On Combining Reinforcement Learning & Adversarial Training

Ankur Deka

CMU-RI-TR-21-32

July 30, 2021

The Robotics Institute
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA

Thesis Committee:
Katia Sycara, *advisor*
Michael Lewis, *advisor*
Deepak Pathak
Wenhao Luo

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Robotics.

To my family, friends and teachers.

Abstract

Reinforcement Learning (RL) allows us to train an agent to excel at a given sequential decision making task by optimizing for a reward signal. Adversarial training involves a joint optimization scheme where an agent and an adversary compete against each other. In this work, we explore some domains involving the combination of RL and adversarial training, yielding practical learning algorithms. Certain domains use adversarial training as a tool to help improve the RL agent's performance whereas others have an adversary built into the problem statement. We explore both these kinds of scenarios and propose new algorithms that outperform existing ones.

1. We formulate a new class of Actor Residual Critic (ARC) RL algorithms for improving Adversarial Imitation Learning (AIL). Unlike most RL settings, the reward in AIL is differentiable and to leverage the exact gradient of the reward, we propose ARC instead of the standard Actor Critic algorithms in RL. ARC aided AIL algorithms outperform existing AIL algorithms in continuous-control tasks.
2. We create a new multi-agent mixed cooperative-competitive simulation environment called FortAttack that addresses several limitations in existing multi-agent environments. We further show that complex multi-agent strategies, that involve coordination and heterogeneous task allocation, can naturally emerge from scratch through competition between two teams of agents.
3. Leader-follower strategy is popular in multi-robot navigation. If a leader-follower multi-robot team is in a crucial mission and there is an external adversarial observer (enemy), hiding the leader's identity is crucial. The external adversary who wishes to sabotage the robot team's mission can simply identify and harm just the leader. This will compromise the whole robot team's mission. We propose a defense mechanism of hiding the leader's identity by ensuring the leader moves in a way that behaviorally camouflages it with the followers, making it difficult for an adversary to identify the leader. We further show that our multi-robot policies trained purely in simulation generalizes to real human observers, making it difficult for them to identify the leader.

Acknowledgments

I am extremely thankful to my advisors Prof. Katia Sycara and Prof. Michael Lewis for allowing me to explore the areas of my interest and supporting me through my master's journey. They taught me how to approach a problem in a structured way and present my work in a cohesive fashion. I am thankful to my thesis committee members Prof. Deepak Pathak and Wenhao Luo for the discussions and feedback on my work.

During my master's program, I have had the opportunity to work with and meet many awesome people. I am thankful to my collaborators Prof. Changliu Liu, Huao Li and Phillip Walker. I have had many long discussions with my lab-mates Tejus Gupta and Swaminathan Gurumurthy on the recent progresses in Reinforcement Learning and also on how I can better furnish my ideas. My lab-mates Siddharth Agrawal, Rohit Jena, Vidhi Jain, Jaskaran Singh Grover, Sha Yi, Chendi Lin, Akshay Sharma, Akshay Dharmavaram, Siddharth Ghiya, Yunfei Shi, Fan Jia, Keitaro Nishimura, Sophie Guo and Andrew Jong are all brilliant and fun people to work and spend time with. I am thankful to Dana Hughes for helping me out with his magic wand of computer engineering skills whenever I was in need. I am thankful to Lynnetta Miller for smoothly handling the logistics in our lab and to Barbara Jean and Prof. George Kantor for smoothly conducting the MS in Robotics program. I am thankful to everyone I met at the Robotics Institute or CMU.

My apartment-mates Yash Belhe and Advait Gadhikar have played a pivotal role in my journey. They tolerated boring speeches on my then ongoing work (much like you the reader will hopefully do in some time) and gave me a homely company. I cannot over-thank my parents, brother, sister-in-law and girlfriend for constantly believing in me and providing me the much needed mental strength and support. Without them, this work wouldn't have been possible.

Funding

This research has been supported by ONR N00014-19-C-1070, AFOSR/AFRL award FA9550-18-1-0251, FA9550-18-1-0097, AFOSR FA9550-15-1-0442 and DARPA Cooperative Agreement No.: HR00111820051.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Reinforcement Learning (RL) allows us to train an agent to learn a sequential decision making task by optimizing a reward signal. The agent learns through trial and error, gathering experience in an environment and gradually improving over time. Eventually, it excels at the given task, often generating astonishing results. Some of the recent feats of RL are - AlphaGo defeating the world champion at the challenging game of Go [61, 62], challenging dexterous robotic manipulation [4], legged robot locomotion that can rapidly adapt to difficult unseen terrains [35] and complex multi-agent coordination strategies [11]. A major reason for the success of RL in such diverse tasks is the flexibility in its formulation - any sequential decision making task can be modeled as an RL problem and moreover the reward can be any general scalar function without assumptions such as continuity or differentiability.

Adversarial training is a joint optimization scheme where an agent and an adversary compete against each other. One particular instance, Generative Adversarial Networks (GANs) [23], have been extremely successful in modeling and replicating data distributions in high dimensional spaces, particularly generating realistic fake images [33, 49, 79].

The combination of RL and adversarial training creates new possibilities, extending the applications of both these lines of work. The domains combining RL and adversarial training can be broadly classified into 2 categories, based on the role of the adversary.

In the first category, an adversary is used as a tool to help improve the RL

agent's performance by artificially creating a competition. One of the most popular class of algorithms in this domain is Adversarial Imitation Learning (AIL). Many popular algorithms such as GAIL [28], AIRL [20] and f-MAX [22] fall in this category. The primary objective in these algorithms is to optimally act in an environment by imitating an expert. However, they model it as a 2-player competitive game between an agent and an adversary (discriminator). The adversary tries to distinguish agent trajectories from expert trajectories. The agent tries to fool the adversary by generating trajectories that are similar to expert trajectories. At convergence, the agent trajectories resemble the expert trajectories and the adversary cannot distinguish between them. Some other popular applications that use an adversary as a tool are those of Robust RL or Adversarial attacks in RL [32, 46, 71].

In the second category, an adversary is a part of the problem statement. This includes the many scenarios where an agent competes against an opponent (adversary). A popular scenario is self-play, where an agent competes against a clone of itself and learns to improve from over time [61, 63]. [5] had two agents competing against each other in the RoboSumo environment. Some other examples are multi-agent teams competing in the game of hide and seek [11] or in the game of StarCraft II [72].

In this work, we explore both these categories that combine RL and adversarial training. In particular, we look at the following scenarios:

1. **Actor Residual Critic for Adversarial Imitation Learning [16]**: Adversarial Imitation Learning (AIL) is a class of popular state-of-the-art Imitation Learning algorithms where an artificial adversary's misclassification is used as a reward signal and is optimized by any standard Reinforcement Learning (RL) algorithm. Unlike most RL settings, the reward in AIL is *differentiable* but model-free RL algorithms do not make use of this property to train a policy. In contrast, we leverage the differentiability property of the AIL reward function and formulate a class of Actor Residual Critic (ARC) RL algorithms that draw a parallel to the standard Actor-Critic (AC) algorithms in RL literature and uses a residual critic, C function (instead of the standard Q function) to approximate only the discounted future return (excluding the immediate reward). ARC algorithms have similar convergence properties as the standard AC algorithms with the additional advantage that the gradient through the immediate reward is exact. For the discrete (tabular) case with finite states, actions, and

known dynamics, we prove that policy iteration with C function converges to an optimal policy. In the continuous case with function approximation and unknown dynamics, we experimentally show that ARC aided AIL outperforms standard AIL in simulated continuous-control tasks. ARC algorithms are simple to implement and can be incorporated into any existing AIL implementation with an AC algorithm.

2. **Emergent Multi-Agent Strategies** [15]: Multi agent strategies in mixed cooperative-competitive environments can be hard to craft by hand because each agent needs to coordinate with its teammates while competing with its opponents. Learning based algorithms are appealing but they require a competitive opponent to train against, which is often not available. Many scenarios require heterogeneous agent behavior for the team's success and this increases the complexity of the learning algorithm. In this work, we develop a mixed cooperative-competitive multi agent environment called FortAttack in which two teams compete against each other for success. FortAttack addresses several drawbacks in existing multi agent environments. We show that modeling agents with Graph Neural Networks (GNNs) and training them with Reinforcement Learning (RL) from scratch, leads to the co-evolution of increasingly complex strategies for each team. Through competition in Multi-Agent Reinforcement Learning (MARL), we observe a natural emergence of heterogeneous behavior among homogeneous agents when such behavior can lead to the team's success. Such heterogeneous behavior from homogeneous agents is appealing because any agent can replace the role of another agent at test time. Finally, we propose ensemble training, in which we utilize the evolved opponent strategies to train a single policy for friendly agents. We were able to train a large number of agents on a commodity laptop, which shows the scalability and efficiency of our approach.
3. **Hiding Leader's Identity in Leader-Follower Navigation** [18, 19]: Leader-follower navigation is a popular class of multi-robot algorithms where a leader robot leads the follower robots in a team. The leader has specialized capabilities or mission critical information (e.g. goal location) that the followers lack which makes the leader crucial for the mission's success. However, this also makes

the leader a vulnerability - an external adversary who wishes to sabotage the robot team's mission can simply harm the leader and the whole robot team's mission would be compromised. Since robot motion generated by traditional leader-follower navigation algorithms can reveal the identity of the leader, we propose a defense mechanism of hiding the leader's identity by ensuring the leader moves in a way that behaviorally camouflages it with the followers, making it difficult for an adversary to identify the leader. To achieve this, we combine Multi-Agent Reinforcement Learning, Graph Neural Networks and adversarial training. Our approach enables the multi-robot team to optimize the primary task performance with leader motion similar to follower motion, behaviorally camouflaging it with the followers. Our algorithm outperforms existing work that tries to hide the leader's identity in a multi-robot team by tuning traditional leader-follower control parameters with Classical Genetic Algorithms. We also evaluated human performance in inferring the leader's identity and found that humans had lower accuracy when the robot team used our proposed navigation algorithm.

Chapter 2

ARC - Actor Residual Critic for Adversarial Imitation Learning

2.1 Introduction

Although Reinforcement Learning (RL) allows us to train agents to perform complex tasks without manually designing controllers [25, 42, 58], it is often difficult to hand-craft a dense reward function that captures the task objective [8, 9, 55]. Imitation Learning (IL) or Learning from Demonstration (LfD) is a popular choice in such situations [2, 8, 9, 55]. Common approaches to IL are Behavior Cloning (BC) [10] and Inverse Reinforcement Learning (IRL) [44].

Within IRL, recent Adversarial Imitation Learning (AIL) algorithms have shown state-of-the-art performance, especially in continuous control tasks which make them relevant to real-world robotics problems. AIL involves an agent and an adversary (discriminator) competing in a 2-player game. The adversary tries to distinguish agent trajectories from expert trajectories. The agent tries to fool the adversary by generating trajectories that are similar to expert trajectories. At convergence, the agent trajectories resemble the expert trajectories and the adversary cannot distinguish between them. Popular AIL algorithms include Generative Adversarial Imitation Learning (GAIL) [28], Adversarial Inverse Reinforcement Learning (AIRL) [20] and f-MAX [22].

The agent in AIL is trained with any standard RL algorithm. There are two popular categories of RL algorithms: (i) on-policy algorithms such as TRPO [56], PPO [58], GAE [57] based on the policy gradient theorem [66, 75]; and (ii) off-policy Actor-Critic (AC) algorithms such as DDPG [38], TD3 [21], SAC [25] that compute the policy gradient through a critic (Q function). These standard RL algorithms were designed for arbitrary scalar reward functions; and they compute an *approximate* gradient for updating the policy. Practical on-policy algorithms based on the policy gradient theorem use several approximations to the true gradient [56, 57, 58] and off-policy AC algorithms first approximate policy return with a critic (Q function) and subsequently compute the gradient through this critic [21, 25, 38].

However, our insight is that the reward function in AIL has a special property, it is *differentiable* which means we can compute the exact gradient through the reward function instead of approximating it. As we will see in section 2.3, naively computing the gradient through reward function would lead to a short-sighted sub-optimal policy. To address this issue, we formulate a class of Actor Residual Critic (ARC) RL algorithms that use a residual critic, C function (instead of the standard Q function) to approximate only the discounted future return (excluding immediate reward).

The contribution of this paper is the introduction of ARC, which can be easily incorporated to replace the AC algorithm in any existing AIL algorithm for continuous-control and helps boost the asymptotic performance by computing the exact gradient through the reward function.

2.2 Related Work

The simplest approach to imitation learning is Behavior Cloning [10] where an agent policy directly regresses on expert actions (but not states) using supervised learning. This leads to distribution shift and poor performance at test time [28, 52]. Methods such as DAgger [52] and Dart [36] eliminate this issue but assume an interactive access to an expert policy, which is often impractical.

Inverse Reinforcement Learning (IRL) approaches recover a reward function which can be used to train an agent using RL [44, 80] and have been more successful than BC. Within IRL, recent Adversarial Imitation Learning (AIL) methods inspired by Generative Adversarial Networks (GANs) [23] have been extremely successful. GAIL

Algorithm	Minimized f-Divergence		$r(s,a)$
	Name	Expression	
GAIL [28]	Jensen-Shannon	$\frac{1}{2}\left\{\mathbb{E}_{\rho^{\text{exp}}} \log \frac{2\rho^{\text{exp}}}{\rho^{\text{exp}}+\rho^{\pi}} + \mathbb{E}_{\rho^{\pi}} \log \frac{2\rho^{\pi}}{\rho^{\text{exp}}+\rho^{\pi}}\right\}$	$\log D(s,a)$
AIRL [20], f-MAX-RKL [22]	Reverse KL	$\mathbb{E}_{\rho^{\pi}} \log \frac{\rho^{\pi}}{\rho^{\text{exp}}}$	$\log \frac{D(s,a)}{1-D(s,a)}$

Table 2.1: Popular AIL algorithms, f-divergence metrics they minimize and corresponding reward functions.

[28] showed state-of-the-art results in imitation learning tasks following which several extensions have been proposed [31, 37]. AIRL [20] imitates an expert as well as recovers a robust reward function. [34] and [22] presented a unifying view on AIL methods by showing that they minimize different divergence metrics between expert and agent state-action distributions but are otherwise similar. [22] also presented a generalized AIL method f-MAX which can minimize any specified f-divergence metric [39] between expert and agent state-action distributions thereby imitating the expert. Choosing different divergence metrics leads to different AIL algorithms, e.g. choosing Jensen-Shannon divergence leads to GAIL [28]. [76] proposed a method that automatically learns a f-divergence metric to minimize. Our proposed Actor Residual Critic (ARC) can be augmented with any of the above AIL algorithms.

Some recent methods have identified that the reward in AIL is differentiable but they have used this property in very different settings. [45] used the gradient of the reward to improve the reward function but not to optimize the policy. [26] used the gradient through the reward to optimize the policy but operated in the model-based setting. We instead propose a model-free approach that can use the gradient of the reward to optimize the policy with the help of a new class of RL algorithms called Actor Residual Critic (ARC).

2.3 Background

Objective Our goal is to imitate an expert from one or more demonstrated trajectories (state-action sequences) in a continuous-control task (state and action spaces are continuous). Given any Adversarial Imitation Learning (AIL) algorithm that uses an off-policy Actor-Critic algorithm RL algorithm, we wish to use our insight on

the availability of a differentiable reward function to improve the imitation learning algorithm.

Notation The environment is modeled as a Markov Decision Process (MDP) represented as a tuple $(\mathcal{S}, \mathcal{A}, \mathcal{P}, r, \rho_0, \gamma)$ with state-space $\mathcal{S}$, action-space $\mathcal{A}$, transition dynamics $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$, reward function $r(s, a)$, initial state distribution $\rho_0(s)$, and discount factor γ. $\pi(.|s)$, $\pi^{\text{exp}}(.|s)$ denote policies and $\rho^\pi, \rho^{\text{exp}} : \mathcal{S} \times \mathcal{A} \to [0, 1]$ denote state-action occupancy distributions for agent and expert respectively. $\mathcal{T} = \{s_1, a_1, s_2, a_2, \ldots, s_T, a_T\}$ denotes a trajectory or episode and (s, a, s', a') denotes a continuous segment in a trajectory. A discriminator or adversary $D(s, a)$ tries to determine whether the particular (s, a) pair belongs to an expert trajectory or agent trajectory, i.e. $D(s, a) = P(\text{expert}|s, a)$. The optimal discriminator is $D(s, a) = \frac{\rho^{\text{exp}}(s,a)}{\rho^{\text{exp}}(s,a)+\rho^\pi(s,a)}$ [23].

Adversarial Imitation Learning (AIL) In AIL, the discriminator and agent are alternately trained. The discriminator is trained to maximize the likelihood of correctly classifying expert and agent data using supervised learning:

$$\max_D \left\{ \mathbb{E}_{s,a\sim\rho^{\text{exp}}}[\log D(s, a)] + \mathbb{E}_{s,a\sim\rho^\pi}[\log(1 - D(s, a)] \right\} \tag{2.1}$$

The agent is trained to maximize the discounted return:

$$\max_\pi \left\{ \mathbb{E}_{s,a\sim\rho_0,\pi,\mathcal{P}} \sum_{t\geq 0} \gamma^t r(s_t, a_t) \right\} \tag{2.2}$$

Here, reward $r(s, a) = h(D(s, a))$ is a function of the discriminator which varies between different AIL algorithms. Different AIL algorithms minimize different f-divergence metrics between expert and agent state-action distribution. Defining a f-divergence metric instantiates different reward functions [22]. Some popular divergence choices are Jensen-Shannon in GAIL [28] and Reverse Kullback-Leibler in f-MAX-RKL [22] and AIRL [20] as shown in Table 2.1.

Any RL algorithm could be used to optimize (2.2) and popular choices are off-policy Actor-Critic algorithms such as DDPG [38], TD3 [21], SAC [25] and on-policy algorithms such as TRPO [56], PPO [58], GAE [57] which are based on the policy

gradient theorem [66, 75]. We focus on off-policy Actor-Critic algorithms as they are usually more sample efficient and stable than on-policy policy gradient algorithms [21, 25].

Continuous-control using off-policy Actor-Critic The objective in off-policy RL algorithms is to maximize expected Q function of the policy, Q^π averaged over the state distribution of a dataset $\mathcal{D}$ (typically past states stored in buffer) and the action distribution of the policy π [60]:

$$\max_{\pi} \mathbb{E}_{s\sim\mathcal{D},a\sim\pi} Q^\pi(s,a) \tag{2.3}$$

$$\text{where, } Q^\pi(s,a) = \mathbb{E}_{s,a\sim\rho_0,\pi,\mathcal{P}}\left[\sum_{k\geq 0}\gamma^k r_{t+k}\middle| s_t = s, a_t = a\right] \tag{2.4}$$

The critic and the policy denoted by Q, π respectively are approximated by function approximators such as neural networks with parameters ϕ and θ respectively. There is an additional target Q function parameterized by . There are two alternating optimization steps:

1. Policy evaluation: Fit critic (Q_ϕ function) by minimizing Bellman Backup error.

$$\min_{\phi} \mathbb{E}_{s,a,s'\sim\mathcal{D}} \left\{Q_\phi(s,a) - y(s,a)\right\}^2 \tag{2.5}$$

$$\text{where, } y(s,a) = r(s,a) + \gamma Q(s',a') \text{ and } a' \sim \pi_\theta(.|s') \tag{2.6}$$

   Q_ϕ is updated with gradient descent without passing gradient through the target $y(s,a)$.

2. Policy improvement: Update policy with gradient ascent over RL objective.

$$\mathbb{E}_{s\sim\mathcal{D}}\big[\nabla_\theta Q_\phi(s, a\sim\pi_\theta(.|s))\big] \tag{2.7}$$

All off-policy Actor Critic algorithms follow the core idea above ((2.5) and (2.7)) along with additional details such as the use of a deterministic policy and target network in DDPG [38], double Q networks and delayed updates in TD3 [21], entropy regularization and reparameterization trick in SAC [25].

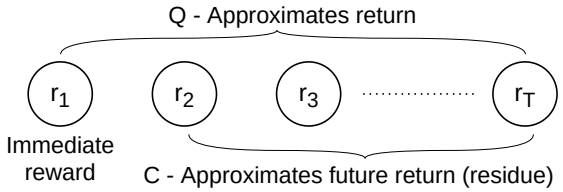


Figure 2.1: Visual illustration of approximating reward via Q function or C function.

Naive-Diff and why it won't work Realizing that the reward in AIL is differentiable, we can formulate a Naive-Diff RL algorithm that updates the policy by differentiating the RL objective (2.2) with respect to the policy parameters θ.

$$\mathbb{E}_{\mathcal{T}\sim\mathcal{D}}\left[\nabla_\theta r(s_1,a_1)+\gamma\nabla_\theta r(s_2,a_2)+\gamma^2\nabla_\theta r(s_3,a_3)+\dots\right] \tag{2.8}$$

Here $\mathcal{T}=\{s_1,a_1,s_2,a_2\dots\}$ is a sampled trajectory in $\mathcal{D}$. It may seem straight forward to obtain the gradients in (2.8) using standard autodiff packages such as Pytorch [47] or Tensorflow [1] but this naive approach would produce incorrect gradients. Apart from the immediate reward $r(s_1,a_1)$, all the terms depend on the transition dynamics of the environment $\mathcal{P}(s_{t+1}|s_t,a_t)$, which is unknown and we cannot differentiate through this term. As a result, autodiff will calculate the gradient of only immediate reward correctly and calculate the rest as 0's. The effective gradient calculated would be $\mathbb{E}_{s\sim\mathcal{D}}\big[\nabla_\theta r(s_1,a_1)\big]$. This will produce a short-sighted policy that maximizes only the immediate reward (without considering future return) and have sub-optimal performance.

2.4 Method

The main lesson we learnt from Naive-Diff is that while we can obtain the gradient of immediate reward, we cannot directly obtain the gradient of future return due to unknown environment dynamics. This directly motivates our formulation of Actor Residual Critic (ARC). Standard Actor Critic algorithms use Q function to approximate the return as described in Eq. 2.4. However, since we can directly obtain the gradient of the reward, we needn't approximate it with a Q function. We, therefore, propose to use C function to approximate only the future return, leaving out the immediate reward. This is the core idea behind Actor Residual Critic

(ARC) and is highlighted in Fig. 2.1. Note that the word "Residual" refers to the residue in return after removing immediate reward. As we will explain in Section 2.5, segregating the immediate reward from future return will allow ARC algorithms to leverage the exact gradient of the immediate reward. We now formally describe Residual Critic (C function) and its relation to the standard critic (Q function) in RL literature.

2.4.1 Definition of Residual Critic (C function)

The Q function under a policy π, $Q^\pi(s,a)$, is defined as the expected discounted return from state s taking action a .

$$Q^\pi(s,a) = \mathbb{E}_{s,a\sim\rho_0,\pi,\mathcal{P}}\left[\sum_{k\geq 0}\gamma^k r_{t+k}\middle| s_t = s, a_t = a\right] \tag{2.9}$$

The C function under a policy π, $C^\pi(s,a)$, is defined as the expected discounted future return, excluding the immediate reward. Note that the summation in (2.10) starts from 1 instead of 0.

$$C^\pi(s,a) = \mathbb{E}_{s,a\sim\rho_0,\pi,\mathcal{P}}\left[\sum_{k\geq 1}\gamma^k r_{t+k}\middle| s_t = s, a_t = a\right] \tag{2.10}$$

Q function can be expressed in terms of C function:

$$\begin{aligned} Q^\pi(s,a) &= \mathbb{E}_{s,a\sim\rho_0,\pi,\mathcal{P}}\left[\sum_{k\geq 0}\gamma^k r_{t+k}\middle| s_t = s, a_t = a\right] \\ &= r(s,a) + \mathbb{E}_{s,a\sim\rho_0,\pi,\mathcal{P}}\left[\sum_{k\geq 1}\gamma^k r_{t+k}\middle| s_t = s, a_t = a\right] \end{aligned} \tag{2.11}$$

$$\therefore Q^\pi(s,a) = r(s,a) + C^\pi(s,a) \tag{2.12}$$

2.4.2 Properties of C function

We now show some useful properties of the C function. We define the optimal C function, C^* as:

$$C^*(s,a) = \max_{\pi} C^{\pi}(s,a) \tag{2.13}$$

There exists an unique optimal C function for any MDP as described in Lemma 1. We can derive the Bellman equation for C^{π} (Lemma 2), similar to the Bellman equations for traditional action value function Q^{π} [66]. Using the recursive Bellman equation, we can define a Bellman backup operation for policy evaluation which converges to the true C^{π} function (Theorem 1). Using the convergence of policy evaluation, we can arrive at the Policy Iteration algorithm using C function as shown in Algorithm 1, which is guaranteed to converge to an optimal policy (Theorem 2), similar to the case of Policy Iteration with Q function or V function [66]. For comparison, the standard Policy Iteration with Q function algorithm is described in Appendix 2.8.1 Algorithm 3.

Lemma 1. *There exists a unique optimum C^* for any MDP.*

Proof. Appendix 2.7.1. □

Lemma 2. *The recursive Bellman equation for C^{π} is as follows*

$$C^{\pi}(s,a) = \gamma \sum_{s'} P(s'|s,a) \sum_{a'} \pi(a'|s') \left(r(s',a') + C^{\pi}(s',a')\right)$$

Proof. Appendix 2.7.1. □

Theorem 1. *The following Bellman backup operation for policy evaluation using C function converges to the true C function, C^{π}*

$$C^{n+1}(s,a) \leftarrow \gamma \sum_{s'} P(s'|s,a) \sum_{a'} \pi(a'|s') \left(r(s',a') + C^{n}(s',a')\right)$$

Here, C^n is the estimated value of C at iteration n.

Proof. Appendix 2.7.2 □

Theorem 2. *The policy iteration algorithm defined by Algorithm 1 converges to the optimal C^* function and an optimal policy π^*.*

Proof. From Theorem 1, the policy evaluation step converges to true C^π function. The policy improvement step is exactly the same as in the case with Q function since $Q^\pi(s,a) = r(s,a) + C^\pi(s,a)$, which is known to converge to an optimum policy [66]. These directly imply that Policy Iteration with C function converges to the optimal C^* function and an optimal policy π^*. □

Algorithm 1 Policy Iteration with C function

Initialize $C^0(s,a) \forall s,a$ **while** π *not converged* **do**
  // Policy evaluation
  for *n=1,2,... until* C_k *converges* **do**
    $C^{n+1}(s,a) \leftarrow \gamma \sum_{s'} P(s'|s,a) \sum_{a'} \pi(a'|s') \left(r(s',a') + C^n(s',a')\right) \quad \forall s,a$
  end
  // Policy improvement
  $\pi(s,a) \leftarrow \begin{cases} 1, & \text{if } a = \text{argmax}_{a'} \left(r(s,a') + C(s,a')\right) \\ 0, & \text{otherwise} \end{cases} \quad \forall s,a$
end

2.5 Continuous-control using Actor Residual Critic

Now that we have a working policy iteration algorithm with C function, we can easily extend it for continuous-control tasks using function approximators instead of discrete C values and a discrete policy (similar to the case of Q function [66]). We call any RL algorithm that uses a policy, π and a residual critic, C function as an Actor Residual Critic (ARC) algorithm. Using the specific details of different existing Actor Critic algorithms, we can formulate analogous ARC algorithms. For example, using a deterministic policy and target network as in [38] we can get ARC-DDPG. Using double C networks (instead of Q networks) and delayed updates as in [21] we can get ARC-TD3. Using entropy regularization and reparameterization trick as in [25] we can get ARC-SAC or SARC (Soft Actor Residual Critic). We show the SARC

algorithm in Algorithm 2.

Algorithm 2 SARC - Soft Actor Residual Critic

Intialization: Environment (env), Policy parameters θ, C-function parameters ϕ_1, ϕ_2, replay buffer $\mathcal{D}$, Target parameters $1 \leftarrow \phi_1$, $2 \leftarrow \phi_2$, Entropy regularization coefficient α **for** *no. of environment interactions* **do**

- $a \sim \pi_\theta(.|s)$ $s', r, d = \text{env.step(a)}$; $d = 1$ if s' is terminal state, 0 otherwise
- Store (s, a, r, s', d) in replay buffer $\mathcal{D}$ **if** *Update interval reached* **then**
  - **for** *no. of update steps* **do**
    - Sample batch $B = (s, a, r, s', d) \sim \mathcal{D}$ Compute C targets $\forall (s, a, r, s', d) \in B$ $y(s, a, d) = \gamma \left(r(s',) + \min_{i=1,2} C_i(s',) - \alpha \log \pi_\theta(|s') \right), \sim \pi_\theta(.|s')$
    - Update C-functions with gradient descent.
    - $\nabla_{\phi_i} \frac{1}{|B|} \sum_{(s,a,r,s',d) \in B} \left(C_{\phi_i}(s, a) - y(s, a, d) \right)^2, \quad \text{for } i = 1, 2$
    - Update policy with gradient ascent.
    - $\nabla_\theta \frac{1}{|B|} \sum_{s \in B} \Big(r(s,) + \min_{i=1,2} C_{\phi_i}(s,) - \alpha \log \pi(|s) \Big), \quad \sim \pi_\theta(.|s)$
    - Update target networks
    - $i \leftarrow \zeta i + (1 - \zeta)\phi_i, \quad \text{for } i = 1, 2; \quad \zeta$ controls polyak averaging
  - **end**
- **end**

end

2.5.1 ARC aided Adversarial Imitation Learning

To incorporate ARC in any Adversarial Imitation Learning algorithm, we simply replace the Actor Critic RL algorithm with an ARC RL algorithm without altering anything else in the pipeline. For example, we can replace SAC [25] with SARC, Algorithm 2. Implementation-wise this is extremely simple and doesn't require any additional functional parts in the algorithm. The same neural network that approximated Q function can be now be used to approximate C function.

2.5.2 Why choose ARC over Actor-Critic in Adversarial Imitation Learning?

The advantage of using an ARC algorithm over an Actor-Critic (AC) algorithm is that we can leverage the exact gradient of the reward. Standard AC algorithms

use Q_ϕ to approximate the immediate reward + future return and then compute the gradient of the policy parameters through the Q_ϕ function (2.14). This is an approximate gradient since the Q_ϕ function is an estimated value. On the other hand, ARC algorithms segregate the immediate reward (which is known in Adversarial Imitation Learning) from the future return (which needs to be estimated). ARC algorithms then compute the gradient of policy parameters through the immediate reward (which is exact) and the C function (which is approximate) separately (2.15).

$$\text{Standard AC} \quad \mathbb{E}_{s\sim\mathcal{D}}\big[\nabla_\theta Q_\phi(s,a)\big], \qquad a \sim \pi_\theta(.|s) \tag{2.14}$$

$$\text{ARC (Our)} \quad \mathbb{E}_{s\sim\mathcal{D}}\big[\nabla_\theta r(s,a) + \nabla_\theta C_\phi(s,a)\big], \qquad a \sim \pi_\theta(.|s) \tag{2.15}$$

2.6 Results

In Theorem 2, we proved that Policy Iteration (PI) with C function converges to an optimal policy. In the next section (Section 2.6.1), we experimentally validate it. In the following section (Section 2.6.2), we show the effectiveness of ARC aided AIL in continuous-control tasks through experiments in 4 different environments.

2.6.1 Policy Iteration on a Grid World

Our objective in this section is to experimentally validate if Policy Iteration (PI) with C function converges to an optimal policy (Theorem 2). We choose a simple Grid World environment to illustrate this. At every time step, the agent can move in one of 4 directions - left, right, up or down. The reward is 1 for reaching the goal (G) and 0 otherwise. The discount factor $\gamma = 0.9$.

On this environment, we run two PI algorithms - PI with C function (Algorithm 1) and the standard PI with Q function (Appendix 2.8.1 Algorithm 3). Fig. 2.2 shows the results of this experiment. Both the algorithms converge to the same optimal policy π^* shown in Fig. 2.2a. This optimal policy receives the immediate reward shown in Fig. 2.2b. Note that the immediate reward is 1 for states adjacent to the goal G as the agent receives 1 reward for taking an action that takes it to the goal. Fig. 2.2c and Fig. 2.2d show the values of C^*, Q^* that PI with C function and PI with Q function respectively converge to. In Fig. 2.2d, $Q^* = r^* + C^*$, which is

↱	↱	↱	↓
↱	↱	↱	↓
↱	↱	↱	↓
→	→	→	G

(a) π^*

0	0	0	0
0	0	0	0
0	0	0	1
0	0	1	G

(b) r^*

0.59	0.65	0.73	0.81
0.65	0.73	0.81	0.90
0.73	0.81	0.90	0.00
0.81	0.90	0.00	G

(c) C^*

0.59	0.65	0.73	0.81
0.65	0.73	0.81	0.90
0.73	0.81	0.90	1.00
0.81	0.90	1.00	G

(d) $Q^* = r^* + C^*$

Figure 2.2: On a Grid World, the results of running two Policy Iteration (PI) algorithms - PI with C function (Algorithm 1) and the standard PI with Q function (Appendix 2.8.1 Algorithm 3). Both algorithms converge in 7 policy improvement steps to the same optimal policy π^* as shown in Fig. 2.2a. The optimal policy gets the immediate reward shown shown in Fig. 2.2b. The C values (Fig. 2.2c) at the convergence of PI with C function and the Q values (Fig. 2.2d) at the convergence of PI with Q function are consistent with their relation $Q^* = r^* + C^*$ (2.12).

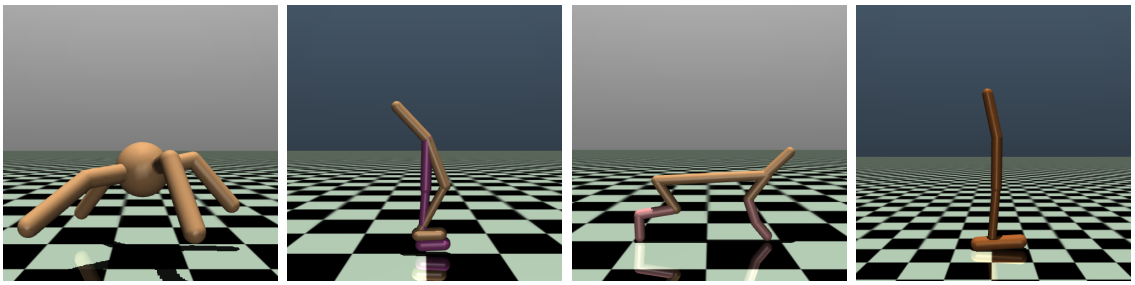
consistent with the relation between Q function and C function (2.12). In Fig. 2.2d, the Q^* values in the states adjacent to the goal are 1 since Q function includes the immediate reward (2.9). C function doesn't include the immediate reward (2.10) and hence the C^* values in these states are 0 (Fig. 2.2c). This experiment validates that PI with C function converges to an optimal policy as already proved in Theorem 2.

2.6.2 Imitation Learning in continuous-control tasks

We use Ant-v2, Walker-v2, HalfCheetah-v2 and Hopper-v2 Mujoco continuous-control environments from OpenAI Gym [12], as shown in Fig. 2.3. We evaluated the benefit of using ARC with two popular Adversarial Imitation Learning (AIL) algorithms, f-MAX-RKL [22] and GAIL [28]. For each of these algorithms, we evaluated the performance of standard AIL algorithms (f-MAX-RKL, GAIL), ARC aided AIL algorithms (ARC-f-MAX-RKL, ARC-GAIL) and Naive-Diff algorithm described in Section 2.3 (Naive-Diff-f-MAX-RKL, Naive-Diff-GAIL). We also evaluated the performance of Behavior Cloning (BC). In a given environment, each AIL algorithm is provided with a single expert trajectory to imitate from. We implemented our algorithm on top of the AIL code of [45]. The pre-implemented algorithms (f-MAX-RKL, GAIL) used SAC [25] as the RL algorithm and the ARC aided AIL algorithms

use SARC (Algorithm 2) as the RL algorithm. Further experimental details are presented in Appendix 2.9.

Fig. 2.4 shows the training plots and Table 2.2 shows the final performance of the different algorithms. Across all environments and across both the AIL algorithms, incorporating ARC shows consistent improvement over standard AIL algorithms (Table 2.2). BC suffers from distribution shift at test time [28, 52] and performs very poorly. As we predicted in Section 2.3, Naive-Diff algorithms don't perform well as naively using autodiff doesn't compute the gradients correctly. Amongst all the algorithms we evaluated, ARC-f-MAX-RKL performed the best.



(a) Ant-v2 (b) Walker-v2 (c) HalfCheetah-v2 (d) Hopper-v2

Figure 2.3: OpenAI Gym's [12] Mujoco continuous-control environments used for evaluation.

Method	Ant	Walker2d	HalfCheetah	Hopper
Expert return	5926.18 ± 124.56	5344.21 ± 84.45	12427.49 ± 486.38	3592.63 ± 19.21
ARC-f-Max-RKL (Ours)	**6275.08** ± 112.61	**4741.28** ± 94.49	**12698.38** ± 330.67	**3453.28** ± 37.95
f-Max-RKL	5983.96 ± 80.94	4107.55 ± 61.16	12003.33 ± 183.04	3410.09 ± 12.6
Naive-Diff f-Max-RKL	997.65 ± 3.31	320.48 ± 7.7	-0.48 ± 0.03	146.43 ± 34.83
ARC-GAIL (Ours)	**6048.67** ± 126.97	**3941.39** ± 83.72	**11615.09** ± 555.05	**3385.28** ± 15.92
GAIL	5895.09 ± 10.44	3318.8 ± 48.55	10814.69 ± 323.91	3322.05 ± 8.31
Naive-Diff GAIL	995.97 ± 2.13	55.63 ± 29.04	-0.46 ± 0.02	126.3 ± 51.55
BC	559.76 ± 129.82	87.51 ± 120.39	-444.5 ± 44.67	233.83 ± 84.82

Table 2.2: Policy return on Mujoco environments using different Adversarial Imitation Learning algorithms. Each algorithms is run with 3 random seeds and each seed is evaluated for 20 episodes.

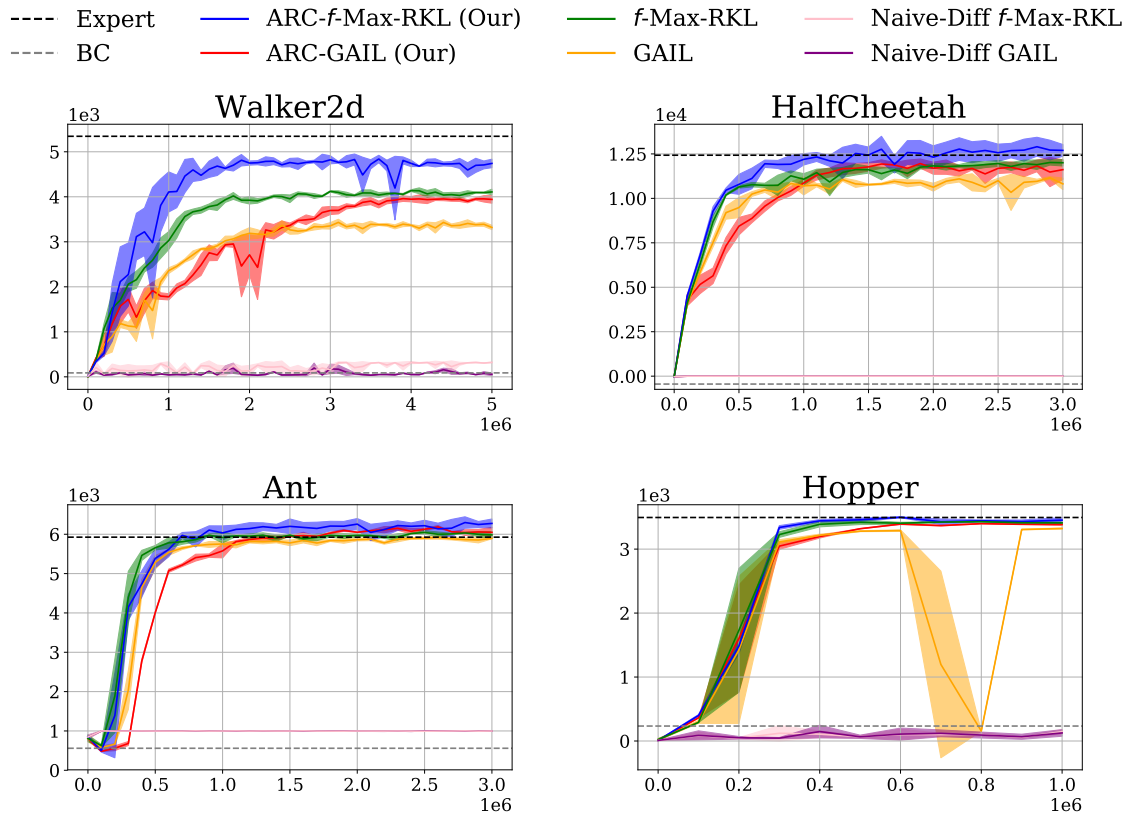


Figure 2.4: Episode return versus number of environment interaction steps for different Imitation Learning algorithms on Mujoco continuous-control environments.

2.7 Appendix

2.7.1 Derivations and Proofs

Unique optimality of C function

We restate Lemma 1 There exists a unique optimum C^* for any MDP.

Proof. The unique optimality of C function can be derived from the optimality ofthe Q function [66]. The optimum Q function, Q^* is defined as:

$$\begin{aligned} Q^*(s,a) &= \max_\pi Q^\pi(s,a) \\ &= \max_\pi [r(s,a) + C^\pi(s,a)] \\ &= r(s,a) + \max_\pi C^\pi(s,a) \\ &= r(s,a) + C^*(s,a) \qquad (2.16) \\ \therefore C^*(s,a) &= Q^*(s,a) - r(s,a) \qquad (2.17) \end{aligned}$$

Since Q^* is unique [66], (2.17) implies C^* must be unique. □

Bellman backup for C function

We restate Lemma 2. The recursive Bellman equation for C^π is as follows

$$C^\pi(s,a) = \gamma \sum_{s'} P(s'|s,a) \sum_{a'} \pi(a'|s') \left(r(s',a') + C^\pi(s',a')\right)$$

Proof. The derivation is similar to that of state value function V^π presented in [66]. We start deriving the Bellman backup equation for C^π function by expressing current $C(s_t, a_t)$ in terms of future $C(s_{t+1}, a_{t+1})$. In the following, the expectation is over the policy π and the transition dynamics $\mathcal{P}$ and is omitted for ease of notation.

$$C^{\pi}(s_t, a_t) = \mathbb{E} \sum_{k \geq 1} \gamma^k r_{t+k} \tag{2.18}$$

$$= \mathbb{E} \left(\gamma r_{t+1} + \sum_{k \geq 2} \gamma^k r_{t+k} \right) \tag{2.19}$$

$$= \gamma \left(\mathbb{E}[r_{t+1}] + \mathbb{E} \sum_{k \geq 1} \gamma^k r_{t+1} \right) \tag{2.20}$$

$$= \gamma \left(\mathbb{E}[r_{t+1}] + \mathbb{E} \sum_{k \geq 1} \gamma^k r_{t+1+k} \right) \tag{2.21}$$

$$= \gamma \mathbb{E} \left(r_{t+1} + C(s_{t+1}, a_{t+1}) \right) \tag{2.22}$$

$$\tag{2.23}$$

Using Eq. 2.22, we can write the recursive Bellman equation of C.

$$C(s, a) = \gamma \sum_{s'} P(s'|s, a) \sum_{a'} \pi(a'|s') \left(r(s', a') + C(s', a') \right) \tag{2.24}$$

□

2.7.2 Convergence of policy evaluation using C function

We restate Theorem 1. The following Bellman backup operation for policy evaluation using C function converges to the true C function, C^{π}

$$C^{n+1}(s, a) \leftarrow \gamma \sum_{s'} P(s'|s, a) \sum_{a'} \pi(a'|s') \left(r(s', a') + C^{n}(s', a') \right)$$

Proof. Let us define $F_{(.)}$ as the Bellman backup operation over the current estimates of C values:

$$F_C(s, a) = \gamma \sum_{s'} P(s'|s, a) \sum_{a'} \pi(a'|s') \left(r(s', a') + C(s', a') \right) \tag{2.25}$$

We prove that F is a contraction mapping w.r.t ∞ norm and hence is a fixed point

iteration. Let, C_1 and C_2 be any 2 sets of estimated C values.

$$||F_{C_1} - F_{C_2}||_\infty = \max_{s,a} |F_{C_1} - F_{C_2}| \tag{2.26}$$

$$= \gamma \max_{s,a} \left| \sum_{s'} P(s'|s,a) \sum_{a'} \pi(a'|s')[r(s',a') + C_1(s',a')] \right.$$

$$\left. - \sum_{s'} P(s'|s,a) \sum_{a'} \pi(a'|s')[r(s',a') + C_2(s',a')] \right| \tag{2.27}$$

$$= \gamma \left| \max_{s,a} \sum_{s'} P(s'|s,a) \sum_{a'} \pi(a'|s')(C_1(s',a') - C_2(s',a')) \right| \tag{2.28}$$

$$\leq \gamma \max_{s,a} \sum_{s'} P(s'|s,a) \sum_{a'} \pi(a'|s') \, |C_1(s',a') - C_2(s',a'))| \tag{2.29}$$

$$\leq \gamma \max_{s,a} \sum_{s'} P(s'|s,a) \max_{a'} |C_1(s',a') - C_2(s',a'))| \tag{2.30}$$

$$\leq \gamma \max_{s'} \max_{a'} |C_1(s',a') - C_2(s',a'))| \tag{2.31}$$

$$= \gamma \max_{s,a} |C_1(s,a) - C_2(s,a))| \tag{2.32}$$

$$= \gamma ||C_1 - C_2||_\infty \tag{2.33}$$

$$\therefore ||F_{C_1} - F_{C_2}||_\infty \leq \gamma ||C_1 - C_2||_\infty \tag{2.34}$$

Eq. 2.34 implies that iterative operation of $F_{(.)}$ converges to a fixed point. The true C^π function satisfies the Bellman equation Eq. 2.24. These two properties imply the policy evaluation converges to the true C^π function. □

2.8 Popular Algorithms

2.8.1 Policy Iteration using Q function

We restate the popular Policy Iteration using Q function algorithm in Algorithm 3.

Algorithm 3 Policy Iteration using Q function

Initialize $Q^0(s,a) \forall s,a$ **while** π *not converged* **do**
  // Policy evaluation
  for *n=1,2,... until* Q^n *converges* **do**
    $Q^{n+1}(s,a) \leftarrow r(s,a) + \gamma \sum_{s'} P(s'|s,a) \sum_{a'} \pi(a'|s') Q^n(s',a') \quad \forall s,a$
  end
  // Policy improvement
  $\pi(s,a) \leftarrow \begin{cases} 1, \text{ if } a = \text{argmax}_{a'} Q(s,a') \\ 0, \text{ otherwise} \end{cases} \quad \forall s,a$
end

2.9 Experimental Details

Environment We use Ant-v2, Walker-v2, HalfCheetah-v2 and Hopper-v2 Mujoco continuous-control environments from OpenAI Gym [12]. All 4 environments use Mujoco, a realistic physics-engine, to model the environment dynamics. The maximum time steps, T is set to 1000 in each environment.

Code We implemented our algorithm on top of the AIL code of [45]. The pre-implemented standard AIL algorithms (f-MAX-RKL, GAIL) used SAC [25] as the RL algorithm and the ARC aided AIL algorithms use SARC (Algorithm 2) as the RL algorithm.

Expert trajectories We used the expert trajectories provided by [45]. They used SAC [25] to train an expert in each environments. The policy network, π_θ was a tanh squashed Gaussian which parameterized the mean and standard deviation with two output heads. Each of the policy network, π_θ and the 2 critic networks, Q_{ϕ_1}, Q_{ϕ_1} was a $(64, 64)$ ReLU MLP. Each of them was optimized by Adam optimizer with a learning rate of 0.003. The entropy regularization coefficient, α was set to 1, the batch size was set to 256, the discount factor γ was set to 0.99 and the polyak averaging coefficient ζ for target networks was set to 0.995. The expert was trained for 1 million time steps on Hopper and 3 million time steps on the other environments. For each environment, we used 1 trajectory from the expert stochastic policies to train the imitation learning algorithms.

Standard AIL For the standard AIL algorithms (f-MAX-RKL [22] and GAIL [28]) we used the code provide by [45]. The standard AIL algorithms used SAC [25] as the RL algorithm. SAC used the same network and hyper-parameters that were used for training the expert policy except the learning rate and the entropy regularization coefficient, α. The learning rate was set to 0.001. α was set to 0.05 for HalfCheetah and to 0.2 for the other environments. The reward scale scale and gradient penalty coefficient were set to 0.2 and 4.0 respectively. In each environment, the observations were normalized in each dimension of the state using the mean and standard deviation of the expert trajectory.

For the discriminator, we used the same network architecture and hyper-parameters suggested by [45]. The discriminator was a (128,128) tanh MLP network with the output clipped within [-10,10]. The discriminator was optimized with Adam optimizer with a learning rate of 0.0003 and a batch size of 128. Once every 1000 environment steps, the discriminator and the policy were alternately trained for 100 iterations each.

Each AIL algorithm was trained for 1 million environment steps on Hopper, 3 million environment steps on Ant, HalfCheetah and 5 million environment steps on Walker2d.

ARC aided AIL For ARC aided AIL algorithms, we modified the SAC implementation of [45] to SARC (Algorithm 2). This was relatively straight forward, we used the same networks to parameterize C_{ϕ_1}, C_{ϕ_2} instead of Q_{ϕ_1}, Q_{ϕ_2} based on the steps of SARC (Algorithm 2). For SARC, we used the same network and hyper-parameters as SAC except the following changes. Learning rate was set to 0.0001. Entropy regularization coefficient, α was set to 0.05 for HalfCheetah and 1 for the other environments. No reward scaling was used (reward scale was set to 1). The C networks were updated 10 times for every update of the policy network. We did so because we noticed that otherwise the C networks (C_{ϕ_1}, C_{ϕ_2}) were slower to update as compared to the policy network.

The discriminator was the same as with standard AIL algorithms except it had 2 Resnet blocks of 128 dimension each, with batch normalization and leaky ReLU activation. These changes were motivated by common tricks to train stable GANs [23]. In GANs, the generator is differentiated through the discriminator and the use

of leaky ReLU and Resnet helps in gradient flow through the discriminator. In ARC aided AIL we have a similar scenario, the policy is differentiated through the reward function. We briefly tried to make the same changes with standard AIL algorithms as well but didn't see an improvement in performance.

Naive-Diff For the Naive-Diff aided AIL algorithms (Naive-Diff-f-MAX-RKL and Naive-Diff-GAIL), we used the same network architectures and hyper-parameters as with ARC aided AIL.

Behavior Cloning For Behavior Cloning, we trained the agent to regress on expert actions by minimizing the mean squared error for 10000 epochs using Adam optimizer with learning rate of 0.001 and batch size of 256.

Evaluation We evaluated all the imitation learning algorithms based on the true environment return achieved by the deterministic version of their policies. Each algorithm was run on 3 different seeds and each run was evaluated for 20 episodes. The final mean reward was used for comparing the algorithms. The results are presented in Table 2.2.

2.10 Extending to Real Robots

Although we have evaluated our proposed method - ARC aided AIL in simulation, we believe that it will perform well with real robots because of:

Better performance in physics-based simulation than prior work In this work we proposed ARC aided AIL, a modification to existing AIL algorithms which are already used with real robots. Since our proposed method outperforms standard AIL algorithms in physics-based simulation, it is likely to perform well on real robots.

Better sample-efficiency As we can see from Fig. 2.4, our proposed ARC aided AIL algorithms have similar sample efficiency as standard AIL algorithms which have been tested on real robots. In fact, ARC-f-MAX-RKL was more sample efficient than f-MAX-RKL and GAIL (Fig. 2.4).

Ease of hyper-parameter tuning We had to perform minimal hyper-parameter tuning both in terms of different ARC aided AIL algorithms (ARC-f-MAX-RKL and ARC-GAIL) and in terms of 4 different physics based simulated environments. This validates the stability of our proposed approach which means that our proposed approach is likely to perform well with minimal hyper-parameter tuning on a real robot.

Chapter 3

Natural Emergence of Heterogeneous Strategies in Artificially Intelligent Competitive Teams

3.1 Introduction

Multi agent systems can play an important role in scenarios such as disaster relief, defense against enemies and games. There have been studies on various aspects of it including task assignment [59], resilience to failure [50], scalability [3] and opponent modeling [74]. Multi agent systems become increasingly complex in mixed cooperative-competitive scenarios where an agent has to cooperate with other agents of the same team to jointly compete against the opposing team. It becomes difficult to model behavior of an agent or a team by hand and learning based methods are of particular appeal.

Our goal is to develop a learning based algorithm for decentralized control of multi agent systems in mixed cooperative-competitive scenarios with the ability to handle a variable number of agents, as some robots may get damaged in a real world scenario or some agents may get killed in a game. To be able to handle a variable

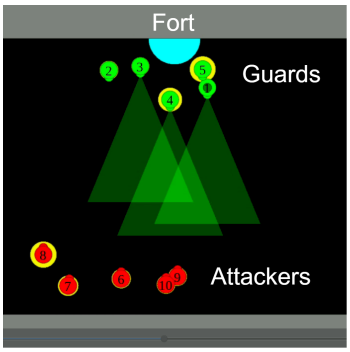


Figure 3.1: The FortAttack environment in which guards (green) need to protect the fort (cyan semicircle at the top) from the attackers (red). The attackers win when any one of them reaches the fort. Each agent can shoot a laser which can kill an opponent.

number of agents and to scale to many agents, we propose to use a Graph Neural Networks (GNNs) based architecture to model inter-agent interactions, similar to [3] and [11]. This approach relies on shared parameters amongst all agents in a team which renders all of them homogeneous. We aim to study if heterogeneous behavior can emerge out of such homogeneous agents.

Our contributions in this work are:

- We have developed a mixed cooperative-competitive multi agent environment called FortAttack with simple rules yet room for complex multi agent behavior.
- We show that using GNNs with a standard off the shelf reinforcement learning algorithm can effectively model inter agent interactions in a competitive multi agent setting.
- To train strong agents we need competitive opponents. Using an approach inspired by self play, we are able to create an auto curriculum that generates strong agents from scratch without using any expert knowledge. Strategies naturally evolved as a winning strategy from one team created pressure for the other team to be more competitive. We were able to achieve this by training on a commodity laptop.
- We show that highly competitive heterogeneous behavior can naturally emerge amongst homogeneous agents with symmetric reward structure (within the same team) when such behavior can lead to the team's success. Such behavior

implicitly includes heterogeneous task allocation and complex coordination within a team, none of which had to be explicitly crafted but can be extremely beneficial for multi agent systems.

3.2 Related Work

3.2.1 Multi-Agent Reinforcement Learning

The recent successes of reinforcement learning in games, [43], [62] and robotics, [58], [25] have encouraged researchers to extend reinforcement learning to multi agent settings.

There are three broad categories of approaches used, centralized, decentralized and a mix of the two. Centralized approaches have a single reinforcement learning agent for the entire team, which has global state information and selects joint actions for the team. However, the joint state and action spaces grows exponentially with the number of agents rendering centralized approaches difficult to scale, [13].

Independent Q-learning, [68], [67] is a decentralized approach where each agent learns separately with Q-learning, [73] and treats all other agents as parts of the environment. Inter agent interactions are not explicitly modeled and performance is generally sub-par.

Centralized learning with decentralized execution has gained attention because it is reasonable to remove communication restrictions at training time. Some approaches use a decentralized actor with a centralized critic, which is accessible only at training time. MADDPG, [40] learns a centralized critic for each agent and trains policies using DDPG, [38]. QMIX, [51] proposes a monotonic decomposition of action value function. However, the use of centralized critic requires that the number of agents be fixed in the environment.

GridNet, [27] addresses the issue of multiple and variable number of agents without exponentially growing the policy representation by representing a policy with an encoder-decoder architecture with convolution layers. However, the centralized execution realm renders it infeasible in many scenarios.

Graphs can naturally model multi agent systems with each node representing an agent. [65] modeled inter agent interactions in multi agent teams using GNNs

which can be learnt through back propagation. [30] proposed to use attention and [3] proposed to use an entity graph for augmenting environment information. However, these settings don't involve two opposing multi agent teams that both evolve by learning.

[11] explored multi agent reinforcement learning for the game of hide and seek. They find that increasingly complex behavior emerge out of simple rules of the game over many episodes of interactions. However, they relied on extremely heavy computations spanning over many millions of episodes of environment exploration.

We draw inspiration from [3] and [11]. For each team we propose to have two components within the graph, one to model the observations of the opponents and one to model the interactions with fellow team mates. Our work falls in the paradigm of centralized training with decentralized execution. We were able to train our agents in the FortAttack environment using the proposed approach on a commodity laptop. We believe that the reasonable computational requirement would encourage further research in the field of mixed cooperative-competitive MARL.

3.2.2 Multi-Agent Environments

Although there are many existing multi-agent environments, they suffer from the following deficiencies:

- Multi-Agent Particle Environment (MAPE) [40] doesn't consider two teams with multiple agents in each team.
- StarCraft II Learning Environment (SC2LE) [72] assumes a centralized controller for all agents in a team which is impractical for real world scenarios.
- Starcraft Multi-Agent Challenge (SMAC) [54] doesn't incorporate learning based opponents.
- RoboSumo [5] Doesn't scale to many agents (only contains 1 vs 1 scenarios).

Moreoever, SC2LE [72], SMAC [54] and SoboSumo [5] are computationally heavy environments.

To overcome these deficiencies, we design a new light-weight (can run on commodity laptop) mixed cooperative-competitive environment called FortAttack (Fig. 3.1) which can handle (1) Large number of agents, (2) Decentralized controllers, (3) Learning

based opponents, (4) Variable number of agents within a single episode and (5) Complex multi-agent strategies as is evident from our results (Section 3.5.1).

3.3 Method

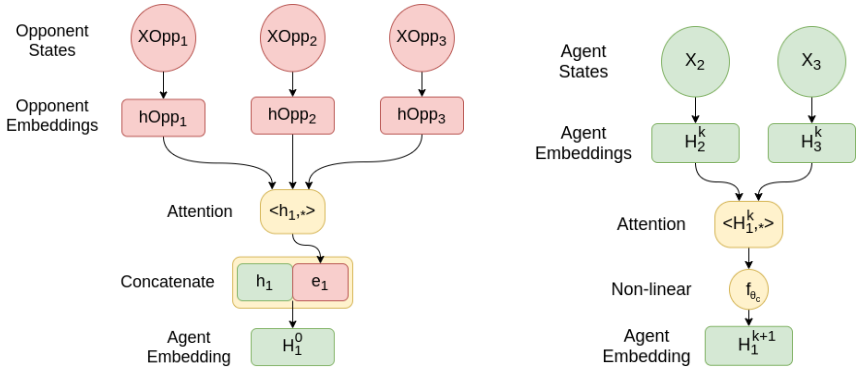


Figure 3.2: Modeling of inter agent interactions with Graph Neural Networks (GNNs) from the perspective of agent 1, in a 3 friendly agents vs 3 opponents scenario. Left: agent 1's embedding, H_1^0 is formed by taking into consideration the states of all opponents through an attention layer. Right: agent 1's embedding gets updated, $(H_1^k \rightarrow H_1^{k+1})$ by taking into consideration its team mates through an attention layer.

The agents in a multi-agent team can be treated as nodes of a graph to leverage the power of Graph Neural Networks (GNNs). GNNs form a deep-learning architecture where the computations at the nodes and edges of the graph are performed by neural networks (parameterized non-linear functions), [3]. Due to the presence of graph structure and multiple neural networks, they are called GNNs.

We describe our use of GNNs from the perspective of one team and use X_i to denote the state of i^{th} friendly agent in the team, which in our case is its position, orientation and velocity. We use $XOpp_j$ to denote the state of the j^{th} opponent in the opposing team. Let $S = \{1, 2, \ldots, N_1\}$ denote the set of friendly agents and $S_{Opp} = \{N_1 + 1, N_1 + 2, \ldots, N_1 + N_2\}$ denote the set of opponents. Note that a symmetric view can be presented from the perspective of the other team.

In the following, we describe how agent 1 processes the observations of its opponents and how it interacts with its teammates. Fig. 3.2 shows this pictorially for a 3 agents vs 3 agents scenario. All the other agents have a symmetric representation of interactions.

3.3.1 Modeling observation of opponents

Friendly agent 1 takes its state, X_1 and passes it through a non-linear function, f_{θ_a} to generate an embedding, h_1. Similarly, it forms an embedding, $hOpp_j$ from each of its opponents with the function f_{θ_b}.

$$h_1 = f_{\theta_a}(X_1) \tag{3.1}$$

$$hOpp_j = f_{\theta_b}(XOpp_j) \quad \forall j \in S_{Opp} \tag{3.2}$$

Note that the opponents don't share their information with the friendly agent 1. Friendly agent 1 merely makes its own observation of the opponents. It then computes a dot product attention, ψ_{1j} which describes how much attention it pays to each of its opponents. The dimension of h_1 and $hOpp_j$ are d_1 each. This attention allows agent 1 to compute a joint embedding, e_1 of all of its opponents.

$$\hat{\psi}_{1j} = \frac{1}{d_1} < h_1, hOpp_j > \quad \forall j \in S_{Opp} \tag{3.3}$$

$$\psi_{1j} = \frac{\exp(\hat{\psi}_{1j})}{\sum_{m \in S_{Opp}} \exp(\hat{\psi}_{1m})} \tag{3.4}$$

$$e_1 = \sum_{j \in S_{Opp}} \psi_{1j} hOpp_j \tag{3.5}$$

In Eq. 3.3, $<,>$ denotes vector dot product. Note that $\sum_{j \in S_{Opp}} \psi_{1j} = 1$ which ensures that the net attention paid by agent 1 to its opponents is fixed. Finally, e_1 is concatenated with h_1 to form an agent embedding, H_1^0:

$$H_1^0 = \text{concatenate}(h_1, e_1) \tag{3.6}$$

3.3.2 Modeling interactions with teammates

Agent 1 forms an embedding for each of its team mates with the non-linear function, f_{θ_a}.

$$H_i^0 = f_{\theta_a}(X_i) \quad \forall i \in S, i \neq 1 \tag{3.7}$$

Dimension of $H_i^k, \forall i \in S$ is d_2. Agent 1 computes a dot product attention, ϕ_{1i} with all of its team mates and updates it's embedding with a non-linear function, f_{θ_c}.

$$\hat{\phi}_{1i} = \frac{1}{d_2} < H_1^k, H_i^k > \quad \forall i \in S, i \neq 1 \tag{3.8}$$

$$\phi_{1i} = \frac{\exp(\hat{\phi}_{1i})}{\sum_{m \in S, m \neq 1} \exp(\hat{\phi}_{1m})} \tag{3.9}$$

$$\hat{H}_1^{k+1} = \sum_{i \in S, i \neq 1} \phi_{1i} H_i^k \tag{3.10}$$

$$H_1^{k+1} = f_{\theta_c}(\hat{H}_1^{k+1}) \tag{3.11}$$

Equations, 3.8 to 3.11 can be run over multiple iterations for $k = \{0, 1, \ldots, K\}$ to allow information propagation to other agents if agents can perceive only its local neighborhood similar to [3].

3.3.3 Policy

The final embedding of friendly agent 1, H_1^K is passed through a policy head. In our experiments, we use a stochastic policy in discrete action space and hence the policy head has a sigmoid activation which outputs a categorical distribution specifying the probability of each action, α_m.

$$\pi(\alpha_m|O_1) = \pi'(\alpha_m|H_1^K) = \text{sigmoid}(f_{\theta_d}(H_1^K)) \tag{3.12}$$
$$\text{where, } O_1 = \{X_i : i \in S\} \cup \{XOpp_j : j \in S_{Opp}\}$$

Here, O_1 is the observation of agent 1, which consists of its own state and the states of all other agents that it observes. This corresponds to a fully connected graph. We do this for simplicity. In practice, we could limit the observation space of an agent within a fixed neighborhood around the agent similar to [3] and [11].

3.3.4 Scalability and real world applicability

Due to the use of GNNs, the learn-able parameters for a team are the shared parameters, $\theta_a, \theta_b, \theta_c$ and θ_d of the functions, $f_{\theta_a}, f_{\theta_b}, f_{\theta_c}$ and f_{θ_d}, respectively which we model with fully connected neural networks. Note that the number of learn-able

parameters is independent of the number of agents and hence can scale to a large number of agents. This also allows us to handle a varying number of agents as agents might get killed during an episode and makes our approach applicable to real world scenarios where a robot may get damaged during a mission.

3.3.5 Training

Our approach follows the paradigm of centralized training with decentralized execution. During training, a single set of parameters are shared amongst teammates. We train our multi agent teams with Proximal Policy Optimization (PPO), [58]. At every training step, a fixed number of interactions are collected from the environment using the current policy for each agent and then each team is trained separately using PPO.

The shared parameters naturally share experiences amongst teammates and allow for training with fewer number of episodes. At test time, each agent maintains a copy of the parameters and can operate in decentralized fashion. We trained our agents on a commodity laptop with i7 processor and GTX 1060 graphics card. Training took about 1-2 days without parallelizing the environment.

3.4 Environment

We design a mixed cooperative-competitive environment called Fortattack with OpenAI Gym, [12] like interface. Fig. 3.1 shows a rendering of our environment. The environment consists of a team of guards, shown in green and a team of attackers, shown in red, that compete against each other. The attackers need to reach the fort which is shown as a cyan semi-circle at the top. Each agent can shoot a laser beam which can kill an opponent if it is within the beam window.

At the beginning of an episode, the guards are located randomly near the fort and the attackers are spawned at random locations near the bottom of the environment. The guards win if they manage to kill all attackers or manage to keep them away for a fixed time interval which is the episode length. The guards lose if even one attacker manages to reach the fort. The environment is built off of Multi-Agent Particle Environment, [40].

3.4.1 Observation space

Each agent can observe all the other agents in the environment. Hence, the observation space consists of states (positions, orientations and velocities) of team mates and opponents. We assume full observability as the environment is small in size. This can possibly be extended to observability in the local neighborhood such as in [3] and [11].

3.4.2 Action space

At each time step, an agent can choose one of 7 actions, accelerate in $\pm x$ direction, accelerate in $\pm y$ direction, rotate clockwise/anti-clockwise by a fixed angle or do nothing.

3.4.3 Reward structure

Each agent gets a reward which has components of its individual and the team's performance as described in Table 3.1. The last two rows show the major reward signals corresponding to winning and losing. The negative reward for wasting a laser shot is higher in magnitude for attackers than for guards. Otherwise, we observed that the attackers always managed to win. This reward structure can also be attributed to the fact that attackers in a real world scenario would like to sneak in and wouldn't want to shoot too often and reveal themselves to the guards.

Table 3.1: Reward structure

Sl. No.	**Event**	**Reward**
1	Guard i leaves the fort	Guard i gets -1 reward.
2	Guard i returns to the fort	Guard i gets +1 reward.
3	Attacker j moves closer to the fort	Attacker j gets small +ve reward = $2[D_j(t-1) - D_j(t)]$. Where, $D_j(t)$ = distance between attacker and fort at time t.
4	Attacker j moves away from the fort	Attacker j gets small -ve reward = $-2[D_j(t-1) - D_j(t)]$.
5	Guard i shoots attacker j with laser	Guard i gets +3 reward and attacker j gets -3 reward.
6	Attacker j shoots guard i with laser	Guard i gets -3 reward and attacker j gets +3 reward.
7	Agent i shoots laser but doesn't hit any opponent	Agent i gets low -ve reward (-0.1 if guard, -1 if attacker).
8	All attackers are killed	All alive guards get high +ve reward (+10). Attacker(s) that just got killed gets high -ve (-10) reward.
9	Attacker j reaches the fort	All alive guards high -ve reward. Attacker j gets high +ve reward.

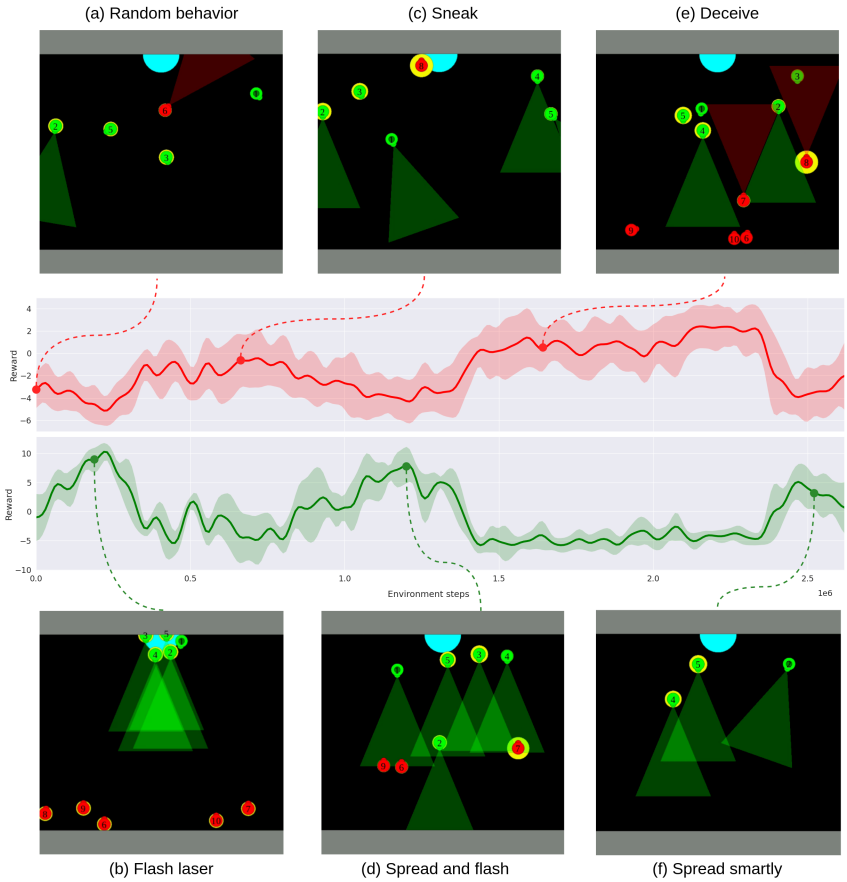


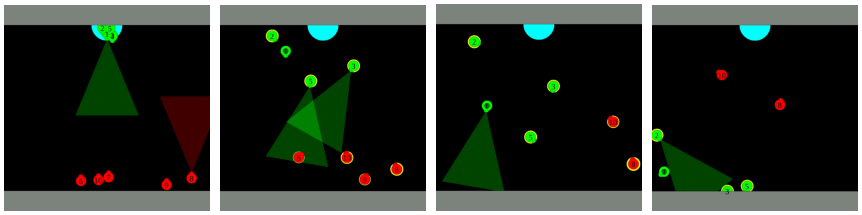
Figure 3.3: Average reward per agent per episode for the teams of attackers and guards as training progresses. The reward plots have distinct extrema and corresponding snapshots of the environment are shown. The x-axis shows the number of steps of environment interaction. The reward is plotted after Gaussian smoothing.

3.5 Results

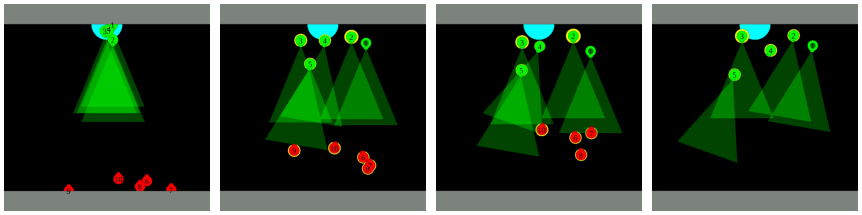
We show the results for the 5 guards vs 5 attackers scenario in the FortAttack environment.

3.5.1 Evolution of strategies

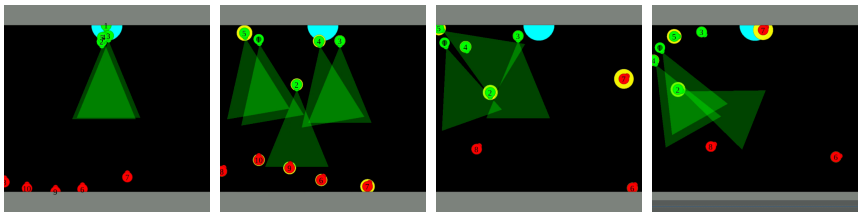
Fig. 3.3 shows the reward plot for attackers and guards and snapshots of specific checkpoints as training progresses. The reward for guards is roughly a mirror image of the reward for attackers as victory for one team means defeat for the other.



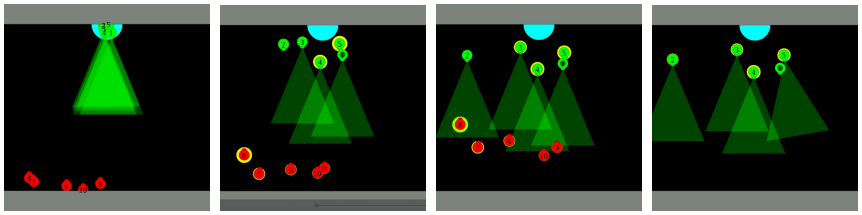
(a) Random exploration



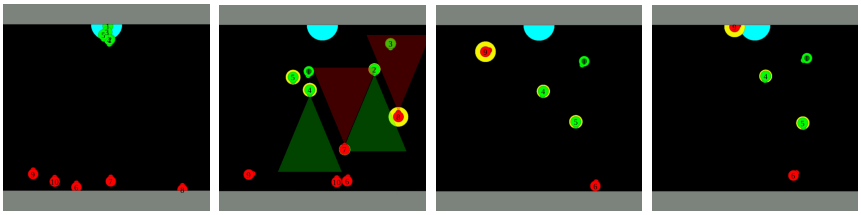
(b) Laser flashing strategy of guards



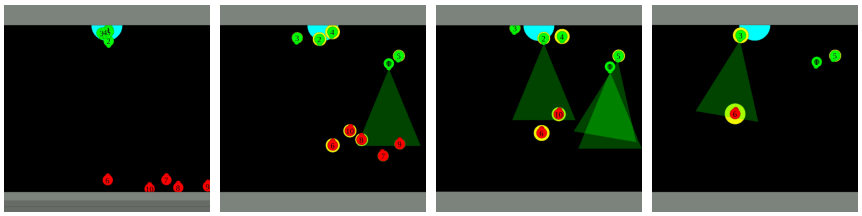
(c) Sneaking strategy of attackers



(d) Spreading and flashing strategy of guards



(e) Deception strategy of attackers



(f) Smartly spreading strategy of guards

Figure 3.4: Sample sequences for different strategies that evolved during training. Each row represents one sequence and time moves from left to right.

The rewards oscillate with multiple local extrema, i.e. maxima for one team and a corresponding minima for the other. These extrema correspond to increasingly complex strategies that evolve naturally - as one team gets better at its task, it creates pressure for the other team, which in turn comes up with a stronger and more complex strategic behavior.

1. *Random behavior*: At the beginning of training, agents randomly move around and shoot in the wild. They explore trying to make sense of the FortAttack environment and their goals in this world.
2. *Flash laser*: Attackers eventually learn to approach the fort and the guards adopt a simple strategy to win. They all continuously flash their lasers creating a protection zone in front of the fort which kills any attacker that tries to enter.
3. *Sneak*: As guards block entry from the front, attackers play smart. They approach from all the directions, some of them get killed but one of them manages to sneak in from the side.
4. *Spread and flash*: In response to the sneaking behavior, the guards learn to spread out and kill all attackers before they can sneak in.
5. *Deceive*: To tackle the strong guards, the attackers come up with the strategy of deception. Most of them move forward from the right while one holds back on the left. The guards start shooting at the attackers on the right which diverts their attention from the single attacker on the left. This attacker quietly waits for the right moment to sneak in, bringing victory for the whole team. Note that this strategy requires heterogeneous behavior amongst the homogeneous agents, which naturally evolved without explicitly being encouraged to do so.
6. *Spread smartly*: In response to this, the guards learn to spread smartly, covering a wider region and killing attackers before they can sneak in.

3.5.2 Being Attentive

In each of the environment snapshots in Fig. 3.3 and Fig. 3.4, we visualize the attention paid by one alive guard to all the other agents. This guard has a dark green dot at it's center. All the other agents have yellow rings around them, with the sizes of the rings being proportional to the attention values. Eg. in Fig. 3.4(e), agent

1 initially paid roughly uniform and low attention to all attackers when they were far away. Then, it started paying more attention to agent 8, which was attacking aggressively from the right. Little did it know that it was being deceived by the clever attackers. When agent 9 reached near the fort, agent 1 finally started paying more attention to the sneaky agent 9 but it was too late and the attackers had successfully deceived it.

3.5.3 Ensemble strategies

To train and generate strong agents, we first need strong opponents to train against. The learnt strategies in Section 3.5.1 give us a natural way to generate strategies from simple rules of the game. If we wish to get strong guards, we can train a single guard policy against all of the attacker strategies, by randomly sampling one attacker strategy for each environment episode. Fig. 3.5 shows the reward for guards as training progresses. This time, the reward for guards continually increases and doesn't show an oscillating behavior.

Figure 3.5: Average reward per agent per episode for guards as ensemble training progresses. The reward is shown after Gaussian smoothing.

Chapter 4

Hiding Leader's Identity in Leader-Follower Navigation through Multi-Agent Reinforcement Learning

Multi-robot systems are useful for many scenarios such as drone-delivery [14], agriculture [6], search-and-rescue [48], disaster relief [24] and defense [15]. A class of multi-robot algorithms called leader-follower navigation is particularly popular as it simplifies the task of controlling multiple robots. It involves at least one leader robot which the other (follower) robots follow. Using leader follower navigation, it is sufficient to command only the leader robot and the follower robots simply follow their leader.

Clearly, the leader is crucial for the robot team's success. Imagine a critical scenario such as a multi-robot team in a disaster relief mission as shown in Fig. 4.1. An external adversary (enemy) who wishes to sabotage the robot team's mission can simply identify and harm just the leader. This will compromise the whole robot team's mission. Thus, it is crucial to hide the leader's identity in such critical scenarios. Even if we make the visual appearance of the leader similar to the followers, it is possible to identify the leader by observing the motion of all the robots over time. E.g. the leader is usually ahead of the followers which an adversary can notice and

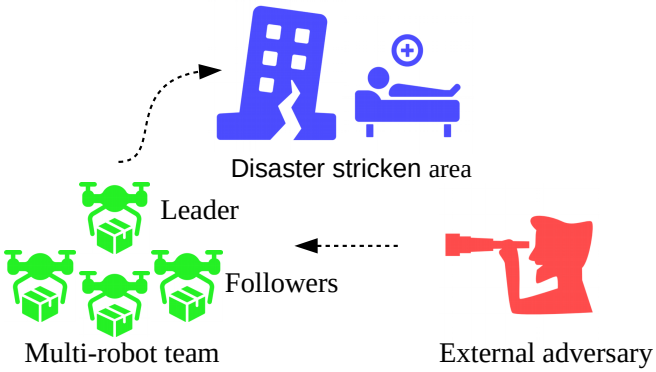


Figure 4.1: A leader-follower multi-robot team in a disaster relief mission in the presence of an external adversary. It is crucial to hide the leader's identity from the adversary because if the adversary harms the leader then the whole team's mission would be compromised.

identify the leader.

We propose a defense mechanism of hiding the leader's identity by ensuring the leader moves in a way that behaviorally camouflages it with the followers, making it difficult for an adversary to identify the leader. Here, by "behavioral camouflage" we refer to leader behavior (motion) that resembles followers' motion, blending it with the followers. Traditional leader-follower controllers are not good at hiding the leader's identity - including an approach [77] that explicitly tried to hide the leader's identity, as we will see in Section 4.4.2). Moreover, it is difficult to design a leader identity hiding multi-agent controller by hand, since such a controller would require complex multi-agent coordination.

Hence, instead of relying on traditional leader-follower controllers, we propose to leverage on the recent advancements in Multi-Agent Reinforcement Learning (MARL), Graph Neural Networks (GNNs) and adversarial training. MARL allows us to define the objective of a multi-agent (in our case multi-robot) team through scalar reward signals given to each agent which it tries to maximize. GNNs form a deep learning architecture that allows us to model and train a large & variable number of agents, which is appealing for real world applications. Adversarial training allows us to incorporate an intelligent artificial adversary that seeks to identify the leader.

We combine the power of MARL, GNNs and adversarial training for the task of hiding the leader's identity through a 3-stage training process. First, agents (robots) are trained with MARL to maximize a primary task reward, without caring about hiding the leader's identity. Then, robot trajectories (spatial coordinates over time)

are collected from this multi-robot team. These trajectories are used as training data to train an artificial adversary with supervised learning to identify the leader. Finally, the agents are trained again with MARL - but this time they are given both a primary task reward and an identity hiding reward.

We test our approach on a simulated multi-robot goal reaching task where the goal is known only to the leader and there is no communication between the robots. Using our proposed approach, the multi-robot team successfully reaches the goal while hiding the leader's identity from an artificial adversary, outperforming the baselines - both traditional leader-follower controller and MARL algorithms. We also evaluate human performance in inferring the leader's identity from multi-robot navigation videos and found that even humans found it hard to identify the leader when the multi-robot team deployed our proposed navigation algorithm.

The paper makes the following contributions:

1. We bring MARL, GNNs and adversarial training under one umbrella and present a multi-stage training process for the task of hiding the leader in a multi-robot team as a defense mechanism against an external adversary.
2. We propose a novel deep learning architecture called Scalable-LSTM for modeling an artificial adversary.
3. Our method is effective not only against an artificial adversary but it also "fools" human observers, i.e human observers do not detect the leader. To the best of our knowledge this is the first time this effect has been reported in the literature.
4. We show that our approach can generalize to multi-robot teams with different sizes in a 0-shot fashion (wihout any fine tuning).

4.1 Related Work

Leader follower navigation is popular in the robotics community and has been studied in various contexts. [53] addressed the issues of obstacle avoidance and connectivity preservation. [70] proposed an adaptive strategy of formation reconfiguration. [64] experimented with a leader-follower control algorithm on various mobile robots. All of these relied on traditional control based algorithms that didn't involve deep learning

or Reinforcement Learning (RL).

Recent work has tried to incorporate RL into the leader-follower navigation problem [41]. However, their method is designed for only 2-member robot teams with a single leader and a single follower. [78] showed brief results with 2 followers and mentioned that they found it intractable to train a large number of agents due to exponentially growing state and action spaces as robot team size increases.

We wish to not only leverage on the power of RL but also have a large robot team with multiple followers. Because of this, we build up on our previous work on Multi Agent Reinforcement Learning (MARL) with Graph Neural Networks (GNNs) [15]. We have previously shown that by combining MARL & GNNs large multi-agent teams can be tractably trained with reasonable computational resources (even a commodity laptop). In this work we suitably modify the GNNs architecture to incorporate the constraints of the leader-follower navigation problem, as delineated in Section 4.3.3.

Once we have multiple robots in the leader-follower problem, protecting the leader's identity becomes crucial, as we have already discussed in Section **??**. The importance of hiding the leader's identity has received limited attention in the past. We only found [77] to have attempted to address this issue. They used a traditional leader-follower controller and tuned its parameters with classical Genetic Algorithms (GAs) to hide leader's identity from an artificial adversary - a Convolutional Neural Network (CNN). However their results have limited evaluation. The adversary isn't benchmarked against other adversaries, which could mean that the adversary isn't smart enough and hence is easily deceived. In our experiments we saw that their approach couldn't hide the leader's identity when humans (instead of their artificial adversary) tried to identify the leader.

We instead propose a novel adversary architecture called Scalable LSTM (Long Short Term Memory) in Section 4.3.4 and show its superiority over existing deep learning architectures in Section 4.4.1. We experimentally show that our approach, relying on MARL & GNNs rather than traditional leader-follower controllers, outperforms existing approach, [77] that tried to hide the leader's identity. Further, we show that even humans have low accuracy in identifying the leader in a multi-robot team that navigates using our proposed approach.

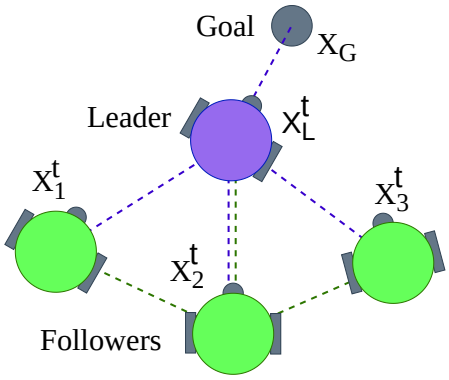


Figure 4.2: A goal reaching task for a leader-follower multi-robot team. All robots can sense the neighboring robots and additionally the leader knows the goal location. All robots have the same visual appearance, the leader is shown in a different color only for illustrative purpose.

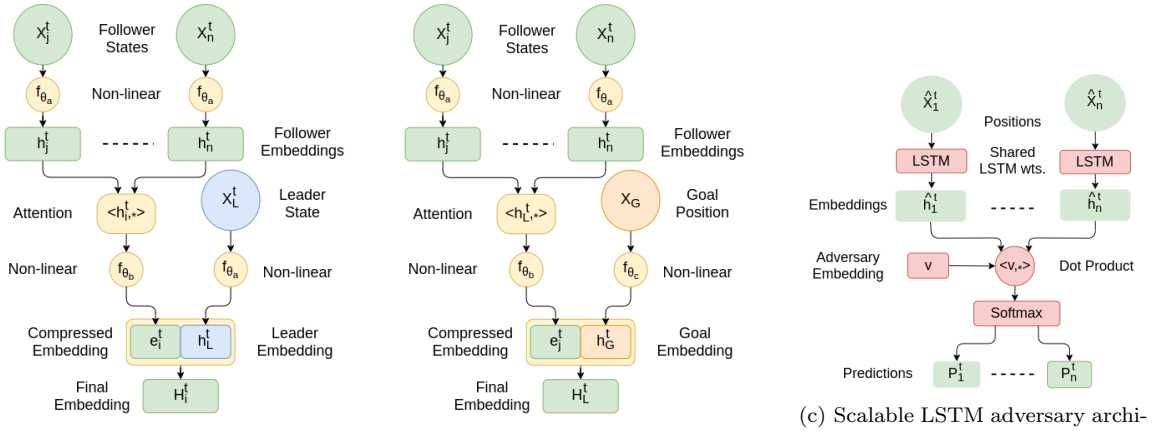
4.2 Problem Statement

There is a n-robot team with 1 leader, L and $(n-1)$ followers, as shown in Fig. 4.2 with no communication between them. At every time step t, each robot can sense its own state, X_i^t and the states of its neighbors, $\{X_j^t \mid j \neq i\}$ using on-board sensors. At the start of the mission, the goal is randomly located in the environment and only the leader knows its location X_G. E.g. in Fig. 4.2 the leader robot knows the goal and senses the states of all the neighboring followers; follower robot 2 only senses the states of the neighboring followers (to its left and right) and the leader.

Our objectives are (i) decentralized control for navigating the leader-follower multi-robot team to the goal location; and (ii) hide the leader's identity from an external adversary.

4.3 Method

We will use the terms *robot* and *agent* interchangeably. Let $Q = \{1, 2, \ldots, n\}$ denote the set of all agents and $S = Q - \{L\}$ denote the set of followers (excluding the leader). $\mathcal{A}$ denotes the external artificial adversary. We now describe the various components of our method.



(a) Follower architecture (b) Leader architecture (c) Scalable LSTM adversary architecture

Figure 4.3: GNNs based multi-agent architecture (Fig. 4.3a,4.3b) and Scalable-LSTM adversary architecture (Fig. 4.3c).

4.3.1 MARL Formulation

The problem is formulated as a Decentralized Partially Observable Markov Decision Process (Dec-POMDP), [7] with the following components:

States and observations

The state of i^{th} agent at time step t is denoted by $X_i^t \in \mathbb{R}^4$, which consists of its planar position and velocity. $X^t = \{X_i^t \mid \forall i \in Q\}$ denotes the collective state of all agents at time t. $\bar{X}^t = X^t \bigcup X_G$ denotes the full environment state at time t.

Action space

Each agent has an action space $a_i^t \in A_i$ that consists of accelerating in the $\pm x$ or $\pm y$ direction or not accelerating at all.

Policy

Each agent follows a policy π_i to select its actions $a_i^t \sim \pi_i$. The leader's policy $a_L^t \sim \pi_L(a \mid \bar{X}^t)$ is conditioned on $\bar{X}^t = X^t \bigcup X_G$ which contains goal information. On the other other hand each follower's policy $a_i^t \sim \pi_i(a \mid X^t)$ is conditioned on X^t which does not contain goal information.

Reward structure

The primary task reward for agent i at time t is:

$$r_i^t = \lambda_r \left(||X_G - \hat{X}_i^{t-1}||_2 - ||X_G - \hat{X}_i^t||_2 \right) \tag{4.1}$$

Here $\hat{X}_i^t$ denotes the position of the agent and $\lambda_r = 100$ is a scaling factor. This reward indicates how much closer the agent reached towards the goal in the current time step. Note that the same reward structure is used for both the leader and the followers. The reward encourages the leader to take large steps towards the goal. The followers don't have access to the goal location since their policy is not conditioned on X_G. They are indirectly encouraged to follow the leader which in turn will lead them to the goal.

The standard MARL objective (Naive MARL) for this reward signal is to maximize the cumulative reward, Eq. 4.2. Here, $\gamma = 0.99$ is the discount factor. We add an additional identity hiding reward, μ_i^t which is indicative of how well the multi-robot team is hiding its leader's identity, Eq. 4.3.

$$\text{Objective}_{\text{Naive MARL}} \equiv \max \mathbb{E} \left[\sum_{i=1}^{n} \sum_{t \geq 0} \gamma^t r_i^t \right] \tag{4.2}$$

$$\text{Objective}_{\text{Proposed}} \equiv \max \mathbb{E} \left[\sum_{i=1}^{n} \sum_{t \geq 0} \gamma^t (r_i^t + \lambda_\mu \mu_i^t) \right] \tag{4.3}$$

$\lambda_\mu = 1$ is a scaling factor controlling the importance of hiding leader's identity over the primary task. Computation of μ_i^t is described in the following sub-section in Eq. 4.5. Note that the rewards r_i^t, μ_i^t are only required for training and are not accessible to the agents at test time.

4.3.2 Adversarial Training

The adversary's goal is to identify the ID of the leader from $i = \{1, 2, \ldots, n\}$ since there are n agents. the adversary ($\mathcal{A}$) observes the positions of all the agents $\hat{X}_i^t$ upto

the current time step $O^t = \{\hat{X}_i^k | i \in Q, k \leq t\}$ and predicts the leader ID l_{pred}^t.

$$l_{pred}^t = \mathcal{A}(O^t) = \underset{i}{\text{argmax}}\, P_{\mathcal{A}}(i \mid O^t) \tag{4.4}$$

$$\mu_i^t = -\mathbb{I}(l_{pred}^t = L) \tag{4.5}$$

The adversary is trained with multi-class (since one amongst the n agents is the leader) cross entropy loss which tries to match the predicted leader ID l_{pred}^t to the true leader ID L. Once the adversary is trained, its prediction is used to generate identity hiding reward, μ_i^t (Eq. 4.5) as a feedback signal to the multi-robot team (Eq. 4.3). The negative sign in Eq. 4.5 denotes that all agents get the same negative identity hiding reward, i.e. if the leader is identified by the adversary all the agents are penalized. This encourages cooperation and team spirit in the multi-agent team to jointly deceive the adversary by coordinating their motion.

4.3.3 Graph Neural Networks Multi-agent Architecture

The agents (robots) in the multi-robot team can be treated as nodes of a graph to leverage the power of Graph Neural Networks (GNNs). GNNs form a deep-learning architecture where the computations at the nodes and edges of the graph are performed by neural networks (parameterized non-linear functions), [15, 17]. Due to the presence of graph structure and multiple neural networks, they are called GNNs.

We incorporate the constraints of the leader-follower problem by ensuring that the goal information is accessible only to the leader and not the followers. In the following we describe the computations performed by an arbitrary follower and the leader.

Followers

Computation performed by an arbitrary follower agent i is pictorially shown in Fig. 4.3a. At every time step t, it takes as input its state, X_i^t and passes it through a non-linear function, f_{θ_a} to compute an embedding, h_i^t. Similarly, it computes embeddings for all the other neighboring agents.

$$h_i^t = f_{\theta_a}(X_i^t) \quad \forall i \in Q \tag{4.6}$$

Follower agent i then computes dot product attention, ψ_{ij} with all the other neighboring followers v_j's.

$$\hat{\psi}_{ij}^t = \frac{1}{d} < h_i^t, h_j^t > \quad \forall j \in S, j \neq i \tag{4.7}$$

$$\psi_{ij}^t = \frac{\exp(\hat{\psi}_{ij}^t)}{\sum_{k \in S, k \neq i} \exp(\hat{\psi}_{ik}^t)} \tag{4.8}$$

$$m_i^t = \sum_{j \in S, j \neq i} \psi_{ij}^t h_j^t \tag{4.9}$$

$$e_i^t = f_{\theta_b}(\text{concat}(h_i^t, m_i^t)) \tag{4.10}$$

d is the dimension of the vectors in dot product $<,>$. ψ_{ij}^t denotes the attention paid by agent follower agent i to follower agent j at time t. The total attention paid to the neighbors sums to 1 due to the normalization in Eq. 4.8. It then concatenates e_i^t with leader embedding, h_L^t to compute its final embedding, H_1^t.

$$H_i^t = \text{concat}(e_i^t, h_L^t) \tag{4.11}$$

Leader

The leader's computation is pictorially shown in Fig. 4.3b. It is similar to followers' computations with the key differnces being (i) Leader state (X_L^t) is replaced by goal position (X_G) and (ii) f_{θ_c} is used instead of f_{θ_a} for computing goal embedding.

Policy and Value function

Once each follower computes its final embedding, it conditions its policy and value function on it.

$$\pi_i(a \mid X^t) = f_{\theta_d}(a \mid H_i^t) \quad \forall i \in S \tag{4.12}$$

$$V_{\pi_i}(X^t) = f_{\theta_e}(H_i^t) \quad \forall i \in S \tag{4.13}$$

The leader computes its policy and value function in a similar way but uses a different set of parameters.

$$\pi_L(a \mid \bar{X}^t) = f_{\theta_f}(a \mid H_L^t) \tag{4.14}$$

$$V_{\pi_L}(\bar{X}^t) = f_{\theta_g}(H_L^t) \tag{4.15}$$

4.3.4 Scalable LSTM Adversary Architecture

Long Short Term Memory (LSTM), [29] is a memory based deep learning architecture that can work with sequential data. It takes data at current time step as input and retains the past data in memory. It is temporally adaptable since it can handle input of any time duration.

However, it is not adaptable to different number of agents. This is because, the input & output sizes at time t depend on the number of agents n and an LSTM can be designed to handle only fixed sizes of input & output. If we directly feed the positions of all agents at time t to an LSTM, the input size would be $2 \times n$ (planar position) and the output size would be n (probability of each agent being the leader).

Scalable-LSTM adversary architecture builds on top LSTM. It inputs the position of each agent separately (to handle variable size inputs) and has a dot product operation (to handle variable size outputs).

At every time step t, the adversary takes as input the position of each agent separately, $\hat{X}_i^t$ and passes it through an LSTM to compute an embedding $\hat{h}_i^t$. Therefore, for n agents there would be n passes through the same LSTM as follows:

$$\hat{h}_i^t, \hat{c}_i^t = \text{LSTM}(\hat{X}_i^t, \hat{h}_i^{t-1}, \hat{c}_i^{t-1}) \quad \forall i \in Q \tag{4.16}$$

Because the adversary does n separate passes, the LSTM gets fixed size (=2) input irrespective of how large n is. Here, $\hat{h}_i^t$ and $\hat{c}_i^t$ denote the hidden state and cell state of LSTM at time t.

The adversary then computes pairwise dot product between each agent's embedding, $\hat{h}_i^t$ and an adversary embedding, v (a trainable parameter vector) as in Eq. 4.17. The output is passed through soft-max function to compute its probabilistic belief of

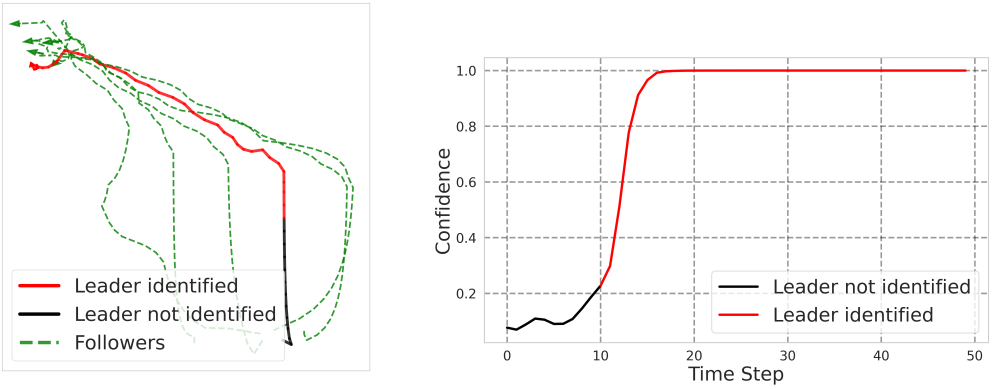
the leader's identity as in Eq. 4.18.

$$y_i^t =< v, \hat{h}_i^t > \quad \forall i \in Q \tag{4.17}$$

$$P_{\mathcal{A}}(i|O^t) = \frac{\exp(y_i^t)}{\sum_{m \in Q} \exp(y_m^t)} \quad \forall i \in Q \tag{4.18}$$

If we have n agents, there would be n dot-products $y_i^t =< v, \hat{h}_i^t >$ for $i = 1$ to n and correspondingly n output probability values $P_{\mathcal{A}}(i|O^t)$, irrespective of dimension of h_i^t. Thus, number of outputs automatically scales with no. of agents. Here, O^t denotes the observations of the adversary upto time step t, i.e. $O^t = \{\hat{X}_i^k | i \in Q, k \leq t\}$.

The adversary architecture consists of two sets of trainable parameters, LSTM weights and a adversary embedding v which we collectively denote as $\theta_{\mathcal{A}}$.



(a) Naive MARL Multi-robot trajectories

(b) Adversary's confidence increases over time

Figure 4.4: An episode of Naive MARL Multi robot trajectories (Fig. 4.4a) which the Scalable-LSTM adversary observes and tries to identify the leader. Scalable-LSTM predicts the leader at every time step t based on the trajectory observation till time t. Initially it fails to identify the leader correctly (shown in black) but within 10 time steps in predicts leader correctly (shown in red). Fig. 4.4b shows confidence of the adversary on its prediction for the same episode. Again we can observe that initially Scalable LSTM has low confidence and fails to identify the leader (shown in black) but within 10 time it identifies the leader correctly (shown in red) and within 20 time steps its confidence is almost 1 (100% confident).

4.3.5 Scalability, Adaptability and Decentralized Control

The trainable parameters in our GNNs based Multi-agent architecture are $\theta = \{\theta_a, \theta_b, \ldots, \theta_g\}$ of the non-linear functions $f_{\theta_a}, f_{\theta_b}, \ldots, f_{\theta_g}$ respectively (we model these functions as neural networks, hence the name GNNs). Since the number of

parameters is independent of the number of agents, the architecture is scalable and adaptable to different number of agents. The scalability and adaptability of Scalable LSTM adversary architecture is already described in the previous section (4.3.4). We validate these in our experimental results (Section 4.4.1 and 4.4.2).

The parameters θ_a, θ_b are shared across all agents and the paramters θ_d, θ_e are shared across all the followers. This means that each agent maintains a separate copy of their respective parameters at test time for decentralized control.

4.3.6 Training

We train a leader-hiding multi-agent strategy in 3 stages:

Training without identity hiding reward

We first train a multi-agent team with the Naive MARL objective (Eq. 4.2, doesn't include identity hiding reward μ_i^t) using Proximal Policy Optimization (PPO) [58], a popular model-free RL algorithm. The multi-agent team learns to navigate to the goal location without caring about hiding the leader's identity.

Training an adversary

Once we have a trained Naive MARL policy, we run the policy for multiple episodes and save the multi-robot trajectories along with the true leader ID in each episode. These trajectories and leader ID's are provided to the adversary as training data. The adversary is trained with Stochastic Gradient Descent (SGD) supervised learning algorithm to predict the leader ID by minimizing the multi-class cross-entropy loss.

Training with identity hiding reward

Now that we have a trained adversary, we train a new multi-agent policy which tries to maximize both goal reaching reward and identity hiding reward (Eq. 4.3 using PPO).

Due to the use of shared parameters in GNNs, our proposed approach is very simple to train with reasonable computational resources. We performed all our training on a commodity laptop with i7-7700HQ CPU, 16GB of RAM and a GTX

1060 GPU.

Remark

We would like to highlight that our formulation is consistent with the leader-follower problem because:

1. Followers are not given goal information as input. As shown in Eq. 4.12, follower policies are conditioned on the embedding H_i^t which doesn't contain any goal information. Goal information X_G is completely absent from the GNNs based follower architecture, Fig. 4.3a.
2. Reward is NOT accessible at test time. Primary task reward is calculated using goal information, Eq. 4.1. This might raise the confusion that the followers have access to goal information at test time. This is not the case as the agents have access to the reward only at training time and not at test time.
3. Goal location is randomly changed in every episode. At the start of every episode, the goal is placed at a random location. The test goal location is different from the training location and it is impossible for the multi-agent policy to memorize a fixed goal-location seen during training.

4.3.7 Environment

The environment is a simulated goal reaching task built using Multi Agent Particle Environment (MAPE) [40].

4.4 Results

4.4.1 Adversary performance

Table 4.1: Comparison of Adversary Architectures

Architecture	Accuracy (max. 1)	No. of Params.	Adaptability	
			Temporal	No. of agents
Random guess	0.20	0	✓	✓
LSTM	0.95	2574	✓	✗
Zheng et al. [77]	0.97	109286	✗	✗
Scalable-LSTM	**0.99**	**936**	✓	✓

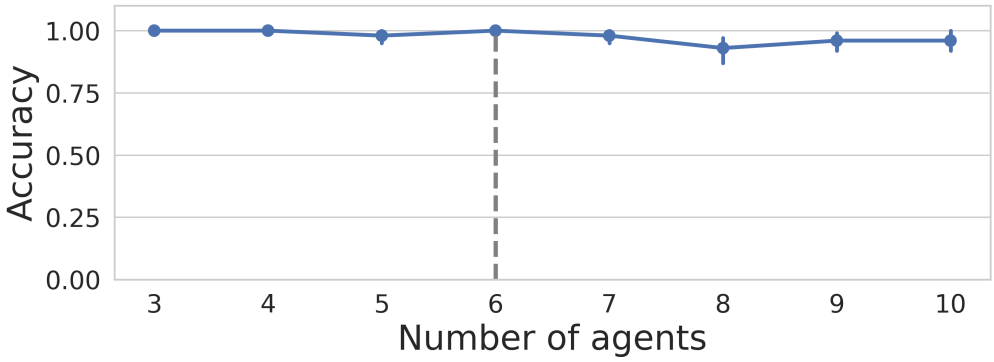
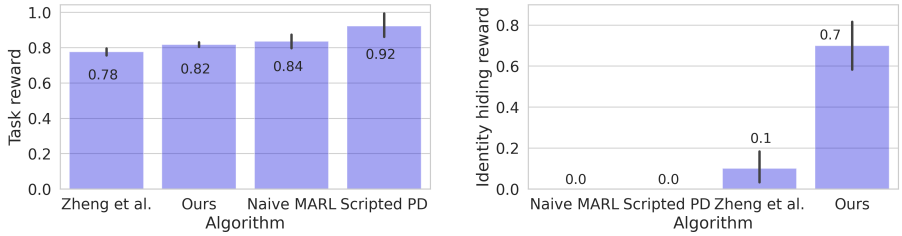


Figure 4.5: Scalable-LSTM adversary's 0-shot generalization to different number of agents. The blue curve, which shows the accuracy of Scalable-LSTM in identifying the leader in robot teams with different number of agents, constantly stays high (close to 1). Although Scalable-LSTM adversary was trained with only 6 agents (shown with dashed vertical line), it had high accuracy in identifying the leader in robot teams with number of agents varying between 3 and 10.



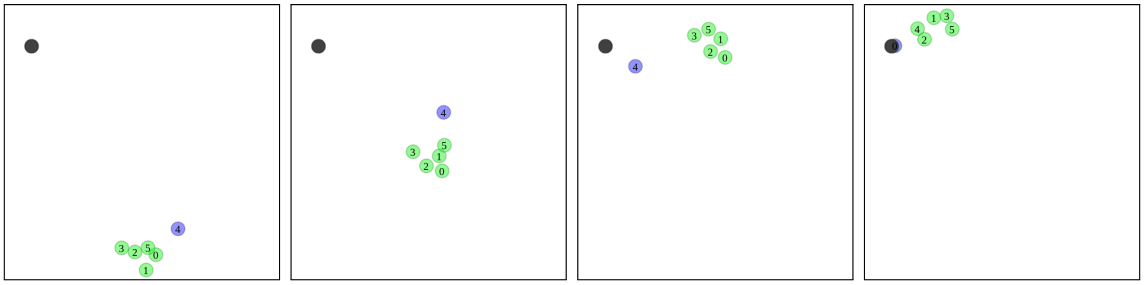
(a) Primary task reward

(b) Identity hiding reward

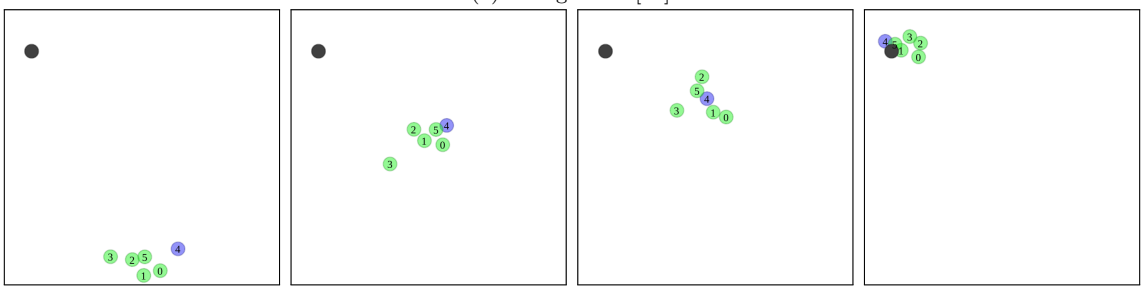
Figure 4.6: Multi-agent performance on primary task reward and identity hiding reward using different algorithms. Values are normalized between 0 and 1.

We first show that our proposed Scalable-LSTM adversary architecture performs extremely well in identifying the leader when the multi-robot team doesn't try to hide it's leader's identity. This is important because there would be no point in designing and evaluating a multi-agent policy against a weak adversary.

We collect 100 episodes of multi-agent (6 agents) trajectories from a Naive MARL policy (Section 4.3.6) and test the performance of Scalable-LSTM along with other baselines in identifying the leader correctly. The baselines includes random guess and deep learning adversary architectures LSTM, [29] and Zheng et al. [77] which has a Convolutional Neural Network (CNN) architecture. Scalable-LSTM has an accuracy of 0.99 which is higher than that of LSTM (0.95) and Zheng et al. (0.97) as shown in



(a) Zheng et al. [77]



(b) Ours

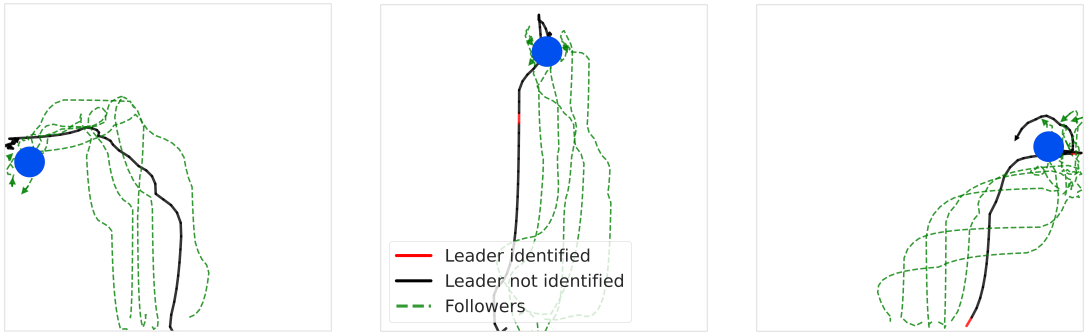
Figure 4.7: Multi-agent navigation using approach of Zheng et al. [77] and our approach for the same goal location (shown as black circle). The leader is shown as a blue circle while the followers are shown as green circles. All agents have a number written on them - e.g. the leader (blue circle) is numbered 4. The leader's identity isn't concealed well by Zheng et al. (Fig. 4.7a) as it is clearly ahead of the followers. Using our approach, the leader smartly moves with the followers as a group, hiding its identity (Fig. 4.7b) - E.g. in 3rd snapshot of Fig. 4.7b, a follower agent (numbered 3) deceptively seems to be leading the multi-agent team while the leader is behind.

Table 4.1. Table 4.1 also shows that Scalable-LSTM has 2 orders of magnitude lesser parameters than Zheng et al. [77] and still performs better.

Another advantage of Scalabe-LSTM over the baseline deep learning architectures is that it is adaptable both temporally and in the number of agents, which is appealing for real world applications as an adversary might observe only a part of the trajectory or the multi-agent team size might change.

Temporal adaptability

We show an example episode in Fig. 4.4 to demonstrate temporal adaptability. The adversary can make a prediction of the leader's identity at every time step, i.e. even from partial trajectories. Within 10 time steps, the adversary correctly identifies the



(a) Goal location 1 (b) Goal Location 2 (c) Goal location 3

Figure 4.8: Leader-follower navigation to different goal locations (shown as blue circle) using our proposed approach. The multi-agent team could successfully reach different goal locations while fooling the adversary which made a wrong prediction of the leader's identity as depicted by the black color of the leader trajectory in most regions.

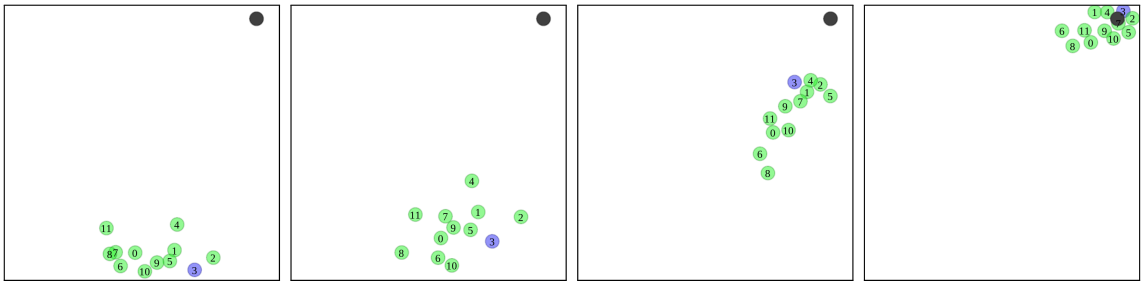
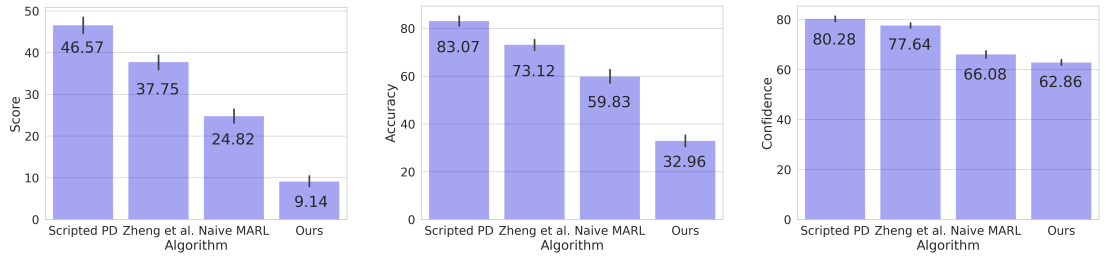


Figure 4.9: 0-shot generalization of our proposed approach to different multi-agent team size. We trained a multi-agent policy with $n = 6$ agents using our approach directly tested the performance with $2n = 12$ agents without any fine-tuning. The multi-agent team with double the number of agents in able to navigate to the goal location while successfully hiding the leader within the followers.

leader and as time flows it predicts the true leader with higher confidence (assigns higher probability to true leader).

Adaptability in no. of agents

We trained the Scalable-LSTM adversary to identify the leader in 6-agent team and tested it on 3 to 10 agent teams without any fine tuning on different team sizes. Leader prediction accuracy remains high in all cases as shown in Fig. 4.5. Thus Scalable-LSTM shows 0-shot generalization to different number of agents.



(a) Humans' Score (b) Humans' Accuracy (c) Humans' Confidence

Figure 4.10: This figure presents comparative results of human observers (adversary) trying to identify the leader in multi-robot teams executing different navigation algorithms (Scirpted PD, Zheng et al. [77], Naive MARL and ours). The results show that our algorithm performs better (**lower value is better**) in terms of overall score (4.10a), accuracy (4.10b) and confidence (4.10c).

4.4.2 Multi-Agent Performance

Baselines

We test the multi-agent policy's performance based on primary task reward (efficiently reaching the goal) and and on its ability to hide the leader's identity from artificial adversary (identity hiding reward). These rewards are only used for evaluation purpose and are not accessible to the agents at test time. We compare the performance against traditional leader-follower controllers - Scripted PD and Zheng et al., [77] as well as Naive MARL policy (Section 4.3.6). Scripted PD is a Proportional-Differential (PD) controller based algorithm where the leader tracks the goal location and the follwers track the leader. Zheng et al., [77] used traditional control based method to track waypoints in the trajectory. In their method, the followers don't know the leader's identity. We adopted their method to track the goal instead of waypoints. Scripted PD and Naive MARL only try to reach the goal location while Zheng et al. [77] and our approaches also try to hide the leader's identity.

Goal reaching task

On the goal reaching task, Fig. 4.6a, all the methods we tested performed well and could successfully navigate the swarm to its goal. The slight drop in performance of our approach and Zheng et al. [77] is expected as these methods might have to trade-off primary task objective with identity hiding objective. Among the methods that considered identity hiding objective, our approach got higher reward than Zheng

et al. [77].

Leader Identity hiding

In terms of identity hiding reward, Fig. 4.6b, our approach significantly outperformed all the baselines, hiding the leader successfully in most of the cases.

Fig. 4.7 shows example multi-robot trajectories using the approach of Zheng et al. [77] and our approach. In case of Zheng et al. [77], the leader clearly leads the followers and this makes it easy for an adversary to identify the leader, Fig. 4.7a. Using our approach, the leader doesn't naively lead the other agents. It rather smartly moves together with the group while informing the target motion direction through hidden movement cues. This smart strategy helps our multi-agent team deceive the adversary while achieving the primary goal, Fig. 4.7b. We show successful leader-follower navigation to different goal locations while hiding the leader's identity using our proposed approach 4.8.

0-shot generalization

We demonstrate the 0-shot generalization of our multi-agent policy to different number of agents in Fig. 4.9, where we train a policy on $n = 6$ agents and test on $2n = 12$ agents without any fine-tuning.

4.4.3 Effectiveness against Human observers

Although we have shown that our trained multi-agent policy can successfully hide its leader's identity from artificial adversary, we wish to push it further and test how well the learnt navigation strategy can deceive humans when they act as adversaries and try to identify the leader. In the following sections we describe the experimental setup and the results.

Experimental setup

We tasked humans with identifying the leader in a multi-agent team by showing videos of leader-follower multi-robot navigation with visually identical leader and followers. The human's goal was to maximize a score that incentivized answers that were both

correct and quick. For each algorithm (our and baselines), we collected data from ~ 30 humans and showed 50 videos to each human. In total, we collected $\sim 30 \times 50 = 1500$ data points for each algorithm to get a good estimate of its performance. We collected our data through a web based interface via Amazon Mechanical Turk (MTurk) [69]. Amazon Mechanical Turk is a crowd-sourcing platform where real humans are paid money to perform tasks.

Results

Fig. 4.10 shows the performance of different algorithms - Scripted PD, Zheng et al. [77], Naive MARL and ours against human adversaries. From a multi-robot team's perspective, a lower score achieved by humans is indicative of better performance of multi-robot team. Across all the metrics, our algorithm outperformed all of the other algorithms. Humans got much lower score (Fig. 4.10a), lower accuracy (Fig. 4.10b) and lacked confidence (Fig. 4.10c) in their decisions. It is interesting to note that the Zheng et al. [77] didn't generalize to human data well and performed worse than the Naive MARL policy we trained. On the other hand, our algorithm generalized well to completely unseen humans making it hard for humans to infer the leader's identity.

Chapter 5

Conclusions

In this work we explored the combination of Reinforcement Learning and adversarial training. This combination is very versatile and is applicable to many scenarios. In particular, we looked at 3 scenarios and the conclusion we derive from them are:

1. **Actor Residual Critic (ARC)** [16]: In this work, we highlighted that the reward in popular Adversarial Imitation Learning (AIL) algorithms are differentiable but this property has not been leveraged by existing model-free RL algorithms to train a policy. We also showed that naively differentiating the policy through this reward function does not perform well. To solve this issue, we proposed a class of Actor Residual Critic (ARC) algorithms that use a C function as an alternative to standard Actor Critic (AC) algorithms which use a Q function. An ARC algorithm can replace the AC RL algorithm in any existing AIL algorithm. We formally proved that Policy Iteration using C function converges to an optimum policy in tabular environments. For continuous-control tasks, using ARC can compute the exact gradient of the policy through the reward function which we believe helps improves the performance of the AIL algorithms. Future work might explore the applicability of ARC algorithm to other scenarios which have a differentiable reward function.
2. **Emergent Multi-Agent Strategies** [15]: In this work we were able to scale to multiple agents by modeling inter agent interactions with a graph containing two attention layers. We studied the evolution of complex multi agent

strategies in a mixed cooperative-competitive environment. In particular, we saw the natural emergence of deception strategy which required heterogeneous behavior amongst homogeneous agents. If instead we wanted to explicitly encode heterogeneous strategies, a simple extension of our work would be to have different sets of policy parameters (f_{θ_d}) within the same team, eg. one set for aggressive guards and one set of defensive guards. We believe that our study would inspire further work towards scaling multi agent reinforcement learning to large number of agents in more complex mixed cooperative-competitive scenarios.

3. **Hiding Leader's Identity in Leader-Follower Navigation** [18, 19] : In this paper we brought together MARL, GNNs and adversarial training for the task of hiding the leader's identity in a multi-robot team as a defense mechanism. Our proposed leader-follower navigation algorithm allows decentralized control, scales to a large and variable number of agents and generalizes to unseen human adversaries showing the effectiveness of our algorithm. Given the formulation of leader identity hiding as a MARL problem, other leader-follower tasks (such as trajectory following) could also be adopted by changing the task-related reward structure of MARL.

Bibliography

[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. Tensorflow: A system for large-scale machine learning. In *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*, pages 265–283, 2016. 2.3

[2] Pieter Abbeel, Adam Coates, and Andrew Y Ng. Autonomous helicopter aerobatics through apprenticeship learning. *The International Journal of Robotics Research*, 29(13):1608–1639, 2010. 2.1

[3] Akshat Agarwal, Sumit Kumar, and Katia Sycara. Learning transferable cooperative behavior in multi-agent teams. *arXiv preprint arXiv:1906.01202*, 2019. 3.1, 3.2.1, 3.3, 3.3.2, 3.3.3, 3.4.1

[4] Ilge Akkaya, Marcin Andrychowicz, Maciek Chociej, Mateusz Litwin, Bob McGrew, Arthur Petron, Alex Paino, Matthias Plappert, Glenn Powell, Raphael Ribas, et al. Solving rubik's cube with a robot hand. *arXiv preprint arXiv:1910.07113*, 2019. 1

[5] Maruan Al-Shedivat, Trapit Bansal, Yuri Burda, Ilya Sutskever, Igor Mordatch, and Pieter Abbeel. Continuous adaptation via meta-learning in nonstationary and competitive environments. *arXiv preprint arXiv:1710.03641*, 2017. 1, 3.2.2

[6] Dario Albani, Joris IJsselmuiden, Ramon Haken, and Vito Trianni. Monitoring and mapping with robot swarms for agricultural applications. In *2017 14th IEEE International Conference on Advanced Video and Signal Based Surveillance (AVSS)*, pages 1–6. IEEE, 2017. 4

[7] Christopher Amato, Girish Chowdhary, Alborz Geramifard, N Kemal Üre, and Mykel J Kochenderfer. Decentralized control of partially observable markov decision processes. In *52nd IEEE Conference on Decision and Control*, pages 2398–2405. IEEE, 2013. 4.3.1

[8] Brenna D Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. A survey of robot learning from demonstration. *Robotics and autonomous systems*, 57(5):469–483, 2009. 2.1

[9] Christopher G Atkeson and Stefan Schaal. Robot learning from demonstration. In *ICML*, volume 97, pages 12–20. Citeseer, 1997. 2.1

[10] Michael Bain and Claude Sammut. A framework for behavioural cloning. In *Machine Intelligence 15*, pages 103–129, 1995. 2.1, 2.2

[11] Bowen Baker, Ingmar Kanitscheider, Todor Markov, Yi Wu, Glenn Powell, Bob McGrew, and Igor Mordatch. Emergent tool use from multi-agent autocurricula. In *International Conference on Learning Representations*, 2019. 1, 3.1, 3.2.1, 3.3.3, 3.4.1

[12] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *arXiv preprint arXiv:1606.01540*, 2016. (document), 2.6.2, 2.3, 2.9, 3.4

[13] Lucian Bu, Robert Babu, Bart De Schutter, et al. A comprehensive survey of multiagent reinforcement learning. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 38(2):156–172, 2008. 3.2.1

[14] Shushman Choudhury, Kiril Solovey, Mykel J Kochenderfer, and Marco Pavone. Efficient large-scale multi-drone delivery using transit networks. *Journal of Artificial Intelligence Research*, 70:757–788, 2021. 4

[15] Ankur Deka and Katia Sycara. Natural emergence of heterogeneous strategies in artificially intelligent competitive teams. In *International Conference on Swarm Intelligence*, pages 13–25. Springer, 2021. 2, 4, 4.1, 4.3.3, 2

[16] Ankur Deka, Changliu Liu, and Katia Sycara. ARC - actor residual critic for adversarial imitation learning. . under review. 1, 1

[17] Ankur Deka, Vishnu K Narayanan, Takahiro Miyashita, and Norihiro Hagita. Adaptive attention-aware pedestrian trajectory prediction for robot planning in human environments. . 4.3.3

[18] Ankur Deka, Michael Lewis, Huao Li, Phillip Walker, and Katia Sycara. Human vs. deep neural network performance at a leader identification task. In *Proceedings of the 65th Annual Meeting of the Human Factors and Ergonomics Society*, 2021. 3, 3

[19] Ankur Deka, Wenhao Luo, Huao Li, Michael Lewis, and Katia Sycara. Hiding leader's identity in leader-follower navigation through multi-agent reinforcement learning. In *Accepted to 2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2021. 3, 3

[20] Justin Fu, Katie Luo, and Sergey Levine. Learning robust rewards with adversarial inverse reinforcement learning. *arXiv preprint arXiv:1710.11248*, 2017. 1, 2.1, **??**, 2.2, 2.3

[21] Scott Fujimoto, Herke Hoof, and David Meger. Addressing function approxi-

mation error in actor-critic methods. In *International Conference on Machine Learning*, pages 1587–1596. PMLR, 2018. 2.1, 2.3, 2.3, 2.5

[22] Seyed Kamyar Seyed Ghasemipour, Richard Zemel, and Shixiang Gu. A divergence minimization perspective on imitation learning methods. In *Conference on Robot Learning*, pages 1259–1277. PMLR, 2020. 1, 2.1, **??**, 2.2, 2.3, 2.6.2, 2.9

[23] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. *Advances in neural information processing systems*, 27, 2014. 1, 2.2, 2.3, 2.9

[24] Jason Gregory, Jonathan Fink, Ethan Stump, Jeffrey Twigg, John Rogers, David Baran, Nicholas Fung, and Stuart Young. Application of multi-robot systems to disaster-relief scenarios with limited communication. In *Field and Service Robotics*, pages 639–653. Springer, 2016. 4

[25] Tuomas Haarnoja, Aurick Zhou, Pieter Abbeel, and Sergey Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In *International Conference on Machine Learning*, pages 1861–1870. PMLR, 2018. 2.1, 2.3, 2.3, 2.5, 2.5.1, 2.6.2, 2.9, 2.9, 2.9, 3.2.1

[26] Danijar Hafner, Timothy Lillicrap, Jimmy Ba, and Mohammad Norouzi. Dream to control: Learning behaviors by latent imagination. In *International Conference on Learning Representations*, 2019. 2.2

[27] Lei Han, Peng Sun, Yali Du, Jiechao Xiong, Qing Wang, Xinghai Sun, Han Liu, and Tong Zhang. Grid-wise control for multi-agent reinforcement learning in video game ai. In *International Conference on Machine Learning*, pages 2576–2585, 2019. 3.2.1

[28] Jonathan Ho and Stefano Ermon. Generative adversarial imitation learning. In *Advances in neural information processing systems*, pages 4565–4573, 2016. 1, 2.1, 2.2, **??**, 2.3, 2.6.2, 2.9

[29] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997. 4.3.4, 4.4.1

[30] Yedid Hoshen. Vain: Attentional multi-agent predictive modeling. In *Advances in Neural Information Processing Systems*, pages 2701–2711, 2017. 3.2.1

[31] Rohit Jena, Changliu Liu, and Katia Sycara. Augmenting gail with bc for sample efficient imitation learning. *arXiv preprint arXiv:2001.07798*, 2020. 2.2

[32] Parameswaran Kamalaruban, Yu-Ting Huang, Ya-Ping Hsieh, Paul Rolland, Cheng Shi, and Volkan Cevher. Robust reinforcement learning via adversarial training with langevin dynamics. *arXiv preprint arXiv:2002.06063*, 2020. 1

[33] Tero Karras, Samuli Laine, and Timo Aila. A style-based generator architecture

for generative adversarial networks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4401–4410, 2019. 1

[34] Liyiming Ke, Matt Barnes, Wen Sun, Gilwoo Lee, Sanjiban Choudhury, and Siddhartha Srinivasa. Imitation learning as f-divergence minimization. *arXiv preprint arXiv:1905.12888*, 2019. 2.2

[35] Ashish Kumar, Zipeng Fu, Deepak Pathak, and Jitendra Malik. Rma: Rapid motor adaptation for legged robots. *arXiv preprint arXiv:2107.04034*, 2021. 1

[36] Michael Laskey, Jonathan Lee, Wesley Hsieh, Richard Liaw, Jeffrey Mahler, Roy Fox, and Ken Goldberg. Iterative noise injection for scalable imitation learning. In *1st conference on robot learning (CoRL),(ed., Sergey Levine and Vincent Vanhoucke and Ken Goldberg), Mountain View, CA, USA*, pages 13–15, 2017. 2.2

[37] Yunzhu Li, Jiaming Song, and Stefano Ermon. Infogail: Interpretable imitation learning from visual demonstrations. In *NIPS*, 2017. 2.2

[38] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015. 2.1, 2.3, 2.3, 2.5, 3.2.1

[39] Jianhua Lin. Divergence measures based on the shannon entropy. *IEEE Transactions on Information theory*, 37(1):145–151, 1991. 2.2

[40] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, Pieter Abbeel, and Igor Mordatch. Multi-agent actor-critic for mixed cooperative-competitive environments. *arXiv preprint arXiv:1706.02275*, 2017. 3.2.1, 3.2.2, 3.4, 4.3.7

[41] Md Suruz Miah, Amr Elhussein, Fazel Keshtkar, and Mohammed Abouheaf. Model-free reinforcement learning approach for leader-follower formation using nonholonomic mobile robots. In *The Thirty-Third International Flairs Conference*, 2020. 4.1

[42] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013. 2.1

[43] 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. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015. 3.2.1

[44] Andrew Y Ng, Stuart J Russell, et al. Algorithms for inverse reinforcement learning. In *Icml*, volume 1, page 2, 2000. 2.1, 2.2

[45] Tianwei Ni, Harshit Sikchi, Yufei Wang, Tejus Gupta, Lisa Lee, and Benjamin

Eysenbach. F-irl: Inverse reinforcement learning via state marginal matching. *arXiv preprint arXiv:2011.04709*, 2020. 2.2, 2.6.2, 2.9, 2.9, 2.9, 2.9

[46] Tuomas Oikarinen, Tsui-Wei Weng, and Luca Daniel. Robust deep reinforcement learning through adversarial loss. *arXiv preprint arXiv:2008.01976*, 2020. 1

[47] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. Pytorch: An imperative style, high-performance deep learning library. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc., 2019. 2.3

[48] Jorge Pena Queralta, Jussi Taipalmaa, Bilge Can Pullinen, Victor Kathan Sarker, Tuan Nguyen Gia, Hannu Tenhunen, Moncef Gabbouj, Jenni Raitoharju, and Tomi Westerlund. Collaborative multi-robot search and rescue: Planning, coordination, perception, and active vision. *IEEE Access*, 8:191617–191643, 2020. 4

[49] Alec Radford, Luke Metz, and Soumith Chintala. Unsupervised representation learning with deep convolutional generative adversarial networks. *arXiv preprint arXiv:1511.06434*, 2015. 1

[50] Ragesh K Ramachandran, James A Preiss, and Gaurav S Sukhatme. Resilience by reconfiguration: Exploiting heterogeneity in robot teams. *arXiv preprint arXiv:1903.04856*, 2019. 3.1

[51] Tabish Rashid, Mikayel Samvelyan, Christian Schroeder De Witt, Gregory Farquhar, Jakob Foerster, and Shimon Whiteson. Qmix: monotonic value function factorisation for deep multi-agent reinforcement learning. *arXiv preprint arXiv:1803.11485*, 2018. 3.2.1

[52] Stéphane Ross, Geoffrey Gordon, and Drew Bagnell. A reduction of imitation learning and structured prediction to no-regret online learning. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 627–635. JMLR Workshop and Conference Proceedings, 2011. 2.2, 2.6.2

[53] Daito Sakai, Hiroaki Fukushima, and Fumitoshi Matsuno. Leader–follower navigation in obstacle environments while preserving connectivity without data transmission. *IEEE Transactions on Control Systems Technology*, 26(4):1233–1248, 2017. 4.1

[54] Mikayel Samvelyan, Tabish Rashid, Christian Schroeder De Witt, Gregory Farquhar, Nantas Nardelli, Tim GJ Rudner, Chia-Man Hung, Philip HS Torr, Jakob Foerster, and Shimon Whiteson. The starcraft multi-agent challenge.

arXiv preprint arXiv:1902.04043, 2019. 3.2.2

[55] Stefan Schaal. Learning from demonstration. In *Advances in neural information processing systems*, pages 1040–1046, 1997. 2.1

[56] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *International conference on machine learning*, pages 1889–1897. PMLR, 2015. 2.1, 2.3

[57] John Schulman, Philipp Moritz, Sergey Levine, Michael Jordan, and Pieter Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015. 2.1, 2.3

[58] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017. 2.1, 2.3, 3.2.1, 3.3.5, 4.3.6

[59] Daigo Shishika, James Paulos, and Vijay Kumar. Cooperative team strategies for multi-player perimeter-defense games. *IEEE Robotics and Automation Letters*, 5 (2):2738–2745, 2020. 3.1

[60] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. Deterministic policy gradient algorithms. In *International conference on machine learning*, pages 387–395. PMLR, 2014. 2.3

[61] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484–489, 2016. 1

[62] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. Mastering the game of go without human knowledge. *nature*, 550(7676): 354–359, 2017. 1, 3.2.1

[63] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018. 1

[64] Aleksander S Simonsen and Else-Line M Ruud. The application of a flexible leader-follower control algorithm to different mobile autonomous robots. 4.1

[65] Sainbayar Sukhbaatar, Rob Fergus, et al. Learning multiagent communication with backpropagation. In *Advances in neural information processing systems*, pages 2244–2252, 2016. 3.2.1

[66] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018. 2.1, 2.3, 2.4.2, 2, 2.5, 2.7.1, 2.7.1, 2.7.1

[67] Ardi Tampuu, Tambet Matiisen, Dorian Kodelja, Ilya Kuzovkin, Kristjan Korjus, Juhan Aru, Jaan Aru, and Raul Vicente. Multiagent cooperation and competition with deep reinforcement learning. *PloS one*, 12(4), 2017. 3.2.1

[68] Ming Tan. Multi-agent reinforcement learning: Independent vs. cooperative agents. In *Proceedings of the tenth international conference on machine learning*, pages 330–337, 1993. 3.2.1

[69] Amazon Mechanical Turk. Amazon mechanical turk. *Retrieved August*, 17:2012, 2012. 4.4.3

[70] José Vilca, Lounis Adouane, and Youcef Mezouar. Adaptive leader-follower formation in cluttered environment using dynamic target reconfiguration. In *Distributed Autonomous Robotic Systems*, pages 237–254. Springer, 2016. 4.1

[71] Eugene Vinitsky, Yuqing Du, Kanaad Parvate, Kathy Jang, Pieter Abbeel, and Alexandre Bayen. Robust reinforcement learning using adversarial populations. *arXiv preprint arXiv:2008.01825*, 2020. 1

[72] Oriol Vinyals, Timo Ewalds, Sergey Bartunov, Petko Georgiev, Alexander Sasha Vezhnevets, Michelle Yeo, Alireza Makhzani, Heinrich Küttler, John Agapiou, Julian Schrittwieser, et al. Starcraft ii: A new challenge for reinforcement learning. *arXiv preprint arXiv:1708.04782*, 2017. 1, 3.2.2

[73] Christopher JCH Watkins and Peter Dayan. Q-learning. *Machine learning*, 8 (3-4):279–292, 1992. 3.2.1

[74] Ying Wen, Yaodong Yang, Rui Luo, Jun Wang, and Wei Pan. Probabilistic recursive reasoning for multi-agent reinforcement learning. *arXiv preprint arXiv:1901.09207*, 2019. 3.1

[75] Ronald J Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3-4):229–256, 1992. 2.1, 2.3

[76] Xin Zhang, Yanhua Li, Ziming Zhang, and Zhi-Li Zhang. f-gail: Learning f-divergence for generative adversarial imitation learning. *arXiv preprint arXiv:2010.01207*, 2020. 2.2

[77] Hehui Zheng, Jacopo Panerati, Giovanni Beltrame, and Amanda Prorok. An adversarial approach to private flocking in mobile robot teams. *IEEE Robotics and Automation Letters*, 5(2):1009–1016, 2020. (document), 4, 4.1, 4.1, 4.4.1, 4.7a, 4.7, 4.10, 4.4.2, 4.4.2, 4.4.2, 4.4.3

[78] Yanlin Zhou, Fan Lu, George Pu, Xiyao Ma, Runhan Sun, Hsi-Yuan Chen, and Xiaolin Li. Adaptive leader-follower formation control and obstacle avoidance via deep reinforcement learning. In *2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4273–4280. IEEE, 2019. 4.1

[79] Jun-Yan Zhu, Taesung Park, Phillip Isola, and Alexei A Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Proceedings of the IEEE international conference on computer vision*, pages 2223–2232, 2017. 1

[80] Brian D Ziebart, Andrew L Maas, J Andrew Bagnell, and Anind K Dey. Maximum entropy inverse reinforcement learning. In *Aaai*, volume 8, pages 1433–1438. Chicago, IL, USA, 2008. 2.2