folder of our GitHub repository) [9]. These, along with the random agent, will help to serve as a baseline point of comparison for how much of an increase in performance we are able to achieve using more strategic forms of reinforcement learning.

As a reference point for what basic reinforcement learning algorithms are able to achieve, we have also implemented a Q-learning algorithm using a table mapping state-action pairs to Q-values (see `scripted_agents/q_table_agent.py`. We built this model based on an existing model used for a different PySC2 minigame called Simple64 [2]. Besides acting as another baseline for what simple RL agents can achieve on our minigames, applying this model allowed us to become more familiar with the mechanics of our minigames and the PySC2 library. In creating this tabular model, we decided to quantize the 64 by 64 minimap grid state space into a more manageable 4 by 4 grid (marking desired locations and enemies), and similarly condense the number of possible actions to 33. These actions include no-op, selecting and moving an individual unit, and selecting and moving all units together. Quantizing the action and state spaces to such a degree, however, clearly sacrifices some level of precision. Unlike deep models, a Q-learning technique such as this does not find larger patterns or generalizable strategies. For instance, it struggled on the simple MoveToBeacon minigame due to its lack of precision or generalizeability of the simple strategy to move to the beacon (see our video on GitHub [9]). Similarly, on the CollectMineralShards task the combinatorial possible state configurations of the presence of shards on a 4 by 4 grid proved difficult to train on. Still, this model is able to achieve decent performance on the combat minigames, where the start and mid-game states are more consistant across episodes, averaging a score of 27 on DefeatRoaches and a score of 68 on DefeatBanelingsAndZerglings (see table above). We noted that the average reward for the policies, ignoring some variance, tended to stabilize after a relatively short 1000-5000 episodes due to the smaller state and action spaces.

4.2. Asynchronous Advantage Actor-Critic (A3C)

Additionally, we have used the state-of-the-art policy gradient method, A3C. This actor-critic algorithm represents its policy as a probability distribution of actions for each state. As described above, the policy is updated by calculating the "advantage" of increasing certain actions probabilities, as discovered by multiple actors which are simultaneously running through separate episodes and updating centralized critic and actor policies. A neural network is used to encode the policy, and yields more generalizeable strategies in comparison to the Q-Learning method detailed above. After going through a series of implementations, a repository on github by Xiaowei (xhujoy) was selected as a base. This repository can be found here: `https://github.com/xhujoy/pysc2-agents` [4]. The network architecture follows Atari_Net [1], with 4 convolutional layers, followed by 3 fully connected layers. The optimizer used is rmsprop. The policy penalty term was replaced with epsilon greedy exploration. We modified the code to be compatible with PySC2 v1.2, and trained and evaluated the model on each of the minigames.

In our experiments, we ran 8 parallel A3C agents sharing and updating the same value function. Each thread interacts with its own Starcraft II environment and plays independently of the others. When threads are spawned and agents created, each A3CAgent builds the part of the tensor specific to state representation, value function, action probabilities, through the functions in `network.py`. The rest of the network, which concerns filtering actions, computing losses and optimizing is built through the functions in `a3c_agent.py` itself. Note that we handle the fact that agents share the same value function by "reusing" Tensorflow variables.

The training alternates playing for an episode, collecting replays for this episode, and updating the network according to the results and observations. The rewards are provided and managed by pysc2 and are part of the observation.

When playing, the agent (each parallel agent) receives at each step an observation of the current state of the game: a set of precomputed feature layers that correspond to the screen and the minimap, and additionnal non spatial information. In our work, we restricted the additionnal non spatial information to the available actions that the agent can take at a given timestep (in StarCraft II, not all actions at available depending on the situation), since most non spatial information in the game is not useful for minigames. These observations are fed into the input layers of our network, which outputs a state representation corresponding to the tensor `feat_fc`. This state representation is then used to output the shared value `value`, a non-spatial action policy `non_spatial_action` and a spatial action policy `spatial_action`. The non-spatial action output corresponds to the probability of taking each of the 523 possible

actions (including actions that the game wouldn't allow it to perform in the current state of the game). The spatial action output corresponds to the screen target that will be fed to this action, if needed (not all actions target a point). After getting this output from the netwoork, the step function filters to keep only available actions in the current state of the game. It then either chooses to apply the action with highest probability, or picks randomly an action as epsilon-greedy exploration. We also perform epsilon-greedy exploration on the target arguments by moving them by random dx and dy. Each step of the game is recorded in a replay buffer.

When updating, the agent (each parallel agent) reads the buffer, retrieves the approximate value (from the network) and reward (from the environment) at each step, and computes the expected value (with finite horizon, the end of the episode). From the expected value, it then computes the advantage for each step, from which it can compute the policy and value losses and optimize the network.

In our experiments, we limit the number of steps per minigame i.e. we choose not to continue playing until the end of the minigame. The rationale is that otherwise, especially for early steps, agents can get stuck in situations where units are isolated and do not act, which slows down the training. Since situations encountered (and therefore states) for longer games are quite similar to the early ones, it seems to be a reasonable choice to speed up training. We chose to interrupt each episode after a maximum of 60 steps i.e. 60 actions taken by the agent..

We trained our A3C agent with 8 threads for a total of 100,000 episodes (12,500 per thread), on a Google Cloud instance with 2 Tesla K80 GPUs and an 8 core CPU. Each episode corresponds to roughly 500 game steps (each game step being an action taken by an agent, which corresponds to 8 real game frames), therefore our experiments correspond to a training over approximately 5 million game steps. After the training, we test our agents on full minigame, i.e. without early stopping the episodes, and collect the scores outputted by pysc2 at the end of each episode. The results are summarized in Table 1, training on a set number of game steps for each minigame and averaging over scores for 300 episodes.

5. Reducing the Action and Observation Spaces

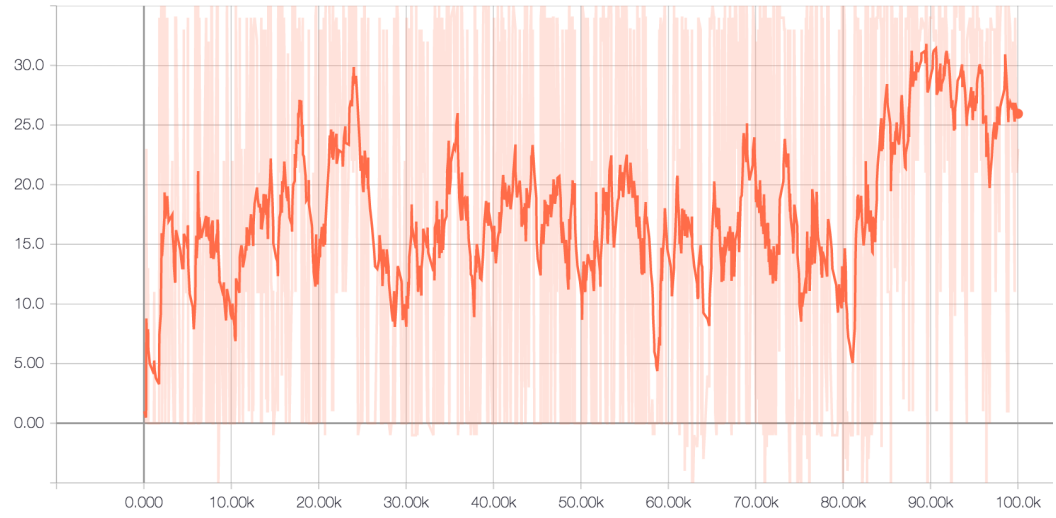
One of the major takeaways from reproducing the state of the art results was that the training time was simply too long to permit for proper tuning of hyper-parameters or experimentation with different architectures. From our conversations with deep reinforcement learning expert and practitioner Niels Justesen, we came to understand that part of the reason for this is that the action space is by default too vast and unmanageable to provide for quick training because so many combinations of actions must be attempted to arrive at a useful strategy.

Since we could not possibly hope to have the computational resources of Google's DeepMind, having to train for 5 million game steps before finding a workable was strategy was undesirable. Given this impracticality in training time, we decided to remove redundant and unnecessary observations and actions which we figured would only mislead the agent before finding the "right" observations and actions to pay attention to. Taking in all the full observation state from the SC2LE library gives a total 59 feature layers describing the the of position units, their health, the type, their player, etc. on the minimap and screen. Of these, we found that only 30 were at all useful for any of the five minigames we were training on. We thus sacrificed some generalizability to other minigames for a reduction in the complexity of the observation state by nearly 50%. Meanwhile, the action space contains a total of 524 non-spatial actions, of which we noticed that the A3C models we trained converged to only ever using 23 of these actions. Many of these actions, such as creating buildings and using powers, would only be relevant in more complex minigames or the full game anyway. We were thus able to remove extraneous actions for a reduction in the action space complexity of approximately 95%. Combined, these reductions in the action and observation space thus incorporated a bit of domain knowledge about these minigames to drastically reduce the number of dead ends which would otherwise be explored by an agent before converging to using the more useful actions and observations.

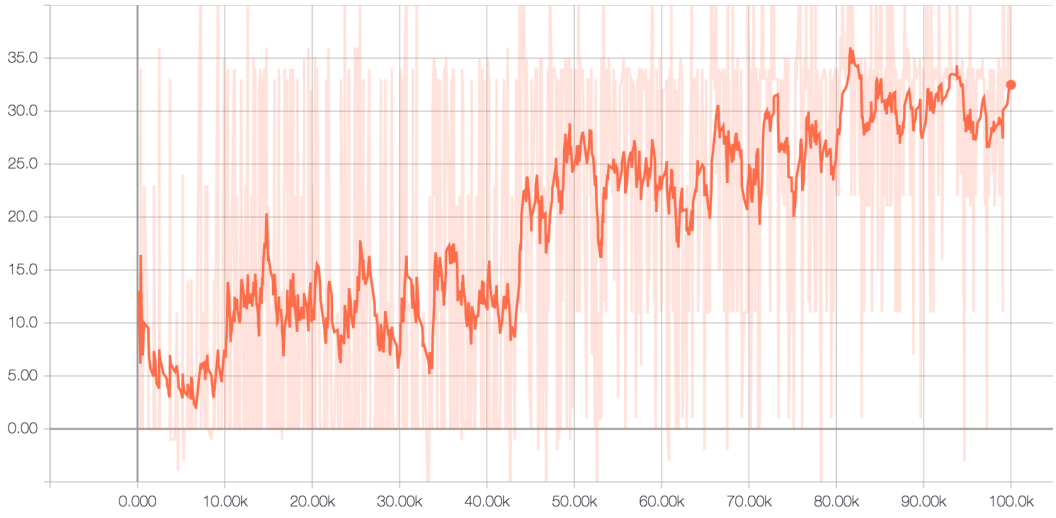
Training the A3C model with these reductions in observation and state complexity, we indeed found that the model trained more quickly. As is visible in Figure 2, we also noted that it seemed to have less of a tendency to regress in mean score. This was likely because the epsilon-greedy approach of our algorithm had previously been adding noise by occasionally trying the 500 useless actions and may have assigned them some probability due to sheer luck when it subsequently received rewards for its other more useful actions. We suspect this noise may have been enough to cause the policy to be temporarily steered away from useful actions before once again converging to them.

6. Meta-Learning Shared Hierarchies

As the above results show, in line with the results released by DeepMind, the current state of the art takes a very large number of training episodes to reach a decent policy. Worse still, these policies are so trained to the specific minigame on which they are trained that they fail to generalize across minigames. This finding leads us to the question of whether there might be a more generalizable approach. In particular, we would prefer our algorithm develop strategies which it might then be able to apply to variety of other minigames with different environments and rewards.



(a)



(b)

Figure 1: (a) Reward over the course of training A3C for a combined 100000 episodes of training (approx. 5M game steps) across 8 threads. Note that these episodes were cut short to 60 game steps during training which reduces the score below the performance for full episodes (240 game steps) as listed in Table II (b) Similarly, reward over the course of training A3C with reduced action and observation spaces.

In order to achieve such behavior, we looked to Frans et al.'s recent October 2017 work in the paper Meta-Learning Shared Hierarchies [3]. Similar to our observations in the previous section, Fran et al. note that approaching a reinforcement learning problem without any prior knowledge demands an unreasonable number of samples. Observing that tasks in similar domains tend to use similar low-level tactics which can be transferred across tasks, the authors propose that instead of learning a single policy the reinforcement learning agent learn a set of sub-policies which can then be applied to new tasks, and a master policy that selects between the learned sub-policies.

Meta-Learning Shared Hierarchies (MLSH) defines sub-policies $\pi_{\phi_k}(a|s)$ which each take in the observation in the state-space S and outputs in the action space A. Additionally, a master policy θ selects a sub-policy $\pi_{\phi_k}(a|s)$. The architecture finds a shared parameter vector ϕ and can easily be adapted to new tasks by modifying θ. The shared parameter vector ϕ is composed of a set of subvectors ϕ_k which each define a sub-policy $\pi_{\phi_k}(a|s)$. θ uses a separate neural network to select the index k corresponding to a specific sub-policy every N timesteps.

Fran et al.'s algorithm for MLSH training is compatible with a variety of reinforcement learning algorithms for

			Agent/Algorithm						
			Random Policy	Script (Specific)	Human Novice	Human Master	Q-Learn Table	A3C	A3C + MLSH
	Lines of Code		< 5	~ 70	N/A	N/A	~ 200	~ 800	~ 1100
Minigame	MoveToBeacon	Mean	1	24	26	28	2	19	1
		Max	2	30	28	28	5	31	4
	CollectMineralShards	Mean	17	94	133	177	12	35	6
		Max	19	97	142	179	38	79	17
	FindAndDefeatZerglings	Mean	3	6	46	61	7	8	13
		Max	14	40	49	61	14	11	22
	DefeatRoaches	Mean	2	108	41	215	27	61	16
		Max	36	304	81	363	96	297	91
	DefeatZerglings&Banelings	Mean	21	36	729	727	68	72	45
		Max	67	98	757	848	164	167	118

Table 1: Scores for each agent. For each non-human agent, computed on 300 episodes of testing after 5M game steps of training. The scores for novice and grandmaster human players were gathered by Google DeepMind [8].

training the the sub-policies and master policies. Given that A3C is the current state of the art on SC2LE, we decided to adapt our version of A3C to use the same advantage function gradient updates for both the master policies and sub-policies. While the MLSH paper had for their purposes used entirely separate networks to learn the master policy and sub-policies, we quickly realized that this could not scale to the complexity of the action and observation spaces in our minigames. We thus continue to share the initial layers of AtariNet–which gave the feature representation feeding into A3C's policy (actor) and value (critic) outputs–across all of the sub-policies and master policy. Each sub-policy and master policy have their own outputs layers. Using the asynchronous approach of A3C, we now have 8 agents running simultaneously (on different threads) across 4 minigames (2 on each). Each thread thus has its own master policy, but the sub-policies are shared across threads.

Our choice of the four minigames for training was MoveToBeacon, CollectMineralShards, FindAndDefeatZerglings, and DefeatRoaches. We felt these minigames covered many of the essential simple tactics in StarCraft II such as moving units to target locations, finding enemy units, and engaging in combat with enemy units. We further left out DefeatZerglingsAndBanelings as a test minigame which would be useful to see how our sub-policies could generalize to unseen (during training) tasks. In testing on this unseen minigame, we would thus train a new master policy without updating the sub-policies which it selects from. This training of just the master policy would require dramatically fewer updates to the network to converge since the master policy operates over longer spans of time.

Our training algorithm for MLSH, based on the MLSH paper, proceeds as follows. For each thread, there is first a warmup period of 100 episodes, during which only the master policy θ is optimized using some randomly initialized sub-policies ϕ. For our implementation of MLSH, we let the master policy select a sub-policy every 5 game steps, which translates to 2.5 seconds. Accordingly, the gradient updates to the master policy are thus based on the rewards obtained over the course of the this set of 5 game steps. We settled on 5 game steps as a suitable frequency for changing sub-policy after observing that it only takes about 5 seconds to lose all of one's units in combat in DefeatRoaches. Our hope was thus that the master policy would have time to change subpolicy before the combat was over. The thinking behind updating only the master policy during this phase (at the end of each episode) is that the agent will more effectively learn sub-policies once it has a master policy that is close to optimal. After this warmup period, there is a joint update period where both the master policy and the sub-policies are updated at the end of each episode. This is done over 500 episodes for each of the four training minigames. After this joint training period, we hope that the sub-policies have picked up useful skills which will be more useful. We thus reset the master policy at this point (for each thread when it finished the joint update period), and then cycle repeatedly through these warmup and joint update periods. Then the sub-policies are applied to the test minigame DefeatZerglingsAndBanelings, where the master policy is optimized.

7. Results

In order to give a fair comparison between our MLSH agents and the A3C agents, we trained MLSH for 20M game steps (5M for each of the four minigames on which it is run). We then computed the mean and max scores for the resulting agent on each of these four trained minigames as well as the unseen minigame DefeatZerglingsAndBanelings. We trained this model twice, one with 2 sub-policies

and one with 3 sub-policies.

Our agents got best results when trained with 3 subpolicies, which are the scores reported in Table 1. Even though the agents did not reach scores as high as A3C, we did notice interesting behaviors. Recall that our goal is to build agents that can master each minigame as well as be able to switch between them, therefore a trade-off appears between generalization and expertise at given minigames.

Figure 2 shows an example of two subpolicies learned by our agent, and gives the frequency of each action type over an episode (all 522 actions are represented on the x axis). Interestingly, one of the policy converged to using more often actions between 0 and 5, which are actions for selecting or moving units, while the other converged to using actions around action 330 which corresponds to moving the camera and the minimap. The ladder types of actions are more useful for games such as FindAndDefeatZerglings (the camera is fixed in all other minigames), while the former are useful for other minigames. In a full game, our agent would need both types of actions and would therefore be able to use the master policy choices in order to alternate between both, which is very promising.

Another promising result is how trained agents use subpolicies on minigames. To generate the results in Table 1, once our agent learned subpolicies training on multiple minigames simultaneously, we train train the master policy on a single game and analyze its performance. While training, we collected the mostly used subpolicy per minigame: When playing CollectMineralShards, our agent uses the first subpolicy; it learns to use the second subpolicy on MoveToBeacon; finally it uses the third subpolicy on the three other games, which we note are the only combat minigames. This promising result shows that our agent learned subpolicies that are good for specific subtasks, and when learning how to play a minigame, the master policy learns quickly which subpolicy is optimal for the current minigame. In the full game, this would mean that our agent could identify easily which subpolicy to use given the context.

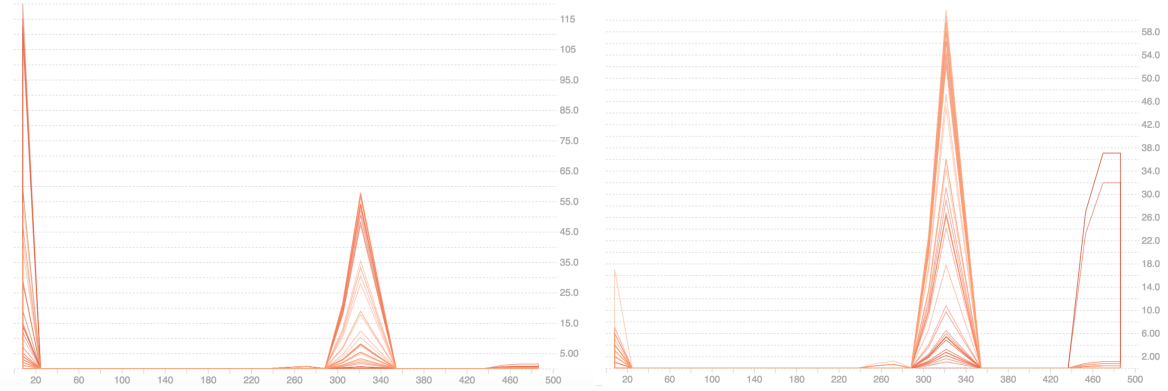
As for our goal of generalizability to new minigames, we observe that our MLSH agent performs reasonably well on the unseen minigame DefeatZerglingsAndBanelings. While the results were significantly below those of A3C, they were much better than random and notably only required a short training of the master policy for 100 episodes, or just 4800 updates to the network before achieving this score. Note that this is orders of magnitude faster than the 5 million policy gradient updates required for our previous training of an A3C agent from scratch for DefeatZerglingsAndBanelings.

One potential difficulty in training which we did not anticipate–which may have resulted in such low scores on MoveToBeacon and CollectMineralShards–is that the minigames on which we train have different magnitudes of rewards. Further, some minigames yield high rewards using only a few actions (DefeatZerglingsAndBanelings and DefeatRoaches) while other minigames require many actions to achieve high rewards (MoveToBeacon and CollectMineralShards). We suspect that this difference in the magnitude of expected rewards given certain actions may cause the magnitude of the advantage function in our A3C policy gradient update to depend on the minigame being trained on. Training the same sub-policies on different minigames simultaneously in MLSH, then, it is likely that some thread's policy updates have larger magnitude than others and thus tend to dominate the shared sub-policies. In the future, some sort of normalization or additional loss term encouraging performance across the minigames may lead to results which do not skew as heavily as our results documented here. This may explain the gap in performance on FindAndDefeatZerglings, which was better than A3C, while performance on CollectMineralShards and MoveToBeacon was on par with a random policy .

8. Future Work

There is clearly much work to be done to refine our work combining A3C with MLSH. One area where there is much room for improvement is in setting hyper-parameters. We note that our implementation ha large number hyper-parameters, including number of sub-policies, epsilon use for epsilon-greedy choices across master and sub-policies, length of warmup and joint update periods, number of steps taken per choice of sub-policy, learning rate and schedule, etc. Our current settings for these hyper-parameters are typically assigned to the standards used in the related works we cite. However, certain parameters such as the length of warmup and joint update periods, number of sub-policies, or number of steps taken per choice of sub-policy will need to be tuned more specifically to the SC2LE environment and the unique challenges it presents. To our knowledge, there is no known method of tuning such hyper-parameters aside from training models with different settings and comparing performance. Unfortunately, due to the high variance in performance over the course of training each model, it takes an intractable (given our computing resources) amount of time before a robust comparison can be made. We hope to run such comparisons in the feature, or perhaps try another method such as running each agent many times for a short number of episodes and comparing the relative speeds improvement rather than performance after convergence.

A more speculative area for potential improvement would be to introduce a more complicated hierarchy of master policy, sub-policies, sub-sub-policies, etc. This would allow for more complex reasoning, such as selecting a general strategy at the master level such as aggressive, defensive, economic focus, etc., learning general sub-policies to



(a) (b)

Figure 2: (a) Action use frequency for each of the 522 actions over an episode, for a given subpolicy (agent trained using 2 subpolicies). Actions around index 0 correspond to selecting, moving, attacking; actions around index 330 as well as actions around index 460 correspond to moving the camera or the minimap. (b) Similar frequencies, for the other learned subpolicy of the agent.

achieve those goals, and then allowing those sub-policies to select among sub-sub-policies which are the more fine-grained controllers. It is unclear what the proper number of levels should be in this hierarchy, but such an approach could be closer to how human players approach the game and transfer their skills between matches with different goals (rewards to the DRL learner). For instance, after training such a model we might only train the upper levels of such a hierarchy while leaving the lower levels as they are. Moving beyond a simple two-level structure of MLSH could further reduce the complexity of mastering a new and unseen task, allowing for training of only the higher levels of the policy hierarchy.

In conclusion, we remark that the overall goal of research on SC2LE agent training is to develop agents competitive with and even surpassing human performance on full matches. Because of the complexity of this task, as outlined in this paper, we believe this will likely require transfer and/or curriculum learning in order to create a model which can generalize from simpler tasks such as the minigames we have focused on here. Learning tabula rasa on the full game using current DRL methods is completely intractable given the combinatorial increases in complexity with longer-lasting games, more unit types, and buildings. Having established the state of the art results and developed an MLSH agent with "system 2" higher level deliberation, we hope that our openly published results and code combining A3C with MLSH may help others pursuing similar methods in the future. Despite the much remaining work to be done toward the goal of a fully generalizable agent, we believe that the results presented here may serve as a first step towards overcoming current performance limitations.

References

[1] M. Bellemare, Y. Naddaf, J. Veness, and M. Bowling. The Arcade Learning Environment: An evaluation platform for general agents. J. Artif. Intell. *Res.(JAIR)*, 47:253 279, 2013.

[2] S. Brown. Building A Spares PySC2 Agent. https://github.com/skjb/pysc2-tutorial

[3] K. Frans, J. Ho, X. Chen, P. Abbeel, J. Schulman. Meta Learning Shared Hierarchies. *arXiv preprint arXiv:1710.09767v2*, 2017.

[4] X. Joy. PySC2 Agents.https://github.com/xhujoy/pysc2-agents

[5] A. Juliani. Deep Reinforcement Learning Agents in TensorFlow. https://github.com/awjuliani/DeepRL-Agents

[6] N. Justesen, P. Bontrager, J. Togelius, R. Sebastian. Deep Learning for Video Game Playing. *arXiv preprint arXiv:1708.07902v2*, 2017.

[7] V. Mnih, A. Badia, M. Mirza1, A. Graves, T. Harley, T. Lillicrap, D. Silver, K. Kavukcuoglu. *Asynchronous Methods for Deep Reinforcement Learning*, 2016.

[8] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev. et al. StarCraft II: A New Challenge for Reinforcement Learning. *Google DeepMind*, 2017.

[9] https://github.com/roop-pal/Deep-Reinforcement-Learning-for-StarCraft-II-Battles