Gym-μRTS: Toward Affordable Full Game Real-time Strategy Games Research with Deep Reinforcement Learning

Shengyi Huang
Drexel University
Philadelphia, PA, USA
sh3397@drexel.edu

Santiago Ontañón*
Drexel University
Philadelphia, PA, USA
sh3397@drexel.edu

Chris Bamford
Queen Mary University
London, United Kindom
c.d.j.bamford@qmul.ac.uk

Lukasz Grela
grotrianster@gmail.com

Abstract—In recent years, researchers have achieved great success in applying Deep Reinforcement Learning (DRL) algorithms to Real-time Strategy (RTS) games, creating strong autonomous agents that could defeat professional players in StarCraft II. However, existing approaches to tackle full games have high computational costs, usually requiring the use of thousands of GPUs and CPUs for weeks. This paper has two main contributions to address this issue: 1) We introduce Gym-μRTS (pronounced "gym-micro-RTS") as a fast-to-run RL environment for full-game RTS research and 2) we present a collection of techniques to scale DRL to play full-game μRTS as well as ablation studies to demonstrate their empirical importance. Our best-trained bot can defeat every μRTS bot we tested from the past μRTS competitions when working in a single-map setting, resulting in a state-of-the-art DRL agent while only taking about 60 hours of training using a single machine (one GPU, three vCPU, 16GB RAM).

Index Terms—Deep reinforcement learning, Real-time strategy games

I. INTRODUCTION

In recent years, researchers have achieved great success in applying Deep Reinforcement Learning (DRL) algorithms to Real-time Strategy (RTS) games. Most notably, DeepMind trained a grandmaster-level AI called AlphaStar with DRL for the popular RTS game StarCraft II [1]. AlphaStar demonstrates impressive strategy and game control, presenting many human-like behaviors, and is able to defeat professional players consistently. Given most previously designed bots fail to perform well in the full-game against humans [2], AlphaStar clearly represents a significant milestone in the field. While this accomplishment is impressive, it comes with high computational costs. In particular, AlphaStar and even further attempts by other teams to lower the computational costs [3] still require thousands of CPUs and GPUs/TPUs to train the agents for an extended period of time, which is outside of the computational budget of most researchers.

This paper has two main contributions to address this issue. The first main contribution is to introduce Gym-μRTS as an

RL testbed for affordable full-game RTS research, which focuses on all aspects of the game such as harvesting resources, defending units, and attack enemies (this is in contrast to *mini-games* that only focus on one aspect of the game). Gym-μRTS is a reinforcement learning interface for the RTS game μRTS [4], which has been a popular platform to test out a variety of AI techniques for RTS games. Despite its simple visuals, μRTS captures the core challenges of RTS games. Although Gym-μRTS shares many similarities to the StarCraft II Learning Environment (PySC2) [5], there are also many key differences (e.g., PySC2 uses a human-like action space whereas Gym-μRTS uses a lower-level action space). Through Gym-μRTS, we are able to conduct full-game RTS research using DRL without extensive technical resources such as high-performance compute clusters.

Despite the simplifications done in μRTS, playing 1v1 competitive matches via DRL is still a daunting task. Thus, our second main contribution is a collection of techniques to scale DRL to play μRTS. We start with a Proximal Policy Optimization (PPO) [6] implementation that matches implementation details of PPO in *openai/baselines* [7], and incrementally stack augmentations to account for Gym-μRTS's combinatorial action space (all units must be controlled simultaneously) and improve training efficiency and performance. Among these augmentations, two are *essential*: 1) action composition and 2) invalid action masking. These two augmentations combined allowed us to bootstrap an initial agent that could compete on the 16×16 map to a reasonable standard. Additionally, we experimented with 3) diversified training opponents, and 4) different neural network architectures. We provide ablation studies to shed insights on the importance of each of these augmentations. Our best-trained agent can defeat every μRTS bot we tested against, from the past μRTS competitions[1] in a single-map setting, establishing a new state-of-the-art for DRL bots in μRTS while only taking about 60 hours of training using a single machine (one GPU, three vCPU, 16GB RAM). We make source code and trained models [2], as well as all the

---

*Currently at Google

[1]https://sites.google.com/site/micrortsaicompetition/home
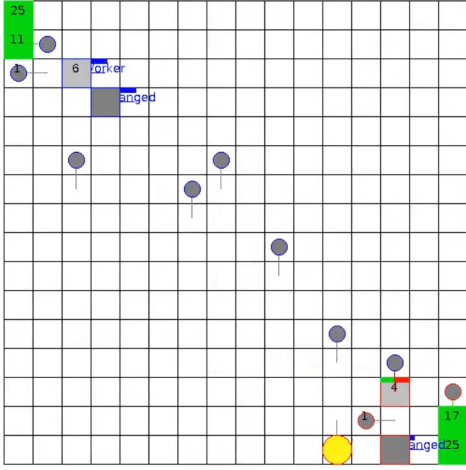[2]https://github.com/vwxyzjn/gym-microrts-paper

Fig. 1: Screenshot of our best-trained agent (top-left) playing against CoacAI (bottom-right), the 2020 μRTS AI competition champion. Strategy-wise, our agent usually defeats CoacAI by harvesting resources (green squares) efficiently using two workers (dark gray circles), doing a highly optimized worker rush that takes out the enemy base in the bottom right (shown with 50% damage), followed by a transition to the mid and late game by producing combat units (colored circles) from the barracks (dark gray squares). The blue and red border suggest the unit is owned by player 1 and 2, respectively. See additional combat videos here: https://bit.ly/3llOhex

metrics, logs, and recorded videos[3] available for comparison.

II. BACKGROUND

Real-time Strategy (RTS) games are complex adversarial domains, typically simulating battles between a large number of combat units, that pose a significant challenge to both human and artificial intelligence [8]. Designing AI techniques for RTS games is challenging due to a variety of reasons: 1) players need to issue actions in real-time, leaving little time computational budget, 2) the action spaces grows combinatorially with the number of units in the game, 3) the rewards are very sparse (win/loss at the end of the game), 4) generalizing against diverse set of opponents and maps is difficult, and 5) stochasticity of game mechanics and partial observability (these last two are not considered in this paper).

StarCraft I & II are very popular RTS games and, among other games, have attracted much research attention. Past work in this area includes reinforcement learning [9], case-based reasoning [10], [11], or game tree search [12]–[15] among many other techniques designed to tackle different sub-problems in the game, such as micromanagement, or build-order generation. In the full-game settings, however, most techniques have had limited success in creating viable agents to play competitively against professional StarCraft players until recently. In particular, DeepMind introduced AlphaStar [1], an

agent trained with DRL and self-play, that sets a new state-of-the-art bot for StarCraft II, defeating professional players in the full-game. In Dota 2, a popular collaborative online-player game that shares many similar challenges as StarCraft, Open AI Five [16] is able to create agents that can achieve super-human performance. Although these two systems achieve great performance, they come with large computational costs. AlphaStar used 3072 TPU cores and 50,400 preemptible CPU cores for a duration of 44 days [1], [3]. This makes it difficult for those with less computational resources to do full-game RTS research using DRL.

There are usually three ways to circumvent this computational costs. The first way is to focus on sub problems such as combat scenarios [17]. The second way is to reduce the full-game complexity by either considering hierarchical actions spaces or incorporating scripted actions [18], [19]. The third way is to use alternative game simulators that run faster such as Mini-RTS [20], Deep RTS [21], and CodeCraft[4].

We show that Gym-μRTS as an alternative that could be used for full-game RTS research with the full action space while using affordable computational resources.

III. PRELIMINARIES

In this paper, we use policy gradient methods to train agents. Let us consider the Reinforcement Learning problem in a Markov Decision Process (MDP) denoted as $(S, A, P, \rho_0, r, \gamma, T)$, where S is the state space, A is the discrete action space, $P : S \times A \times S \rightarrow [0, 1]$ is the state transition probability, $\rho_0 : S \rightarrow [0, 1]$ is the initial state distribution, $r : S \times A \rightarrow \mathbb{R}$ is the reward function, γ is the discount factor, and T is the maximum episode length. A stochastic policy $\pi_\theta : S \times A \rightarrow [0, 1]$, parameterized by a parameter vector θ, assigns a probability value to an action given a state. The goal is to maximize the expected discounted return of the policy:

$$J = \mathbb{E}_\tau \left[\sum_{t=0}^{T-1} \gamma^t r_t \right]$$

where τ is the trajectory $(s_0, a_0, r_0, \ldots, s_{T-1}, a_{T-1}, r_{T-1})$ and $s_0 \sim \rho_0, s_t \sim P(\cdot|s_{t-1}, a_{t-1}), a_t \sim \pi_\theta(\cdot|s_t), r_t = r(s_t, a_t)$

The core idea behind policy gradient algorithms is to obtain the *policy gradient* $\nabla_\theta J$ of the expected discounted return with respect to the policy parameter θ. Doing gradient ascent $\theta = \theta + \nabla_\theta J$ therefore maximizes the expected discounted reward. Earlier work proposes the following policy gradient estimate to the objective J [22]:

$$\nabla_\theta J = \mathbb{E}_{\tau \sim \pi_\theta} \left[\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) G_t \right], \quad G_t = \sum_{k=0}^{\infty} \gamma^k r_{t+k}$$

This gradient estimate, however, suffers from high variance [22] and there are many techniques and extensions to address it [6], [23], [24].

---

[3]https://wandb.ai/vwxyzjn/gym-microrts-paper

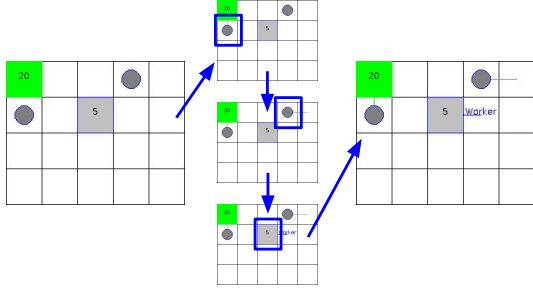[4]https://github.com/cswinter/DeepCodeCraft

Fig. 2: Unit Action Simulation (UAS) calls the policy iteratively until all units have actions (at each step, the policy chooses a unit and issue a unit action to it).
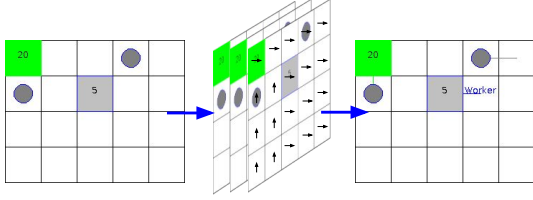

Fig. 3: Gridnet predicts an action for each cell in the map (it predicts all the action component planes in one step), which then define which actions each unit will perform.

IV. GYM-μRTS

Gym-μRTS[5] is a reinforcement learning interface for the RTS games simulator μRTS[6]. Despite having a simplified implementation, μRTS captures the core challenges of RTS games, such as combinatorial action space, real-time decision-making, optionally partial observability and stochasticity. Gym-μRTS's observation space provides a series of feature maps similar to PySC2 (the StarCraft II Learning environment [5]). Its action space design, however, is more low-level due to its lack of AI assisted actions. In this section, we introduce their technical details.

A. Observation Space.

Given a map of size $h \times w$, the observation is a tensor of shape (h, w, n_f), where n_f is a number of feature planes that have binary values. The observation space used in this paper uses 27 feature planes as shown in Table I. The different feature planes result as the concatenation of multiple one-hot encoded features. As an example, if there is a worker player 1 with hit points equal to 1, not carrying any resources, and currently not executing any actions, then the one-hot encoding features will look like this (see Table I):

$$[0, 1, 0, 0, 0], [1, 0, 0, 0, 0], [1, 0, 0],$$
$$[0, 0, 0, 0, 1, 0, 0, 0], [1, 0, 0, 0, 0, 0]$$

TABLE I: Observation features and action components. $a_r = 7$ is the maximum attack range.

Observation Features	Planes	Description
Hit Points	5	0, 1, 2, 3, ≥ 4
Resources	5	0, 1, 2, 3, ≥ 4
Owner	3	player 1, -, player 2
Unit Types	8	-, resource, base, barrack, worker, light, heavy, ranged
Current Action	6	-, move, harvest, return, produce, attack

Action Components	Range	Description
Source Unit	$[0, h \times w - 1]$	the location of the unit selected to perform an action
Action Type	$[0, 5]$	NOOP, move, harvest, return, produce, attack
Move Parameter	$[0, 3]$	north, east, south, west
Harvest Parameter	$[0, 3]$	north, east, south, west
Return Parameter	$[0, 3]$	north, east, south, west
Produce Direction Parameter	$[0, 3]$	north, east, south, west
Produce Type Parameter	$[0, 6]$	resource, base, barrack, worker, light, heavy, ranged
Relative Attack Position	$[0, a_r^2 - 1]$	the relative location of the unit that will be attacked

Each feature plane contains one value for each coordinate in the map. The values for the 27 feature planes for the position in the map of such worker will thus be:

$$[0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 0]$$

B. Action Space.

Compared to traditional reinforcement learning environments, the design of the action space of RTS games is more difficult because, depending on the game state, there is a different number of units to control, and each unit might have different number of actions available. This poses a challenge for directly applying off-the-shelf DRL algorithm such as PPO that generally assume a fixed output size for the actions. Early work on RL in RTS games simply learned policies for individual units, rather than having the policy control all the units at once [9]. To address this issue, we decompose the action space into two parts: the *unit action space* (the space of possibilities for issuing actions to only one unit) and the *player action space* (the space of unit actions for all the units a player owns).

In the unit action space, given a map of size $h \times w$, the unit action is an 8-dimensional vector of discrete values as specified in Table I. The first component of the unit action vector represents the unit in the map to issue commands to, the second is the unit action type, and the rest of components represent the different parameters different unit action types can take. Depending on which unit action type is selected, the game engine will use the corresponding parameters to execute the action. As an example, if the RL agent issues a "move

south" unit action to the worker at $x = 3, y = 2$ in a 16×16 map, the unit action will be encoded in the following way:

$$[3 + 2 * 16, 1, 2, 0, 0, 0, 0, 0]$$

In the player action space, we compare two ways to issue player actions to a variable number of units at each frame: **Unit Action Simulation (UAS)** and **Gridnet** [25].

Their mechanisms are best illustrated through an example as shown in Figs. 2 and 3, where the player owns two workers and a base in a 4×5 map.

UAS calls the RL policy iteratively. At each step, the policy chooses a unit based on the *source unit masks* (a vector of $h \times w$ scalars). It then chooses the action type and parameter via the *unit action masks* (a vector of $6+4+4+4+4+7+49$ scalars). We then compute a "simulated game state" where that action has been issued (and any potential rewards collected). Once all three units have been issued actions, the simulated game states are discarded, and the three actions are collected and sent to the actual game environment.

Under Gridnet [25], The RL agent receives a *player action mask* (a tensor of shape $(h, w, 1+6+4+4+4+4+7+49)$, where the first plane indicates if the source unit is available). It then issues actions to each cell in this map in one single step, that is, issues in total $4*5 = 20$ unit actions. The environment executes the three valid actions (actions in cells with no player-owned units are ignored).

C. The Action Spaces of Gym-μRTS and PySC2

Although Gym-μRTS is heavily inspired by and shares many similarities with PySC2 [5], their action space designs are considerably different. Specifically, PySC2 has designed its action space to mimic the human interface, while Gym-μRTS has a more low-level action that require actions being issued for each individual unit. This distinction is rather interesting from a research standpoint because certain challenges are easier for an AI agent and some more difficult.

Consider the canonical task of harvesting resources and returning them to the base. In PySC2, the RL agent would need to issue two actions at two timesteps 1) select an area that has workers and 2) move the selected workers towards to a coordinate that has resources. Then, the workers will continue harvesting resources until otherwise instructed. Note that this sequence of actions is assisted by AI algorithms such as path-finding. After the workers harvest the resources, the engine automatically determines the closest base for returning the resources, and repeating these actions to continuously harvest resources. So the challenge for the RL agent is to learn to select the correct area and move to the correct coordinates. In Gym-μRTS, however, the RL agent can only issue primitive actions to the workers such as "move north for one cell" or "harvest resource that is one cell away at north". Therefore, it needs to constantly issue actions to control units at all times,

having to learn how to perform these AI-assisted decisions from scratch[7].

The benefit of PySC2's approach is that it makes it easier to do imitation learning from human datasets and the resulting agent will have a fairer comparison when evaluated against humans since the AI and the human are mostly playing the same game. That being said, the human interface could be an artificial limitation to the AI system. In particular, the human interface is constructed to accommodate the human limitations: humans' eyes have limited range, so camera locations are designed to help capture larger maps, and humans have limited physical mobility, so hotkeys are set to help control a group of units with one mouse click. However, machines don't have these limitations and can observe the entire map and issue actions to all units individually.

D. Reward Function

We use a *shaped reward* function to train the agents, which gives the agent +10 for winning, 0 for drawing, -10 for losing, +1 for harvesting one resource, +1 for producing one worker, +0.2 for constructing a building, +1 for each valid attack action it issues, +4 for each combat unit it produces. It gives the rewards to the frame at which the events are initialized (e.g. attack takes 5 game frames to finish, but the attack reward is given at the first frame). For reporting purposes, we also keep track of the *sparse reward*, which is +1 for winning, 0 for drawing, -1 for losing. The shaped reward weights are picked by hand with very little tuning.

Note this shaped reward function is similar to the one used in Open AI Five for Dota 2 [16]. Like in Open AI Five, it is possible for the agents to gain more shaped rewards by doing other good behaviors than winning the game outright. Notice we have avoided using very large win/lose rewards because anecdotally large reward numbers could cause worse performance for RL algorithms, which might be the reason why reward normalization [26] or reward clipping [27] have been used in previous work.

V. SCALING DRL TO GYM-μRTS

We use PPO [6], a popular policy gradient algorithm, to train agents for all experiments in this paper. In addition to PPO's core algorithm, many implementation details and empirical setting also have a huge impact on the algorithm's performance [26].

We start with a PPO implementation that matches the implementation details and benchmarked performance in *openai/baselines* [7][8], and use it along with the architecture from Mnih, et al. (denoted as Nature-CNN) [27] as the baseline. We train the RL agents using UAS and Gridnet by playing against CoacAI, the 2020 μRTS competition winner, in the standard `16x16basesWorkers` map, where the RL agents always spawn from the top left position and end episodes after 2000

---

[7]Notice, however that μRTS offers both the low-level interface and a PySC2-style interface with AI-assisted actions, but for Gym-μRTS, we only expose the former.

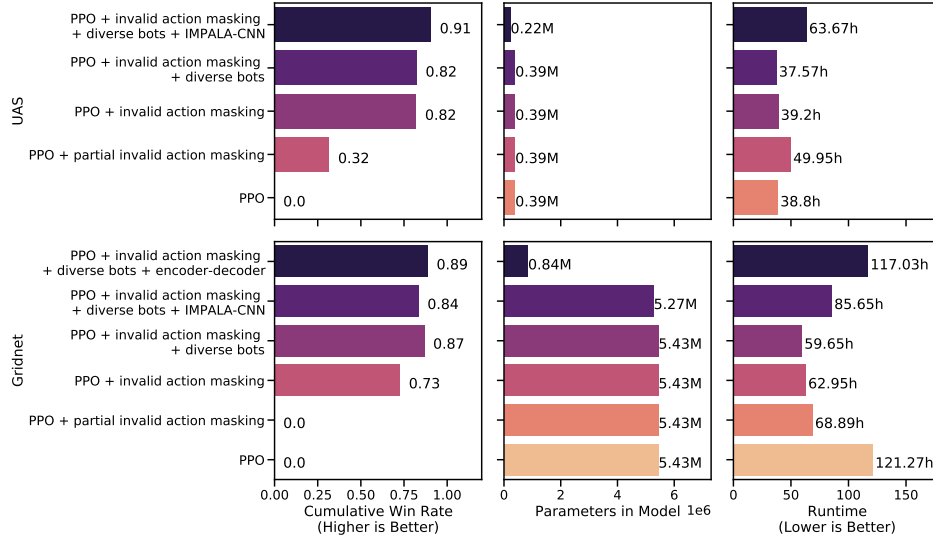[8]See https://costa.sh/blog-the-32-implementation-details-of-ppo.html

Fig. 4: Ablation study for UAS and Gridnet.

game ticks. We then incrementally include augmentations for both UAS and Gridnet and compare their relative performance.

We run each ablation with 4 random seeds each. Then, we select the best performing seeds according to the reported sparse reward function and evaluate them against a pool of 11 bots with various strategies that have participated in previous μRTS competitions (other competition bots are not included due to either staleness or difficulty to set up) and 2 baseline bots which are mainly used for testing. All μRTS bots are configured to use their μRTS competition parameters and setups. The name, category and best result of these bots are listed in Table II. The evaluation involves playing 100 games against each bot in the pool for 4000 maximum game ticks, and we report the cumulative win rate, the model size, and total run time in Figure 4. To further provide insights, we record videos of the RL agents against each of the bots in the pool and make them publicly available[9]. Let us now describe the different augmentations we added on top of PPO.

A. Action Composition

After having solved the problem of issuing actions to a variable number of units (via either UAS or Gridnet), the next problem is that even the action space of a single unit is too large. Specifically, to issue a single action a_t in μRTS using UAS, according to Table I, we have to select a Source Unit, Action Type, and its corresponding action parameters. So in total, there are $hw \times 6 \times 4 \times 4 \times 4 \times 4 \times 6 \times a_r^2 = 9216(hwa_r^2)$ number of possible discrete actions, which includes many invalid actions, which is huge even for small maps (about 50 million in the map size we use in this paper).

To address this problem, we use *action composition*, where we consider an action as composed of some smaller

independent discrete actions. Namely, a_t is composed of a set of smaller actions $D = \{a_t^{\text{Source Unit}}, a_t^{\text{Action Type}}, a_t^{\text{Move Parameter}}, a_t^{\text{Harvest Parameter}}, a_t^{\text{Return Parameter}}, a_t^{\text{Produce Direction Parameter}}, a_t^{\text{Produce Type Parameter}}, a_t^{\text{Relative Attack Position}}\}$. And the policy gradient is updated in the following way (without considering the PPO's clipping for simplicity):

$$\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_t|s_t) G_t = \sum_{t=0}^{T-1} \nabla_\theta \left(\sum_{a_t^d \in D} \log \pi_\theta(a_t^d|s_t) \right) G_t$$

$$= \sum_{t=0}^{T-1} \nabla_\theta \log \left(\prod_{a_t^d \in D} \pi_\theta(a_t^d|s_t) \right) G_t$$

Implementation-wise, for each action component, the logits of the corresponding shape are output by the policy, which we refer to as action component logits. Each action a_t^d is sampled from a softmax distribution parameterized by these action component logits. In this way, the algorithm has to generate $hw + 6 + 4 + 4 + 4 + 4 + 6 + a_r^2 = hw + 36 + a_r^2$ logits, significantly less than $9216(hwa_r^2)$ (301 vs 50 million).

B. Invalid Action Masking

The next most important augmentation in our experiments is *invalid action masking*, which "masks out" invalid actions out of the action space (by exploiting the fact that we know the rules of the game), significantly reducing it. This is used in PySC2 [5], OpenAI Five [16], and a number of related work with large action spaces [17].

Masks are generated and being applied as shown in Figure 5. Under UAS, the agent would first sample a source unit based on the source units masks of shape (hw), then query the game client for the action type and parameter mask of the said units with shape (78). Under Gridnet, the agent would receive all the masks up front on source unit, action type and parameter with shape $(hw, 79)$, where the first plane of 79
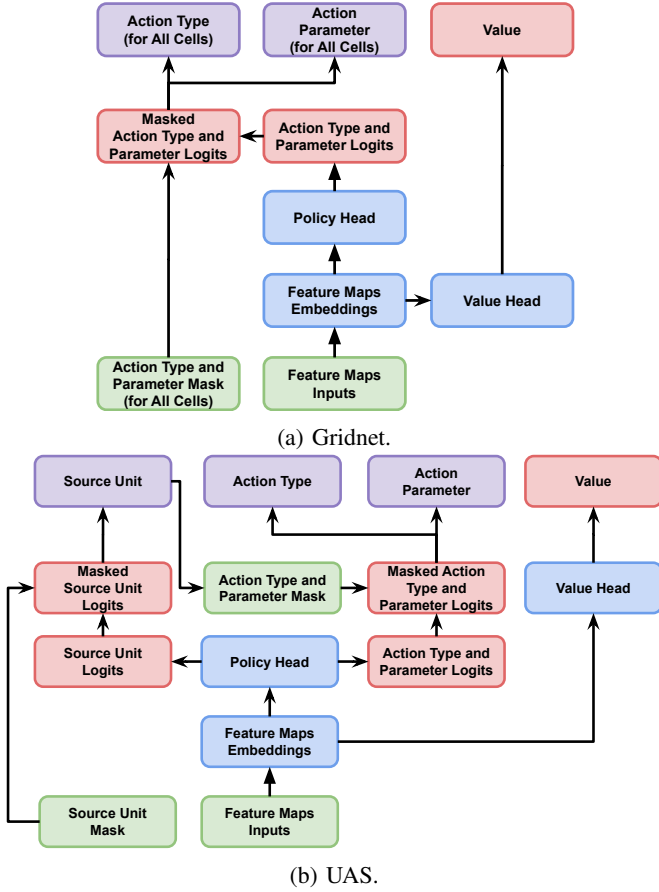
(a) Gridnet.

(b) UAS.

Fig. 5: Neural network architectures for Gridnet and UAS. The green boxes are (conditional) inputs from the environments, blue boxes are neural networks, red boxes are outputs, and purple boxes are sampled outputs.

is the mask on the source unit selection. Note that in both cases, the agent received a *full* action mask that in a sense significantly reduce the search space. In contrast, PySC2 and SMAC (the StarCraft Multi-Agent Challenge) [17] would only provide a *partial* mask on the action type, and the logits of action parameters are unmasked (our action types and action parameters are function identifiers and arguments in PySC2's term). This could explain why invalid action masking does not seem to cause as drastic of a difference in PySC2 as shown by Kanervisto et al. [28].

In the interest of ablation study, we also conduct experiments that provide masking on the action types but not the action parameters, which is more similar to PySC2's settings. As shown in Figure 4, we see that having only a partial mask has little impact whereas having the full mask considerably improves performance. Although the action space and PySC2 is quite different as discussed above, masking all invalid actions maximally reduces the action space, hence simplifying the learning task. We therefore believe that the PySC2 agents could receive a performance boost by providing masks on function arguments as well.

TABLE II: The previous μRTS competition bots.

Name	Category	Best result
CoacAI	Scripted	1st place in 2020
Tiamat	MCTS-based	1st place in 2018
MixedBot	MCTS-based	2nd place in 2019
Droplet	MCTS-based	3rd place in 2019
Izanagi	MCTS-based	4th place in 2019
Rojo	MCTS-based	5th place in 2020
LightRush	Scripted	6th place in 2020
GuidedRojoA3N	MCTS-based	7th place in 2020
WorkerRush	Scripted	8th place in 2020
NaiveMCTS	MCTS-based	9th place in 2020
RandomBiasedAI	Scripted	10th place in 2020
Random	Scripted	-
PassiveAI	Scripted	-

C. Other augmentations

This section details other additional augmentations that contribute to the agents' performance, but not as much as the previous two (which are essential for having an agent that even starts learning to play the full game).

1) Diverse Opponents: The baseline setting is to train the agents against CoacAI. However, this lacks a diversified experience and when evaluating, we frequently see the agents being defeated by AIs as simple as WorkerRush. To help alleviate this problem, we train the agents against a diverse set of built-in bots. Since we train with 24 parallel environments for PPO, we set 18 of these environments to have CoacAI as the opponent, 2 to have RandomBiasedAI, 2 to have WorkerRush, and 2 to have LightRush. Per Figure 4, we see a rather significant performance boost for Gridnet, whereas in UAS the performance boost is milder.

2) Nature-CNN vs Impala-CNN vs Encoder-Decoder: To seek better neural network architectures, we experimented with the use of residual blocks [29] (denoted IMPALA-CNN), which have been shown to improve the agents' performance in several domains like DMLab [24]. Additionally, Han et al. [25] also experimented with an encoder-decoder network in Gridnet, so we also conducted experiments using this architecture. Per the ablation study in Figure 4, we see IMPALA-CNN helps with the performance of UAS whereas encoder-decoder benefits Gridnet.

VI. DISCUSSION

Establishing a SOTA in Gym-μRTS. According to Figure 4, our best agent consists of *ppo + coacai + invalid action masking + diverse opponents + impala cnn*, reaching the cumulative win rate of 91%. Additionally, Figure 6 shows the specific match results, showing this agent can outperform all other bots in the pool. Note that in the μRTS competition settings the players could start in two different locations of the map whereas our agent always start from the top left. Nevertheless, due to the symmetric nature of the map, we could address this issue by "rotating" the map when needed so that both starting locations look the same to our agent. Therefore, our agent establishes the state of the art for μRTS in the `16x16basesWorkers` map. Note that generalizing to
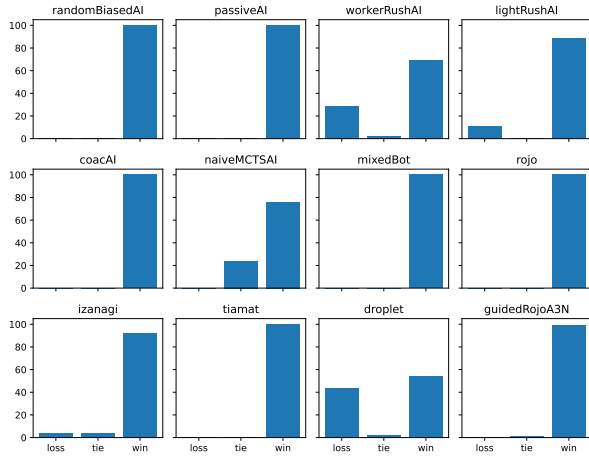
Fig. 6: Match results: the y-axis shows the number of losses, ties, and wins against AIs listed in Table II. The Random bot's match result is excluded for presentation purposes.
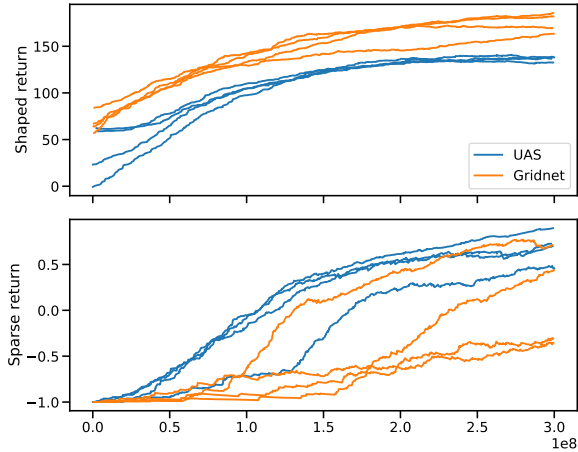

Fig. 7: The shaped and sparse return over training steps for all 4 random seeds of *PPO + invalid action masking* for Gridnet and UAS. The curve is smoothed using exponential moving average with weight 0.99.

handle a variety of maps (including the asymmetric ones) in μRTS competition settings is part of our future work (also note that some work on StarCraft II also focused in the one map setting [3], while still requiring large computation budgets).

Our best agent struggles the most against the Droplet bot, which typically uses a worker rush strategy, but enhanced thanks to MCTS search. Droplet usually defeats our agent by destroying the first barracks our agent makes, which is a rare experience with other bots. As a result, our agent would keep trying to build a barracks until it exhausts its resources, at which point, Droplet would have more left over resources, build more workers and eventually defeat our agent. However, if by chance our agent successfully builds and protects the first barracks and combat units, it is usually able to defeat Droplet. As part of our future work, we would like to include agents

like Droplet in our training process. However, search-based bots like Droplet significantly decrease the speed of training.

Hardware Usage and Training Time. Most of our experiments are conducted using 3 vCPUs, 1 GPU, and 16GB RAM. According to Figure 4, the experiments take anywhere from 37 hours to 117 hours, where our SOTA agent takes 63 hours.

Model size vs performance. Overall, Gridnet models have more parameters compared to the UAS models. This is because Gridnet predicts the action type and parameter logits for every cell in the map. We did not find a strong correlation between the model's size (in number of trainable parameters) and the performance of the agents. As shown in Figure 4, it is clear that the techniques such as invalid action masking or different neural network architectures are more important to the performance than the sheer number of the model's trainable parameters in our experiments.

Variance w.r.t. Shaped and Sparse Reward. In almost all experiments conducted in this paper, we observe the RL agents are able to optimize against the shaped rewards well, showing little variance across different random seeds; however, this is not the case with respect to the sparse reward (win/loss). We report the sum of shaped rewards and sparse rewards in the episode as *shaped return* and *sparse return* respectively in Figure 7, where we usually see little difference in the shaped return when the sparse (win/loss) return could be drastically different. This is a common drawback with reward shaping: agents sometimes overfit to the shaped rewards instead of sparse rewards.

UAS vs Gridnet. Figure 7 shows a typical result where Gridnet is able to get much higher shaped return, but it receives relatively similar sparse return as UAS. Upon further inspection of the agents actual behaviors, we found the Gridnet agents obtain higher shaped return by 1) producing more barracks, 2) producing more combat units, and 3) harvesting more resources effectively. In fact, Gridnet agents learn to harvest resources using three workers, which is a behavior we haven't observed in any existing bots. We suspect this difference is due to how rewards are attributed in UAS vs Gridnet. UAS attributes rewards to unit actions *individually*, while Gridnet attributes the rewards to the player action *collectively*.

Depending on the implementation, Gridnet agents usually have many more trainable parameters. Also, when the player owns a relatively small amount of units, it is faster to step the environment using UAS because Gridnet has to predict an action for all the cells in the map; however, when the player owns a large number of units, Gridnet's mechanism becomes faster because UAS has to do more simulated steps and thus more inferences.

The Amount of Human Knowledge Injected. In our best trained agents, there are usually three sources of human knowledge injected: 1) the reward function, 2) invalid action masking, and 3) the use of human-designed bots such as CoacAI. In comparison, AlphaStar uses 1) human replays, 2) its related use of Statistics z and Supervised KL divergence [1], and 3) invalid action masking.

VII. Conclusions and Future Work

We present a new efficient library, Gym-μRTS, which allows DRL research to be realized in the complex RTS environment μRTS. Through Gym-μRTS, we conducted ablation studies on techniques such as action composition, invalid action masking, diversified training opponents, and novel neural network architectures, providing insights on their importance to scale agents to play the full game of μRTS. Our agents can be trained on a single CPU+GPU within 2-4 days, which is a reasonable hardware and time budget that is available to many researchers outside of large research labs

For future work, we would like to consider multiple maps and the partial observability setting of μRTS (i.e. fog-of-war). Additionally, we also want to experiment with selfplay, which further reduces human knowledge injected such as the human-designed bots we used in this paper.

References

[1] O. Vinyals, I. Babuschkin, W. M. Czarnecki, M. Mathieu, A. Dudzik, J. Chung, D. H. Choi, R. Powell, T. Ewalds, P. Georgiev, J. Oh, D. Horgan, M. Kroiss, I. Danihelka, A. Huang, L. Sifre, T. Cai, J. P. Agapiou, M. Jaderberg, A. S. Vezhnevets, R. Leblond, T. Pohlen, V. Dalibard, D. Budden, Y. Sulsky, J. Molloy, T. L. Paine, C. Gulcehre, Z. Wang, T. Pfaff, Y. Wu, R. Ring, D. Yogatama, D. Wünsch, K. McKinney, O. Smith, T. Schaul, T. Lillicrap, K. Kavukcuoglu, D. Hassabis, C. Apps, and D. Silver, "Grandmaster level in StarCraft II using multi-agent reinforcement learning." *Nature*, vol. 575, no. 7782, pp. 350–354, oct 2019. [Online]. Available: http://www.nature.com/articles/s41586-019-1724-z

[2] S. Ontanón, G. Synnaeve, A. Uriarte, F. Richoux, D. Churchill, and M. Preuss, "A survey of real-time strategy game ai research and competition in starcraft," *IEEE Transactions on Computational Intelligence and AI in games*, vol. 5, no. 4, pp. 293–311, 2013.

[3] L. Han, J. Xiong, P. Sun, X. Sun, M. Fang, Q. Guo, Q. Chen, T. Shi, H. Yu, and Z. Zhang, "Tstarbot-x: An open-sourced and comprehensive study for efficient league training in starcraft ii full game," *arXiv preprint arXiv:2011.13729*, 2020.

[4] S. Ontanón, "The combinatorial multi-armed bandit problem and its application to real-time strategy games," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 9, no. 1, 2013.

[5] O. Vinyals, T. Ewalds, S. Bartunov, P. Georgiev, A. S. Vezhnevets, M. Yeo, A. Makhzani, H. Küttler, J. Agapiou, J. Schrittwieser *et al.*, "Starcraft ii: A new challenge for reinforcement learning," *arXiv preprint arXiv:1708.04782*, 2017.

[6] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv*, jul 2017. [Online]. Available: https://arxiv.org/abs/1707.06347

[7] P. Dhariwal, C. Hesse, O. Klimov, A. Nichol, M. Plappert, A. Radford, J. Schulman, S. Sidor, Y. Wu, and P. Zhokhov, "Openai baselines," https://github.com/openai/baselines, 2017.

[8] M. Buro, "Real-time strategy games: A new ai research challenge," in *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, ser. IJCAI'03. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2003, p. 1534–1535.

[9] U. Jaidee and H. Muñoz-Avila, "Classq-l: A q-learning algorithm for adversarial real-time strategy games," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 8, no. 1, 2012.

[10] B. Weber and M. Mateas, "Case-based reasoning for build order in real-time strategy games," in *Proceedings of the AAAI Conference on Artificial Intelligence and Interactive Digital Entertainment*, vol. 4, no. 1, 2009.

[11] S. Ontanón, K. Mishra, N. Sugandh, and A. Ram, "On-line case-based planning," *Computational Intelligence*, vol. 26, no. 1, pp. 84–119, 2010.

[12] R.-K. Balla and A. Fern, "Uct for tactical assault planning in real-time strategy games," in *Proceedings of the 21st international joint conference on Artifical intelligence*, 2009, pp. 40–45.

[13] D. Churchill, A. Saffidine, and M. Buro, "Fast heuristic search for rts game combat scenarios," in *Proceedings of the AAAI conference on artificial intelligence and interactive digital entertainment*, vol. 8, no. 1, 2012.

[14] N. Justesen, B. Tillman, J. Togelius, and S. Risi, "Script-and cluster-based uct for starcraft," in *2014 IEEE Conference on Computational Intelligence and Games*. IEEE, 2014, pp. 1–8.

[15] S. Ontanón, "Combinatorial multi-armed bandits for real-time strategy games," *Journal of Artificial Intelligence Research*, vol. 58, pp. 665–702, 2017.

[16] C. Berner, G. Brockman, B. Chan, V. Cheung, P. Debiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, R. Józefowicz, S. Gray, C. Olsson, J. W. Pachocki, M. Petrov, H. P. de Oliveira Pinto, J. Raiman, T. Salimans, J. Schlatter, J. Schneider, S. Sidor, I. Sutskever, J. Tang, F. Wolski, and S. Zhang, "Dota 2 with large scale deep reinforcement learning," *ArXiv*, vol. abs/1912.06680, 2019.

[17] M. Samvelyan, T. Rashid, C. S. De Witt, G. Farquhar, N. Nardelli, T. G. Rudner, C.-M. Hung, P. H. Torr, J. Foerster, and S. Whiteson, "The starcraft multi-agent challenge," *arXiv preprint arXiv:1902.04043*, 2019.

[18] P. Sun, X. Sun, L. Han, J. Xiong, Q. Wang, B. Li, Y. Zheng, J. Liu, Y. Liu, H. Liu *et al.*, "Tstarbots: Defeating the cheating level builtin ai in starcraft ii in the full game," *arXiv preprint arXiv:1809.07193*, 2018.

[19] D. Lee, H. Tang, J. O. Zhang, H. Xu, T. Darrell, and P. Abbeel, "Modular architecture for starcraft ii with deep reinforcement learning," in *Fourteenth Artificial Intelligence and Interactive Digital Entertainment Conference*, 2018.

[20] Y. Tian, Q. Gong, W. Shang, Y. Wu, and C. L. Zitnick, "Elf: An extensive, lightweight and flexible research platform for real-time strategy games," in *Advances in Neural Information Processing Systems*, 2017, pp. 2659–2669.

[21] P.-A. Andersen, M. Goodwin, and O.-C. Granmo, "Deep rts: a game environment for deep reinforcement learning in real-time strategy games," in *2018 IEEE conference on computational intelligence and games (CIG)*. IEEE, 2018, pp. 1–8.

[22] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction (2nd), chapter 13.4, p329*. MIT press, 2018.

[23] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," *arXiv preprint arXiv:1506.02438*, 2015.

[24] L. Espeholt, H. Soyer, R. Munos, K. Simonyan, V. Mnih, T. Ward, Y. Doron, V. Firoiu, T. Harley, I. Dunning *et al.*, "Impala: Scalable distributed deep-rl with importance weighted actor-learner architectures," in *International Conference on Machine Learning*. PMLR, 2018, pp. 1407–1416.

[25] L. Han, P. Sun, Y. Du, J. Xiong, Q. Wang, X. Sun, H. Liu, and T. Zhang, "Grid-wise control for multi-agent reinforcement learning in video game ai," in *International Conference on Machine Learning*. PMLR, 2019, pp. 2576–2585.

[26] L. Engstrom, A. Ilyas, S. Santurkar, D. Tsipras, F. Janoos, L. Rudolph, and A. Madry, "Implementation matters in deep rl: A case study on ppo and trpo," in *International conference on learning representations*, 2019.

[27] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[28] A. Kanervisto, C. Scheller, and V. Hautamäki, "Action space shaping in deep reinforcement learning," in *2020 IEEE Conference on Games (CoG)*. IEEE, 2020, pp. 479–486.

[29] K. He, X. Zhang, S. Ren, and J. Sun, "Identity mappings in deep residual networks," *Lecture Notes in Computer Science*, p. 630–645, 2016. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-46493-0_38

[30] T. King, S. Butcher, and L. Zalewski, *Apocrita - High Performance Computing Cluster for Queen Mary University of London*, Mar. 2017. [Online]. Available: https://doi.org/10.5281/zenodo.438045

[31] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[32] S. Huang and S. Ontañón, "A closer look at invalid action masking in policy gradient algorithms," 2020.