5-2021

Action selection for composable modular deep reinforcement learning

Vaibhav GUPTA

Daksh ANAND

Praveen PARUCHURI

Akshat KUMAR
Singapore Management University, akshatkumar@smu.edu.sg

Citation

GUPTA, Vaibhav; ANAND, Daksh; PARUCHURI, Praveen; and KUMAR, Akshat. Action selection for composable modular deep reinforcement learning. (2021). *Conference of the International Conference on Autonomous Agents and Multiagent Systems, Virtual Online, May 3-7*. 565-573.
Available at: https://ink.library.smu.edu.sg/sis_research/6900

Action Selection For Composable Modular Deep Reinforcement Learning

Conference Paper · May 2021

CITATIONS
0

READS
168

4 authors, including:

Praveen Paruchuri
International Institute of Information Technology, Hyderabad
69 PUBLICATIONS **1,254** CITATIONS

SEE PROFILE

Action Selection For Composable Modular Deep Reinforcement Learning

Vaibhav Gupta*
IIIT, Hyderabad
gupta.vaibhav@research.iiit.ac.in

Daksh Anand*
IIIT, Hyderabad
daksh.anand@research.iiit.ac.in

Praveen Paruchuri†
IIIT, Hyderabad
praveen.p@iiit.ac.in

Akshat Kumar†
Singapore Management University
akshatkumar@smu.edu.sg

ABSTRACT

In modular reinforcement learning (MRL), a complex decision making problem is decomposed into multiple simpler subproblems each solved by a separate module. Often, these subproblems have conflicting goals, and incomparable reward scales. A composable decision making architecture requires that even the modules authored separately with possibly misaligned reward scales can be combined coherently. An arbitrator should consider different module's action preferences to learn effective global action selection. We present a novel framework called GRACIAS that assigns fine-grained importance to the different modules based on their relevance in a given state, and enables composable decision making based on modern deep RL methods such as deep deterministic policy gradient (DDPG) and deep Q-learning. We provide insights into the convergence properties of GRACIAS and also show that previous MRL algorithms reduce to special cases of our framework. We experimentally demonstrate on several standard MRL domains that our approach works significantly better than the previous MRL methods, and is highly robust to incomparable reward scales. Our framework extends MRL to complex Atari games such as Qbert, and has a better learning curve than the conventional RL algorithms.

KEYWORDS

Reinforcement Learning; Coordination and Control; Deep Learning

ACM Reference Format:
Vaibhav Gupta, Daksh Anand, Praveen Paruchuri, and Akshat Kumar. 2021. Action Selection For Composable Modular Deep Reinforcement Learning. In *Proc. of the 20th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2021), Online, May 3–7, 2021*, IFAAMAS, 9 pages.

1 INTRODUCTION

In a reinforcement learning (RL) setting, an agent learns to make decisions by repeatedly interacting with the environment to optimize the long term reward [26]. Thanks to several advances in model-free RL algorithms [13, 18], RL has been successfully applied to several challenging domains in robotics [8], complex games [22, 31], and industrial control [12]. Despite these advances, challenges such as high sample complexity remain when scaling RL algorithms to problems with large state spaces [8]. Fortunately, in several domains, a significant structure exists that can be exploited to yield scalable and sample efficient solution approaches. In hierarchical RL (HRL), a complex problem is formulated using a hierarchy of simpler subproblems via temporal decomposition of the problem and modeling intermediate tasks as higher-level actions [2, 27].

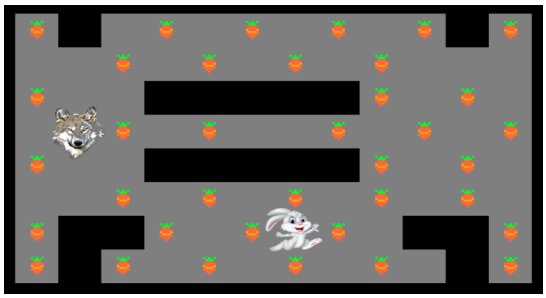


Figure 1: Bunny World Domain

In several settings, it is not possible to extract a hierarchy among different subtasks for a given problem. Modular reinforcement learning (MRL) has been proposed to model settings where an agent's complex decision making problem is decomposed into multiple simpler subproblems each solved by a separate *module* [10, 19, 24]. Each module learns (concurrently with other modules) to solve a simpler subproblem using RL. Although modules solve different subproblems, they are still linked as they share the action taken by the agent. The MRL setting enables modeling problems where modules may have different and possibly conflicting goals. For example, in the popular Bunny World domain [24, 25] shown in figure 1, an agent (bunny) needs to avoid a predator and simultaneously eat all the food particles in the environment. This task can be decomposed into two modules: a) module that learns to collect the food particles, and b) module whose task is to avoid the predator. Each module only deals with the observations and rewards relevant to its subtask.

A key benefit of such a modular decomposition is that each module solves a problem with a smaller state space (e.g., predator module can ignore food locations), which helps address the *curse of dimensionality* in RL [24]. Furthermore, it helps to address the issue of structural credit assignment [1], as each module receives a reward specific to its local subgoal. Such module specific rewards

*Equal Contribution
†Equal advising.

Proc. of the 20th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2021), U. Endriss, A. Nowé, F. Dignum, A. Lomuscio (eds.), May 3–7, 2021, Online.

provide a clearer signal about the contribution of different actions to the individual subtasks as opposed to a single global reward signal in the standard monolithic RL setting. Such improved credit assignment helps in faster convergence of the learning algorithm leading to lower sample complexity and a better learning curve, which we also observe empirically. As highlighted in [23, 24], another major benefit of MRL is towards enabling learning-based software engineering whereby even the modules having separate authors can be composed together to solve a larger and complex learning problem. It also enables module reuse across different projects.

The modules in MRL, while learning concurrently, aim to solve different goals that may be conflicting in nature [10]. Therefore, the action preferences can be different across modules. One key challenge in MRL is to learn a policy for an *arbitrator* (AR), that takes as input the action preferences of different modules and selects the action to be taken by the agent, which is then shared with all the modules. This is called the action selection problem in MRL [10] which is challenging since different techniques for combining the action preferences of different modules give rise to a multitude of policies. Another key challenge for effective action selection is that although the module specific reward signals are consistent within a module, they might be incomparable to the rewards used in other modules [24]. Designing comparable rewards across all the modules is burdensome for domain experts as defining a suitable reward function is known to be challenging even in standard RL [7, 17]. This further hinders the modularity and composability in MRL where different modules can be authored separately (e.g., in a software engineering context [23, 24]). Therefore, the arbitrator's policy should be able to select effective actions even in these challenging circumstances.

Contributions: Our main contributions are as follows:

(1) We develop a novel framework, GRACIAS (**G**lobal **R**eward **A**ware **C**omposition of **I**ndividual **A**ction **U**tilitie**S**), based on deep RL algorithms. It enables the arbitrator (AR) to assign appropriate relative importance to the modules in different states of the environment to address the action selection problem effectively. GRACIAS also allows concurrent learning of action-value functions of different modules using standard deep RL methods such as Double-DQN.
(2) We provide insights into convergence properties of GRACIAS by exploiting connections to the two timescale stochastic approximation [4]. We also show that previous approaches such as GM algorithms [10, 19], and Arbi-Q [24] reduce to special cases of our framework.
(3) Empirically, we demonstrate the effectiveness of our approach on two well known MRL domains, namely "Bunny World" [24] and "Racetrack" [19]. Besides these, we also use a complex domain named Qbert (from the Arcade Learning Environment [3]) and propose to use it as a new benchmark for performance comparisons in MRL.

We compare our approach with popular MRL techniques GM-Q and Arbi-Q. Our approach is highly robust to different reward scales across modules, whereas previous approaches significantly degrade in performance. On Qbert, when compared against Double-DQN, our approach provides similar solution quality while having better sample complexity. To further illustrate composable learning, we pre-train some modules which are made available to the agent beforehand. Empirically, GRACIAS fully leverages the advantage of pre-trained modules, has faster convergence, and significantly outperforms existing approaches. In contrast, conventional RL methods are not able to leverage the availability of such pre-trained modules.

2 RELATED WORK

In the command arbitration paradigm [6, 9, 10, 24], an arbitration policy is defined that selects a single module to execute the agent's action in a given state. This module obtains complete control of the agent in that state, and its most preferred action is executed by the agent. The MRL literature has witnessed a variety of *command arbitration* policies. Top-Q [10] uses a simple arbitration policy, where each module suggests the value of its most preferred action, and the module with the highest action-value gets selected. W-learning [6, 9] selects the module that would suffer the most if its most preferred action is not executed. Arbi-Q models an intelligent arbitrator that tries to maximize the global reward by selecting the most relevant module in a given state [24]. All the command arbitration algorithms give complete control to a single module in any given state. As highlighted in previous works [19], these approaches fail to identify *compromise* actions (i.e., actions agreeable for all the modules) and hence, generally lead to suboptimal policies.

Command fusion techniques [5, 20, 21] try to address this issue by performing a combination (like average) of the most preferred action of each module. However, these techniques converge to suboptimal policies that suffer from oscillations and instability [11]. GM algorithms [10] address this issue by considering the action preferences of all the modules. In particular, for each action, the cumulative value (or expected long term reward) is obtained by adding action values for this action across all the modules. The action with the highest aggregate is executed by the agent. This procedure enables the agent to find the action which can be reasonable for all the modules. One of the major drawbacks of GM-algorithms is that action selection is completely based on the individual modules' action-values, without considering the global reward. It makes these algorithms sensitive to the modules' reward scales and necessitates careful crafting of reward functions of each module.

GRACIAS addresses all the major concerns mentioned above. Unlike GM algorithms, it is not sensitive to the modules' reward scales and considers the global reward signals. Simultaneously, unlike command arbitration algorithms, it considers the individual action preferences of the modules and can find actions reasonable from every modules' perspective.

3 MODULAR REINFORCEMENT LEARNING

The standard RL problem is modeled using a Markov decision process (MDP). An MDP is defined using a tuple $\langle S, A, T, r, \gamma \rangle$. An agent can be in one of the states $s \in S$, and takes an action $a \in A$. As a result of the action, the agent transitions stochastically to a new state s' with probability $\Pr(s'|s, a) = T(s, a, s')$ and receives a reward $r(s, a)$. We assume an infinite-horizon setting with reward discounting factor $\gamma < 1$. In the model-free RL setting, an agent only observes the tuple $\langle s, a, s', r \rangle$ after each interaction with the environment. The policy of the agent is denoted using $\pi(a|s)$ denoting the probability $\Pr(a|s)$. Let the discounted future return be

denoted by $R_t = \sum_{k=0}^{\infty} \gamma^k r_{k+t}$. The value function induced by policy π is denoted as:

$$V^{\pi}(s_t) = \mathbb{E}_{s_{t+1:\infty}, a_{t:\infty}}\left[R_t | s_t, a_t\right] \tag{1}$$

The action-value function is denoted as:

$$Q^{\pi}(s_t, a_t) = \mathbb{E}_{s_{t+1:\infty}, a_{t+1:\infty}}\left[R_t | s_t, a_t\right] \tag{2}$$

The goal is to find the best policy π to maximize the value for the starting belief b_0:

$$J(\pi) = \sum_{s} b_0(s) V^{\pi}(s) \tag{3}$$

Modular RL: We follow the MRL framework as presented in [24]. The framework assumes that there are N modules, each of which addresses a smaller subproblem of the main RL task. To enable local learning of action-value functions for modules, we assume that the environment generates $N+1$ reward signals $\langle r, r_1, \ldots, r_N \rangle$ where r is the original global reward signal, and r_i is the local reward signal for module i $\forall_{i=1\ldots N}$. The global reward r quantifies the actual performance of the agent for the given state and action. Our goal is still to find the policy that maximizes the value function defined using this global reward r analogous to (3). Each r_i quantifies the agent's performance w.r.t. the specific subtask addressed by module i. Notice that we *do not* require that $r = \sum_{i=1}^{N} r_i$. Therefore, global rewards need not be on the same scale as module rewards. Although modules may receive different rewards, they are still coupled as they share the action taken. Given the agent's policy π, we can also define the action-value function for each module i as:

$$Q_i^{\pi}(s_t, a_t) = \mathbb{E}\left[r_i(s_t, a_t) + \gamma Q_i^{\pi}(s_{t+1}, \pi(s_{t+1}))\right] \tag{4}$$

A key benefit of MRL is that since each module addresses a smaller subtask, it may be sufficient to define their local action-value functions using state abstraction $\phi_i(s)$, which may have smaller dimensionality than the original state s [10, 24]. Furthermore, having access to local rewards r_i enables better structural credit assignment as it provides a clearer signal about the contribution of the different actions to the individual subtasks as opposed to a single global reward signal r [1]. It can make the learning sample efficient, resulting in a better learning curve than monolithic RL algorithms.

As in HRL, some domain expertise may be required to define the subtasks addressed by different modules, and the local reward and state abstraction functions. However, different modules are allowed to interfere with each other and pursue different subgoals, and they can have incomparable reward scales. Therefore, the overhead for domain experts is relatively low. We also show that extracting such a modular structure is possible even for complex Atari games such as Qbert. In settings, such as in Qbert, hand coding of state abstraction ϕ_i is not possible as the state represents the game screen (i.e., pixels). In such settings, state abstraction can also be automatically learned using convolutional neural nets.

4 OUR APPROACH

Our goal is to find the agent's policy π that allows the agent to select actions to optimize the objective (3). However, if we directly define and optimize π, it is not clear how to exploit the information learned by the different modules. Therefore, we introduce *arbitrator*

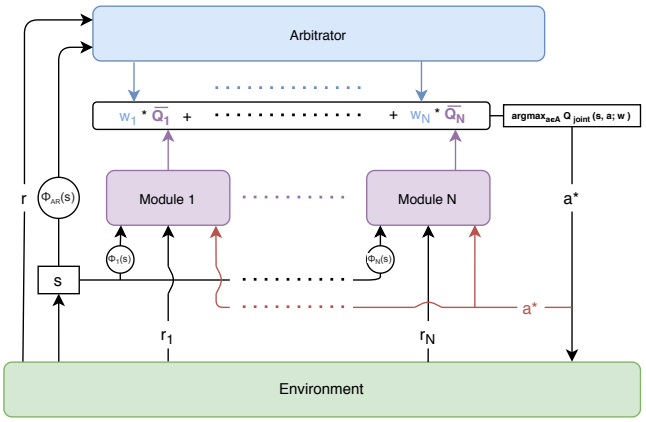


Figure 2: MRL and Arbitration Architecture. The arbitrator observes abstracted environment state $\phi_{AR}(s)$, global reward r, and modules' state-action values Q_i ($\overline{Q}_i$ denotes vector of state-action values, one for each action). The arbitrator performs action selection to execute the action $a^{\star}$ in the environment (details in text). This action is also communicated to each module. Each module observes the state features $\phi_i(s)$ and the local reward signal r_i for their specific subtask, and update their Q-values following Q-learning.

AR, which itself is a reinforcement learning module, and its policy defines the action to be taken by the agent at each time step.

4.1 Arbitrator Architecture

GRACIAS uses an *arbitrator* (AR) to which each module communicates its action preferences in the form of action-values Q_i. Figure 2 shows the information flow from the modules and environment to the AR. The arbitrator observes the global reward signal r and selects the action to maximize the expected long term reward (3). It aims to learn the relative importance of each module i in a given state s, denoted by appropriate weights. Using these weights, the joint Q-value (Q_{joint}) is computed by taking a linear combination of the action-values of the individual modules. Formally, the state, action spaces and reward signal for AR are defined as:

- **State**: The AR observes the environment state s. In several settings, it may be possible to define a state abstraction function $\phi_{AR}(s)$ that has lower dimensionality than s, and is still sufficient for making decisions by the AR. However, this is not required; in complex domains such as Qbert, the AR can learn ϕ_{AR} using standard convolution neural networks (CNNs).
- **Actions**: The action space of AR is *continuous* and with N-dimensions, or $\boldsymbol{w} \in [0,1]^N$ (where $\boldsymbol{w}$ denotes AR's action and N is the number of modules). There is one action component $w_i \in [0,1]$ for each module i. Intuitively, w_i corresponds to the *weight* or importance assigned to module i. We can enforce additional constraints on $\boldsymbol{w}$ such as normalization (or $\sum_i w_i = 1$).
- **Reward and action selection**: Notice that AR's action does not directly provide the action taken by the agent in the environment. Similarly, it is not clear what should be the reward signal for the continuous action $\boldsymbol{w}$ of AR. To allow

AR to connect the modules' preferences and global reward, the (environment) action selection mechanism is defined as:

(1) The AR computes $\mathrm{Q}_{\text{joint}}(s, a; \boldsymbol{w}) = \sum_{i=1}^{N} w_i Q_i(\phi_i(s), a)\ \forall a \in A$. Notice that a is the environment action. $\mathrm{Q}_{\text{joint}}$ is a weighted combination of module action-values with weights given by the AR action $\boldsymbol{w}$.
(2) The action selected to execute in the environment is $a^\star \leftarrow \text{argmax}_{a \in A}\, \mathrm{Q}_{\text{joint}}(s, a; \boldsymbol{w})$
(3) The action $a^\star$ is then executed in the environment, and the global reward r is the reward corresponding to AR's action $\boldsymbol{w}$ or $r_{\text{AR}}(s, \boldsymbol{w}) = r(s, a^\star)$.
(4) After taking action $a^\star$, the environment transitions to a new state s'.

Arbitrator training: Notice that the above action selection process takes module preferences into account, by computing $\mathrm{Q}_{\text{joint}}$ as the weighted combination of the module state-action values, and also connects to the global reward signal r. Since the reward r_{AR} given to the arbitrator is essentially the global reward signal, optimizing arbitrator's actions $\boldsymbol{w}$ would also optimize the global objective (3). The arbitrator would therefore learn to weigh the different modules appropriately in different states such that it results in the optimization of the long term global reward.

Using tuples $\langle s, \boldsymbol{w}, s', r_{\text{AR}} \rangle$, we can now train the arbitrator policy with standard deep RL techniques like DDPG [15] or SAC [8] that are able to optimize continuous actions. Thus, our novel action selection mechanism can exploit advances in deep RL methods, and also exploits the modular nature of the underlying problem. Such connections were not developed in previous MRL methods such as GM-Q and Arbi-Q.

For our experiments, arbitrator is trained using DDPG approach with policy $\mu(\cdot; \theta)$, and a critic $Q(\cdot; \theta')$ approximating the action value function of the arbitrator. The gradient of the objective w.r.t. policy parameters θ is given as [15]:

$$\nabla_\theta J = \mathbb{E}_{s_t \sim \rho}\left[\nabla_{\boldsymbol{w}} Q(s, \boldsymbol{w}; \theta')|_{s=s_t, \boldsymbol{w}=\mu(s_t)} \nabla_\theta \mu(s; \theta)|_{s=s_t}\right] \quad (5)$$

where ρ is the state-visitation distribution; critic Q can also be trained using samples as shown in [15]. We use a publicly available implementation of DDPG [33].

4.2 Module Architecture

Each interaction with the environment generates an experience tuple $\langle \phi_i(s), a^\star, \phi_i(s'), r_i \rangle$ for module i as shown in figure 2, where a is the action chosen by the arbitrator using the action selection mechanism in section 4.1. We assume that each module implements an RL approach such as Q-learning or SARSA [26], to learn its local state-action value function Q_i. Notice that we allow the arbitrator and the modules to learn concurrently. We will discuss later how to make such concurrent learning stable by appropriately choosing learning rates for the different RL modules by exploiting connections to the two timescale stochastic approximation [4].

For our implementation, we chose Q-learning that updates the module's action-value function as:

$$Q_i(\phi_i(s_t), a_t) \leftarrow (1 - \alpha_i^t) Q_i(\phi_i(s_t), a_t) + \alpha_i^t \left[r_i + \gamma \max_{a \in A} Q_i(\phi_i(s_{t+1}), a)\right] \quad (6)$$

where α_i^t is a learning rate that decays over time. Such updates can be done in parallel for each module, and have been used in previous MRL approaches as well [19]. Notice that this local Q-learning gives the *illusion of control* to module i since Q_i is updated under the assumption that in the future states, the agent's policy will be optimal from the perspective of module i (indicated by $\max_{a \in A} Q_i$). This is an approximate way of updating action-value function of a module. It has been noted that SARSA is a better choice for updating a module's Q-values as SARSA is on-policy and does not overestimate future returns unlike Q-learning [19, 25]. However, in our setting, we chose Q-learning for modules for the following reasons.

First, Q-learning is an off-policy algorithm. Therefore, it is compatible with the experience replay mechanism, an integral part of several modern deep RL algorithms such as DQN [16]. In contrast, since SARSA is on-policy, experience replay is not compatible with it in a straightforward fashion. Second, under some standard regularity assumptions, Q-learning updates in equation (6) will converge to locally greedy estimates regardless of the arbitrator's policy [19, 29]. This makes learning stable even with multiple concurrently learning modules. Third, previous MRL approaches that used Q-learning, like GM-Q [10], did not employ an intelligent arbitrator. Only the modules' action-values decided the agent's policy. It resulted in suboptimal policies due to the overestimation caused by the *illusion of control*. In contrast, since GRACIAS's arbitrator directly optimizes the global reward, it can assign appropriate importance to each module to learn good quality policies. Empirically, we observed that the arbitrator in our case, was able to accurately learn how to weight the Q-values of the different modules to maximize the long term global reward.

Algorithm 1 GRACIAS

Defines:
N ▷ The number of modules
Requires:
Arbitrator ▷ Central arbitrator AR
Modules ▷ Array containing N modules
$\phi_{\text{AR}}(s)$ ▷ State abstraction function of the arbitrator
$\phi_i(s)$ ▷ State abstraction function of module i

for each episode **do**
  Receive the initial state s
  for each time step **do**
    $s_{\text{AR}} \leftarrow \phi_{\text{AR}}(s)$ ▷ Abstracted state for the arbitrator
    $\boldsymbol{w} \leftarrow$ Arbitrator.getWeights(s_{AR})
    $Q_i \leftarrow$ Modules$[i]$.getActionValues($\phi_i(s)$) $\quad \forall i = 1 : N$
    $\mathrm{Q}_{\text{joint}}(s, a; \boldsymbol{w}) = \sum_{i=1}^{N} w_i Q_i(\phi_i(s), a) \quad \forall a \in A$
    $a^\star \leftarrow \text{argmax}_a\, \mathrm{Q}_{\text{joint}}(s, a; \boldsymbol{w})$
    Execute $a^\star$, observe rewards $\langle r, r_i \forall i = 1 : N \rangle$, state s'
    Arbitrator.update($\phi_{\text{AR}}(s)$, $\boldsymbol{w}$, r, $\phi_{\text{AR}}(s')$)
    for $i \in \{1, \ldots, N\}$ **do**
      Modules$[i]$.update($\phi_i(s)$, $a^\star$, r_i, $\phi_i(s')$)
    end for
    $s \leftarrow s'$
  end for
end for

Pre-trained modules: To enable a truly modular and composable RL, we also allow incorporating pre-trained modules in our framework. The Q-values of such modules are not updated during the learning process. Since GRACIAS does not require comparable rewards across the different modules, our arbitrator can still learn to weigh the state-action values of such pre-trained modules. A key benefit is that, in some settings, a pre-trained module can be reused across multiple RL tasks. This should make the learning faster and sample efficient. We also show the effects of such pre-training empirically which confirms these observations.

Algorithm 1 provides the high level design of GRACIAS. The modules and the arbitrator have their individual state abstraction functions. In a given state s, Arbitrator assigns a relative importance w_i to the action-values (Q_i) of each module i. For each action a, $\mathrm{Q}_{\text{joint}}(s, a; \boldsymbol{w})$ is calculated as a linear weighted sum of the action-value of the individual modules. The agent executes the action $a^\star$ having the maximum aggregate value. The agent then observes the $N+1$ reward signals $\langle r, r_i \forall i = 1:N \rangle$ and the next state s'. Using their respective experience tuples, the Arbitrator and the modules can be trained using the standard deep RL techniques. We use DDPG and Double-DQN for training the Arbitrator and the modules respectively. For completeness the updates for all the components, and two routines: Arbitrator.getWeights and Modules$[i]$.getActionValues are provided in Appendix B.

4.3 Connections with GM-Q and Arbi-Q

Algorithm 2 Action selection in MRL

Defines:

Π_{AR} ▷ Policy of the arbitrator

Q_i ▷ Q function of module i

1: $\boldsymbol{w} \leftarrow \Pi_{\text{AR}}(s)$

2: $\mathrm{Q}_{\text{joint}}(s, a) \leftarrow \sum_{i=1}^{N} w_i \cdot Q_i(s, a); \quad \forall a \in A$

3: $a^\star \leftarrow \underset{a}{\text{argmax}}\, \mathrm{Q}_{\text{joint}}(s, a)$ ▷ Action to execute in environment

We next show that the policies executed by the GM algorithms like GM-Q as well as the command arbitration algorithms like Arbi-Q can be reduced to special cases of GRACIAS. Algorithm 2 highlights the key steps in the action selection mechanisms of Arbi-Q, GRACIAS and GM algorithms. At an abstract level, all the three algorithms assign weights to the modules to compute $\mathrm{Q}_{\text{joint}}$, the joint value function, which is then used to select the agent's action. These algorithms differ in the possible values that can be assigned to the weight vector $\boldsymbol{w}$ which in turn is defined by the action space of the arbitrator. Intuitively, we show that the range of possible $\boldsymbol{w}$ for Arbi-Q and GM algorithms are subsets of possible $\boldsymbol{w}$ in GRACIAS.

Proposition 1. *Arbi-Q is a special case of* GRACIAS.

Proposition 2. *GM algorithms are a special case of* GRACIAS.

Proofs are provided in appendix A.1 and A.2 in supplementary.

4.4 Convergence Properties

Notice that there are multiple components that are learning concurrently (arbitrator and N modules). It might lead to unstable and non-stationary learning. Indeed, this seems to be the case for the arbitrator's learning. The reward function r_{AR} for the arbitrator is not stationary (especially in the initial learning epochs) as r_{AR} depends on the modules' Q-values, which themselves are getting updated as learning proceeds. Therefore, it is not clear why such interleaved learning will converge. To provide clarity into this learning process, we connect it to the two timescales stochastic approximation that provides insights regarding learning rate schedules of different components and other regularity assumptions necessary for convergence [4, Chapter 6].

Two timescale approximation involves x and y as variables ($x, y \in \mathbb{R}^d$) which are updated in the following fashion iteratively [4, Chapter 6]:

$$x_{t+1} = x_t + \eta_1(t)\left[h(x_t, y_t) + \epsilon_{t+1}^{(1)}\right] \tag{7}$$

$$y_{t+1} = y_t + \eta_2(t)\left[g(x_t, y_t) + \epsilon_{t+1}^{(2)}\right] \tag{8}$$

where $\eta_1(t)$ and $\eta_2(t)$ are the learning rates, and ϵ is the noise. Notice that in this scheme, updates of both x and y are interleaved. As noted in [4, Chapter 6], the intuition for these updates is as follows: The setup involves an iterative algorithm. Let us assume that the outer loop of the algorithm corresponds to updating the arbitrator's policy. For the current parameters θ_t of arbitrator's policy, we run another subroutine that updates action-values of each module until (near) convergence. Using the converged Q-values of modules, a stable estimate of r_{AR} can be extracted, and we can update the arbitrator's policy using policy gradient in equation (5). The theory of two timescale approximation shows that we can achieve the same effect by running the outer loop (for arbitrator policy update) and inner loop (to update Q-values of modules) *concurrently*, but on *different* timescales. Essentially, using higher learning rate for modules than the arbitrator, the arbitrator perceives modules' Q-values to be almost converged, and modules perceive arbitrator's policy to be quasi-static.

Let $\eta_1(t)$ and $\eta_2(t)$ be the learning rates for module Q-value updates and the arbitrator policy update respectively at time t. The key assumption required for this two timescale approximation to work is:

Assumption 1.

$$\sum_t \eta_1(t) = \sum_t \eta_2(t) = \infty, \sum_t (\eta_1(t)^2 + \eta_2(t)^2) < \infty, \frac{\eta_2(t)}{\eta_1(t)} \to 0 \tag{9}$$

Notice that the condition $\frac{\eta_2(t)}{\eta_1(t)} \to 0$ implies that rate η_1 (for module learning) is higher than η_2 (for arbitrator). Let x denote $\boldsymbol{Q}$ (or the vector of Q-values for all modules, and states and actions), and y denote θ, the arbitrator policy. For simplicity, we assume tabular action-value function for modules, proof can be extended to parametric policies also [28]. We state the below result.

Proposition 3. *Under assumption 1 and other standard assumptions for the stability of iterates and bounded noise [4, Chapter 6], the iterates $(\theta_t, \boldsymbol{Q}_t)$ converge to a fixed point (local optima) almost surely.*

We show additional results in the Appendix A.3 to prove the above proposition. The connections to two timescale approximation provides useful insights (e.g., about relation between learning rates

of different components), and shows convergence properties under some assumptions. In practical implementation, there are additional details which we did not include for clarity of exposition (e.g., there is a critic which is also learned for the arbitrator, Q-values for modules are approximated using function approximators). The above two timescale approximation analysis can also be extended to such multi-timescale approximation also [28].

5 EXPERIMENTS

We showcase the effectiveness of GRACIAS on two well known MRL domains, namely "Bunny World" [24, 25] and "Racetrack" [19]. Besides these, we also use a complex Atari domain Qbert and propose it as a new MRL benchmark. We compare against popular MRL methods GM-Q and Arbi-Q as baselines. We also use Double-DQN (DDQN) algorithm [30] as the standard deep RL baseline.

5.1 Hyperparameters and setup

Appendix G (in supplementary) describes all the three domains and their respective subtasks in detail. The arbitrator in GRACIAS can use any RL algorithm that can optimize continuous actions. For our experiments, the arbitrator is trained using DDPG algorithm [14]. The modules for all the approaches (GM-Q, Arbi-Q, and GRACIAS) use the Double-DQN [30] algorithm to learn their subtasks. All the hyperparameters and components such as network architecture, state abstraction functions and experience replay memory for the modules were kept the same across all the algorithms. For Qbert, the modules use the same network architecture as proposed in [16]. The network architecture used in the Racetrack and Bunny World is explained in the Appendix E (supplementary).

All the agents are trained using standard exploration techniques. The DDQN baseline uses the epsilon-greedy exploration policy [32]. Similarly, GM-Q computes Q_{joint} and uses the epsilon greedy policy to select the action. In GRACIAS, the arbitrator outputs the modules' weights, and adds a uniform random noise to these weights with epsilon probability. The Q_{joint} is calculated using these weights, and the action with the highest aggregate is executed.

We perform a set of four experiments (sections 5.2-5.5), each highlighting a specific aspect of our algorithm. Five different instances of each algorithm were trained using different random seeds. The x-axis represents the training epochs, while the y-axis represents the average episode reward over 15 trials based on the fixed policy at that point (averaged over 5 random seeds). The solid curves correspond to the mean, and the shaded regions capture the minimum and maximum returns over five trials (plots with min/max returns are shown in Appendix F). A running average of 15 was used in plotting the results.

5.2 Incomparable Reward Scales Setting

Figure 3 shows the performance comparison among different algorithms when the modules have incomparable or misaligned reward scales. As defined in Appendix C, incomparable rewards imply that rewards for individual modules are internally consistent, but are incomparable to other modules' reward scales. Designing comparable rewards across all the modules is burdensome for domain experts [24] and becomes even harder with a large number of modules. We generated several candidate reward settings (shown in tables 1, 2, 3 in supplementary). Only the GM-Q algorithm is known to be highly sensitive to modules' reward scales; GRACIAS and Arbi-Q are robust to different reward scales (also shown in section 5.5). Therefore, we selected a reward setting (indexed #5 in table 1 and indexed #2 in table 2 and 3 in supplementary) for which GM-Q did not perform well.

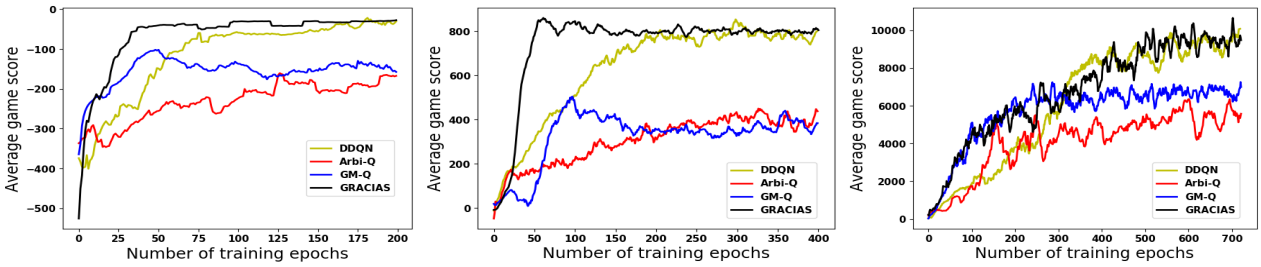
We observe that GM-Q cannot adapt to the misaligned reward scale and converges to a suboptimal solution. Although both Arbi-Q and GRACIAS are robust to reward scales, Arbi-Q has a slower learning curve and converges to a suboptimal solution. In contrast, GRACIAS can learn a good policy and converge faster than other approaches. The graphs also show that even with a slow learning curve DDQN baseline can eventually converge to a similar solution as GRACIAS. We present below some insights for these results.

GM algorithms are unable to learn the importance of different modules and assign equal weight to them. With misaligned reward scales, some modules might overpower the contribution of other modules in deciding the joint policy. For instance, in the Bunny World domain, if the rewards of the *CollectFood* module are much larger compared to those of the *AvoidPredator* module (which is the case for the chosen reward setting), then the agent will tend to ignore the action-values (and hence the preferences) of the *AvoidPredator* module. Therefore, the overall policy will be unable to dodge the predator, and the agent will die quickly. As shown in the graphs, such behavior causes the agent to converge to a suboptimal policy.

As discussed earlier, Arbi-Q transfers the agent's complete control to a single module and fails to find actions reasonable to all modules. For instance, in the Bunny World domain, the predator follows the agent almost always; hence the agent can not ignore it in any state. If the control is given to the food module in any state, the agent will completely ignore the predator, which can lead to its death. On the other hand, if the control always remains with the predator module, the agent ignores collecting food. This behavior can be observed in the graphs where Arbi-Q converges to suboptimal policies. It also has a much slower learning curve compared to other algorithms. Similar trends can also be observed in the experiments conducted in figures 2 & 3 of [24]. Note that the figures for GM-SARSA and Arbi-Q have different scales on the x-axis. Arbi-Q is slower than GM-SARSA by orders of magnitude, the reasons for which were not discussed in that paper.

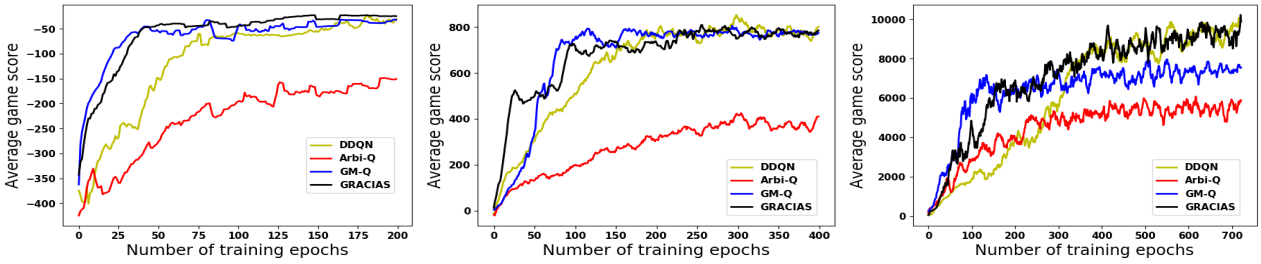
GRACIAS explores different weights for the modules and learns to find a weight $\boldsymbol{w}$ that results in a good action selection that also optimizes the global reward. By assigning appropriate importance to different modules, GRACIAS can handle cases where the rewards of some of the modules are very high and prevent them from dominating the joint policy. Results in figure 3 for different domains validate this observation. Experiments show that our approach adapts well even to misaligned reward scales and converges to a good quality policy.

In DDQN, a monolithic agent deals with the joint state space containing complete information about all the subtasks. State space grows exponentially in the number of state features due to the *curse of dimensionality*, resulting in slower learning. However, the availability of all this information also enables DDQN to converge to a good policy eventually. On the other hand, in MRL, each module solves a problem with a smaller state space resulting in better



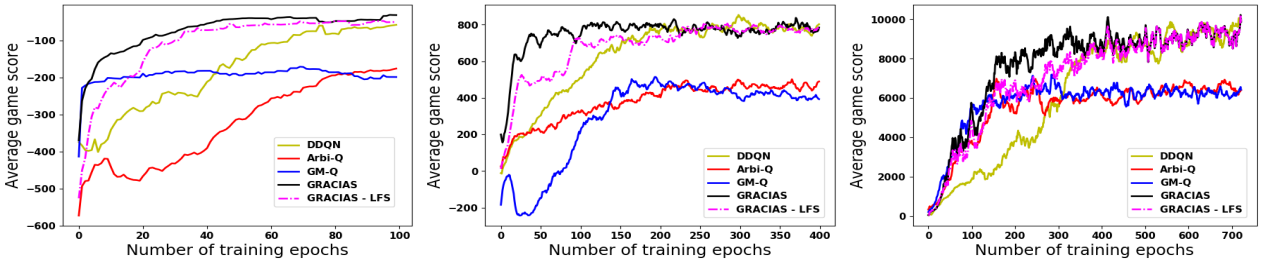
(a) Racetrack (b) Bunny World (c) Qbert

Figure 3: Training curves of different algorithms having modules with incomparable reward scales



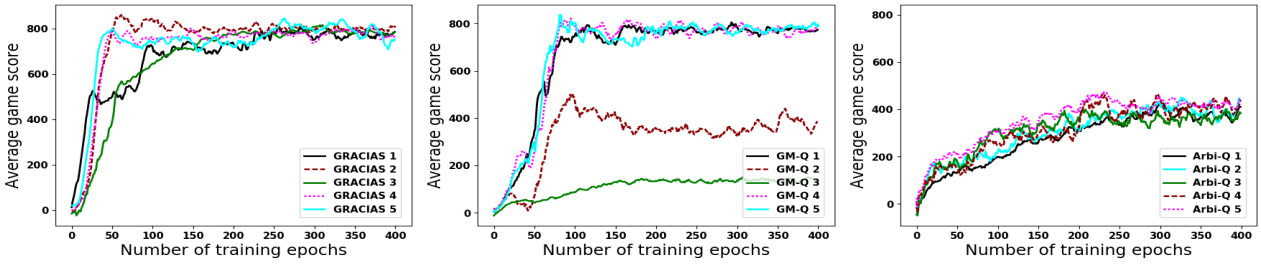
(a) Racetrack (b) Bunny World (c) Qbert

Figure 4: Training curves of different algorithms having modules with comparable reward scales



(a) Racetrack (b) Bunny World (c) Qbert

Figure 5: Training curves of different algorithms when one module is pre-trained. All components of DDQN and GRACIAS-LFS are *"Learning From Scratch"* and have been added for comparison.



(a) GRACIAS (b) GM-Q (c) Arbi-Q

Figure 6: Training curves of different MRL algorithms with different reward scales of the modules on Bunny World

learning curves. Even for Qbert, where each module's state space is the same as DDQN, GRACIAS has better convergence than DDQN as each module receives its subtask specific reward signal, which results in better credit assignment and faster learning.

5.3 Comparable Reward Scale Setting

Figure 4 shows the performance comparison of different algorithms when all the modules have comparable reward scales. As discussed above, only GM-Q is sensitive to the reward scales of the modules. In Appendix C (supplementary), we do a grid search over the reward scales of different modules (tables 1-3). We call a reward setting comparable using which GM-Q can achieve a solution whose quality is similar to that achieved by DDQN on convergence. This set of comparable rewards was then used for all the algorithms.

As evident from the graphs, both GRACIAS and GM-Q have better learning curves and a faster rate of convergence as compared to DDQN. Arbi-Q has a slow learning curve and converges to suboptimal policies. We can also see that despite the slower learning curve, DDQN baseline is able to achieve good performance eventually. All these observations are in line with the previous set of experiments.

A key observation is that even after prolonged training, GM-Q is unable to learn a good policy in Qbert. We believe that this happens because the GM algorithms fail to assign state-specific importance to modules. Different modules fulfill different subtasks, and therefore, their importance across states might vary. Hence, assigning a fixed importance to a module across all the states might hamper the agent's performance. However, in some domains like the Bunny World, this phenomenon might not cause any significant performance degradation. In the Bunny World, the agent has to dodge the predator and consume all the food particles. It can not afford to ignore the predator since that might cause its death. Moreover, the predator module is indifferent between all the actions which do not lead to its immediate death. An optimal policy would choose from only these actions. This can be achieved by giving importance to the $CollectFood$ module that is just enough to break the tie between these actions. Any static scaling that assigns adequate importance to the $AvoidPredator$ module satisfies this requirement. Therefore a single scaling might work equally well in Bunny World.

We also notice that GRACIAS is slightly slower than GM-Q in the initial phases. It is because the GRACIAS' arbitrator has to learn the weights of different modules from scratch. It does so by exploring different weight assignments $\boldsymbol{w}$ before converging to a good policy. GM-Q on the other hand simply adds up the action values of the modules and does not require any learning.

5.4 Learning with Pre-trained Modules

The modules of a given complex task can be authored by different dedicated experts. To realise the complete potential of MRL, the knowledge from these modules should be composable, and the components usable without any modification. In many cases, a module might have been trained for a particular use case and made available to the agent in a completely different context. An ideal arbitrator would directly incorporate such modules and result in sample efficient training. Therefore, we test the setting wherein some of the modules are pre-trained and made available to the arbitrator beforehand.

Figure 5 shows performance comparisons of different algorithms. Conventional RL techniques for solving the monolithic RL tasks fail to leverage pre-trained modules. The plots for DDQN and GRACIAS ("Learning From Scratch" in the normal setting) have been added for comparisons. As in previous experiments, Arbi-Q converges to suboptimal policies, and GM-Q suffers from the reward scale incomparability and converges to suboptimal solutions. This experiment highlights that GRACIAS can synchronize different modules even when they are at different stages of their individual learning, and fully leverages pre-trained modules for faster convergence. We observe similar performance boost in GRACIAS when all the modules are pre-trained and provided to the agent beforehand, and only the arbitrator's policy is learned (Appendix D, supplementary). Even in this setting, GRACIAS learns a good policy.

5.5 Sensitivity to Modules' Reward Scales

Figure 6 shows the training curves of each MRL algorithm when trained across a spectrum of five different reward scales of the modules on Bunny World. The graphs for Racetrack and Qbert, the reward settings used for this experiment and the process of selecting those rewards is described in Appendix C in supplementary.

We observe that GM-Q is unstable and converges to a lower score on many reward scales. Arbi-Q, although indifferent to the reward scales, consistently has slower learning curves and converges to suboptimal policies. In contrast, GRACIAS is robust and adapts well to even highly skewed reward scales. This robustness of GRACIAS is the chief enabler of the truly modular RL framework allowing modules to be transferred across different systems without re-engineering their reward signals.

6 CONCLUSION

This work is a first step towards making MRL scalable and compatible with modern deep RL approaches. We developed a novel framework called GRACIAS for modular RL to solve complex real world problems by decomposing it into simpler subtasks. GRACIAS allows concurrent learning for such subtasks via separate modules, which may have different subgoals. It can also leverage the knowledge of pre-trained modules. A key enabler of such composable and modular learning is our novel technique to train the arbitrator using deep RL algorithms by considering both the global reward and the modules' action preferences. Empirically, we demonstrate the efficacy of GRACIAS on a range of problems against several MRL algorithms. GRACIAS is able to overcome all the major limitations of the existing MRL approaches and was observed to be highly robust even when the rewards of the modules were incomparable. Thus, our framework provides significant benefits for practical applications of MRL for large scale learning problems.

ACKNOWLEDGMENTS

We thank Kohli Center on Intelligent Systems, International Institute of Information Technology, Hyderabad for the generous support. This research is also supported by the Ministry of Education, Singapore under its MOE Academic Research Funding Tier 2 (Award MOE2018-T2-1-179). Any opinions, findings, and conclusions are those of the author(s) and do not reflect the views of the Ministry of Education, Singapore.

REFERENCES

[1] Adrian K. Agogino and Kagan Turner. 2004. Unifying temporal and structural credit assignment problems. In *International Joint Conference on Autonomous Agents and Multiagent Systems*, Vol. 2. 980–987.

[2] Pierre Luc Bacon, Jean Harb, and Doina Precup. 2017. The option-critic architecture. In *AAAI Conference on Artificial Intelligence*. 1726–1734.

[3] Marc G Bellemare, Yavar Naddaf, Joel Veness, and Michael Bowling. 2013. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research* 47 (2013), 253–279.

[4] Vivek Borkar. 2008. *Stochastic Approximation: A Dynamical Systems Viewpoint*.

[5] Hyunjin Chang and Taeseok Jin. 2013. Command fusion based fuzzy controller design for moving obstacle avoidance of mobile robot. In *Future information communication technology and applications*. Springer, 905–913.

[6] Ivana Dusparic and Vinny Cahill. 2009. Distributed w-learning: Multi-policy optimization in self-organizing systems. In *2009 Third IEEE international conference on self-adaptive and self-organizing systems*. IEEE, 20–29.

[7] Carlos Florensa, David Held, Markus Wulfmeier, Michael Zhang, and Pieter Abbeel. 2017. Reverse curriculum generation for reinforcement learning. *arXiv preprint arXiv:1707.05300* (2017).

[8] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. 2018. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International Conference on Machine Learning*. 2976–2989.

[9] Mark Humphrys. 1995. *W-learning: Competition among selfish Q-learners*. Technical Report. University of Cambridge.

[10] Mark Humphrys. 1996. Action selection methods using reinforcement learning. *From Animals to Animats* 4 (1996), 135–144.

[11] Yoram Koren, Johann Borenstein, et al. 1991. Potential field methods and their inherent limitations for mobile robot navigation.. In *ICRA*, Vol. 2. 1398–1404.

[12] Nevena Lazic, Tyler Lu, Craig Boutilier, Moonkyung Ryu, Eehern Wong, Binz Roy, and Greg Imwalle. 2018. Data center cooling using model-predictive control. In *Advances in Neural Information Processing Systems*. 3814–3823.

[13] Sergey Levine, Chelsea Finn, Trevor Darrell, and Pieter Abbeel. 2016. End-to-end training of deep visuomotor policies. *The Journal of Machine Learning Research* 17, 1 (2016), 1334–1373.

[14] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2015. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971* (2015).

[15] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2016. Continuous control with deep reinforcement learning. In *International Conference on Learning Representations*.

[16] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *nature* 518, 7540 (2015), 529–533.

[17] Andrew Y. Ng, Daishi Harada, and Stuart Russell. 1999. Policy invariance under reward transformations : Theory and application to reward shaping. In *International Conference on Machine Learning*. 278–287.

[18] Martin Riedmiller, Roland Hafner, Thomas Lampe, Michael Neunert, Jonas Degrave, Tom Van de Wiele, Volodymyr Mnih, Nicolas Heess, and Jost Tobias Springenberg. 2018. Learning by playing-solving sparse reward tasks from scratch. *arXiv preprint arXiv:1802.10567* (2018).

[19] Stuart J Russell and Andrew Zimdars. 2003. Q-decomposition for reinforcement learning agents. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*. 656–663.

[20] Alessandro Saffiotti, Kurt Konolige, and Enrique H Ruspini. 1995. A multivalued logic approach to integrating planning and control. *Artificial intelligence* 76, 1-2 (1995), 481–526.

[21] Gregor Schöner and Michael Dose. 1992. A dynamical systems approach to task-level system integration used to plan and control autonomous vehicle motion. *Robotics and Autonomous systems* 10, 4 (1992), 253–267.

[22] D Silver, J Schrittwieser, K Simonyan, I Antonoglou Nature, and Undefined 2017. 2016. Mastering the game of Go without human knowledge. *Nature* 550, 7676 (2016), 354.

[23] Christopher Simpkins, Sooraj Bhat, Charles Isbell, and Michael Mateas. 2008. Towards adaptive programming integrating reinforcement learning into a programming language. In *International Conference on Object-Oriented Programming Systems, Languages, and Applications*. 603–613.

[24] Christopher Simpkins and Charles Isbell. 2019. Composable modular reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 33. 4975–4982.

[25] Nathan Sprague and Dana Ballard. 2003. Multiple-goal reinforcement learning with modular sarsa (0). (2003).

[26] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.

[27] Richard S. Sutton, Doina Precup, and Satinder Singh. 1999. Between MDPs and semi-MDPs: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence* 112, 1 (1999), 181–211.

[28] Chen Tessler, Daniel J. Mankowitz, and Shie Mannor. 2019. Reward constrained policy optimization. In *International Conference on Learning Representations, ICLR 2019*.

[29] John N Tsitsiklis. 1994. Asynchronous Stochastic Approximation and Q-Learning. *Machine Learning* 16, 3 (1994), 185–202.

[30] Hado Van Hasselt, Arthur Guez, and David Silver. 2015. Deep reinforcement learning with double q-learning. *arXiv preprint arXiv:1509.06461* (2015).

[31] Mnih Volodymyr, Kavukcuoglu Koray, Silver David, Rusu Andrei A, Veness Joel, Bellemare Marc G, Graves Alex, Riedmiller Martin, Fidjeland Andreas K, and Ostrovski Georg. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529.

[32] Christopher John Cornish Hellaby Watkins. 1989. Learning from delayed rewards. (1989).

[33] Shangtong Zhang. 2018. Modularized Implementation of Deep RL Algorithms in PyTorch. https://github.com/ShangtongZhang/DeepRL.

Appendix

A Proofs

A.1 Proposition 1

Proposition 1. *Arbi-Q [Simpkins and Isbell (2019)] is a special case of* GRACIAS.

Proof Sketch. In any given state s, the arbitrator in Arbi-Q is restricted to select a single module that decides the agent's action in that state. Essentially, its weight assignment $\boldsymbol{w}$ is a 1-hot vector that assigns one to the selected module. The arbitrator in GRACIAS is free to assign weights from a continuous action space $[0,1]^N$. Since this continuous action space covers the 1-hot vectors, GRACIAS can potentially learn all the weight assignments that Arbi-Q is capable of learning. □

A.2 Proposition 2

Proposition 2. *GM algorithms [Humphrys (1996); Russell and Zimdars (2003)] are a special case of* GRACIAS.

Proof Sketch. Previous work does not explicitly associate an arbitrator with GM algorithms. However, GM algorithms can be assumed to have an arbitrator whose policy is state-agnostic as well as module-agnostic, i.e., it assigns equal and the same weight to all the modules in every state. Essentially $w_i = c \; \forall_{i=1..N}$. Such an arbitrator ignores both the information about the state-space S and the global reward signal r_{AR}.

The arbitrator in GRACIAS, on the other hand, makes use of such information-rich signals. Its continuous action space $\boldsymbol{w} \in [0,1]^N$ also covers the action space of GM's arbitrator. Therefore GRACIAS can potentially learn all the weight assignments that GM algorithms are capable of learning.

□

A.3 Proposition 3

Assumption 1.

$$\sum_t \eta_1(t) = \sum_t \eta_2(t) = \infty, \sum_t (\eta_1(t)^2 + \eta_2(t)^2) < \infty, \frac{\eta_2(t)}{\eta_1(t)} \to 0 \tag{1}$$

Proposition 3. *Under assumption 1 and other standard assumptions for the stability of iterates and bounded noise (Borkar, 2008, Chapter 6), the iterates $(\theta_t, \boldsymbol{Q}_t)$ converge to a fixed point (local optima) almost surely.*

Proof. We follow a similar proof outline as other deep RL methods that follow two timescale approximation, such as Tessler, Mankowitz, and Mannor (2019). We first show two additional results that help prove the overall convergence of arbitrator and module learning. We also note below the standard greedy update for Q-learning in (2).

$$Q_i(\phi_i(s_t), a_t) \leftarrow (1-\alpha_i^t) Q_i(\phi_i(s_t), a_t) + \alpha_i^t \big[r_i + \gamma \max_{a \in A} Q_i(\phi_i(s_{t+1}), a) \big] \tag{2}$$

Convergence of $\boldsymbol{Q}$ for a given θ: The $\boldsymbol{Q}$ parameter for modules is updated at a faster time scale, and perceive arbitrator's policy θ as quasi-static or constant. We utilize the fact that Q-values for modules are updated using greedy Q-learning (2), which is an off-policy algorithm. Regardless of the policy followed by the arbitrator, under standard assumptions (as noted in (Russell and Zimdars, 2003, Theorem 1)), Q-learning converges to $\tilde{Q}_i$ for each module i satisfying:

$$\tilde{Q}_i(s,a) = \sum_{s'} T(s,a,s') \big[r_i(s,a) + \gamma \max_{a' \in A} \tilde{Q}_i(s',a') \big] \tag{3}$$

Thus, we have shown that $\boldsymbol{Q}$ would converge for any θ.

Convergence of θ for arbitrator: For a given θ, let us assume that module Q-values (or $\boldsymbol{Q}$) have converged to $\tilde{\boldsymbol{Q}}$ due to timescale separation (arbitrator perceives $\boldsymbol{Q}$ as converged due to faster learning rate for $\boldsymbol{Q}$). Using the current θ and $\tilde{\boldsymbol{Q}}$, the θ is updated as:

$$\theta_{t+1} = \Gamma_\theta [\theta_t + \eta_2(t) \nabla_\theta J(\theta_t, \tilde{\boldsymbol{Q}})] \tag{4}$$

where $\nabla_\theta J$ is the arbitrator's policy gradient as defined in (5). The Γ_θ is a projection operator to ensure that θ stays within a compact convex set $\Theta := \Pi_{i=1}^{d}[\theta^i_{\min}, \theta^i_{\max}]$, assuming θ is d-dimensional. We also note that objective J for the arbitrator depends on $\tilde{\boldsymbol{Q}}$ as the rewards r_{AR} are influenced by module Q-values $\tilde{\boldsymbol{Q}}$ using the action selection mechanism presented in section 4.1 of the main text. Using results from (Borkar, 2008, Chapter 2), the update (4) can be seen as the ODE $\dot{\theta} = \Gamma_\theta(\nabla_\theta J(\theta_t, \tilde{\boldsymbol{Q}}))$, and will converge following standard results and regularity conditions in (Borkar, 2008, Chapter 2).

$$\nabla_\theta J = \mathbb{E}_{s_t \sim \rho}\big[\nabla_{\boldsymbol{w}} Q(s, \boldsymbol{w}; \theta')|_{s=s_t, \boldsymbol{w}=\mu(s_t)} \nabla_\theta \mu(s; \theta)|_{s=s_t}\big] \tag{5}$$

Given the above results and assumptions, it has been shown in (Borkar, 2008, Chapter 6), $(\theta_t, \boldsymbol{Q}_t)$ will converge to a locally optimal solution $(\theta^\star, \tilde{\boldsymbol{Q}})$ almost surely.

□

B Algorithm and Update Rules

Algorithm 1 Arbitrator

Defines:

N	▷ The number of modules
$\Pi_{AR}(s)$	▷ Policy of the arbitrator
D_{AR}	▷ Experience Replay Memory of Arbitrator

Routine: getWeights(s)

1: $weights \leftarrow \Pi_{AR}(s)$
2: **return** $weights$

Routine: update(s, a, r, s')

1: Store transition (s, a, r, s') in D_{AR}
2: Sample a random minibatch of transitions (s, a, r, s') from replay memory D_{AR}
3: Update critic network of arbitrator using these tuples as given in equation (6)
4: Update actor network of arbitrator using these tuples as given in equation (8)

B.1 Arbitrator's Updates

For the case of this paper, GRACIAS's arbitrator uses the standard DDPG algorithm Lillicrap et al. (2016) to learn its policy. Details of actor and critic are given below.

1. Policy structure: $\mu(s_{AR}|\theta^\mu) \rightarrow A_{AR}$, where θ^μ are the parameters of actor network, A_{AR} is the action space of the arbitrator of the form $[-1, 1]^N$. We use $\theta^{\mu'}$ for the actor's target network.
2. Q function: $Q_{AR}(s_{AR}, a_{AR}|\theta^{Q_{AR}}) \rightarrow \mathbb{R}$, where $\theta^{Q_{AR}}$ are the parameters of critic network. We use $\theta^{Q'_{AR}}$ for the critic's target network.
3. Q function update equation:

$$L = \sum (y_{AR} - Q_{AR}(s_{AR}, a_{AR}|\theta^{Q_{AR}}))^2 \tag{6}$$

   where:

$$y_{AR} = r_{AR} + \gamma Q'_a(s'_{AR}, \mu'(s'_{AR}|\theta^{\mu'})|\theta^{Q'_{AR}}), \tag{7}$$

4. Policy update equation:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N} \sum \nabla_{AR} Q_{AR}(s_{AR}, a_{AR}|\theta^{Q_{AR}})|_{s_a = s_{AR}, a_{AR} = \mu(s_{AR})} \nabla_{\theta^\mu} \mu(s_{AR}|\theta^\mu)|_{s_{AR}} \tag{8}$$

Algorithm 2 Module i

Defines:

D_i ▷ Experience Replay Memory of module i

$Q_i(s)$ ▷ Value function of module i

Routine: getQValues(s)

1: $values \leftarrow Q_i(s)$
2: **return** $values$

Routine: update(s, a, r, s')

1: Store transition (s, a, r, s') in D_i
2: Sample a random minibatch of transitions (s_j, a_j, r_j, s'_j) from replay memory D_i
3: Set $y_j = \begin{cases} r_j & \text{for terminal } s'_j \\ \mathrm{r}_j + \lambda \underset{a'}{\operatorname{argmax}} Q_{i(s',a')} & \text{otherwise} \end{cases}$
4: Perform gradient descent step on $(y_j - Q_i(s_j, a_j))^2$ as given in equation (9)

B.2 Modules' Updates

For the case of this paper, the modules use the standard Double-DQN algorithm [Van Hasselt, Guez, and Silver (2015)] to learn their policy.

1. Q-function for the module i: $Q_i(s, a|\theta_i) \rightarrow \mathbb{R}$, where θ_i are the parameters of Q_i.
2. Q-function update equation:

$$\nabla_{\theta_i} L_i(\theta_i) = \mathbb{E}_{s_j, a_j, r_j, s'_j}[(y - Q_i(s_j, a_j; \theta_i))\nabla_{\theta_i} Q_i(s_j, a; \theta_i)] \tag{9}$$

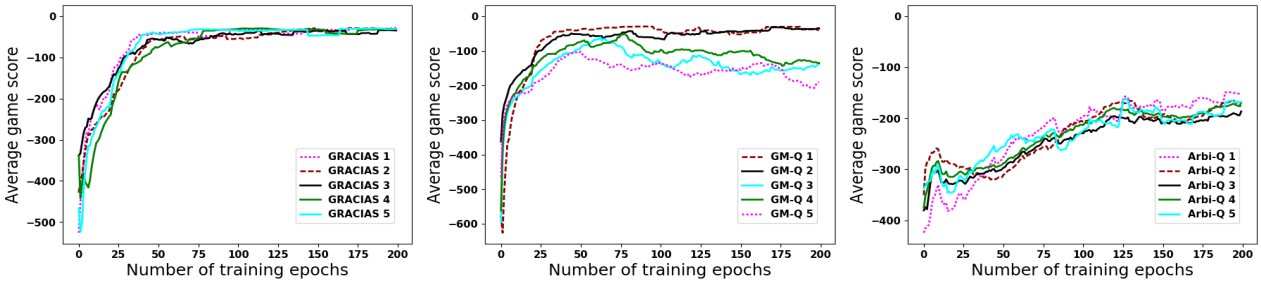
C Performance Comparison in Different Reward Settings

As discussed in the main text, only GM algorithms are highly sensitive to the reward scales of the modules; GRACIAS and Arbi-Q are robust to different reward scales. Defining and designing *comparable rewards scales* is not a straightforward task. We define a comparable reward setting as follows. First, a DDQN agent was trained till convergence using the global rewards as in conventional standard RL. Then a grid search was done over the rewards of all the modules to train different instances of GM-Q. The reward setting on which GM-Q resulted in similar solution quality as DDQN on convergence is defined as a *comparable reward setting*. Conversely, a reward setting is called *incomparable* if GM-Q can not attain similar solution quality as that of DDQN on convergence.

There is a huge spectrum of reward settings. All the MRL algorithms (GRACIAS, Arbi-Q, and GM-Q) were analyzed in five different reward settings. It should be noted that all the reward scales were modified such that the rewards within an individual module remain consistent. Table 1, 2 and 3 list the exact rewards used for Racetrack, Bunny World and Qbert respectively. The first entry in all the tables corresponds to the comparable reward setting. Only a relative change in the reward structure of the different modules affects the overall policy. Therefore in the case of two modules, it can be sufficient to vary the rewards of a single module. Table 4 shows the global rewards for each domain.

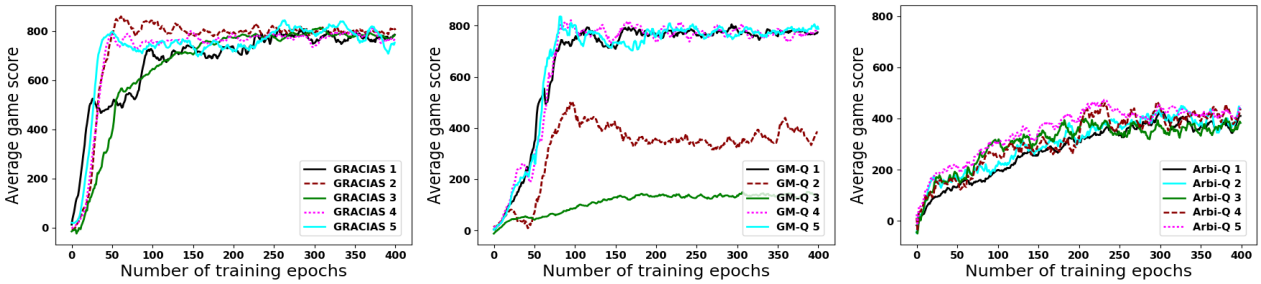
1. In **Racetrack**, different rewards settings were tried for the $AvoidCollision$ module while fixing the rewards of the $FinishRace$ module. In the first setting, the rewards of the modules are comparable with each other. In the next settings, the internal rewards of the $AvoidCollision$ module are increased gradually.
2. In **Bunny World**, we vary the rewards settings of both the modules (although, as stated above, varying the rewards of a single module is sufficient). In the first setting, the rewards of the modules are comparable with each other. In the second and third settings, all the $AvoidPredator$ module's internal rewards were decreased by four and ten times, respectively, with respect to the comparable rewards setting (the first reward setting). In the fourth and fifth settings, all the $CollectFood$ module's internal rewards were decreased by four and ten times, respectively, with respect to the comparable rewards setting. As pointed out in the main text, any scaling that gives adequate importance to the $AvoidPredator$ module works well for Bunny World. Therefore GM-Q's performance does not degrade in the last two cases for Bunny World.
3. In **Qbert**, similar to Bunny World, we vary the relative rewards of all the modules. The first setting corresponds to the comparable reward scaling. In each of the next four cases, we increase all the internal rewards of the respective modules by five times as compared to the comparable reward setting (the first reward setting).

As explained in the main text, GRACIAS shows similar performances in all the five settings for all the domains. On the other hand, it can be observed in Figures 1, 2, 3, that GM-Q is highly unstable and converges to a lower score on many reward scales.



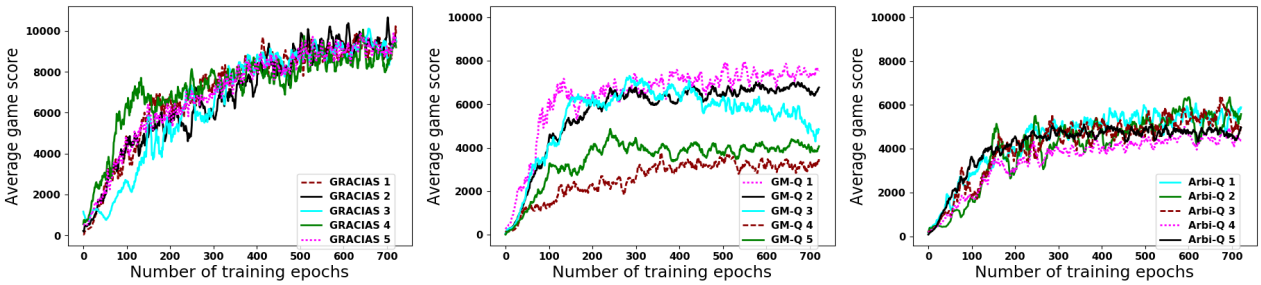
(a) GRACIAS (b) GM-Q (c) Arbi-Q

Figure 1: Training curves of different MRL algorithms with different reward scales of the modules on Racetrack



(a) GRACIAS (b) GM-Q (c) Arbi-Q

Figure 2: Training curves of different MRL algorithms with different reward scales of the modules on Bunny World



(a) GRACIAS (b) GM-Q (c) Arbi-Q

Figure 3: Training curves of different MRL algorithms with different reward scales of the modules on Qbert

	Finish Module		Collision Module	
S.No.	Finish Reward	Time Penalty	Collision Penalty	Time Reward
1	15	-1	-8	-1
2	15	-1	-20	-1
3	15	-1	-25	-3
4	15	-1	-25	-2
5	15	-1	-30	-5

Table 1: Different Reward Scalings for Racetrack

	Collect Food Module		Avoid Predator Module	
S.No.	Food Reward	Time Penalty	Death Penalty	Time Reward
1	10	-10	-20	1
2	10	-10	-5	.25
3	10	-10	-2	0.1
4	2.5	-2.5	-20	1
5	1	-1	-20	1

Table 2: Different Reward Scalings for Bunny World

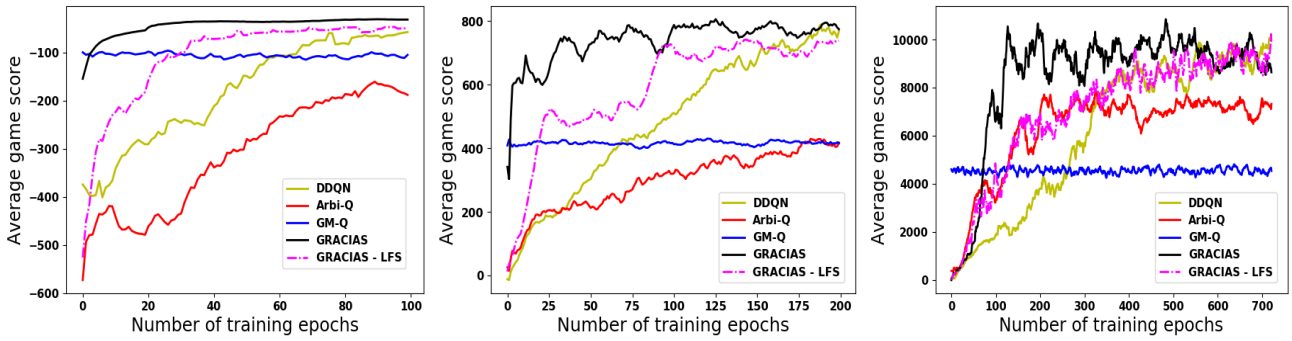
	Life Module		Color Transition Module		Disc Module	Green Module
S.No.	Death Penalty	Timestep Reward	Transition Reward	Episode Completion	Successful Luring	Caught Object
1	1	-10	5	20	10	10
2	5	-50	5	20	10	10
3	1	-10	25	100	10	10
4	1	-10	5	20	50	10
5	1	-10	5	20	10	50

Table 3: Different Reward Scalings for Qbert

Bunny World	
Food Reward	10
Death Penalty	-20
Time Penalty	-1
Racetrack	
Finish Reward	20
Collision Penalty	-5
Time Penalty	-1
Qbert	
Color transition reward	25
Catching Sam	300
Catching Green Ball	100
Luring the snake	500
Episode Completion	3100

Table 4: List of Global Rewards for all games

D Performance Comparison with All Modules are Pre-trained



(a) Racetrack (b) Bunny World (c) Qbert

Figure 4: Training curves of different algorithms when all the modules are pre-trained. All components of DDQN and GRACIAS-LFS are *"Learning From Scratch"* and have been added for comparison.

E Model Architecture and Hyper-parameters

The modules for all the approaches (GM-Q, Arbi-Q and GRACIAS) use Double-DQN to learn their specific subtasks Van Hasselt, Guez, and Silver (2015). All the hyper-parameters and components like network architecture, state abstraction functions and experience replay memory for the modules were kept the same across all the algorithms. The DDQN architecture of the modules in Qbert was kept the same as proposed by Mnih et al. (2015). The DDPG architecture of the Arbitrator in Qbert was kept the same as proposed by Lillicrap et al. (2016). The DDQN and DDPG architecture used in the Racetrack and Bunny world are explained in Table 5 and Table 6. The output layer is a fully connected linear layer with a single output for each valid action. The number of valid actions for Racetrack is nine and for Bunny World is five.

Parameter Name	Value
Optimizer	Adam
Learning Rate	10^{-2}
Discount Factor	0.9
Replay Buffer Size	50k
Samples per minibatch	32
Non-linearity	ReLU
Number of Hidden layers (all networks)	3
Layer sizes	64, 64, 32 (all fully connected)
Target network update Frequency	200
Gradient steps	1

Table 5: DDQN Parameters

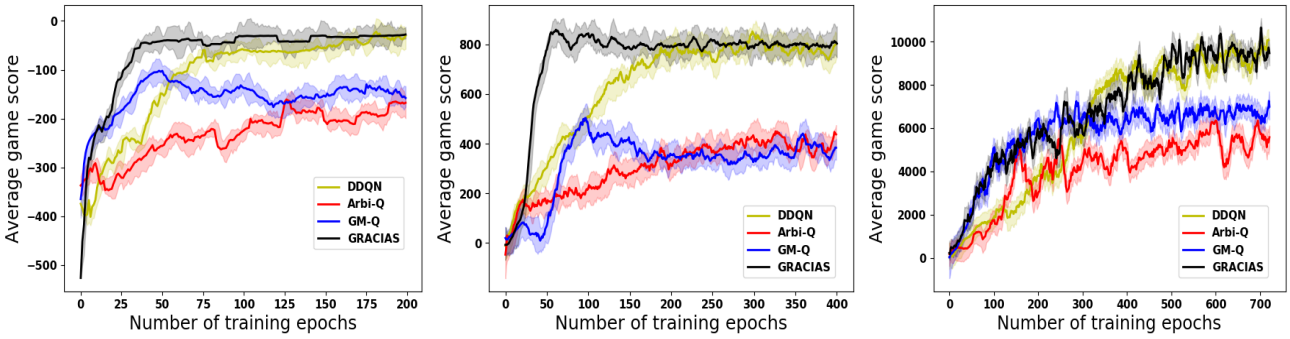
Training details for the different domains are given below.

1. **Racetrack**: All the agents were trained for 2000 episodes with the size of each training epoch being ten episodes. Each training epoch is followed by a testing epoch where the performance of the algorithm is averaged over ten episodes. Both the training and testing episodes were truncated at 200 steps.
2. **Bunny World**: All the agents were trained for 4000 episodes with the size of each training epoch being ten episodes. Each training epoch is followed by a testing epoch where the performance of the algorithm is averaged over ten episodes. Both the training and testing episodes were truncated at 1000 steps.
3. **Qbert**: All the agents were trained for 30M steps with the size of each training epoch being 40k steps. Each training epoch is followed by a testing epoch where the performance of the algorithm is averaged over 20 episodes.

Parameter Name	Value
Optimizer	Adam
Learning Rate	10^{-3}
Discount Factor	0.9
Replay Buffer Size	50k
Samples per minibatch	32
Non-linearity	ReLU
Number of Hidden layers (all networks)	3
Layer sizes	64, 128, 64 (all fully connected)
Target network smoothing coefficient	0.01
Target network update interval	1
Gradient steps	1

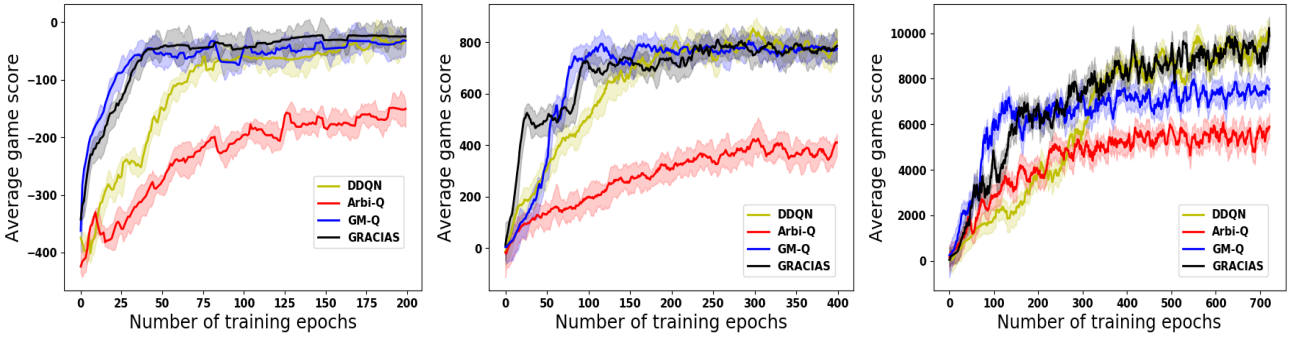
Table 6: DDPG Parameters

F Training curves with Min-Max Returns



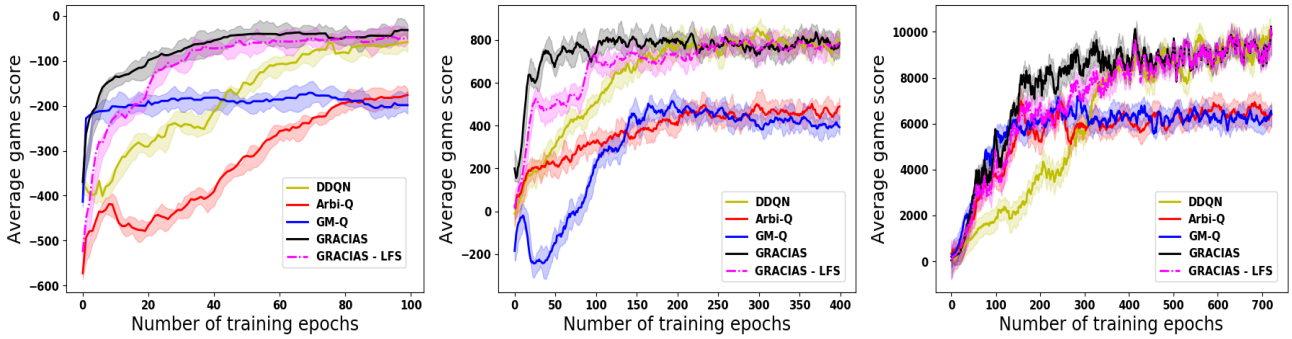
(a) Racetrack (b) Bunny World (c) Qbert

Figure 5: Training curves of different algorithms having modules with incomparable reward scales



(a) Racetrack (b) Bunny World (c) Qbert

Figure 6: Training curves of different algorithms having modules with comparable reward scales



(a) Racetrack (b) Bunny World (c) Qbert

Figure 7: Training curves of different algorithms when one module is pre-trained. All components of DDQN and GRACIAS-LFS are *"Learning From Scratch"* and have been added for comparison.

G Experimental Domains

G.1 Racetrack

The version of the well-known "racetrack" problem used in this paper is similar to the one presented in Russell and Zimdars (2003). The agent is situated at the start line of a rectangular race track. Every time the agent touches the finish line, it receives a positive reward, and the episode ends. At each time step, the agent receives a negative time penalty. To make things a little more interesting: every time the agent collides with any of the walls, it incurs a collision penalty for damaging the car. The agent tries to maximize the accumulated reward by reaching the finish line in the shortest time possible by making the minimum number of collisions with the walls.

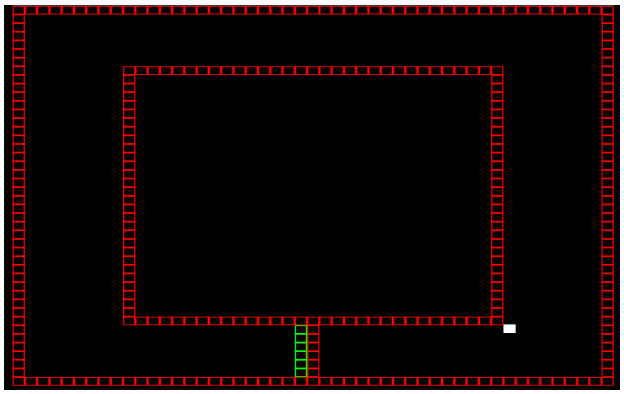


Figure 8: Racetrack domain

State Space: The environment consists of two concentric rectangles. Dimensions of the outer and the inner rectangle are 50*50 and 30*30, respectively. The area between these rectangles forms the racing track with a uniform width of eight units. These rectangles act as the walls at the boundaries of the racetrack. Initially, the car is at rest and placed at one of the randomly selected points on the start line. The agent has a continuous state space defined by its current position (x and y coordinates) and velocity (in x and y-direction).

Action Space: At each time step, the car can accelerate or decelerate its velocity by one unit. It results in nine actions (acceleration of -1, 0, or 1 unit in each of the two components).

Environment Dynamics: It is a stochastic environment, and the chosen action is executed with .9 probability. With .03 probability each, the car accelerates 45 ° to the left or right of the desired acceleration vector. With .01 probability each, the car accelerates 90 ° to the left or right of the desired acceleration vector, and with probability .02, it does nothing. The rotation of the acceleration vector by 45 ° makes the velocity, and hence the car's position is a *continuous variable*. There is also a limit on the maximum magnitude of the car's velocity in each component, capped at three units. When the car tries to increase the speed beyond this limit, acceleration for that particular component is set to zero. Upon collision, the car is projected back onto the track at the same point of collision, and the magnitude of each component

of velocity is reduced by one without changing its direction. If the agent does not accelerate away from the wall, it will not go out of the racetrack but will continue to slip parallel to the wall.

Reward Structure: The agent receives a reward of twenty units on touching the finish line, five units of penalty for every collision, and one unit time step penalty. The agent also receives a shaping reward Ng, Harada, and Russell (1999) proportional to the measure of the arc swept by its action.

The task of the agent is divided into two modules.

1. **Finish Race Module** M_F: The sole aim of this module is to reach the finish line in the shortest time possible. It receives a positive reward on reaching the finish line and obtains a time penalty for each step it takes in the environment. This module does not have any information about the penalty incurred on collision.
2. **Avoid Collision Module** M_C: This module learns to prevent collisions. Whenever the agent collides with the walls, this agent receives a collision penalty. For each time step, the agent does not collide with the walls; it gets a positive time reward. It is not concerned about finishing the race and doesn't receive any information about it.

G.2 Bunny World

Bunny World is one of the most popular benchmark domains in MRL literature Simpkins and Isbell (2019); Sprague and Ballard (2003). The agent (bunny) is situated in a continuous rectangular domain. Food particles are distributed over the entire space. A predator exists in the environment who wants to kill the agent. The agent aims to eat all the available food particles while avoiding predator. If the predator catches the agent, the agent dies, the episode ends, and the agent receives a death penalty. Once all the food particles are consumed, the agent wins, and the episode ends. Various walls obstruct the movement of the agent and the predator. At each timestep, the agent receives a negative time penalty. The agent tries to maximize the accumulated reward by eating all the food particles in the shortest time possible and dodging the predator throughout gameplay.

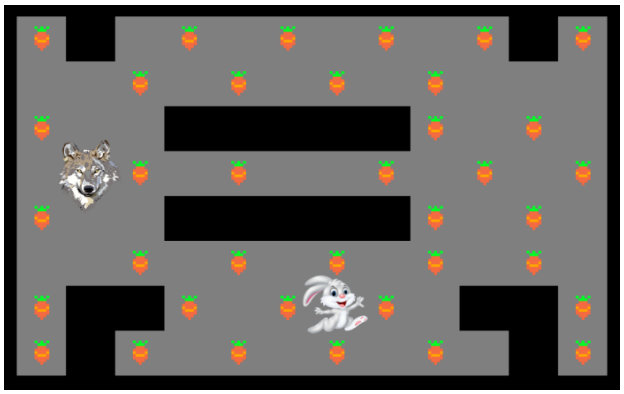


Figure 9: Bunny World domain

State Space: The environment consists of a continuous 2D rectangular world with numerous walls. The dimensions of the rectangular area are 50*50. Initially, the agent is placed at any random point in the world. At any given instant, the agent's state is defined by its current position (x and y coordinates), the current position of the predator, the position of the closest food particle, and its distance from the walls in all the four directions.

Action Space: At each time step, the agent can move one step forward or backward in either x or y direction or choose to stay at the current position. It results in five actions (up, down, left, right, and stay).

Environment Dynamics: The environment is stochastic, and the action chosen by the agent is executed with .8 probability. If the agent chooses to stay, all the other actions are executed with .05 probability each. If the agent chooses any other action, the opposite action is executed with .02 probability and the remaining three actions with .06 probability each. Further, a uniform random noise, $\zeta \sim N(\mu = 0, \sigma^2 = 0.01)$ is added to the final action making the position of the agent and hence its *state continuous*. If the agent tries to move to a location where a wall is situated, it stays in its current position. The predator follows a handcrafted policy that tries to minimize its distance from the agent. Due to stochasticity, the predator's chosen action is also followed with .8 probability, and a random action is chosen otherwise.

At any instant, if the distance between the agent and the predator becomes less than 0.1 unit, the agent dies, and the episode ends.

Reward Structure: The agent receives a reward of ten units on consuming each food particle, twenty units of penalty for getting caught by the predator, and a penalty of one unit time step.

The task of the agent is divided into two modules.

1. **Collect Food Module** M_F: The sole aim of this module is to learn to eat food. It gets a positive reward for each food particle it consumes and a time penalty for each step it takes in the environment. The optimal policy for this agent is to eat all the food as soon as possible. This module does not have any information about the predator operating in the environment.
2. **Avoid Predator Module** M_G: This module only learns to run away from the predator. It gets a negative dying penalty when the predator catches the agent. For each time step, it evades the predator; it gets a positive time reward. Again this agent does not receive any information about the food units in the environment.

G.3 Q*bert

Qbert is one of the popular Atari 2600 domains from the Arcade Learning Environment Bellemare et al. (2013). It is a multi-level action game where the agent hops over a 28-cube pyramid. The agent starts at the top cube of a pyramid and moves by hopping diagonally from cube to cube. Initially, all the cubes are colored with a "starting color". Once the agent lands on a cube, its color changes to the "target color". The aim is to change the color of all the cubes to the target color. Once it happens, the agent moves to the next level and the game continues.

Environment Dynamics: Qbert is a complex game with several different game objects. At some point, the agent has to dodge a ball or escape a snake, and at some other point, it has to catch some objects or use the flying discs. There are several types of enemies that impede the agent from achieving its goal. Red and Purple balls roll down from the top of the pyramid and squash the agent on contact. There are flying discs that help the agent to reach back to the top of the pyramid. There is a snake (Coily) that can kill the agent on contact. The agent needs to learn to lure the snake off the pyramid using the flying discs. A green character (Sam) reverts the color of the cube it lands on, back to the "starting color". The agent can kill Sam on contact. The agent has four lives in total. There is a green ball, which freezes all the objects except the agent upon contact with the agent. Its high-level objectives of gathering maximum points and finishing different levels can be naturally decomposed into simpler objectives. Therefore, it seems to be a great domain to establish the efficacy of MRL algorithms.

Figure 10: Qbert domain

State Space: Since extracting all the relevant features from a given frame is itself a research problem, for experimentation purposes, we feed the complete frame as input to all the modules. Our approach can further benefit from techniques that can provide the modules with relevant features instead of the complete game screen.

Action Space: At each time step, the agent can hop to an adjacent cube or stay on the same cube. A cube can have at most four adjacent cubes, one in each direction (up-left, up-right, down-left, and down-right). It results in five actions (up, down, left, right, and stay).

Reward Structure: The agent receives a reward based on the action taken. There are rewards for catching different objects, completing a level, and changing each cube's color to the "target color". At each level, the agent aims to collect as much reward as possible without getting killed. There is a reward of 25 units when the color of any cube transitions to the "target color", a reward of 300 units and 100 units respectively for catching Sam and the green ball. The agent receives 500 units for luring the snake off the pyramid and a bonus of 3100 units for completing a level. For space considerations, a few intricacies of the gameplay have been omitted. Complete details can be found online [1].

We decompose the high-level objective into the following four subtasks, a) Keeping the agent alive, b) Changing color of all the cubes, c) Luring the snake off the pyramid using flying discs, and d) Catching Sam and the Green ball. Below is the description of these modules:

1. **Life Module** M_L: The goal of this module is to keep the agent alive forever. The agent can potentially die when a) falls from the pyramid or collides with b) a snake, c) a Red ball, or d) a Purple ball. Every time step the agent survives, this module receives a positive reward, and whenever the agent dies, it receives a penalty. This module effectively learns to dodge the snake and the Red & Purple balls without falling from the bridge.
2. **Color Transition Module** M_C: This module aims to change all the cubes' color to the "target color". It receives a positive reward when the color of any cube transitions to the "target color". Whenever the color of the last remaining cube transitions to the target color, this module receives a large episode completion reward, and the next round starts.
3. **Disc Module** M_D: This module tries to use the flying discs to lure the snake off the pyramid. For every successful attempt to lure the snake off the pyramid, it receives a positive reward.
4. **Green Module** M_G: This module tries to catch all the green objects in the gameplay, namely, Sam and Green ball. For every green object, the agent catches, it receives a positive reward.

References

Bellemare, M. G.; Naddaf, Y.; Veness, J.; and Bowling, M. 2013. The arcade learning environment: An evaluation platform for general agents. *Journal of Artificial Intelligence Research* 47: 253–279.

Borkar, V. 2008. *Stochastic Approximation: A Dynamical Systems Viewpoint*.

Humphrys, M. 1996. Action selection methods using reinforcement learning. *From Animals to Animats* 4: 135–144.

Lillicrap, T. P.; Hunt, J. J.; Pritzel, A.; Heess, N.; Erez, T.; Tassa, Y.; Silver, D.; and Wierstra, D. 2016. Continuous control with deep reinforcement learning. In *International Conference on Learning Representations*.

Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A. A.; Veness, J.; Bellemare, M. G.; Graves, A.; Riedmiller, M.; Fidjeland, A. K.; Ostrovski, G.; et al. 2015. Human-level control through deep reinforcement learning. *nature* 518(7540): 529–533.

Ng, A. Y.; Harada, D.; and Russell, S. 1999. Policy invariance under reward transformations: Theory and application to reward shaping. In *ICML*, volume 99, 278–287.

Russell, S. J.; and Zimdars, A. 2003. Q-decomposition for reinforcement learning agents. In *Proceedings of the 20th International Conference on Machine Learning (ICML-03)*, 656–663.

Simpkins, C.; and Isbell, C. 2019. Composable modular reinforcement learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, 4975–4982.

Sprague, N.; and Ballard, D. 2003. Multiple-goal reinforcement learning with modular sarsa (0) .

Tessler, C.; Mankowitz, D. J.; and Mannor, S. 2019. Reward constrained policy optimization. In *International Conference on Learning Representations, ICLR 2019*.

Van Hasselt, H.; Guez, A.; and Silver, D. 2015. Deep reinforcement learning with double q-learning. *arXiv preprint arXiv:1509.06461* .

[1] `https://atariage.com/manual_html_page.php?SoftwareLabelID=379`