APPENDICES

A DETAILS ON μRTS AS A REINFORCEMENT LEARNING ENVIRONMENT

For all of our experiments in this paper, we use the *gym-microrts* (Huang & Ontañón, 2019) library which provides a reinforcement learning interface of μRTS similar to OpenAI Gym's interface (Brockman et al., 2016). In this section, we hope to provide details on the implementation of *gym-microrts* as well as its limitations.

A.1 OBSERVATION AND ACTION SPACE OF μRTS

Here is a description of *gym-microrts*'s observation and action space:

- **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 2. A feature plane can be thought of as a concatenation of multiple one-hot encoded features. As an example, if there is a worker with hit points equal to 1, not carrying any resources, owner being Player 1, and currently not executing any actions, then the one-hot encoding features will look like the following:

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

  The 27 values of each feature plane for the position in the map of such worker will thus be:

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

- **Action Space.** Given a map of size $h \times w$, the action is an 8-dimensional vector of discrete values as specified in Table 2. The first component of the action vector represents the unit in the map to issue actions to, the second is the action type, and the rest of components represent the different parameters different action types can take.

A.2 LIMITATIONS

In general, there are two limitations associated with *gym-microrts*'s reinforcement learning interface that could hinder the trained agents from competing against existing μRTS bots. As a result, the agent trained in this paper would subject to these limitations.

Frame-skipping Each action in μRTS takes some internal game time, measured in ticks, for the action to be completed. *gym-microrts* sets the time of performing harvest action, return action, and move action to be 10 game ticks. Once an action is issued to a particular unit, the unit would be considered as a "busy" unit and would take additional 9 game ticks for the actions to be finalized. To speed up training, *gym-microrts* by default performs frame skipping of 9 frames such that from the agent's perspective, once it executes the harvest action, return action, or move action given the current observation, those actions would be finished in the next observation.

Limited Unit Action per Tick In general, the game engine of μRTS allows the bots to issue actions to as many units as the bots own at each game tick. This means the action space will have "varied size" depending on the number of units available for control. For simplicity, *gym-microrts* only allows the agent to issue one action to one unit at each tick.

B DETAILS ON THE TRAINING ALGORITHM PROXIMAL POLICY OPTIMIZATION

The DRL algorithm that we use to train the agent is Proximal Policy Optimization (PPO) Schulman et al. (2017), one of the state of the art algorithms available. The hyper-parameters of our experiments can be found in Table 3. There are three important details regarding our PPO implementation that warrants explanation. The first detail concerns how to generate an action in the `MultiDiscrete` action space as defined in the OpenAI Gym environment (Brockman et al., 2016) of gym-microrts (Huang & Ontañón, 2019), the second details involves the implementation of invalid action masking (Vinyals et al., 2017; Berner et al., 2019; Huang & Ontañón, 2020) with PPO, and the third detail is about the various code-level optimizations utilized to augment performance. As pointed out by Engstrom, Ilyas, et al. (Engstrom et al., 2019), such code-level optimizations could be critical to the performance of PPO.

Table 2: The descriptions of observation features and action components.

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 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	$[0, 5]$	resource, base, barrack, worker, light, heavy, ranged
Attack Target Unit	$[0, h \times w - 1]$	the location of unit that will be attacked

B.1 Multi Discrete Action Generation

To perform an action a_t in μRTS, according to Table 2, we have to select a Source Unit, Action Type, and its corresponding action parameters. So in total, there are $hw \times 6 \times 4 \times 4 \times 4 \times 4 \times 6 \times hw = 9216(hw)^2$ number of possible discrete actions (including invalid ones), which grows exponentially as we increase the map size. If we apply the PPO directly to this discrete action space, it would be computationally expensive to generate the distribution for $9216(hw)^2$ possible actions. To simplify this combinatorial action space, `openai/baselines` Dhariwal et al. (2017) library proposes an idea to consider this discrete action to be composed from some smaller *independent* discrete actions. Namely, a_t is composed of smaller actions

$$a_t^{\text{Source Unit}}, a_t^{\text{Action Type}}, a_t^{\text{Move Parameter}}, a_t^{\text{Harvest Parameter}},$$
$$a_t^{\text{Return Parameter}}, a_t^{\text{Produce Direction Parameter}}, a_t^{\text{Produce Type Parameter}}, a_t^{\text{Attack Target Unit}}$$

And the policy gradient is updated in the following way (without considering the PPO's clipping for simplicity)

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

$D = \{\text{Source Unit}, \text{Action Type}, \text{Move Parameter}, \text{Harvest Parameter}, \text{Return Parameter}, \text{Produce Direction Parameter}, \text{Produce Type Parameter}, \text{Attack Target Unit}, \}$

Implementation wise, for each Action Component of range $[0, x-1]$, the logits of the corresponding shape x is generated, which we call Action Component logits, and each a_t^d is sampled from this Action Component logits. Because of this idea, the algorithm now only has to generate $hw + 6 + 4 + 4 + 4 + 4 + 6 + hw = 2hw + 36$ number of logits, which is significantly less than $9216(hw)^2$. To the best of our knowledge, this approach of handling large multi discrete action space is only mentioned by Kanervisto et, al Kanervisto et al. (2020).

B.2 Invalid Action Masking

Invalid action masking is a technique that "masks out" invalid actions and then just sample from those actions that are valid (Vinyals et al., 2017; Berner et al., 2019). Huang & Ontañón (2020) show invalid action masking is crucial in helping the agents explore in μRTS

Table 3: The list of hyperparameters and their values.

Parameter Names	Parameter Values
N_{total} Total Time Steps	10,000,000
N_{mb} Number of Mini-batches	4
N_{envs} Number of Environments	8
N_{steps} Number of Steps per Environment	128
γ (Discount Factor)	0.99
λ (for GAE)	0.95
ε (PPO's Clipping Coefficient)	0.1
η (Entropy Regularization Coefficient)	0.01
ω (Gradient Norm Threshold)	0.5
K (Number of PPO Update Iteration Per Epoch)	4
α Learning Rate	0.00025 Linearly Decreased to 0 over the Total Time Steps
$c1$ (Value Function Coefficient, see Equation 3)	0.5
$c2$ (Entropy Coefficient, see Equation 3)	0.01
$N_{updates}$ (Total Number of Updates)	$N_{total}/(N_{mb}N_{envs})$

B.3 Code-level Optimizations

Here is a list of code-level optimizations utilized in this experiments. For each of these optimizations, we include a footnote directing the readers to the files in the *openai/baselines* (Dhariwal et al., 2017) that implements these optimization.

1. **Normalization of Advantages**[5]**:** After calculating the advantages based on GAE, the advantages vector is normalized by subtracting its mean and divided by its standard deviation.
2. **Normalization of Observation**[6]**:** The observation is pre-processed before feeding to the PPO agent. The raw observation was normalized by subtracting its running mean and divided by its variance; then the raw observation is clipped to a range, usually $[-10, 10]$.
3. **Rewards Scaling**[7]**:** Similarly, the reward is pre-processed by dividing the running variance of the discounted the returns, following by clipping it to a range, usually $[-10, 10]$.
4. **Value Function Loss Clipping**[8]**:** The PPO implementation of *openai/baselines* clips the value function loss in a manner that is similar to the PPO's clipped surrogate objective:

$$V_{loss} = \max\left[\left(V_{\theta_t} - V_{targ}\right)^2, \left(V_{\theta_{t-1}} + \text{clip}\left(V_{\theta_t} - V_{\theta_{t-1}}, -\varepsilon, \varepsilon\right)\right)^2\right]$$

   where V_{targ} is calculated by adding $V_{\theta_{t-1}}$ and the A calculated by General Advantage EstimationSchulman et al. (2015).
5. **Adam Learning Rate Annealing**[9]**:** The Adam Kingma & Ba (2014) optimizer's learning rate is set to decay as the number of timesteps agent trained increase.
6. **Mini-batch updates**[10]**:** The PPO implementation of the *openai/baselines* also uses mini-batches to compute the gradient and update the policy instead of the whole batch data such as in *open/spinningup*. The mini-batch sampling scheme, however, still makes sure that

[5] https://github.com/openai/baselines/blob/ea25b9e8b234e6ee1bca43083f8f3cf974143998/baselines/ppo2/model.py#L139

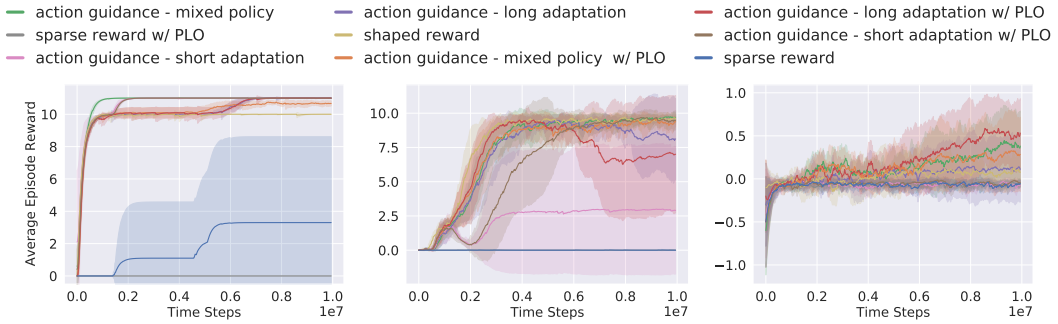
[6] https://github.com/openai/baselines/blob/ea25b9e8b234e6ee1bca43083f8f3cf974143998/baselines/common/vec_env/vec_normalize.py#L4

[7] https://github.com/openai/baselines/blob/ea25b9e8b234e6ee1bca43083f8f3cf974143998/baselines/common/vec_env/vec_normalize.py#L4

[8] https://github.com/openai/baselines/blob/ea25b9e8b234e6ee1bca43083f8f3cf974143998/baselines/ppo2/model.py#L68-L75

[9] https://github.com/openai/baselines/blob/ea25b9e8b234e6ee1bca43083f8f3cf974143998/baselines/ppo2/ppo2.py#L135

[10] https://github.com/openai/baselines/blob/ea25b9e8b234e6ee1bca43083f8f3cf974143998/baselines/ppo2/ppo2.py#L160-L162



(a) LearnToAttack (b) ProduceCombatUnit (c) DefeatRandomEnemy

Figure 4: The learning curves of agents. The x-axis shows the number of time steps and y-axis shows the average sparse return gathered.

every transition is sampled only once, and that the all the transitions sampled are actually for the network update.

7. **Global Gradient Clipping**[11]**:** For each update iteration in an epoch, the gradients of the policy and value network are clipped so that the "global ℓ_2 norm" (i.e. the norm of the concatenated gradients of all parameters) does not exceed 0.5.
8. **Orthogonal Initialization of weights**[12]**:** The weights and biases of fully connected layers use with orthogonal initialization scheme with different scaling. For our experiments, however, we always use the scaling of 1 for historical reasons.

B.4 Neural Network Architecture

The input to the neural network is a tensor of shape $(10, 10, 27)$. The first hidden layer convolves 16 3×3 filters with stride 2 with the input tensor followed by a rectifier nonlinearityNair & Hinton (2010). The second hidden layer similarly convolves 32 2×2 filters followed by a rectifier nonlinearity. The final hidden layer is a fully connected linear layer consisting of 128 rectifier units. The policy's output layer is a fully connected linear layer with $2hw + 36 = 236$ number of output and the value output layer is a fully connected linear layer with a single output.

C Learning Curves

The learning curves of all experiments are shown in Figure 4.

Algorithm 1 PPO with Action Guidance

Let S be the set of policies and reward functions $= \{(\pi_{\theta_{\mathcal{M}}}, R_{\mathcal{M}}), (\pi_{\theta_{\mathcal{A}_1}}, R_{\mathcal{A}_1})\}$
for update = $1, 2, ..., N_{updates}$ **do**
  for step = $1, 2, ..., N_{steps}$ **do**
    With probability ϵ select a policy $(\pi_{\theta_b}, R) \in S$ ▷ Action Guidance
    Perform rollouts under current policy π_{θ_b} and store rewards based on $R_{\mathcal{M}}, R_{\mathcal{A}_1}$
  for policy and their reward function $(\pi_\theta, R) \in S$ **do**
    if PLO is used and the sum of rewards according to R in rollouts is 0 **then**
      continue
    for epoch = $1, 2, ..., K$ **do**
      Optimize $L^{CLIP}(\theta) - c_1 L^{VF}(\theta) + c_2 S[\pi_{\theta_b}]$ ▷ Policy Epochs

[11] https://github.com/openai/baselines/blob/ea25b9e8b234e6ee1bca43083f8f3cf974143998/baselines/ppo2/model.py#L107

[12] https://github.com/openai/baselines/blob/ea25b9e8b234e6ee1bca43083f8f3cf974143998/baselines/a2c/utils.py#L58