Grounded Reinforcement Learning: Learning to Win the Game under Human Commands

Shusheng Xu¹, Huaijie Wang¹ and Yi Wu¹,²
¹ IIIS, Tsinghua University, Beijing, China
² Shanghai Qi Zhi Institute, Shanghai, China
{xuss20, wanghuai19}@mails.tsinghua.edu.cn
jxwuyi@gmail.com

Abstract

We consider the problem of building a reinforcement learning (RL) agent that can both accomplish non-trivial tasks, like winning a real-time strategy game, and strictly follow high-level language commands from humans, like “attack”, even if a command is sub-optimal. We call this novel yet important problem, Grounded Reinforcement Learning (GRL). Compared with other language grounding tasks, GRL is particularly non-trivial and cannot be simply solved by pure RL or behavior cloning (BC). From the RL perspective, it is extremely challenging to derive a precise reward function for human preferences since the commands are abstract and the valid behaviors are highly complicated and multi-modal. From the BC perspective, it is impossible to obtain perfect demonstrations since human strategies in complex games are typically sub-optimal. We tackle GRL via a simple, tractable, and practical constrained RL objective and develop an iterative RL algorithm, REinforced demonstration Distillation (RED), to obtain a strong GRL policy. We evaluate the policies derived by RED, BC and pure RL methods on a simplified real-time strategy game, MiniRTS. Experiment results and human studies show that the RED policy is able to consistently follow human commands and, at the same time, achieve a higher win rate than the baselines. We release our code and present more examples at https://sites.google.com/view/grounded-rl.

1 Introduction

Building assistive agents that can help humans accomplish complex tasks remains a long-standing challenge in artificial intelligence. Natural language, as the most generic protocol for humans to exchange and share knowledge, becomes a general and important interface for human-AI interaction. Decades of research efforts have been continuously made to develop AI systems that can ground agent behaviors to natural language commands [69, 64, 46].

Recently, as deep reinforcement learning (RL) techniques have been widely used to develop interactive agents to solve a variety of challenging problems, it becomes a new trend to cast language grounding as an RL problem to train agents that can automatically ground language concepts to visual objects or behaviors in an interactive fashion [29, 13, 15]. In particular, an RL agent is typically presented with a language command describing the goal of the RL task and will receive a success reward when the desired goal state under the command is achieved. A well-trained RL agent can often exhibit strong generalization capabilities to novel commands. Despite the simplicity and effectiveness, such an RL formulation requires a language generator to repeatedly output random commands and an explicit reward function to determine whether the command is accomplished. Hence, most existing RL works focus on simple navigation problems with template-based compositional languages over objects and attributes, e.g., “go to the red box next to the blue wall” [56, 68, 34, 15].
Figure 1: The grounded reinforcement learning (GRL) problem on the MiniRTS environment. (A) MiniRTS is a real-time strategy game where the player in blue needs to control its units to kill the enemy units in red. (B) A conventional RL agent. (C) MiniRTS provides a dataset of human demonstrations in the form of paired abstract language commands (e.g., “attack”) and control action sequences. Human actions are often sub-optimal. (D) GRL aims to learn a command-conditioned agent such that it plays a winning strategy stronger than the human executor. (E) A GRL agent should strictly follow the human command even if it is sub-optimal.

To train agents that can interpret general human languages, another research direction is to leverage paired behavior-command data collected from human demonstrators. Representative applications include robotic manipulation, where the robot needs to map language commands to predefined motion skills [60, 35, 61, 65], and vision-and-language navigation, where an agent learns from human demonstrations to navigate towards a desired location in a 3D environment [51, 13, 29, 71]. Since an accurate command-following reward is no longer accessible, most works apply behavior cloning (BC) to directly imitate human behaviors or adopt inverse reinforcement learning (IRL) to learn a language-conditioned reward function [19, 52, 80, 23, 79, 17]. Although the use of human data enables natural commands, pure imitation-based methods require massive demonstrations, which can be expensive to collect. More importantly, both BC and IRL algorithms assume that the demonstrations are optimal [19, 52, 80, 23, 79, 17], in the sense that (i) each behavior demonstration is optimal under its paired language command, and (ii) the language commands are optimal for the underlying task. Such an assumption is reasonable in restricted domains like manipulation or navigation, where the optimal behavior is simply the shortest trajectory to the goal while human commands typically provide strong guidance towards task completion. However, in more complex problems like playing real-time strategy games, it is often the case that human players can be highly biased or sub-optimal [10].

Let’s consider a concrete example in a simplified real-time strategy game, MiniRTS [33]. Fig. [1C] shows a scenario where a human commander gives an abstract command “attack” to a human executor to attack enemy units. However, the action taken by the human executor is sub-optimal: a unit with full HP is under attack while a clear better strategy is to attack another dying unit. Moreover, the human commander can be sub-optimal as well. As shown in Fig. [1E], very few enemy units are remaining, so the optimal command should be simply an “attack” for the win. However, the commander sends a
“retreat” command. This could be possibly due to the partially observed game state or human biases, but an obedient executor should still faithfully control the units to stop attacking and retreat.

We study this novel RL challenge, i.e., learning an strong agent capable of not only achieving a high win rate in a real-time strategy game, e.g., MiniRTS, but also strictly following high-level language commands, even if a sub-optimal command is given. We call this problem, Grounded Reinforcement Learning (GRL), which assumes an interactive RL environment with a dataset of (sub-optimal) human demonstrations reflecting proper human behaviors under language commands. In this setting, the rewards are not associated with the language commands, and it is non-trivial to verify whether a high-level language command is completed or not. So the policy can only learn about how humans follow commands from the demonstrations. We remark that GRL is different from standard language grounding problems since the primary mission is not towards better language understanding via interactions [11]. Instead, GRL focuses on the opposite direction, i.e., learning strong winning strategies while taking human commands as high-level behavior regulations, which is more related to the concept of developing human-compatible AI [57].

To tackle the GRL problem, we propose a simple yet effective constrained RL objective as a tractable approximation and developed an iterative RL algorithm, REinforced demonstration Distillation (RED), to derive a strong language-conditioned policy. RED adopts unconditioned RL training to encourage the policy to explore stronger winning strategies and periodically applies self-distillation and BC over demonstrations to ensure that the policy behavior is consistent with human preferences. We evaluate the performance of RED as well as baseline methods including BC-based methods and RL variants on the MiniRTS environment. Simulation results show that RED policy achieves strong command-following capabilities and a higher win rate than baselines under various types of test-time commands. We also conduct human evaluation by inviting 30 volunteers to play with policies trained by different methods. RED policy appears to be the most obedient under human votes and leads to a much higher collaborative win rate with human commanders.

2 Related Work

Language grounding. The idea of grounding languages to behaviors or concepts can be traced back to 1970s [69], and most early works assume simple domain-specific languages and a pre-defined set of skills [45] [77] [37] [14] [47] [5]. Thanks to the recent advances of deep RL, it becomes feasible to ground languages to visual concepts and non-trivial behaviors in a simulated world. Many 3D environments with language-described goals have been developed over different domains, including maze exploration [15], visual navigation [13] [29] [71] and robot control [35] [61] [65], allowing end-to-end concept learning and grounding. RL training requires a goal generator and a reward function. Hence, these testbeds only adopt template-based languages over objects (e.g., box or cube) and attributes (e.g., spatial relation or color) and assume an oracle that can verify whether a state is consistent with the language description. We focus on following high-level natural commands.

Regarding natural commands, recent visual navigation benchmarks [4] [49] [61] [62] started to provide large-scale human demonstrations with the goal or pathway specified by natural languages. Behavior cloning (BC) is feasible since every human description is paired with a successful trajectory [19] [52] [80]. RL fine-tuning can be also applied, since a precise success measurement (i.e., distance to the goal position) w.r.t. each language instruction is known [22] [67]. Some works adopt inverse RL (IRL) to learn a language-conditioned reward function [23] [79] thanks to the fact that the demonstrations are actually optimal. We consider a more challenging game MiniRTS [33] with abstract language commands and a complex behavior space. BC and IRL methods work poorly in MiniRTS since both language commands and game trajectories from human annotators are highly sub-optimal.

There are also successful attempts to leverage powerful pretrained language models to map general languages to the goal space [36] [44] or a predefined set of skills [31] [60] [2] so that massive paired demonstrations can be no longer necessary. We primarily consider the problem of policy learning, so the use of pretrained model is orthogonal to our current focus but remains an exciting future direction.

There are also parallel works on language grounding in other domains, such as text games [38], answer questioning [4] and communication [39], where languages serve as state descriptor or action space rather than commands to follow. Our work is also related to ad-hoc team play [69] [12] since the policy cooperates with arbitrary commands. The difference is that the commander is not assumed to be optimal under the game reward. The policy must be assistive [28] [70] or even obedient [50].
Constrained reinforcement learning. We cast grounded RL as a constrained RL (CRL) formulation for tractability. CRL algorithms often leverage the Lagrangian multiplier \([55]\) or projected gradients \([1]\) \([2]\) for policy improvement assuming simple (closed-form) constraints over states. In our setting, constraints can be only implicitly learned from human demonstrations. We simply adopt the behavior cloning objective, i.e., negative log likelihood, as the constraint satisfying metric, which is related to RL with demonstrations (RLwD) \([32]\) \([25]\). However, due to limited data and a significant distribution shift between demonstrations and on-policy trajectories, RLwD methods can perform poorly. In addition, \([73]\) iteratively optimizes the RL objective, projects the policy on a region around a reference policy, and enforces the policy satisfies the cost constraint. \([27]\) updates the policies on the mixture of the expert replay buffer and the rollout trajectories. The most related work to our RED algorithm is Supervised Seeded Iterated Learning (SSIL) \([42]\), which is designed for learning grounded communication and adopts a conceptually similar iterative framework by alternating between RL and self-distillation. SSIL suggests to jointly optimize RL and BC objectives assuming human communications are perfect demonstrations. By contrast, we find that an additional BC loss can be harmful to RL training due to the distribution shift issue. Similar empirical phenomena have been also reported in multi-task RL literature \([75]\) \([63]\) \([74]\).

3 Preliminary

3.1 The MiniRTS Environment and Dataset

MiniRTS \([33]\) is a grid-world RL environment (Fig. 1) that distills the key features of complex real-time strategy games. It has two parties, a player (blue) controlled by a human/policy against a built-in script AI (red). The player controls units to collect resources, do construction and kill all the enemy units or destroy the enemy base to win a game. A total of 7 unit types form a rock-paper-scissors attacking dynamics. Learning a strong policy is challenging due to partially observed states, complicated micro-management over dozens of units, and extremely diverse possible strategies.

In addition to an RL environment, MiniRTS provides a command-following dataset, which is collected by two humans playing collaboratively against the script opponent. One is the commander, who gives high-level strategic language commands like “attack”, “retreat”, or “build an archer”. The other one is the executor, who controls the units to follow the language command. We remark that human data were collected against much weaker scripts than the default game opponent. Even in such an easier mode, the collected game episodes are highly sub-optimal with a win rate of merely 47.2%. The dataset contains a total of 63,285 commands with an average of 13.7 per game while the average game horizon is 151.7. So, each single command may take a long horizon to accomplish. For notation simplicity in this paper, we assume an independent command is given to the executor at each time step, although a command should be repeated for a few steps. Full details are in appendix.

3.2 Notation

Reinforcement learning. We formulate the MiniRTS environment as a Partially Observable Markov Decision Process (POMDP), which is represented by a tuple \((S, A, O, O, r, p)\). \(S\) is the state space. \(A\) is the action space. \(O\) is the observation space. \(r: S \times A \times S \rightarrow \mathbb{R}\) is the reward function. \(p: S \times A \times S \rightarrow [0, 1]\) is the transition probability with \(p(s'|s, a)\) denoting the probability from state \(s\) to state \(s'\) after taking action \(a\). At each time step, the agent (i.e., the game player) observes \(o_t = O(s_t)\), produces an action \(a_t\) according to its policy \(\pi_\theta\) parameterized by \(\theta\), and receives a reward \(r_t = r(s_t, a_t)\). The optimal policy should achieve high win rates in the game, namely \(\theta^* = \arg\max_{\theta} \mathbb{E}_{o_t, s_t} \left[ \sum_t r(s_t, a_t) \right] \). Note that we omit the discount factor \(\gamma\) for conciseness only.

Command-following policy. We represent the policy by \(\pi_\theta(a_t|o_t, c_t)\), which conditions on an observation \(o_t \in O\) and a language command \(c_t \in C\) at each time step \(t\). \(c_t\) can be generated from arbitrary distributions. \(C\) denotes the space of possible natural language commands. An LSTM is used to encode a command \(c\) to an embedding. We use a special command “NA” to denote “no command”, which corresponds to a zero embedding. When setting every \(c_t\) to be NA, the command-following policy \(\pi_\theta\) degenerates to a conventional policy in classical RL without language commands.

---

\(^1\)Our numbers are slightly different from \([33]\) because the original data-processing script is not released.
Human demonstrations. We denote human demonstrations as a dataset of trajectories, i.e., \( D = \{ \tau_1, \tau_2, \ldots \} \). Each trajectory \( \tau_i \), denoted by \( \tau_i = (s_{i0}, a_{i0}, o_{i0}, s_{i1}, a_{i1}, o_{i1}, \ldots) \), consists of transitions from a single game played by a human executor and the paired command sequence from a human commander. Human behaviors can be sub-optimal: i.e., given the observation \( o_{i0} = O(s_{i0}) \) and the command \( c_{i1} \), the chosen action \( a_{i1} \) can result in low rewards. Likewise, a command \( c_{i2} \) can be arbitrary w.r.t. human preferences. Finally, since human commanders are always asked to generate language commands, \( D \) does not contain any \( \text{NA} \) command, i.e., \( c_{i2} \neq \text{NA} \).

4 Method

4.1 Grounded Reinforcement Learning: Problem Formulation

The mission of GRL is to learn a policy \( \pi_\theta \) such that \( \pi_\theta \) can achieve high rewards in the environment and follow any possible command \( c \in C \) by achieving consistent behaviors with human demonstrations \( D \). We formulate the GRL problem as the following RL objective \( J_G(\theta) \):

\[
J_G(\theta) = \mathbb{E}_{a_t \sim \pi_\theta(o_t, c_t)} \left[ \sum_t r(s_t, a_t) \right] \quad \text{subject to } K(\pi_\theta, D) \leq \delta, \tag{1}
\]

where \( K \) denotes a distance metric between the policy behaviors from \( \pi_\theta \) and the human data \( D \).

Remark #1: The commands can be arbitrary. \( \pi_\theta \) needs to follow any possible command \( c \in C \) while still attains the highest rewards under the behavior constraint \( K(\pi_\theta, D) \leq \delta \).

Remark #2: Since the human actions in \( D \) can be sub-optimal w.r.t. the paired commands, it can be problematic to directly perform behavior cloning (BC) over \( D \). Therefore, we utilize a threshold \( \delta \) so that the command-following policy \( \pi_\theta \) can possibly deviate from the sub-optimal behaviors in \( D \).

Note that Eq. (1) is generally intractable and is presented only for the purpose of problem definition.

4.2 Constrained RL as a Tractable Approximation for GRL

There are two issues when directly optimizing Eq. (1). First, the command \( c_t \) can be arbitrary. Second, the choice of behavior distance metric \( K(\pi_\theta, D) \) should be specified.

Training commands. An obvious choice is to sample a random command \( c_t \) from \( C \) at each time step \( t \), which is perfectly aligned with Eq. (1). However, inconsistent commands from random sampling are typically harmful for a win, making the RL agent tend to ignore the commands. An alternative is a “human proxy” by learning a command model \( h_\phi(c_t|s_t) \) over human data \( D \), leading to the objective \( \mathbb{E}_{c_t \sim h_\phi(s_t)} \left[ \sum_t r(a_t, s_t) \right] \). Although this is reasonable, we need to be aware that human commands are sub-optimal and biased, which may further limit the chance for the policy \( \pi_\theta \) to discover strong strategies during RL training. Thus, we propose an extreme version: i.e., directly performing unconditioned RL training by setting every command \( c_t \) to be \( \text{NA} \). This allows the policy to freely explore the strategy space during RL training for the best winning strategies.

Command-following metric. The simplest distance metric \( K \) is the behavior cloning (BC) loss, i.e., negative log-likelihood (NLL) of \( \pi_\theta \) for \( (s_t, a_t, c_t) \in D \). Specifically, we can define BC objective \( L(\theta) = \mathbb{E}_D [-\log \pi_\theta(a_t|o_t, c_t)] \) and constrain \( L(\theta) \leq \delta \) to ensure the policy behavior does not deviate too much from the demonstrations too much. There can be alternatives, such as KL-divergence or a learned metric. Regarding KL-divergence, since we are measuring the distance between samples \( D \) and a distribution \( \pi_\theta \), \( KL(D||\pi_\theta) \) is equivalent to \( L(\theta) \) while \( KL(\pi_\theta||D) \) requires an behavior proxy over \( D \), which may involve additional biases. Regarding a learned metric, we can apply inverse RL to derive a reward function on whether the command is accomplished \([23, 6, 7]\). However, we empirically notice that a learning-based metric works poorly on MiniRTS. We hypothesis that this is because the demonstrations are sub-optimal and possibly limited in size for such a complex game.

Tractable objective. To sum up, we propose the following constrained RL objective \( J_C(\theta) \) as a tractable objective for the GRL problem, i.e.,

\[
J_C(\theta) = \mathbb{E}_{a_t \sim \pi_\theta(o_t, \text{NA})} \left[ \sum_t r(s_t, a_t) \right] \quad \text{subject to } L(\theta; D) = \mathbb{E}_D [-\log \pi_\theta(a_t|o_t, c_t)] \leq \delta. \tag{2}
\]
We highlight some critical implementation factors here. More details are in appendix.

We present an intuitive justification of the self-distillation process from the perspective of sample distributions into a single neural policy \( \pi_{\theta} \). We introduce an additional 0/1 selection action on each unit denoting whether this unit is controllable or not. This phenomenon can be also justified by some recent theoretical findings [48, 16].

## 4.3 Reinforced Demonstration Distillation

A straightforward way to solve the constrained optimization problem in Eq. (2) is to adopt the Lagrangian multiplier, which leads to a joint optimization problem, i.e.,

\[
J_{C}^{\text{soft}}(\theta) = J_{C}(\theta) - \beta L(\theta; D).
\]

Eq. (3) treats the BC objective \( L(\theta; D) \) as an auxiliary loss of the standard RL objective \( J_{C}(\theta) \). The constraint in Eq. (2) can be satisfied by tuning the coefficient \( \beta \). However, due to the distribution shift issue between on-policy samples and demonstrations, the supervised learning loss may interfere with the policy gradient and further makes the training process sensitive and unstable [21, 40].

**Iterative solution.** Inspired by the recent advances in self-imitation [53] and self-distillation [78, 43, 42], we adopt an iterative framework to decouple the RL loss and the BC loss in Eq. (3). The idea is simple: for RL training, we solely estimate the unconstrained policy gradient over \( J_{C} \) without considering the BC loss; after policy improvement, we distill the demonstrations into the policy by running pure behavior cloning over both human demonstrations and self-generated winning trajectories. By repeatedly alternating between pure RL and pure BC, we are able to achieve a stable learning process. We call this method, **REinforced demonstration Distillation** (RED).

Specifically, let \( \theta_{k} \) denote the parameters at iteration \( k \), then we have the following update rule.

\[
\begin{align*}
\text{RL phase:} & \quad \theta_{k}^{\text{RL}} \leftarrow \theta_{k-1} + \alpha \nabla J_{C}(\theta_{k-1}) ; \\
\text{BC phase:} & \quad \theta_{k} \leftarrow \theta_{k}^{\text{RL}} - \alpha \nabla L(\theta_{k}^{\text{RL}} ; D \cup D_{k}) , \quad D_{k} = \{ \tau | \text{is win}(\tau) , \tau \sim \pi_{\theta_{k}}(\cdot ; \text{NA}) \}.
\end{align*}
\]

Here \( \alpha \) denotes the learning rate. By repeating the update rules for \( N \) iterations, we will derive our final parameter \( \theta^{*} \). The remaining issue is to ensure the constraint is satisfied, i.e., \( L(\theta ; D) \leq \delta \). We empirically notice that this can be accomplished by controlling the ratio between the size of on-policy samples \( \| D_{k} \| \) and the size of demonstrations \( \| D \| \).

We present an intuitive justification of the self-distillation process from the perspective of sample distributions [9, 8]. We adopt RL for the highest rewards, so the unconditioned policy may frequently visit those states approaching a win. As a result, for games played by strong humans with aggressive commands, the paired game states will be more aligned with the RL state distribution while the states produced by biased or defensive human commanders will be more apart. As we are distilling two drift distributions into a single neural policy \( \pi_{\theta} \), it will be more likely to get actions over those overlapping states improved within a bounded BC loss during optimization (see evidences in Sec. 5.3).

## 4.4 Implementations

We highlight some critical implementation factors here. More details are in appendix.

**Policy architecture.** We adopt a similar policy architecture to the provided executor model in MiniRTS [43] but with a modified action space and an additional value head as the critic. The original model outputs an action for every controllable unit, which makes RL training extremely slow. We introduce an additional 0/1 selection action on each unit denoting whether this unit should act or not, which substantially reduces the action dimension.

**Policy learning.** We use PPO [59] for RL training, which involves multiple mini-batch policy gradient steps. Since RL from scratch fails completely in practice, we fine-tune the BC policy and pretrain the value head with the policy parameters frozen. We also empirically find learning rate

\[
\text{REinforced demonstration Distillation (RED).}
\]
warmup particularly helpful for stabilizing training with a pretrained model: we start with learning rate 0 and then linearly increase it to the desired value $\alpha$.

Creating $D_k$. Note that in the early stage of training phase, the winning rate can be low. In order to obtain sufficient data for $D_k$, we implement a queue to store all past winning trajectories and use bootstrapped sampling if the queue is still not full. For the ratio between $\|D_k\|$ and $\|D\|$, we ensure the BC loss (i.e., negative log likelihood) of $\pi^*_\theta$ over the validation set is at most 10% more than the pure BC policy (i.e., no RL training). We can also run sub-sampling over $D$ to reduce the required sample size. Empirically, we find that simply setting $\|D_k\| = \|D\|$ leads to strong performances.

5 Experiment

All the simulation win rates are based on 1200 test games and repeated over 3 random seeds. More details and additional results including emergent behaviors are deferred to appendix.

Baselines. We consider the following methods in addition to our RED algorithm as baselines.
1. “RL” first pretrains the policy over demonstrations and then performs pure unconditioned RL training (i.e., conditioning on “NA” command) without any command-following constraint.
2. “Switch” is the “switching” policy, which consists of two policies, a pure BC policy from demonstrations and a pure RL policy without commands. When a language command is given, it runs the BC policy and uses the RL policy otherwise.
3. “Joint” optimizes the soft objective $J_{\text{soft}}^{C}$ in Eq. (3). $\beta$ is selected via a grid-search process.
4. “IRL” learns a reward function from demonstrations [24] and combines the game reward and the learned language reward to train a conditioned policy.

Evaluation. To evaluate a strong GRL policy, we need to answer the following questions.
1. Does the policy learn a strong winning strategy? The win rate can be a direct metric for this question. A RED policy should largely improves a pure BC policy with a high win rate.
2. Does the policy follow the commands well? A good GRL policy should keep its winning strategy while follow arbitrary commands. A precise evaluation of this question is non-trivial since the constraint satisfying condition, i.e., BC loss, can be a very misleading signal. We adopt an approximate measurement via win rates in Sec. 5.1 and then conduct human evaluation in Sec. 5.4.

We also conduct study on OOD commands and algorithmic hyper-parameters in Sec. 5.2 and Sec. 5.3.

5.1 Main Results

Q#1: is the RED policy strong? We measure the win rates of different policies under two test-time command strategies, i.e., a pure NA strategy, which always gives an NA, and an Oracle strategy, which gives “optimal” commands. For the NA strategy, the GRL problem degenerates to the conventional RL setting, so a strong GRL policy should at least achieve comparable performances to the RL policy and substantially outperforms the BC (i.e., Switch) policy. For the Oracle strategy, it was scripted to give carefully tuned commands based on the ground-truth game state to instruct the policy to build dominating units according to the rock-paper-scissors dynamics. Hence, the GRL problem is converted to a fully cooperative game, and following Oracle commands becomes the optimal executor strategy. A strong GRL policy should achieve the highest possible win rate. Meanwhile, the policy with the highest win rate must produce a good command-following behavior. We also report the win rates of different methods with random commands and a “human proxy” commander, which is learned from human data. Since Random and Human Proxy are sub-optimal, a good GRL policy should have a substantially lower win rate than the NA case but still outperforms BC (Switch) policy.

The results are summarized in Tab. 1. The win rate by RED is the highest under oracle commands and is comparable to RL and Switch without commands (NA). Both RED and Joint policies substantially outperform Switch (BC) policy with sub-optimal commands (Random and Human Proxy), suggesting the conditioned behaviors are improved. Neither RL nor IRL is able to make the policy obedient – even under random commands, their win rates remain comparable to the NA case. We also note that commands from human proxy generally yield an even lower win rate than random commands. We remark that learning a good human commander model is extremely hard – not only due to the limited human command data in the dataset but also because most of the commands are highly sub-optimal.

We empirically notice that a lower validation NLL does not necessarily yield a higher prediction accuracy.
We also try to evaluate the command-following ability directly. At the beginning of a game, we adopt an LSTM encoder to encode arbitrary natural language commands into fixed-length sentence embeddings. During the training process, there are a total of 38,558

<table>
<thead>
<tr>
<th>Test Commands</th>
<th>Command-Ignorant</th>
<th>Command-Following</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracle (%)</td>
<td>89.3 ± 0.7</td>
<td>57.3 ± 3.9</td>
</tr>
<tr>
<td>NA (%)</td>
<td>57.5 ± 1.1</td>
<td>45.6 ± 2.9</td>
</tr>
<tr>
<td>Random (%)</td>
<td>53.3 ± 0.4</td>
<td>47.9 ± 2.8</td>
</tr>
<tr>
<td>Human Proxy (%)</td>
<td>43.6 ± 0.6</td>
<td>48.9 ± 2.9</td>
</tr>
</tbody>
</table>

Table 1: Win rates of different policies under various test-time commands. For Oracle and NA, a better policy should have a higher reward. For sub-optimal commands, i.e., Random and Human Proxy, an obedient policy should have a much lower win rate than the case of no commands (i.e., NA). For Adversarial Oracle, the win rates of command-following policies drop to almost 0.

Q#2: is the RED policy grounded? Note that Tab. [1] has already provided evidences on the command-following performance of the RED policy, since the highest win rates can be only achieved by following the Oracle commands. We also implement an "Adversarial Oracle" commander, which always chooses the worst dominating units to build. We can observe that the win rates of Switch, Joint and RED drop to almost 0, which suggests that they follow commands even when the commands are divergent from the winning strategies.

Here we present an additional criterion: intuitively, as a command-following policy, its win rate should decrease if ‘worse’ commands are fed; otherwise, if the win rate does not decay, the policy must not follow the commands well. Hence, we interpolate between the NA strategy and the full Random strategy and evaluate the win rate of different policies with different ratios of random commands. The results are shown in Fig. 2. We can observe that the RL policy and IRL policy are clearly not following the commands. Among the remaining command-following policies, RED policy generally produces a higher win rate than Switch and Joint, which provides evidences that our RED algorithm leads to improved strategies under the command-following requirement.

We also try to evaluate the command-following ability directly. At the beginning of a game, we provide a sequence of commands to guide the policies to build a specific army unit, and evaluate the success rates in 30 steps. The results are listed in Tab. 2. We can observe that the Switch, Joint, and RED policies achieve significantly higher success rates than IRL. This suggests that Switch, Joint, and RED faithfully follow the commands, while IRL tends to be command-ignorant.

<table>
<thead>
<tr>
<th>Unit Type</th>
<th>Command-Ignorant</th>
<th>Command-Following</th>
</tr>
</thead>
<tbody>
<tr>
<td>Spearman (%)</td>
<td>98.9 ± 0.2</td>
<td>82.6 ± 2.7</td>
</tr>
<tr>
<td>Swordman (%)</td>
<td>95.2 ± 0.6</td>
<td>70.4 ± 5.2</td>
</tr>
<tr>
<td>Cavalry (%)</td>
<td>98.4 ± 0.6</td>
<td>80.4 ± 7.1</td>
</tr>
<tr>
<td>Dragon (%)</td>
<td>81.8 ± 1.6</td>
<td>1.8 ± 0.2</td>
</tr>
<tr>
<td>Archer (%)</td>
<td>85.7 ± 0.6</td>
<td>2.3 ± 0.3</td>
</tr>
<tr>
<td>Catapult (%)</td>
<td>85.3 ± 0.8</td>
<td>2.2 ± 0.9</td>
</tr>
</tbody>
</table>

Table 2: The success rates of building army units given the corresponding commands.

It is also worth noting that RL still achieves reasonable success rates. This is probably caused by the fact that RL policy is initialized with the BC policy and keeps a weak command-following ability. Building a single unit at the beginning of a game is relatively easy. However, following commands in a complex situation later in the game may be more difficult.

5.2 Out-of-Distribution Commands

As mentioned in Sec.[3.2] we adopt an LSTM encoder to encode arbitrary natural language commands into fixed-length sentence embeddings. During the training process, there are a total of 38,558
different commands in the training set. We also evaluate how the policies perform on Out-of-Distribution (OOD) commands. As shown in Tab. 3, we try 4 different ways of adding noise to Oracle commands and report the win rates. We can observe that the agents under noisy oracle commands perform better than Random commands but worse than Oracle. Intuitively, we can observe that the win rate roughly follows “Random” < “Drop” < “Replace” < “Shuffle” < “Oracle”. This suggests that the LSTM encoder is sensitive to input words and word order. We also find that strong-following policies are affected more by noisy commands than weak-following policies, indicating that strong-following policies are more sensitive to the commands. In addition, we also investigate the influence of Out-of-Vocabulary (OOV) words in Appendix D.5, and visualize the command representations in Appendix D.6.

<table>
<thead>
<tr>
<th>Noise Type</th>
<th>Command-Ignorant</th>
<th>Command-Following</th>
</tr>
</thead>
<tbody>
<tr>
<td>Random (%)</td>
<td>53.3 ± 0.4</td>
<td>12.3 ± 0.2</td>
</tr>
<tr>
<td>Drop (%)</td>
<td>61.9 ± 1.0</td>
<td>38.8 ± 0.6</td>
</tr>
<tr>
<td>Replace (%)</td>
<td>59.1 ± 2.4</td>
<td>27.6 ± 0.9</td>
</tr>
<tr>
<td>Insert (%)</td>
<td>75.4 ± 0.7</td>
<td>78.3 ± 0.7</td>
</tr>
<tr>
<td>Shuffle (%)</td>
<td>75.1 ± 1.8</td>
<td>68.6 ± 1.4</td>
</tr>
<tr>
<td>Oracle (%)</td>
<td>89.3 ± 0.7</td>
<td>78.7 ± 0.9</td>
</tr>
</tbody>
</table>

Table 3: We add noise to Oracle commands. For Drop, we delete 50% of words in each command. For Replace, we replace 50% of the words with random words. For Insert, we insert random words and make the original command twice longer. For Shuffle, we shuffle the words in each command.

5.3 Ablation Study

Training commands. We test the performance of RED with different training commands in Fig. 3. The default RED yields the best strategy, i.e., the highest win rate with NA and oracle commands. Training with random commands or human proxy makes the policy command-ignorant – both two variants achieve similar win rates under NA and random commands.

![Figure 3: Win rates of RED with various training commands. RED has the strongest winning strategy with NA commands. Both random command and human proxy result in a command-ignoring policy.](image)

Ratio of $\|D_k\|$ and $\|D\|$. We test different ratios of on-policy samples, i.e., $\|D_k\|$, to the demonstration size, i.e., $\|D\|$, in Tab. 4. We report the test-time win rates with oracle and NA commands as well as the constraint satisfaction condition, including both NLL (i.e., $L(\theta; D)$ from Eq. 2) and the action prediction accuracy, on the validation set. The results show that the validation NLL can be indeed controlled by tuning the dataset ratio, i.e., more RL samples consistently yielding a higher NLL. By contrast, we also find that a lower NLL does NOT necessarily lead to a higher action prediction accuracy. But we empirically find that simply setting a 1:1 ratio leads to a sufficiently good policy, which is the default choice for our RED algorithm.

<table>
<thead>
<tr>
<th>$|D_k| : |D|$</th>
<th>Test Win Rate with Different Commands</th>
</tr>
</thead>
<tbody>
<tr>
<td>2:1</td>
<td>87.7 92.6 92.0 89.6</td>
</tr>
<tr>
<td>1:1</td>
<td>54.6 57.8 56.1 57.0</td>
</tr>
<tr>
<td>1:0.5</td>
<td>3.15 3.13 3.00 2.92</td>
</tr>
<tr>
<td>1:0.25</td>
<td>68.9 68.5 69.0 69.2</td>
</tr>
</tbody>
</table>

Table 4: Ablation study on the ratio of $\|D_k\|$ and $\|D\|$. 0.5 means sub-sampling a half of data from $D$. The data ratio leads to a controlled NLL.
**Learned strategy.** We identify the five most common categories of commands and measure the validation action prediction accuracy for both RED and BC policies to examine under what conditions the actions are changed – noting that RED policy uses BC policy as a warm-start. Fig. 4 shows the categorized accuracy differences with the dashed line denoting the overall mean difference. We can observe that RED policy disagrees with human actions the most under commands belonging to “attack” and “defend” categories, which both represent a highly complicated space of possible behaviors (e.g., which enemy unit to attack, how to defend). We believe these action changes are caused by policy improvement. While for commands with a low uncertainty belonging to “mine” and “stop” categories, RED policy simply follows humans. These findings provide empirical supports to our intuitive justification in Sec. 4.3.

5.4 Human Evaluation

We pick 4 policies, i.e., RED, Switch, Joint and RL, and invite 30 college student volunteers under department permission to complete a two-stage study. In the first stage, we shuffle the order of the 4 policies and ask each student to keep playing with the 4 policies using any commands. Once the student feels familiar with the game and policies enough, he/she is asked to rank the policies by the level of how they follow commands. In the second part, each student is asked to play 2 games per policy, and we measure the average winning rate of different policies. The results of ranking votes and win rates are shown in Fig. 5 and Fig. 6. Regarding the votes, the average rank of RED, Switch, Joint and RL are 2.3, 2.4, 2.5, 2.8, respectively. Fig. 5 visualizes the vote percentage for different policies. We notice a very interesting phenomenon that every policy has almost the same 1st-rank votes. Knowing that we shuffle the policy order while the students often spend a long time on the first policy to get familiar with MiniRTS, we hypothesize that the students are often biased towards the first policy they encountered. Nevertheless, RED has substantially more votes for the 2nd-rank, indicating a better command-following capacity. RL has the most 4th-rank votes, which is consistent with our expectation. Regarding the win rates (Fig. 6), RED outperforms Switch and Joint with a clear margin. The win rate of RL is comparable with RED. We believe this is due to the fact that RL policy often ignores human commands and simply acts for winning.

6 Conclusion

We tackle a new problem, grounded reinforcement learning (GRL), which aims to learn an agent that can not only get high rewards but also faithfully follow natural language commands from humans. We proposed a tractable objective, and developed an iterative RL algorithm RED, and evaluated the derived policy on a real-time strategy game MiniRTS. Both simulation and human results show that a RED policy achieves high win rates while exhibits strong command-following capabilities.

**Limitation and social impact.** As an initial study on the GRL problem, the RED algorithm is only evaluated on MiniRTS. This is because MiniRTS is the only public environment that is both complex (beyond “reaching goals”) and provides natural language data. In addition, although we draw connections to existing theories in multi-task learning to justify why RED works, the theoretical principle of the underlying optimization process remains an open problem. Finally, even though it is in general debatable whether a robot should be fully obedient [18, 26, 50, 58], we believe that in restricted single-agent domains, like playing video games, strictly enforcing command-following behaviors should not result in worse negative social impact than BC or other RL methods.

Acknowledgement

Yi Wu is supported by 2030 Innovation Megaprojects of China (Programme on New Generation Artificial Intelligence) Grant No. 2021AAA0150000.
References


Checklist

1. For all authors...
   (a) Do the main claims made in the abstract and introduction accurately reflect the paper’s contributions and scope? [Yes] See Section.1
   (b) Did you describe the limitations of your work? [Yes] See Section.6
   (c) Did you discuss any potential negative societal impacts of your work? [Yes] See Section.6
   (d) Have you read the ethics review guidelines and ensured that your paper conforms to them? [Yes]

2. If you are including theoretical results...
   (a) Did you state the full set of assumptions of all theoretical results? [N/A]
   (b) Did you include complete proofs of all theoretical results? [N/A]

3. If you ran experiments...
   (a) Did you include the code, data, and instructions needed to reproduce the main experimental results (either in the supplemental material or as a URL)? [Yes] In supplemental material.
   (b) Did you specify all the training details (e.g., data splits, hyperparameters, how they were chosen)? [Yes] See Appendix.B.
   (c) Did you report error bars (e.g., with respect to the random seed after running experiments multiple times)? [Yes] See Section.5
   (d) Did you include the total amount of compute and the type of resources used (e.g., type of GPUs, internal cluster, or cloud provider)? [Yes] See Appendix.B.

4. If you are using existing assets (e.g., code, data, models) or curating/releasing new assets...
   (a) If your work uses existing assets, did you cite the creators? [Yes] See Section.3
   (b) Did you mention the license of the assets? [Yes] See Appendix.A.
   (c) Did you include any new assets either in the supplemental material or as a URL?[N/A]
   (d) Did you discuss whether and how consent was obtained from people whose data you’re using/curating? [Yes] See Appendix.A.
   (e) Did you discuss whether the data you are using/curating contains personally identifiable information or offensive content? [Yes] See Appendix.A.

5. If you used crowdsourcing or conducted research with human subjects...
   (a) Did you include the full text of instructions given to participants and screenshots, if applicable? [Yes] See Appendix.E.
   (b) Did you describe any potential participant risks, with links to Institutional Review Board (IRB) approvals, if applicable? [Yes] See Appendix.E.
   (c) Did you include the estimated hourly wage paid to participants and the total amount spent on participant compensation? [Yes] See Appendix.E.

A MiniRTS Details & Dataset

In this section, we describe the details of MiniRTS Environment and human dataset. Our work is based on the game and dataset released at https://github.com/facebookresearch/minirts under CC BY-NC 4.0 license. The data do not contain any personally identifiable information or offensive content.

A.1 Game Design
Figure 7: MiniRTS \[33\] implements the rock-paper-scissors attack graph, each army type has some units it is effective against and vulnerable to. For example, “swordman” restrains “spearman” but is restrained by “cavalry”. “swordman”, “spearman” and “cavalry” are all effective against “archer”.

Game Units  There are 3 kinds of units in MiniRTS, including resource units, building units, and army units.

- **Resource Units**: Resource units are stationary and neutral. Resource units cannot be constructed by anyone and are created at the beginning of a game. Only “peasant” (an army unit type) of both teams can mine from the resource units. One mine action could gather resources from the resource units, and the mined resources are necessary to build new building units or army units.

- **Building Units**: MiniRTS supports 6 different building unit types. 5 of the building unit types can produce particular army units by consuming resources (Fig. 8). The “guard tower” can not produce army units but can attack enemies. All the building units cannot move. Building units can be constructed by “peasant” at any available map location.

- **Army Units**: 7 types of army units can move and attack enemies. Specifically, a “peasant” can mine resources from resource units and construct building units with mined resources, but its attack power is low. The other 6 army unit types and “guard tower” are designed with a rock-paper-scissors dynamic. As shown in Fig. 7, each type has some units that it is effective against and vulnerable to.

Game Map  The game map is a discrete grid of 32x32, where each cell can either be grass or river. The grass cell is available for constructing building units and is passable for any army unit, while the river cell only allows the “dragon” to go through. The map of each game is generated randomly. When initializing the map, one “town hall” and three “peasant” are placed for each player, then river cells and several resource units are added randomly. This generation phase ensures at least one path between two players’ “town hall”, and there are approximately equal resource units around each “town hall”. In addition, the map is partially observable due to “fog-of-war”.

A.2 MiniRTS as an RL Environment

As an RL environment, a player is controlled by the RL agent while the opponent is a built-in script AI. The RL agent needs to control units to collect resources, do construction and kill all the enemy units or destroy the enemy base (i.e., “town hall”) to win a game. We limit the length of a game to a maximum of 256 time steps.
**Table 5:** The action types and the corresponding action outputs.

<table>
<thead>
<tr>
<th>Action Type</th>
<th>Action Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDLE</td>
<td>NULL</td>
</tr>
<tr>
<td>CONTINUE</td>
<td>NULL</td>
</tr>
<tr>
<td>GATHER</td>
<td>ID of resource unit</td>
</tr>
<tr>
<td>ATTACK</td>
<td>ID of enemy unit</td>
</tr>
<tr>
<td>TRAIN</td>
<td>Army unit type</td>
</tr>
<tr>
<td>BUILD</td>
<td>Building unit type &amp; (x, y)</td>
</tr>
<tr>
<td>MOVE</td>
<td>(x, y)</td>
</tr>
</tbody>
</table>

**Observation Space**  The observation of an agent includes a 32x32 map and extra states of the game (e.g., health points of the observable units, the amount of resources, etc.). The regions not visited are masked in the observation, and unseen enemy units are removed.

**Action Space**  At each time step, the environment requires an action for each controlled unit, leading to a particularly large action space. We introduce an additional 0/1 action for each unit, denoting whether this unit should act or not. We generate the same action for those selected units, which reduces the action dimension substantially. In particular, our policy network outputs a common action as well as a 0/1 flag for each unit. Then we convert this into the standard MiniRTS format in the following way: for a unit assigned “1”, it executes the common output action, for a unit assigned “0”, it executes the action CONTINUE. After determining which units should act, following [33], we first predict an action type (e.g., MOVE, ATTACK), then predict the action outputs based on the action type. We summarize all available action types and their structure in Tab. [5] For IDLE, the selected units would do nothing. For CONTINUE, the units would continue their previous actions. For GATHER, we should also tell the units the ID of the target resource unit. For ATTACK, we should tell the units the enemy unit ID. For TRAIN, we should tell the units the army type they should train. For BUILD, we should tell the units the building type and the position (x,y) to build. For MOVE, we should tell the units the target position.

**Reward**  This environment supports a sparse reward. At the end of a game, the reward is 1 if the agent wins and -1 if the agent loses. And the agent would receive the reward of 0 all the other time steps.

**A.3 Built-in Script AI**

The authors of MiniRTS [33] provide several built-in script AIs. We find that an unconditioned RL agent achieves comparable win rates against both medium level and strong level script AIs, and we choose the medium level AI as the opponent for the convenience of designing the oracle commands. This script first sends all 3 initially available peasants to mine from the closest resource unit. It randomly chooses one army unit type from “spearman”, “swordman”, “cavalry”, “archer”, “dragon” and “catapult” and determines an army size $n$ between 3 and 7. It constructs a building unit corresponding to the selected army (Fig. [8]), then trains $n$ units of the selected army type and sends them to attack. The script continuously trains the army units and maintains the army size of $n$.

**A.4 Dataset**

The authors of MiniRTS [33] split the dataset into a training set and a validation set. The training set includes 4,171 trajectories, 57,293 commands and 634,799 transitions. While the validation set contains 433 trajectories, 5,992 commands, and 63,649 transitions. When training the policy, we only use data in the training set.
B Implementation Details

In this section, we introduce the implementation details and hyper-parameters. We train all the policies on a server with 8 RTX-3090 GPUs. All the policies warm-start with the BC pretrained model.

B.1 RED Implementation

Policy Architecture We adopt a similar policy architecture to the provided executor model in MiniRTS [33] but with a modified action space and an additional value head as the critic. The original model outputs an action for every controllable unit, which makes RL training extremely slow. We introduce an additional 0/1 selection action on each unit, denoting whether this unit should act or not, and only generate the same action for those selected units based on the average of their features, which substantially reduces the action dimension.

RL Phase We train the policy through a parallel PPO training process. We use 1024 parallel workers to collect transitions \((s, a, r, s')\) from the environments synchronously (\(c\) is NA for RED). Once 128 workers get 256 data points each, we split the collected data into 4 batches and run one epoch of PPO training by setting the discount factor \(\gamma\) as 0.999. We use GAE advantage for each data point. In each iteration, we repeat 100 training epochs as described above.

We collect winning trajectories and store them as \(D_k\). In the early stage of the training phase, the winning rate can be low. To obtain sufficient data for \(D_k\), we implement a queue to store all past winning trajectories and use bootstrapped sampling if the queue is still not full. Note that in each iteration, 100 epochs of PPO training produces \(100 \times 128 \times 256 = 3,276,800\) transitions, and there are \(634,799\) transitions in the training set of \(D\). We limit the capacity of \(D_k\) to the same as the size of \(D\). We report the results on different ratios of \(\|D_k\|\) and \(\|D\|\) in Sec. 5.2.

BC Phase We adopt a similar BC training process to MiniRTS [33]. Since we introduce an additional 0/1 action to denote whether each controllable unit should act or not, we also need to train this action in the BC phase. In a transition of \(D\), each controllable unit is paired with an action type. We first find the most common action type across all units, then label the units with this action type as 1 and the rest units as 0. Finally, based on these labels, we train the additional 0/1 actions for all units. We run one epoch of BC training in each iteration by setting the batch size as 2048.

Optimizer & Learning Rate We use Adam optimizer and adopt separate optimizers for RL and BC training. We run a total of 40 iterations (i.e. 4,000 PPO epochs) to train a RED policy. For BC training, we set \(\beta = (0.9, 0.999)\) and fix the learning rate as \(2e^{-4}\). For RL training, we also set \(\beta = (0.9, 0.999)\) but adapt the learning rate throughout all the 4,000 PPO epochs. In the first 500 epochs, we start with a learning rate of 0 and linearly increase it to the desired value \(5e^{-5}\). In the rest 3500 epochs, we decrease the learning rate from \(5e^{-5}\) to 0 linearly.

B.2 Baseline Implementation

RL Implementation For pure RL baseline, we adopt the same architecture and hyper-parameters as RED. We run 4,000 PPO epochs. The batch size and number of transitions in each epoch are the same as RED. We also adopt the same optimizer and learning rate. The only difference is that there is no BC phase during the pure RL training. RL training also starts with the BC pretrained policy.

Joint Implementation For joint RL baseline, we modify the loss in the pure RL training by adding the NLL loss. The batch size to compute the NLL loss is the same as RL training. The architecture and all the hyper-parameters are the same as RED and pure RL training. We tune the weight \(\beta\) of the NLL loss via a grid-search process, which is described in Sec. D.1.

IRL Implementation We adopt AIRL [24] for IRL training. The policy network has the same architecture and hyper-parameters as RED. We train a discriminator network \(D_\phi(s, c, a)\) in the form of

\[
D_\phi(s, c, a) = \frac{\exp f_\phi(s, c, a)}{\exp f_\phi(s, c, a) + \pi_\theta(a|s, c)}, \tag{6}
\]
where \( \pi \) is the policy network. The hyper-parameters are the same as the policy network. For the first 11 iterations, we run 25 discriminator epochs before each policy epoch. For the rest of the iterations, we run a single discriminator epoch before each policy epoch.

The architecture of \( f_{\phi}(s, c, a) \) is similar to the policy network. We extract a fixed-length global feature vector, a fixed-length command feature vector, and fixed-length feature vectors for each unit and each position on the map in the same way as in MiniRTS [33]. Recall that an action contains an action type and possibly action outputs. We encode action type using an embedding layer. Action types IDLE and CONTINUE do not have any action output. For action types GATHER, ATTACK, and TRAIN, we encode action outputs by their corresponding unit embedding. For action type MOVE, we encode action outputs by the embedding of the moving location. For action type BUILD, we encode the building type using an embedding layer and then add it to the embedding of the building location to obtain the embedding of action output. The embedding of action is then obtained by concatenating the embedding of the action type and action outputs. In addition, we encode unit selection features by averaging the extracted features of the units that are selected. Finally, we concatenate the global feature, the command feature, the action feature, the unit selection feature, and the global continue flag together and feed it to a linear layer to get the value of \( f_{\phi}(s, c, a) \).

The discriminator objective is given by

\[
E_{D}[\log D_{\phi}(s, c, a)] + E_{\phi}[\log(1 - D_{\phi}(s, c, a))].
\] (7)

We combine environment rewards and the intrinsic rewards, namely

\[
r(s, a) = r_{\text{env}}(s, a) + \beta_i \text{clip} \left( \frac{r_{\text{disc}}(s, c, a) - \mu}{\sigma}, -1, 1 \right),
\] (8)

where

\[
r_{\text{disc}}(s, c, a) = \log D_{\phi}(s, c, a) - \log(1 - D_{\phi}(s, c, a)),
\] (9)

and \( \mu, \sigma \) are the mean and variance of \( r_{\text{disc}}(s, c, a) \) within a sample batch, respectively. The tuning process of weight \( \beta_i \) of the intrinsic reward is described in Sec. D.3.

C Experimental Details

C.1 Oracle Commander

Since the built-in script AI only builds army units of a single type in a game. We script an oracle command strategy according to the ground truth of enemy units and the attack graph (Fig. 7). This commander would ask the agent to build the appropriate army units step by step in the early stage of the game. Then, it sends NA and asks the policy to play on its own. A strong GRL policy would follow these commands to build the correct army units, play itself in the game’s reset, and achieve a higher win rate.

- **Enemy “spearman”:** (i) “mine with all idle peasant” in the first 2 time steps. (ii) “build 3 peasant” until there are 6 peasants. (iii) “build a blacksmith” until there is a blacksmith. (iv) “build another cavalry” when the number of cavalry is less than 5 otherwise NA.
- **Enemy “swordman”:** (i) “mine with all idle peasant” in the first 2 time steps. (ii) “build 3 peasant” until there are 6 peasants. (iii) “build a stable” until there is a stable. (iv) “build another cavalry” when the number of cavalry is less than 5 otherwise NA.
- **Enemy “cavalry”:** (i) “mine with all idle peasant” in the first 2 time steps. (ii) “build 3 peasant” until there are 6 peasants. (iii) “build a barrack” until there is a barrack. (iv) “build a spearman” when the number of spearman is less than 5 otherwise NA.
- **Enemy “dragon”:** (i) “mine with all idle peasant” in the first 2 time steps. (ii) “build 3 peasant” until there are 6 peasants. (iii) “build a workshop” until there is a workshop. (iv) “make archer” when the number of archer is less than 5 otherwise NA.
- **Enemy “archer”:** (i) “mine with all idle peasant” in the first 2 time steps. (ii) “build 3 peasant” until there are 6 peasants. (iii) Randomly select a building command from “build a blacksmith”, “build a stable” and “build a barrack” until the corresponding building unit is constructed. (iv) Select from “build another swordman”, “build another cavalry” and “build a spearman” according to the constructed building unit when the number of the corresponding army unit is less than 5 otherwise NA.
• **Enemy “catapult”**: We empirically find that the best strategy to deal with the “catapult” is to send **NA** all the time (according to the policies of Joint, RED and Switch). We hypothesize it is because that all the other army types are effective against “catapult”, and building commands as before are not helpful to play against catapult.

### C.2 “Human Proxy” Command Generator

Training a command generator is not our focus. We use the best commander (instructor) model trained and released by MiniRTS [33]. Please refer to the model at https://github.com/facebookresearch/minirts.

### C.3 Learned Strategy

**How to Categorize Commands**  
We identify the five most common categories of commands as follows:

- **Attack**: Commands containing “attack” or “kill” belong to this category. This type of command typically instructs the agent to attack enemy units, but does not necessarily specify which unit to attack.
- **Defend**: Commands containing “defend” belong to this category. This type of command typically asks the agent to fight with invading enemy units but usually does not give more detail.
- **Build**: Commands containing “build”, “create” or “make” belong to this category. This type of command typically instructs the model to build a specific type of building unit or army unit.
- **Mine**: Commands containing “mine”, “mining” or “mineral” belong to this category. This type of command typically instructs the model to mine from the resource units.
- **Stop**: Commands containing “stop” belong to this category. Such commands typically claim that the current action should be terminated.

**Action Prediction Accuracy**  
The actions in the transitions of \( \mathcal{D} \) are extremely complex. In each transition, there are many controllable units, and each unit has its action type and the corresponding action output (see Tab. 5). So it is not so direct to compute the action prediction accuracy. When predicting the action prediction, we concentrate on the units whose action types are not CONTINUE, since the units with CONTINUE would continue their previous actions, which should be considered in the previous transitions. In detail, for a single transition, we skip the actions of determining whether the units should act or not, and treat the units whose actions are not CONTINUE as the selected units. We predict the action type and action output based on the selected units. We think the prediction is correct if the predicted action type and action output exactly match with any selected unit. The accuracy is then computed as the correct prediction ratio over all the transitions. In addition to action prediction accuracy, we also evaluate NLL difference of RED policy and BC policy in Sec. D.4.

### D Additional Results

#### D.1 Grid Search on \( \beta \) for Baseline “Joint”

We conduct the experiments on joint training with different \( \beta \), and the results are listed in Tab. 5. We can observe that the policy with a larger \( \beta \) performs worse when no commands are provided (NA), which indicates that the joint objective may be harmful to RL training. In addition, the win rates tested with random commands suggest that a smaller \( \beta \) may result in the policy being less obedient.

#### D.2 Adaptive \( \beta \)

Although it is empirically a common practice to use a fixed \( \beta \) in the Lagrangian method, we additionally provide experiments in which the coefficient \( \beta \) is updated during the training process. It is worth noting that a similar technique can also be applied to our proposed method RED. In
We can observe that the win rates tested with random commands are comparable with those tested with NA. We adopt AIRL [24] as the algorithm for IRL experiments. We learn a reward function from joint under the Oracle commander. This suggests that iterating between RL and BC ensures better RL performance than mixing the two objectives.

D.3 Grid Search on $\beta_i$ for Baseline “IRL”

We adopt AIRL [24] as the algorithm for IRL experiments. We learn a reward function from demonstrations and combine the game reward and the learned language reward to train a conditioned policy (See Eq. 8). We use the human proxy command generator to provide commands when training, since the learned language reward is conditional on game states and human commands. We evaluate the performance with different $\beta_i$, and the results are listed in Tab. 8.

<table>
<thead>
<tr>
<th>$\beta$</th>
<th>0</th>
<th>0.5</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracle (%)</td>
<td>89.3 ± 0.7</td>
<td>89.9 ± 0.3</td>
<td><strong>90.2 ± 0.7</strong></td>
<td>87.8 ± 0.8</td>
</tr>
<tr>
<td>NA (%)</td>
<td><strong>57.5 ± 1.1</strong></td>
<td>54.9 ± 0.9</td>
<td>51.8 ± 0.8</td>
<td>48.4 ± 0.6</td>
</tr>
<tr>
<td>Random (%)</td>
<td>53.3 ± 0.4</td>
<td>34.3 ± 0.9</td>
<td>32.3 ± 0.3</td>
<td>28.9 ± 0.2</td>
</tr>
<tr>
<td>Human Proxy (%)</td>
<td>43.6 ± 0.6</td>
<td>12.1 ± 0.7</td>
<td>11.4 ± 0.7</td>
<td>10.4 ± 0.4</td>
</tr>
</tbody>
</table>

Table 6: Results of Joint on different $\beta$, where $\beta = 0$ is equal to pure RL training.

<table>
<thead>
<tr>
<th></th>
<th>Switch</th>
<th>Joint (fixed $\beta$)</th>
<th>RED (fixed $\beta$)</th>
<th>Joint (adaptive $\beta$)</th>
<th>RED (adaptive $\beta$)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracle (%)</td>
<td>78.7 ± 0.9</td>
<td>90.2 ± 0.7</td>
<td>92.6 ± 0.6</td>
<td><strong>93.7 ± 0.1</strong></td>
<td>93.1 ± 0.1</td>
</tr>
<tr>
<td>NA (%)</td>
<td>57.5 ± 1.1</td>
<td>51.8 ± 0.8</td>
<td>57.8 ± 1.3</td>
<td>57.7 ± 0.3</td>
<td><strong>61.2 ± 0.3</strong></td>
</tr>
<tr>
<td>Random (%)</td>
<td>12.3 ± 0.2</td>
<td>32.3 ± 0.3</td>
<td>29.8 ± 0.8</td>
<td>38.1 ± 1.1</td>
<td>37.6 ± 1.2</td>
</tr>
</tbody>
</table>

Table 7: Results of Joint and RED using adaptive $\beta$.

One can see that using an adaptive $\beta$ improves the performance of both Joint and RED. While having similar performance under the Oracle commander, RED with adaptive $\beta$ significantly outperforms Joint under the NA commander. This suggests that iterating between RL and BC ensures better RL performance than mixing the two objectives.

We adopt AIRL [24] as the algorithm for IRL experiments. We learn a reward function from demonstrations and combine the game reward and the learned language reward to train a conditioned policy (See Eq. 8). We use the human proxy command generator to provide commands when training, since the learned language reward is conditional on game states and human commands. We evaluate the performance with different $\beta_i$, and the results are listed in Tab. 8.

<table>
<thead>
<tr>
<th>$\beta_i$</th>
<th>0.001</th>
<th>0.005</th>
<th>0.01</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oracle (%)</td>
<td><strong>57.3 ± 3.9</strong></td>
<td>53.7 ± 2.8</td>
<td>43.2 ± 2.0</td>
</tr>
<tr>
<td>NA (%)</td>
<td><strong>45.6 ± 2.9</strong></td>
<td>41.5 ± 2.8</td>
<td>33.7 ± 3.3</td>
</tr>
<tr>
<td>Random (%)</td>
<td>47.9 ± 2.8</td>
<td>43.9 ± 1.2</td>
<td>30.1 ± 2.1</td>
</tr>
<tr>
<td>Human Proxy (%)</td>
<td>48.0 ± 2.9</td>
<td>43.6 ± 3.4</td>
<td>33.9 ± 2.2</td>
</tr>
</tbody>
</table>

Table 8: Results of IRL on different $\beta_i$.

We can observe that the win rates tested with random commands are comparable with those tested with NA, which indicates that IRL is not able to make the policy obedient, although we increase the weight of the learned reward. In addition, the learned reward may be harmful to the RL training, larger $\beta_i$ leads to lower win rates tested on the oracle and NA commands.

D.4 Validation NLL for Various Command Types

In addition to action prediction accuracy, we also evaluate NLL difference of RED policy and BC policy in Fig. 8 on the validation set. The dashed line denotes the overall validation difference. The conclusion is the same as action prediction accuracy. RED disagrees with the commands with a highly complicated space of possible behaviors (e.g., the command belonging “attack”, “defend”) the most, and follows humans’ action on the commands with low uncertainty (e.g., the commands belonging to “stop”).

22
D.5 Performance on Commands with OOV Words

We also evaluate the policy performances on commands with Out-of-Vocabulary (OOV) words. It is worth noting that we use a special token [unk] to handle all the OOV words. We insert the special token in different positions of the oracle commands to evaluate the influence of OOV words. We describe the ways to insert the OOV words as follows. And the results are listed in Tab. 9.

- All-OOV: We replace all words with [unk] tokens in each command.
- Pre-OOV: We insert a sequence of [unk] tokens in front of each command, which makes the command twice the length of the original.
- Post-OOV: We insert a sequence of [unk] tokens at the end of each command, which makes the command twice the length of the original.
- Mid-OOV: We insert [unk] tokens between any two words of each command.

<table>
<thead>
<tr>
<th>Test Commands</th>
<th>Command-Ignorant</th>
<th>Command-Following</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>RL</td>
<td>IRL</td>
</tr>
<tr>
<td>Oracle (%)</td>
<td>89.3 ± 0.7</td>
<td>57.3 ± 3.9</td>
</tr>
<tr>
<td>NA (%)</td>
<td>57.5 ± 1.1</td>
<td>45.6 ± 2.9</td>
</tr>
<tr>
<td>All-OOV (%)</td>
<td>56.3 ± 0.7</td>
<td>41.9 ± 4.4</td>
</tr>
<tr>
<td>Pre-OOV (%)</td>
<td>89.4 ± 0.4</td>
<td>56.1 ± 1.9</td>
</tr>
<tr>
<td>Post-OOV (%)</td>
<td>78.9 ± 0.4</td>
<td>50.9 ± 1.8</td>
</tr>
<tr>
<td>Mid-OOV (%)</td>
<td>71.1 ± 0.9</td>
<td>50.9 ± 2.4</td>
</tr>
</tbody>
</table>

Table 9: Win rates under commands with OOV words.

The results show that “All-OOV” commands are terrible for our strong following policies (i.e., RED, Joint, and Switch). RL and IRL policies perform near the same under the NA command and All-OOV commands, which suggests they are command-ignorant. Performances on Pre-OOV, Post-OOV and Mid-OOV indicate that the policies can also perform well when some OOV words are inserted. In Pre-OOV and Post-OOV commands, the original sequences are kept, which outperform Mid-OOV. Particularly in Pre-OOV commands, the performances are near the same with Oracle.

D.6 Clustering Analysis of Command Representations.

We generate commands using some templates and visualize their embeddings with t-SNE. The templates are listed as follows:

- build building units: {build | create | make} {blacksmith | stable | barrack | workshop | guard tower}, {build | create | make} a {blacksmith | stable | barrack | workshop | guard tower}, {build | create | make} another {blacksmith | stable | barrack | workshop | guard tower}.
• build army units:
  {build | create | make} {swordman | cavalry | spearman | archer | dragon | catapult},
  {build | create | make} a {swordman | cavalry | spearman | archer | dragon | catapult},
  {build | create | make} another {swordman | cavalry | spearman | archer | dragon | catapult},
  {build | create | make} 3 {swordman | cavalry | spearman | archer | dragon | catapult},
  {build | create | make} 5 {swordman | cavalry | spearman | archer | dragon | catapult},
  {build | create | make} 7 {swordman | cavalry | spearman | archer | dragon | catapult}.

• attack: attack, kill.

• scout: scout, scout the map, go around the map.

• mine: mine, {gather | collect | mine} {minerals | resources}.

Figure 10: Visualization of command embeddings, similar commands are encoded into similar embeddings.

As shown in Fig. 10, similar commands are encoded into similar embeddings. In addition, the commands that build building units are close to each other and far from the commands that build army units, which suggests that the LSTM module is able to learn the semantic meanings of different commands.

E Human Evaluation Details

E.1 Ethics Statement

The experiments are permitted under our department committee. There is not any personally identifiable information or sensitive personally identifiable information involved in the experiment.

When conducting the human evaluation experiments, we informed the participants in advance of the purpose of the experiment and the time the experiment might take. All the participants are fully informed and participate voluntarily with signed confirmation. We have controlled each experiment to be completed within 1 hour.

We invite 28 undergrad students and 2 Ph.D. students under department permission to complete a two-stage study. We release this human evaluation as an optional project in an undergrad course. The students who are interested in this project can participate. The Ph.D. students are well paid as research assistants.
E.2 Evaluation Process

We first introduce the rules of the game as described in Sec. A.1 and Fig. 7, and ask them to play the game by providing the commands. To avoid the participants not knowing what to do at the beginning, we give some basic commands as shown in Tab. 10. In addition to these basic commands, the participants can also input arbitrary commands. For example, some participants find that “go around the map” and “scout the map” can make the units explore the map and find the enemies.

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>mine</td>
</tr>
<tr>
<td>attack</td>
</tr>
<tr>
<td>build a blacksmith</td>
</tr>
<tr>
<td>build a barrack</td>
</tr>
<tr>
<td>build a stable</td>
</tr>
<tr>
<td>build a workshop</td>
</tr>
<tr>
<td>build dragon</td>
</tr>
<tr>
<td>build a guard tower</td>
</tr>
<tr>
<td>build peasant</td>
</tr>
<tr>
<td>all peasant mine</td>
</tr>
<tr>
<td>defend</td>
</tr>
<tr>
<td>build swordman</td>
</tr>
<tr>
<td>build spearman</td>
</tr>
<tr>
<td>build cavalry</td>
</tr>
<tr>
<td>build archer</td>
</tr>
<tr>
<td>build catapult</td>
</tr>
<tr>
<td>build a town hall</td>
</tr>
</tbody>
</table>

Table 10: Basic commands presented to volunteers. The volunteers can also input arbitrary commands.

We pick 4 policies. i.e., RED, Switch, Joint and RL. In the first stage of the human evaluation, we shuffle the order of 4 policies and ask each volunteer to play with these policies using arbitrary commands. Once the volunteer feels familiar with the game policies, he/she is asked to rank the policies according to how the policies follow commands. In the second stage, each volunteer is asked to play 2 games per policy and try their best to win. In this stage, the volunteers can turn off the “fog-of-war” and observe the enemy units (“fog-of-war” is always turned on for the executor policies).

Note that we shuffle the policy order, and the students often spend a long time on the first policy to get familiar with MiniRTS, which can make the students biased towards the first policy they encountered. We think that a better evaluation process is to arrange an additional practice stage, allowing participants to play with a BC policy and get familiar with the game.

E.3 Gameplay Interface

We present a screenshot of the gameplay interface in Fig. 11. The user can select a command from a set of recommendations using the "Select Command" button or input an arbitrary (English) command in the text box below. If the user types ENTER in the text box while leaving it empty, the previous command will be issued. This functionality is designed because we notice that it is a natural choice to keep sending the same command for several continuous time steps. To send an empty command (i.e., the NA command), the user can press the “Send Empty Command” button. Since some users may not be good at playing real-time strategy games, one can optionally toggle the fog of war using the “Fog of War: ON(OFF)”. This lowers the difficulty of the game.

E.4 Emergent Behaviors

We find some interesting emergent behaviors during human evaluation. Please refer to our videos at https://sites.google.com/view/grounded-rl.

Some human participants prefer to build dragons since dragons sound powerful and cool. But building dragons makes it more challenging to win because dragons are expensive. If the player builds dragons at the beginning of a game, he/she will get few dragons ready when the enemies start to attack him/her, leading to a game loss. So the RL trained policies prefer to build “spearman”, “swordman” or “cavalry”, since win rates of these units are higher.

Fortunately, humans are intelligent and creative. Some participants figure out the behavior pattern of the built-in script AI. They find that the script AI will quickly build several army units and send them to attack. It is vital to resist the first two waves of attacks. A participant instructs the policy to construct several towers in the early stage of the game to resist the attack. After ensuring the safety, the student asks the policy to build dragons and then sends the dragons to find the enemies to attack, which finally results in a win.
It is difficult for pure RL to learn such a complicated strategy since building other army units (e.g., “spearman”, “swordman” or “cavalry”) is more direct for a fast win. A well-trained RED policy can follow human commands to execute this complicated strategy. In addition, although the human commands are vague, RED policy can take suitable actions to follow the commands. For example, it builds towers close to each other to strengthen the defense. After finding the enemies, it automatically sends the dragons to attack them.

Here we present the example of how the human commander instructs the RED policy to win a game using dragon by several commands in Fig. 12. There are 3 phases. In phase 1, the human player asks the policy to build towers by sending “build a guard tower”. In phase 2, the human player instructs the policy to build dragons using commands “build a workshop” and “build dragon”. In phase 3, the human player sends “scout the map”, and the policy makes the dragons fly around the map and find the enemies. We describe these 3 phases in detail in the following.

**Phase 1: Build Towers** As shown in Fig. 13, in the early stage of the game, to ensure safety, the human player gives the command “build a guard tower” at every time step. So the RED policy lets one peasant build the tower, and the rest peasants keep mining resources from the resource units. Since the command is provided at every time step, the peasant keeps constructing towers one after another.

**Phase 2: Build Dragons** As shown in Fig. 14, after building enough towers, the human player changes the command to “build a workshop”, where the workshop can produce dragons. Then one peasant starts to build the workshop immediately. Once the workshop is built, the human player changes the command and keeps asking the policy to “build dragon”. Then, the workshop keeps building the dragons, and all the peasants return to mine resources. In this phase, some enemy units started to attack but are all defended by the towers.

**Phase 3: Find the Enemies** As shown in Fig. 15, after building 4 dragons, the human player changes the command to “scout the map”, and the dragons fly around the map to explore. Once the dragons find the enemies, RED policy sends them to attack the enemies automatically.

Figure 11: The gameplay interface.
Figure 12: The overview of the “tower defense and dragon rush” strategy. The human player first asks the RED policy to build several guard towers to ensure safety, then build the dragons. The enemies try to attack when the RED policy is building dragons but are defended by the towers. After enough dragons are prepared, the human policy sends the command “scout the map”, then the dragons fly around the map, find and attack the enemies.

Figure 13: Phase 1: Build towers. The human player provides the command “build a guard tower” at every time step.
Figure 14: Phase 2: Build dragons. The human player first gives the command “build a workshop”, after the workshop is built, the human player changes the command to “build dragon”.

Figure 15: Phase 3: Find the enemies. The human player keeps giving the command “scout the map”.