IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received December 22, 2020, accepted January 6, 2021, date of publication January 8, 2021, date of current version January 20, 2021.

Digital Object Identifier 10.1109/ACCESS.2021.3050338

INVITED PAPER

Collision Avoidance in Pedestrian-Rich Environments With Deep Reinforcement Learning

MICHAEL EVERETT[1], YU FAN CHEN[2], AND JONATHAN P. HOW[1], (Fellow, IEEE)
[1]Massachusetts Institute of Technology, Cambridge, MA 02139, USA
[2]Facebook Reality Labs, Redmond, WA 98052, USA

Corresponding author: Michael Everett (mfe@mit.edu)

This work was supported in part by the Ford Motor Company, and in part by the Amazon Web Services.

ABSTRACT Collision avoidance algorithms are essential for safe and efficient robot operation among pedestrians. This work proposes using deep reinforcement (RL) learning as a framework to model the complex interactions and cooperation with nearby, decision-making agents, such as pedestrians and other robots. Existing RL-based works assume homogeneity of agent properties, use specific motion models over short timescales, or lack a principled method to handle a large, possibly varying number of agents. Therefore, this work develops an algorithm that learns collision avoidance among a variety of heterogeneous, non-communicating, dynamic agents without assuming they follow any particular behavior rules. It extends our previous work by introducing a strategy using Long Short-Term Memory (LSTM) that enables the algorithm to use observations of an arbitrary number of other agents, instead of a small, fixed number of neighbors. The proposed algorithm is shown to outperform a classical collision avoidance algorithm, another deep RL-based algorithm, and scales with the number of agents better (fewer collisions, shorter time to goal) than our previously published learning-based approach. Analysis of the LSTM provides insights into how observations of nearby agents affect the hidden state and quantifies the performance impact of various agent ordering heuristics. The learned policy generalizes to several applications beyond the training scenarios: formation control (arrangement into letters), demonstrations on a fleet of four multirotors and on a fully autonomous robotic vehicle capable of traveling at human walking speed among pedestrians.

INDEX TERMS Collision avoidance, deep reinforcement learning, motion planning, multiagent systems, decentralized execution.

I. INTRODUCTION

A fundamental challenge in autonomous vehicle operation is to safely negotiate interactions with other dynamic agents in the environment. For example, it is important for self-driving cars to take other vehicles' motion into account, and for delivery robots to avoid colliding with pedestrians. While there has been impressive progress in the past decade [1], fully autonomous navigation remains challenging, particularly in uncertain, dynamic environments cohabited by other mobile agents. The challenges arise because the other agents' intents and policies (i.e., goals and desired paths) are typically not known to the planning system, and, furthermore, explicit communication of such hidden quantities is often impractical due to physical limitations. These issues motivate the use of decentralized collision avoidance algorithms.

Existing work on decentralized collision avoidance can be classified into cooperative and non-cooperative methods. Non-cooperative methods first predict the other agents' motion and then plan a collision-free path for the vehicle with respect to the other agents' predicted motion. However, this can lead to the *freezing robot problem* [2], where the vehicle fails to find any feasible path because the other agents' predicted paths would occupy a large portion of the traversable space. Cooperative methods address this issue by modeling interaction in the planner, such that the vehicle's action can influence the other agent's motion, thereby having all agents share the responsibility for avoiding collision.

The associate editor coordinating the review of this manuscript and approving it for publication was Aysegul Ucar.

Cooperative methods include reaction-based methods [3]–[6] and trajectory-based methods [7]–[9].

This work seeks to combine the best of both types of cooperative techniques – the computational efficiency of reaction-based methods and the smooth motion of trajectory-based methods. To this end, the work presents the collision avoidance with deep reinforcement learning (CADRL) algorithm, which tackles the aforementioned trade-off between computation time and smooth motion by using reinforcement learning (RL) to offload the expensive online computation to an offline learning procedure. Specifically, a computationally efficient (i.e., real-time implementable) interaction rule is developed by learning a policy that implicitly encodes cooperative behaviors.

Learning the collision avoidance policy for CADRL presents several challenges. A first key challenge is that the number of other agents in the environment can vary between timesteps or experiments, however the typical feed-forward neural networks used in this domain require a fixed-dimension input. Our prior work defines a maximum number of agents that the network can observe, and other approaches use raw sensor data as the input [10], [11]. This work instead uses an idea from Natural Language Processing [12], [13] to encode the varying size state of the world (e.g., positions of other agents) into a fixed-length vector, using long short-term memory (LSTM) [14] cells at the network input. This enables the algorithm to make decisions based on an arbitrary number of other agents in the robot's vicinity.

A second fundamental challenge is in finding a policy that makes realistic assumptions about other agents' belief states, policies, and intents. This work learns a collision avoidance policy without assuming that the other agents follow any particular behavior model and without explicit assumptions on homogeneity [10] (e.g., agents of the same size and nominal speed) or specific motion models (e.g., constant velocity) over short timescales [15], [16].

The main contributions of this work are:

- a new collision avoidance algorithm that greatly outperforms prior works as the number of agents in the environments is increased: a key factor in that improvement is to relax the assumptions on the other agents' behavior models during training and inference,
- a novel use of LSTM in that it encodes spatial representations, rather than temporal, to address the challenge that the number of neighboring agents could be large and could vary in time,
- simulation results that show significant improvement in solution quality compared with other recently published state-of-the-art methods (such as [5], [10], [16]), and
- hardware experiments with aerial and ground robots to demonstrate that the proposed algorithm can be deployed in real time on robots with real sensors.

Open-source software based on this manuscript includes a pre-trained collision avoidance policy (as a ROS package) `cadrl_ros`,[1] the GA3C-CADRL learning algorithm,[2] and a simulation/training environment with several implemented policies, `gym_collision_avoidance`.[3] Videos of the experimental results are posted at https://youtu.be/Bjx4ZEov0yE.

This work is based on [15]–[17] and extends them as follows: (i) expanded discussion and example of the limitations of the prior work, (ii) further explanation of the proposed algorithm, including pseudo-code, (iii) analysis on the effect of sequence ordering in LSTM, which addresses a primary gap in the prior work, (iv) quantifying input gate activation to provide deeper intuition on why the proposed use of LSTM works, (v) additional comparisons to model- and learning-based collision avoidance algorithms, (vi) ablation study of the proposed algorithm, and (vii) experiments with formation control and on real multirotors to demonstrate generalizability of the learned policy.

II. BACKGROUND

A. PROBLEM FORMULATION

The non-communicating, multiagent collision avoidance problem can be formulated as a sequential decision making problem [15], [16]. In an n-agent scenario ($\mathbb{N}_{\leq n} = \{1, 2, \ldots, n\}$), denote the joint world state, $\mathbf{s}_t^{jn}$, agent i's state, $\mathbf{s}_{i,t}$, and agent i's action, $\mathbf{u}_{i,t}$, $\forall i \in \mathbb{N}_{\leq n}$. Each agent's state vector is composed of an observable and unobservable (hidden) portion, $\mathbf{s}_{i,t} = [\mathbf{s}_{i,t}^o, \mathbf{s}_{i,t}^h]$. In the global frame, observable states are the agent's position, velocity, and radius, $\mathbf{s}^o = [p_x, p_y, v_x, v_y, r] \in \mathbb{R}^5$, and unobservable states are the goal position, preferred speed, and orientation,[4] $\mathbf{s}^h = [p_{gx}, p_{gy}, v_{pref}, \psi] \in \mathbb{R}^4$. The action is a speed and heading angle, $\mathbf{u}_t = [v_t, \psi_t] \in \mathbb{R}^2$. The observable states of all $n-1$ other agents is denoted, $\tilde{\mathbf{S}}_{i,t}^o = \{\tilde{\mathbf{s}}_{j,t}^o : j \in \mathbb{N}_{\leq n} \setminus i\}$. A policy, $\pi : \left(\mathbf{s}_{0:t}, \tilde{\mathbf{S}}_{0:t}^o\right) \mapsto \mathbf{u}_t$, is developed with the objective of minimizing expected time to goal $\mathbb{E}[t_g]$ while avoiding collision with other agents,

$$\underset{\pi_i}{\operatorname{argmin}} \ \mathbb{E}\left[t_g | \mathbf{s}_i, \tilde{\mathbf{S}}_i^o, \pi_i\right] \tag{1}$$

$$s.t. \ ||\mathbf{p}_{i,t} - \tilde{\mathbf{p}}_{j,t}||_2 \geq r_i + r_j \qquad \forall j \neq i, \forall t \tag{2}$$

$$\mathbf{p}_{i,t_g} = \mathbf{p}_{i,g} \quad \forall i \tag{3}$$

$$\mathbf{p}_{i,t} = \mathbf{p}_{i,t-1} + \Delta t \cdot \pi_i(\mathbf{s}_{i,t-1}, \tilde{\mathbf{S}}_{i,t-1}^o) \quad \forall i, \tag{4}$$

where (2) is the collision avoidance constraint, (3) is the goal constraint, (4) is the agents' kinematics, and the expectation in (1) is with respect to the other agent's unobservable states (intents) and policies.

Although it is difficult to solve for the optimal solution of (1)-(4), this problem formulation can be useful for understanding the limitations of the existing methods.

[1] https://github.com/mit-acl/cadrl_ros
[2] https://github.com/mit-acl/rl_collision_avoidance
[3] https://github.com/mit-acl/gym-collision-avoidance
[4] Other agents' positions and velocities are straightforward to estimate with a 2D Lidar, unlike human body heading angle

In particular, it provides insights into the approximations/assumptions made by existing works.

B. RELATED WORK

Most approaches to collision avoidance with dynamic obstacles employ model-predictive control (MPC) [18] in which a planner selects a minimum cost action sequence, $\mathbf{u}_{i,t:t+T}$, using a prediction of the future world state, $P(\mathbf{s}^{jn}_{t+1:t+T+1}|\mathbf{s}^{jn}_{0:t}, \mathbf{u}_{i,t:t+T})$, conditioned on the world state history, $\mathbf{s}^{jn}_{0:t}$. While the first actions in the sequence are being implemented, the subsequent action sequence is updated by re-planning with the updated world state information (e.g., from new sensor measurements). The prediction of future world states is either prescribed using domain knowledge (model-based approaches) or learned from examples/experiences (learning-based approaches).

1) MODEL-BASED APPROACHES

Early approaches model the world as a static entity, $[v_x, v_y] = \mathbf{0}$, but replan quickly to try to capture the motion through updated (p_x, p_y) measurements [19]. This leads to time-inefficient paths among dynamic obstacles, since the planner's world model does not anticipate future changes in the environment due to the obstacles' motion.

To improve the predictive model, *reaction-based* methods use one-step interaction rules based on geometry or physics to ensure collision avoidance. These methods [4]–[6] often specify a Markovian policy, $\pi(\mathbf{s}^{jn}_{0:t}) = \pi(\mathbf{s}^{jn}_t)$, that optimizes a one-step cost while satisfying collision avoidance constraints. For instance, in velocity obstacle approaches [5], [6], an agent chooses a collision-free velocity that is closest to its preferred velocity (i.e., directed toward its goal). Given this one-step nature, reaction-based methods do account for current obstacle motion, but do not anticipate the other agents' hidden intents – they instead rely on a fast update rate to react quickly to the other agents' changes in motion. Although computationally efficient given these simplifications, reaction-based methods are myopic in time, which can sometimes lead to generating unnatural trajectories [8], [15].

Trajectory-based methods compute plans on a longer timescale to produce smoother paths but are often computationally expensive or require knowledge of unobservable states. A subclass of non-cooperative approaches [20], [21] propagates the other agents' dynamics forward in time and then plans a collision-free path with respect to the other agents' predicted paths. However, in crowded environments, the set of predicted paths could occupy a large portion of the space, which leads to the *freezing robot problem* [2]. A key to resolving this issue is to account for interactions, such that each agent's motion can affect one another. Thereby, a subclass of cooperative approaches [7]–[9] has been proposed, which solve (1)-(4) in two steps. First, the other agents' hidden states (i.e., goals) are inferred from their observed trajectories, $\hat{\tilde{\mathbf{S}}}^h_t = f(\tilde{\mathbf{S}}^o_{0:t})$, where $f(\cdot)$ is a inference function. Second, a centralized path planning algorithm, $\pi(\mathbf{s}_{0:t}, \tilde{\mathbf{S}}^o_{0:t}) = \pi_{central}(\mathbf{s}_t, \tilde{\mathbf{S}}^o_t, \hat{\tilde{\mathbf{S}}}^h_t)$, is employed to find jointly feasible paths. By planning/anticipating complete paths, trajectory-based methods are no longer myopic. However, both the inference and the planning steps are computationally expensive, and need to be carried out online at each new observation (sensor update $\tilde{\mathbf{S}}^o_t$).

2) LEARNING-BASED APPROACHES

Our recent works [15], [16] proposed a third category that uses a reinforcement learning framework to solve (1)-(4). As in the reactive-based methods, we make a Markovian assumption: $\pi(\mathbf{s}^{jn}_{0:t}) = \pi(\mathbf{s}^{jn}_t)$. The expensive operation of modeling the complex interactions is learned in an offline training step, whereas the learned policy can be queried quickly online, combining the benefits of both reactive- and trajectory-based methods. Our prior methods pre-compute a value function, $V(\mathbf{s}^{jn})$, that estimates the expected time to the goal from a given configuration, which can be used to select actions using a one-step lookahead procedure described in those works. To avoid the lookahead procedure, this work directly optimizes a policy $\pi(\mathbf{s}^{jn})$ to select actions to minimize the expected time to the goal. The differences from other learning-based approaches will become more clear after a brief overview of reinforcement learning.

C. REINFORCEMENT LEARNING

RL [22] is a class of machine learning methods for solving sequential decision making problems with unknown state-transition dynamics. Typically, a sequential decision making problem can be formulated as a Markov decision process (MDP), which is defined by a tuple $M = \langle S, A, P, R, \gamma \rangle$, where S is the state space, A is the action space, P is the state-transition model, R is the reward function, and γ is a discount factor. By detailing each of these elements and relating to (1)-(4), the following provides a RL formulation of the n-agent collision avoidance problem.

1) STATE SPACE

The joint world state, $\mathbf{s}^{jn}$, was defined in Section II-A.

2) ACTION SPACE

The choice of action space depends on the vehicle model. A natural choice of action space for differential drive robots is a linear and angular speed (which can be converted into wheel speeds), that is, $\mathbf{u} = [s, \omega]$. The action space is either discretized directly, or represented continuously by a function of discrete parameters.

3) REWARD FUNCTION

A sparse reward function is specified to award the agent for reaching its goal (3), and penalize the agent for getting too

close or colliding with other agents (2),

$$R(\mathbf{s}^{jn}, \mathbf{u}) = \begin{cases} 1, & \text{if } \mathbf{p} = \mathbf{p}_g \\ -0.1 + d_{min}/2, & \text{if } 0 < d_{min} < 0.2 \\ -0.25, & \text{if } d_{min} < 0 \\ 0, & \text{otherwise,} \end{cases} \quad (5)$$

where d_{min} is the distance to the closest other agent. Optimizing the hyperparameters (e.g., -0.25) in R_{col} is left for future work. Note that we use discount $\gamma < 1$ to encourage efficiency instead of a step penalty.

4) STATE TRANSITION MODEL

A probabilistic state transition model, $P(\mathbf{s}^{jn}_{t+1}|\mathbf{s}^{jn}_t, \mathbf{u}_t)$, is determined by the agents' kinematics as defined in (4). Since the other agents' actions also depend on their policies and hidden intents (e.g., goals), the system's state transition model is unknown.

5) VALUE FUNCTION

One method to find the optimal policy is to first find the optimal value function,

$$V^*(\mathbf{s}^{jn}_0) = \mathbb{E}\left[\sum_{t=0}^{T} \gamma^t R(\mathbf{s}^{jn}_t, \pi^*(\mathbf{s}^{jn}_t))\right], \quad (6)$$

where $\gamma \in [0, 1)$ is a discount factor. Many methods exist to estimate the value function in an offline training process [22].

6) DEEP REINFORCEMENT LEARNING

To estimate the high-dimensional, continuous value function (and/or associated policy), it is common to approximate with a deep neural network (DNN) parameterized by weights and biases, θ, as in [23]. This work's notation drops the parameters except when possible, e.g., $V(\mathbf{s}; \theta) = V(\mathbf{s})$.

7) DECISION-MAKING POLICY

A value function of the current state can be implemented as a policy,

$$\pi^*(\mathbf{s}^{jn}_{t+1}) = \underset{\mathbf{u}}{\operatorname{argmax}} R(\mathbf{s}_t, \mathbf{u}) + \gamma^{\Delta t \cdot v_{pref}} \int_{\mathbf{s}^{jn}_{t+1}} P(\mathbf{s}^{jn}_t, \mathbf{s}^{jn}_{t+1}|\mathbf{u}) V^*(\mathbf{s}^{jn}_{t+1}) d\mathbf{s}^{jn}_{t+1}. \quad (7)$$

Our previous works avoid the complexity in explicitly modeling $P(\mathbf{s}^{jn}_{t+1}|\mathbf{s}^{jn}_t, \mathbf{u})$ by assuming that other agents continue their current velocities, $\hat{\mathbf{V}}_t$, for a duration Δt, meaning the policy can be extracted from the value function,

$$\hat{\mathbf{s}}^{jn}_{t+1,\mathbf{u}} \leftarrow [f(\mathbf{s}_t, \Delta t \cdot \mathbf{u}), f(\tilde{\mathbf{S}}^o_t, \Delta t \cdot \hat{\mathbf{V}}_t)] \quad (8)$$

$$\pi_{CADRL}(\mathbf{s}^{jn}_t) = \underset{\mathbf{u}}{\operatorname{argmax}} R_{col}(\mathbf{s}_t, \mathbf{u}) + \gamma^{\Delta t \cdot v_{pref}} V(\hat{\mathbf{s}}^{jn}_{t+1,\mathbf{u}}), \quad (9)$$

under the simple kinematic model, f.

However, the introduction of parameter Δt leads to a difficult trade-off. Due to the the approximation of the value function in a DNN, a sufficiently large Δt is required such that each propagated $\hat{\mathbf{s}}^{jn}_{t+1,\mathbf{u}}$ is far enough apart, which ensures $V(\hat{\mathbf{s}}^{jn}_{t+1,\mathbf{u}})$ is not dominated by numerical noise in the network. The implication of large Δt is that agents are assumed to follow a constant velocity for a significant amount of time, which neglects the effects of cooperation/reactions to an agent's decisions. As the number of agents in the environment increases, this constant velocity assumption is less likely to be valid. Agents do not actually reach their propagated states because of the multiagent interactions.

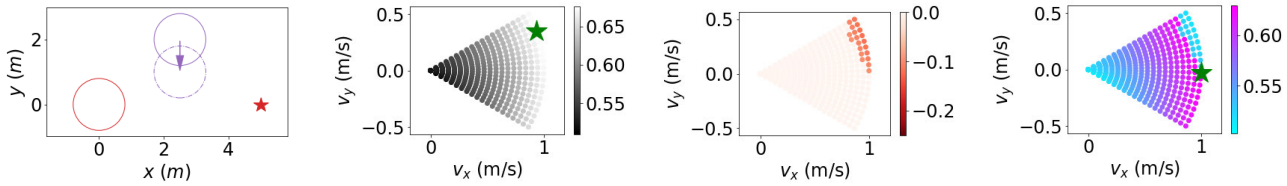
The impact of separately querying the value function and performing collision checking is illustrated in Fig. 1. In (a), a red agent aims to reach its goal (star), and a purple agent is traveling at 1 m/s in the $-y$-direction. Because CADRL's value function only encodes time-to-goal information, (b) depicts that the DNN appropriately recommends that the red agent should cut above the purple agent. However, there is a second term in (9) to convert the value function into a policy. This second term, the collision cost, $R_{col}(\mathbf{s}_t, \mathbf{u})$, shown in (c), penalizes actions that move toward the other agent's predicted position (dashed circle). This model-based collision checking procedure requires an assumption about other agents' behaviors, which is difficult to define ahead of time; the prior work assumed a constant-velocity model. When the value and collision costs are combined to produce $\pi_{CADRL}(\mathbf{s}^{jn}_t)$, the resulting objective-maximizing action is for the red agent to go straight, which will avoid a collision but be inefficient for both agents. The challenge in defining a model for other agents' behaviors was a primary motivation for learning a value function; even with an accurate value function, this example demonstrates an additional cause of inefficient paths: an inaccurate model used in the collision checking procedure.

In addition to not capturing decision making behavior of other agents, our experiments suggest that Δt is a crucial parameter to ensure convergence while training the DNNs in the previous algorithms. If Δt is set too small or large, the training does not converge. A value of $\Delta t = 1$ sec was experimentally determined to enable convergence, though this number does not have much theoretical rationale.

In summary, the challenges of converting a value function to a policy, choosing the Δt hyperparameter, and our observation that the learning stability suffered with more than 4 agents in the environment each motivate the use of a different RL framework. To address the concerns raised about ΔT propagation, this work proposes a new algorithm that does not project agents forward during policy evaluation, thus eliminating the need for tuning the Δt hyperparameter.

8) POLICY LEARNING

Therefore, this work considers RL frameworks which generate a policy that an agent can execute directly, without any arbitrary assumptions about state transition dynamics. A recent actor-critic algorithm called A3C [24] uses a single DNN to approximate both the value (critic) and policy (actor)



(a) 2-Agent Scenario (b) Time-to-Goal Estimate, $V(\hat{\mathbf{s}}^{jn}_{t+1,\mathbf{u}})$ from DNN (c) Collision Cost, $R_{col}(\mathbf{s}_t, \mathbf{u})$, using const. velocity model (d) Objective in $\pi_{CADRL}(\mathbf{s}^{jn}_t)$

FIGURE 1. Issue with checking collisions and state-value separately, as in (9). In (a), the red agent's goal is at the star, and the purple agent's current velocity is in the $-y$-direction. In (b), the CADRL algorithm propagates the other agent forward at its current velocity (dashed purple circle), then queries the DNN for candidate future states. The best action (green star) is one which cuts above the purple agent, which was learned correctly by the CADRL V-Learning procedure. However, the constant velocity model of other agents is also used for collision checking, causing penalties of $R_{col}(\mathbf{s}_t, \mathbf{u})$, shown in (c). CADRL's policy combines these terms (d), instead choosing to go straight (green star), which is a poor choice that ignores that a cooperative purple agent likely would adjust its own velocity as well. This fundamental issue of checking collisions and state-values separately is addressed in this work by learning a policy directly. .

functions, and is trained with two loss terms

$$f_v = (R_t - V(\mathbf{s}^{jn}_t))^2, \tag{10}$$

$$f_\pi = \log \pi(u_t|s^{jn}_t)(R_t - V(s^{jn}_t)) + \beta \cdot H(\pi(s^{jn}_t)), \tag{11}$$

where (10) trains the network's value output to match the future discounted reward estimate, $R_t = \sum_{i=0}^{k-1} \gamma^i r_{t+i} + \gamma^k V(\mathbf{s}^{jn}_{t+k})$, over the next k steps, just as in CADRL. For the policy output in (11), the first term penalizes actions which have high probability of occurring ($\log \pi$) that lead to a lower return than predicted by the value function ($R - V$), and the second term encourages exploration by penalizing π's entropy with tunable constant β.

In A3C, many threads of an agent interacting with an environment are simulated in parallel, and a policy is trained based on an intelligent fusion of all the agents' experiences. The algorithm was shown to learn a policy that achieves super-human performance on many video games. We specifically use GA3C [25], a hybrid GPU/CPU implementation that efficiently queues training experiences and action predictions. Our work builds on open-source GA3C implementations [25], [26].

Other choices for RL policy training algorithms (e.g., PPO [27], TD3 [28]) are architecturally similar to A3C. Thus, the challenges mentioned above (varying number of agents, assumptions about other agents' behaviors) would map to future work that considers employing other RL algorithms or techniques [29] in this domain.

D. RELATED WORKS USING LEARNING

There are several concurrent and subsequent works which use learning to solve the collision avoidance problem, categorized as non-RL, RL, and agent-level RL approaches.

Non-RL-based approaches to the collision avoidance problem include imitation learning, inverse RL, and supervised learning of prediction models. Imitation learning approaches [11] learn a policy that mimics what a human pedestrian or human teleoperator [30] would do in the same state but require data from an expert. Inverse RL methods learn to estimate pedestrians' cost functions, then use the cost function to inform robot plans [7], [31], but require real pedestrian trajectory data. Other approaches learn to predict pedestrian paths, which improves the world model used by the planner [32], but decoupling the prediction and planning steps could lead to the freezing robot problem (Section II-B1). A key advantage of RL over these methods is the ability to explore the state space through self-play, in which experiences generated in a low-fidelity simulation environment can reduce the need for expensive, real-world data collection efforts.

Within RL-based approaches, a key difference arises in the state representation: sensor-level and agent-level. Sensor-level approaches learn to select actions directly from raw sensor readings (either 2D laserscans [10] or images [11]) with end-to-end training. This leads to a large state space ($\mathbb{R}^{w \times h \times c}$ for a camera with resolution $w \times h$ and c channels, e.g., $480 \times 360 \times 3 = 5184000$), which makes training challenging. CNNs are often used to extract low-dimensional features from this giant state space, but training such a feature extractor in simulation requires an accurate sensor simulation model. The sensor-level approach has the advantage that both static and dynamic obstacles (including walls) can be fed into the network with a single framework. In contrast, this work uses interpretable clustering, tracking, and multi-sensor fusion algorithms to extract an agent-level state representation from raw sensor readings. Advantages include a much smaller state space ($\mathbb{R}^{9+5(n-1)}$) enabling faster learning convergence; a sensor-agnostic collision avoidance policy, enabling sensor upgrades without retraining; and increased introspection into decision making, so that decisions can be traced back to the sensing, clustering, tracking, or planning modules.

Within agent-level RL, a key challenge is that of representing a variable number of nearby agents in the environment at any timestep. Typical feedforward networks used to represent the complex decision making policy for collision avoidance require a pre-determined input size. The sensor-level methods do maintain a fixed size input (sensor resolution), but have the

limitations mentioned above. Instead, our first work trained a 2-agent value network, and proposed a mini-max rule to scale up to n agents [15]. To account for multiagent interactions (instead of only pairwise), our next work defines a maximum number of agents that the network can handle, and pads the observation space if there are actually fewer agents in the environment [16]. However, this maximum number of agents is limited by the increased number of network parameters (and therefore training time) as more agents' states are added. This work uses a recurrent network to convert a sequence of agent states at a particular timestep into a fixed-size representation of the world state; that representation is fed into the input of a standard feedforward network. Beyond the scope of collision avoidance, recent work [33] introduced attention mechanisms, another tool popularized in NLP, as another method for embedding the variable number of other agents' states.

There are also differences in the reward functions used in RL-based collision avoidance approaches. Generally, the non-zero feedback provided at each timestep by a dense reward function (e.g., [10]) makes learning easier, but reward shaping quickly becomes a difficult problem in itself. For example, balancing multiple objectives (proximity to goal, proximity to others) can introduce unexpected and undesired local minima in the reward function. On the other hand, sparse rewards are easy to specify but require a careful initialization/exploration procedure to ensure agents will receive *some* environment feedback to inform learning updates. This work mainly uses sparse reward (arrival at goal, collision) with smooth reward function decay in near-collision states to encourage a minimum separation distance between agents. Additional terms in the reward function are shown to reliably induce higher-level preferences (social norms) in our previous work [16].

While learning-based methods have many potential advantages over model-based approaches, learning-based approaches typically lack the guarantees (e.g., avoiding deadlock, zero collisions) desired for safety-critical applications. A key challenge in establishing guarantees in multiagent collision avoidance is what to assume about the world (e.g., policies and dynamics of other agents). Unrealistic or overly conservative assumptions about the world invalidate the guarantees or unnecessarily degrade the algorithm's performance: striking this balance may be possible in some domains but is particularly challenging in pedestrian-rich environments. A survey of the active research area of Safe RL is found in [34].

III. APPROACH

A. GA3C-CADRL

Recall the RL training process seeks to find the optimal policy, $\pi : \left(\mathbf{s}_t, \tilde{\mathbf{S}}_t^o\right) \mapsto \mathbf{u}_t$, which maps from an agent's observation of the environment to a probability distribution across actions and executes the action with highest probability. We use a local coordinate frame (rotation-invariant) as in [15], [16] and separate the state of the world in two pieces: information about the agent itself, and everything else in the world. Information about the agent can be represented in a small, fixed number of variables. The world, on the other hand, can be full of any number of other objects or even completely empty. Specifically, there is one $\mathbf{s}$ vector about the agent itself and one $\tilde{\mathbf{s}}^o$ vector per other agent in the vicinity:

$$\mathbf{s} = [d_g,\ v_{pref},\ \psi,\ r] \tag{12}$$

$$\tilde{\mathbf{s}}^o = [\tilde{p}_x,\ \tilde{p}_y,\ \tilde{v}_x,\ \tilde{v}_y,\ \tilde{r},\ \tilde{d}_a,\ \tilde{r}+r]\,, \tag{13}$$

where $d_g = ||\mathbf{p}_g - \mathbf{p}||_2$ is the agent's distance to goal, and $\tilde{d}_a = ||\mathbf{p} - \tilde{\mathbf{p}}||_2$ is the distance to the other agent.

The agent's action space is composed of a speed and change in heading angle. It is discretized into 11 actions: with a speed of v_{pref} there are 6 headings evenly spaced between $\pm\pi/6$, and for speeds of $\frac{1}{2}v_{pref}$ and 0 the heading choices are $[-\pi/6,\ 0,\ \pi/6]$. These actions are chosen to mimic real turning constraints of robotic vehicles.

This multiagent RL problem formulation is solved with GA3C in a process we call GA3C-CADRL (GPU/CPU Asynchronous Advantage Actor-Critic for Collision Avoidance with Deep RL). Since experience generation is one of the time-intensive parts of training, this work extends GA3C to learn from multiple agents' experiences each episode. Training batches are filled with a mix of agents' experiences ($\{\mathbf{s}_t^{jn},\ \mathbf{u}_t,\ r_t\}$ tuples) to encourage policy gradients that improve the joint expected reward of all agents. Our multiagent implementation of GA3C accounts for agents reaching their goals at different times and ignores experiences of agents running other policies (e.g., non-cooperative agents).

B. HANDLING A VARIABLE NUMBER OF AGENTS

Recall that one key limitation of many learning-based collision avoidance methods is that the feedforward NNs typically used require a fixed-size input. Convolutional and max-pooling layers are useful for feature extraction and can modify the input size but still convert a fixed-size input into a fixed-size output. Recurrent NNs, where the output is produced from a combination of a stored cell state and an input, accept an arbitrary-length sequence to produce a fixed-size output. Long short-term memory (LSTM) [14] is recurrent architecture with advantageous properties for training.[5]

Although LSTMs are often applied to time sequence data (e.g., pedestrian motion prediction [35]), this article leverages their ability to encode a sequence of information that is not time-dependent (see [36] for a thorough explanation of LSTM calculations). LSTM is parameterized by its weights, $\{W_i, W_f, W_o\}$, and biases, $\{b_i, b_f, b_o\}$, where $\{i, f, o\}$ correspond to the input, forget, and output gates. The variable number of $\tilde{\mathbf{s}}_i^o$ vectors is a sequence of inputs that encompass everything the agent knows about the rest of the world. As depicted in Fig. 2, each LSTM *cell* has three inputs: the

[5] In practice, TensorFlow's LSTM implementation requires a known maximum sequence length, but this can be set to something bigger than the number of agents agents ever expected (e.g., 20)

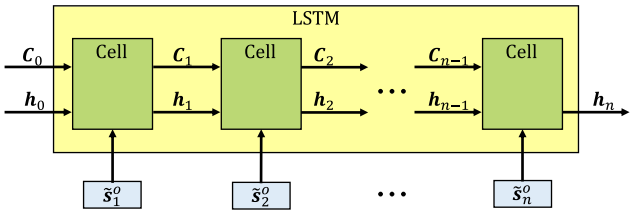


FIGURE 2. LSTM unrolled to show each input. At each decision step, the agent feeds one observable state vector, $\tilde{\mathbf{s}}_i^o$, for each nearby agent, into a LSTM cell sequentially. LSTM cells store the pertinent information in the hidden states, $\mathbf{h}_i$. The final hidden state, $\mathbf{h}_n$, encodes the entire state of the other agents in a fixed-length vector, and is then fed to the feedforward portion of the network. The order of agents is sorted by decreasing distance to the ego agent, so that the closest agent has the most recent effect on $\mathbf{h}_n$.

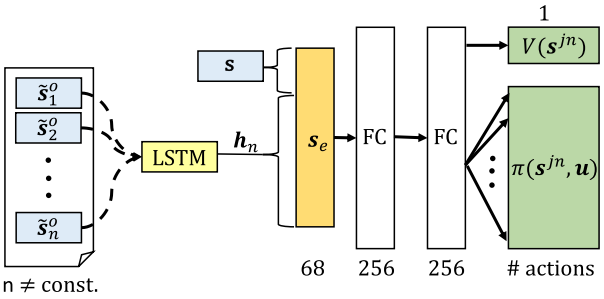


FIGURE 3. Network Architecture. Observable states of nearby agents, $\tilde{\mathbf{s}}_j^o$, are fed sequentially into the LSTM, as unrolled in Fig. 2. The final hidden state is concatenated with the agent's own state, $\mathbf{s}$, to form the vector, $\mathbf{s_e}$. For any number of agents, $\mathbf{s_e}$ contains the agent's knowledge of its own state and the state of the environment. The encoded state is fed into two fully-connected layers (FC). The outputs are a scalar value function (top, right) and policy represented as a discrete probability distribution over actions (bottom, right).

state of agent j at time t, the previous hidden state, and the previous cell state, which are denoted $\tilde{\mathbf{s}}_{j,t}^o$, $\mathbf{h}_j$, C_j, respectively. Thus, at each decision step, the agent feeds each $\tilde{\mathbf{s}}_i^o$ (observation of i^{th} other agent's state) into a LSTM cell sequentially. That is, the LSTM initially has empty states ($\mathbf{h}_0$, $\mathbf{C}_0$ set to zeros) and uses $\{\tilde{\mathbf{s}}_1^o, \mathbf{h}_0, C_0\}$ to generate $\{\mathbf{h}_1, C_1\}$, then feeds $\{\tilde{\mathbf{s}}_2^o, \mathbf{h}_1, C_1\}$ to produce $\{\mathbf{h}_2, C_2\}$, and so on. As agents' states are fed in, the LSTM "remembers" the pertinent information in its hidden/cell states, and "forgets" the less important parts of the input (where the notion of memory is parameterized by the trainable LSTM weights/biases). After inputting the final agent's state, we can interpret the LSTM's final hidden state, $\mathbf{h}_n$ as a fixed-length, encoded state of the world, for that decision step. The LSTM contains n cells, so the entire module receives inputs $\{\tilde{\mathbf{S}}_t^o, \mathbf{h}_{t-1}, \mathbf{C}_{t-1}\}$ and produces outputs $\{\mathbf{h}_n, \mathbf{C}_n\}$, and $\mathbf{h}_n$ is passed to the next network layer for decision making.

Given a sufficiently large hidden state vector, there is enough space to encode a large number of agents' states without the LSTM having to forget anything relevant. In the case of a large number of agent states, to mitigate the impact of the agent forgetting the early states, the states are fed in reverse order of distance to the agent, meaning the closest agents (fed last) should have the biggest effect on the final hidden state, $\mathbf{h}_n$. Because the list of agents needs to be ordered in some manner, reverse distance is one possible ordering heuristic – we empirically compare to other possibilities in Section IV-D.

Another interpretation of the LSTM objective is that it must learn to combine an observation of a new agent with a representation of other agents (as opposed to the architectural objective of producing a fixed-length encoding of a varying size input). This interpretation provides intuition on how an LSTM trained in 4-agent scenarios can generalize reasonably well to cases with 10 agents.

The addition of LSTM to a standard actor-critic network is visualized in Fig. 3, where the box labeled $\mathbf{s}$ is the agent's own state, and the group of boxes is the n other agents' observable states, $\tilde{\mathbf{s}}_i^o$. After passing the n other agents' observable states into the LSTM, the agent's own state is concatenated with $\mathbf{h}_n$ to produce the encoded representation of the joint world state, $\mathbf{s}_e$. Then, $\mathbf{s}_e$ is passed to a typical feedforward DNN with 2 fully-connected layers (256 hidden units each with ReLU activation).

The network produces two output types: a scalar state value (critic) and a policy composed of a probability for each action in the discrete action space (actor). During training, the policy *and* value are used for Equations (10) and (11); during execution, only the policy is used. During the training process, the LSTM's weights are updated to learn how to represent the variable number of other agents in a fixed-length $\mathbf{h}$ vector. The whole network is trained end-to-end with back-propagation.

C. TRAINING THE POLICY

The original CADRL and SA-CADRL (Socially Aware CADRL) algorithms used several clever tricks to enable convergence when training the networks. Specifically, forward propagation of other agent states for Δt seconds was a critical component that required tuning, but does not represent agents' true behaviors. Other details include separating experiences into successful/unsuccessful sets to focus the training on cases where the agent could improve. The new GA3C-CADRL formulation is more general, and does not require such assumptions or modifications.

The training algorithm is described in Algorithm 1. In this work, to train the model, the network weights are first initialized in a supervised learning phase, which converges in less than five minutes. The initial training is done on a large, publicly released set of state-action-value tuples, $\{\mathbf{s}_t^{jn}, \mathbf{u}_t, V(\mathbf{s}_t^{jn}; \phi_{CADRL})\}$, from an existing CADRL solution. The network loss combines square-error loss on the value output and softmax cross-entropy loss between the policy output and the one-hot encoding of the closest discrete action to the one in D, described in Lines 1-6 of Algorithm 1.

The initialization step is necessary to enable any possibility of later generating useful RL experiences (non-initialized agents wander randomly and probabilistically almost never obtain positive reward). Agents running the

Algorithm 1 GA3C-CADRL Training

Input: trajectory training set, D
Output: policy network $\pi(\cdot;\theta)$

// Initialization
1: **for** N_{epochs} **do**
2: $\{\mathbf{s}_t^o, \tilde{\mathbf{S}}_t^o, u_t, V_t\} \leftarrow$ grabBatch(D)
3: $\bar{u}_t \leftarrow$ closestOneHot(u_t)
4: $\mathcal{L}_V = (V_t - V(\mathbf{s}_t^o, \tilde{\mathbf{S}}_t^o; \phi)^2$
5: $\mathcal{L}_\pi =$ softmaxCELogits($\bar{\mathbf{u}}_t, \mathbf{s}_t^o, \tilde{\mathbf{S}}_t^o, \theta$)
6: $\pi(\cdot;\theta), V(\cdot;\phi) \leftarrow$ trainNNs($\mathcal{L}_\pi, \mathcal{L}_V, \theta, \phi$)
7: **end for**
// Parallel Environment Threads
8: **for all** env **do**
9: $\mathbf{S}_0 \leftarrow$ randomTestCase()
10: **while** some agent not done **do**
11: **for all** agent, j **do**
12: $\mathbf{s}_g^o, \tilde{\mathbf{S}}_g^o \leftarrow$ sensorUpdate()
13: $\mathbf{s}^o, \tilde{\mathbf{S}}^o \leftarrow$ transform($\mathbf{s}_g^o, \tilde{\mathbf{S}}_g^o$)
14: **end for**
15: $\{\mathbf{u}_{t,j}\} \sim \pi(\mathbf{s}_{t,j}^o, \tilde{\mathbf{S}}_{t,j}^o; \theta) \forall j$
16: **for all** not done agent, j **do**
17: **if** agent not running GA3C-CADRL **then**
18: $\mathbf{u}_{t,j} \leftarrow$ policy($\mathbf{s}_{t,j}^o, \tilde{\mathbf{S}}_{t,j}^o$)
19: **end if**
20: $\mathbf{s}_{j,t+1}, \tilde{\mathbf{S}}_{j,t+1}, r_{j,t} \leftarrow$ moveAgent($\mathbf{u}_{j,t}$)
21: **end for**
22: **for all** not done GA3C-CADRL agent, j **do**
23: $r_{t,j} \leftarrow$ checkRewards($\mathbf{S}_{t+1}, \mathbf{u}_{t,j}$)
24: addToExperienceQueue($\mathbf{s}_{t,j}^o, \tilde{\mathbf{S}}_{t,j}^o, \mathbf{u}_{t,j}, r_{t,j}$)
25: **end for**
26: **end while**
27: **end for**
// Training Thread
28: **for** $N_{episodes}$ **do**
29: $\{\mathbf{s}_{t+1}^o, \tilde{\mathbf{S}}_{t+1}^o, \mathbf{u}_t, r_t\} \leftarrow$ grabBatchFromQueue()
30: $\theta, \phi \leftarrow$ trainGA3C($\theta, \phi, \{\mathbf{s}_{t+1}^o, \tilde{\mathbf{S}}_{t+1}^o, \mathbf{u}_t, r_t\}$)
31: **end for**
32: **return** π
33: **return** P

initialized GA3C-CADRL policy reach their goals reliably when there are no interactions with other agents. However, the policy after this supervised learning process still performs poorly in collision avoidance. This observation contrasts with CADRL, in which the initialization step was sufficient to learn a policy that performs comparably to existing reaction-based methods, due to relatively-low dimension value function combined with manual propagation of states. Key reasons behind this contrast are the reduced structure in the GA3C-CADRL formulation (no forward propagation), and that the algorithm is now learning both a policy and value function (as opposed to just a value function), since the policy has an order of magnitude higher dimensionality than a scalar value function.

Algorithm 2 GA3C-CADRL Execution

Input: goal position, (g_x, g_y)
Output: next motor commands, $\mathbf{u}$
1: $\mathbf{s}_g^o, \tilde{\mathbf{S}}_g^o \leftarrow$ sensorUpdate()
2: $\mathbf{s}^o, \tilde{\mathbf{S}}^o \leftarrow$ transform($\mathbf{s}_g^o, \tilde{\mathbf{S}}_g^o$)
3: $s_{des}, \theta_{des} \leftarrow \pi(\mathbf{s}^o, \tilde{\mathbf{S}}^o)$
4: $\mathbf{u} \leftarrow$ control(s_{des}, θ_{des})
5: **return** $\mathbf{u}$

To improve the solution with RL, parallel simulation environments produce training experiences, described in Lines 8-24 of Algorithm 1. Each episode consists of 2-10 agents, with random start and goal positions, running a random assortment of policies (Non-Cooperative, Zero Velocity, or the learned GA3C-CADRL policy at that iteration) (Line 9). Agent parameters vary between $r \in [0.2, 0.8]$m and $v_{pref} \in [0.5, 2.0]$m/s, chosen to be near pedestrian values. Agents sense the environment and transform measurements to their ego frame to produce the observation vector (Lines 12, 13). Each agent sends its observation vector to the policy queue and receives an action sampled from the current iteration of the GA3C-CADRL policy (Line 15). Agents that are not running the GA3C-CADRL policy use their own policy to overwrite $\mathbf{u}_{t,j}$ (Line 18). Then, all agents that have not reached a terminal condition (collision, at goal, timed out), simultaneously move according to $\mathbf{u}_{t,j}$ (Line 20). After all agents have moved, the environment evaluates $R(\mathbf{s}^{jn}, \mathbf{u})$ for each agent, and experiences from GA3C-CADRL agents are sent to the training queue (Lines 23,24).

In another thread, experiences are popped from the queue to produce training batches (Line 29). These experience batches are used to train a single GA3C-CADRL policy (Line 30) as in [25].

An important benefit of the new framework is that the policy can be trained on scenarios involving any number of agents, whereas the maximum number of agents had to be defined ahead of time with CADRL/SA-CADRL.[6] This work begins the RL phase with 2-4 agents in the environment, so that the policy learns the idea of collision avoidance in reasonably simple domains. Upon convergence, a second RL phase begins with 2-10 agents in the environment.

D. POLICY INFERENCE

Inference of the trained policy for a single timestep is described in Algorithm 2. As in training, GA3C-CADRL agents sense the environment, transfer to the ego frame, and select an action according to the policy (Lines 1-3). Like many RL algorithms, actions are sampled from the stochastic policy during training (exploration), but the action with highest probability mass is selected during inference (exploitation). A necessary addition for hardware is a low-level

[6]Experiments suggest this number should be below about 6 for convergence

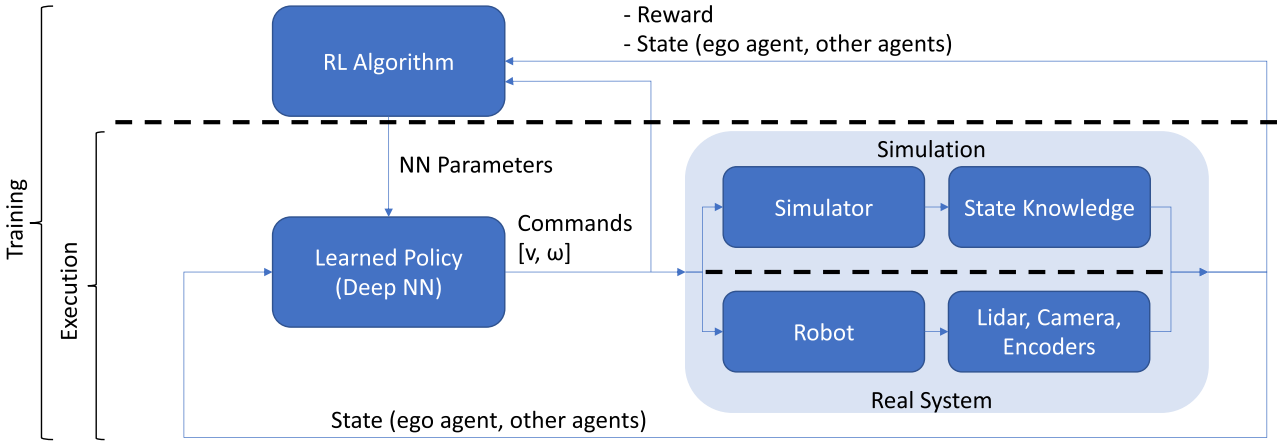


FIGURE 4. System Architecture. During training, the policy receives state measurements to compute robot commands, and the environment returns next states and rewards. Collections of (state, action, reward) tuples enable an RL algorithm to update the parameters of the learned policy. During execution, only the blocks below the upper dashed line run (NN parameters are fixed). The key difference between executing in simulation vs. the real robot is that the robot's onboard sensors (lidar, cameras, encoders) estimate the state of the environment (e.g., agents' positions, velocities).

controller to track the desired speed and heading angle (Line 4). Note that the value function is not used during inference; it is only learned to stabilize estimates of the policy gradients during training.

The architecture of the training and inference steps for the simulated and real robot system are shown in Fig. 4.

IV. RESULTS

A. COMPUTATIONAL DETAILS

The DNNs in this work were implemented with TensorFlow [37] in Python. Each query of the GA3C-CADRL network only requires the current state vector, and takes on average 0.4-0.5ms on a i7-6700K CPU, which is approximately 20 times faster than before [16]. Note that a GPU is not required for online inference in real time, and CPU-only training was faster than hybrid CPU-GPU training on our hardware.

In total, the RL training converges in about 24 hours (after $2 \cdot 10^6$ episodes) for the multiagent, LSTM network on a computer with an i7-6700K CPU with 32 parallel environment threads. A limiting factor of the training time is the low learning rate required for stable training. Recall that earlier approaches [16] took 8 hours to train a 4-agent value network, but now the network learns both the policy and value function and without being provided any structure about the other agents' behaviors. The larger number of training episodes can also be attributed to the stark contrast in initial policies upon starting RL between this and the earlier approach: CADRL was fine-tuning a decent policy, whereas GA3C-CADRL learns collision avoidance entirely in the RL phase.

The performance throughout the training procedure is shown as the "closest last" curve in Fig. 16 (the other curves are explained in Section IV-D2). The mean $\pm 1\sigma$ rolling reward over 5 training runs is shown. After initialization, the agents receive on average 0.15 reward per episode. After RL phase 1 (converges in $1.5 \cdot 10^6$ episodes), they average 0.90 rolling reward per episode. When RL phase 2 begins, the domain becomes much harder (n_{max} increases from 4 to 10), and rolling reward increases until converging at 0.93 (after a total of $1.9 \cdot 10^6$ episodes). Rolling reward is computed as the sum of the rewards accumulated in each episode, averaged across all GA3C-CADRL agents in that episode, averaged over a window of recent episodes. Rolling reward is only a measure of success/failure, as it does not include the discount factor and thus is not indicative of time efficiency. Because the maximum receivable reward on any episode is 1, an average reward < 1 implies there are some collisions (or other penalized behavior) even after convergence. This is expected, as agents sample from their policy distributions when selecting actions in training, so there is always a non-zero probability of choosing a sub-optimal action in training. Later, when executing a trained policy, agents select the action with highest probability.

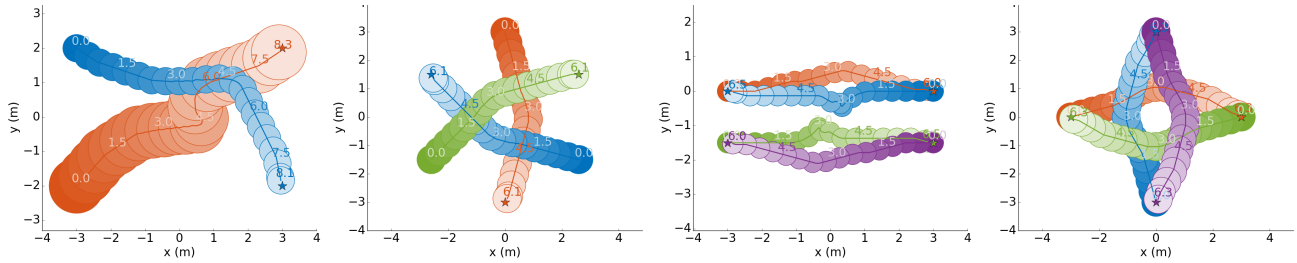
Key hyperparameter values include: learning rate $L_r = 2 \cdot 10^{-5}$, entropy coefficient $\beta = 1 \cdot 10^{-4}$, discount $\gamma = 0.97$, training batch size $b_s = 100$, and we use the Adam optimizer [38].

B. SIMULATION RESULTS

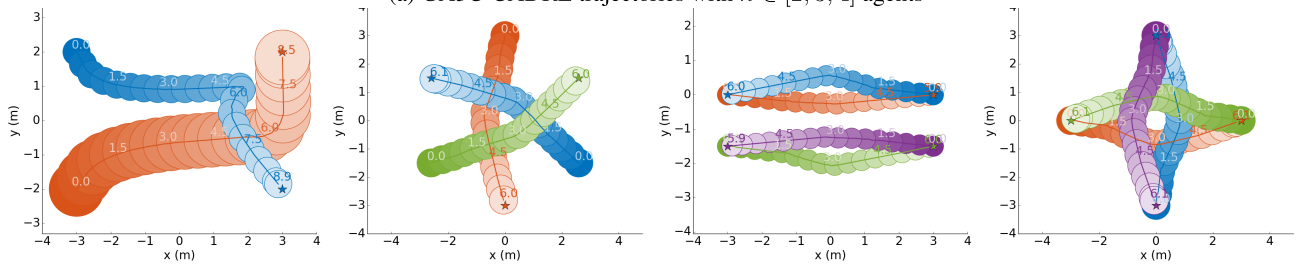
1) BASELINES

This section compares the proposed GA3C-CADRL algorithm to ORCA [5], SA-CADRL [16], and, where applicable, DRLMACA [10].

We first briefly summarize the ORCA, SA-CADRL, and DRLMACA algorithms. In ORCA, agents solve a one-step optimization problem to make a minimal adjustment to the desired velocity vector, such that the new velocity does not collide with other agents in the future (assuming they travel at a constant velocity). Because the one-step horizon and constant velocity assumption leads to mypoic planning,



(a) GA3C-CADRL trajectories with $n \in [2, 3, 4]$ agents



(b) SA-CADRL trajectories with $n \in [2, 3, 4]$ agents

FIGURE 5. Scenarios with $n \leq 4$ agents. The top row shows agents executing GA3C-CADRL-10-LSTM, and the bottom row shows same scenarios with agents using SA-CADRL. Circles lighten as time increases, the numbers represent the time at agent's position, and circle size represents agent radius. GA3C-CADRL agents are slightly less efficient, as they reach their goals slightly slower than SA-CADRL agents. However, the overall behavior is similar, and the more general GA3C-CADRL framework generates desirable behavior without many of the assumptions from SA-CADRL.

SA-CADRL improves on that approach by learning a value function that encodes the time-to-goal from various states. Thus, the online optimization of SA-CADRL also considers the long-horizon impact of a local control command, via a quick lookup (DNN query). Like SA-CADRL, DRLMACA also uses deep RL, but the inputs to the policy include raw sensor data (laserscans) to inform collision avoidance, rather than using agent position, velocity, and radius estimates as in SA-CADRL and this work.

In our experiments, ORCA agents must inflate agent radii by 5% to reduce collisions caused by numerical issues. Without this inflation, over 50% of experiments with 10 ORCA agents had a collision. This inflation led to more ORCA agents getting stuck, which is better than a collision in most applications. The time horizon parameter in ORCA impacts the trajectories significantly; it was set to 5 seconds.

Although the original 2-agent CADRL algorithm [15] was also shown to scale to multiagent scenarios, its minimax implementation is limited in that it only considers one neighbor at a time as described in [16]. For that reason, this work focuses on the comparison against SA-CADRL which has better multiagent properties - the policy used for comparison is the same one that was used on the robotic hardware in [16]. That particular policy was trained with some noise in the environment ($\mathbf{p} = \mathbf{p}_{actual} + \sigma$) which led to slightly poorer performance than the ideally-trained network as reported in the results of [16], but more acceptable hardware performance.

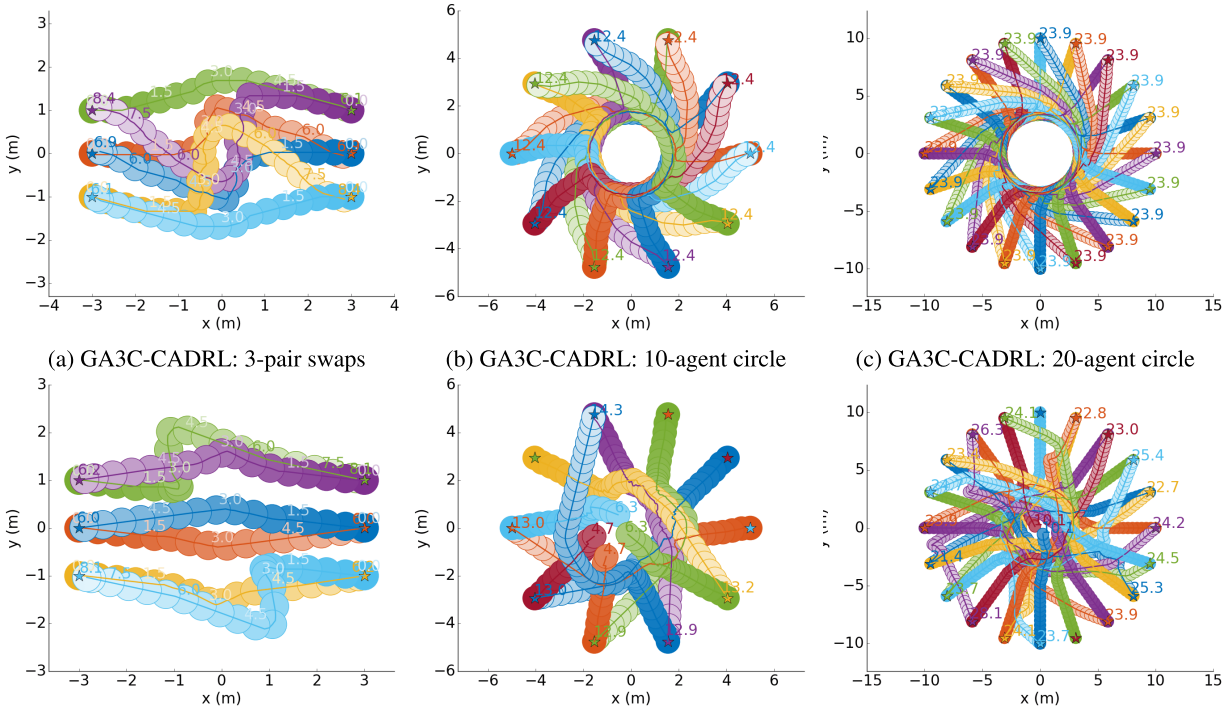
The version of the new GA3C-CADRL policy after RL phase 2 is denoted GA3C-CADRL-10, as it was trained in scenarios of up to 10 agents. To create a more fair comparison with SA-CADRL which was only trained with up to 4 agents, let GA3C-CADRL-4 denote the policy after RL phase 1 (which only involves scenarios of up to 4 agents). Recall GA3C-CADRL-4 can still be naturally implemented on $n > 4$ agent cases, whereas SA-CADRL can only accept up to 3 nearby agents' states regardless of n.

2) $n \leq 4$ AGENTS: NUMERICAL COMPARISON TO BASELINES

The previous approach (SA-CADRL) is known to perform well on scenarios involving a few agents ($n \leq 4$), as its trained network can accept up to 3 other agents' states as input. Therefore, a first objective is to confirm that the new algorithm can still perform comparably with small numbers of agents. This is not a trivial check, as the new algorithm is not provided with any structure/prior about the world's dynamics, so the learning is more difficult.

Trajectories are visualized in Fig. 5: the top row shows scenarios with agents running the new policy (GA3C-CADRL-10-LSTM), and the bottom row shows agents in identical scenarios but using the old policy (SA-CADRL). The colors of the circles (agents) lighten as time increases and the circle size represents agent radius. The trajectories generally look similar for both algorithms, with SA-CADRL being slightly more efficient. A rough way to assess efficiency in these plotted paths is time indicated when the agents reach their goals.

Although it is easy to pick out interesting pros/cons for any particular scenario, it is more useful to draw conclusions after aggregating over a large number of randomly-generated cases. Thus, we created test sets of 500 random scenarios, defined by ($p_{start}, p_{goal}, r, v_{pref}$) per agent, for many different numbers of agents. Each algorithm is evaluated on the same 500 test cases. The comparison metrics are the percent of



(a) GA3C-CADRL: 3-pair swaps (b) GA3C-CADRL: 10-agent circle (c) GA3C-CADRL: 20-agent circle

(d) SA-CADRL: 3-pair swaps (e) SA-CADRL: 10-agent circle (f) SA-CADRL: 20-agent circle

FIGURE 6. Scenarios with $n > 4$ agents. In the 3-pair swap Figs. 6a and 6d, GA3C-CADRL agents exhibit interesting multiagent behavior: two agents form a pair while passing the opposite pair of agents. SA-CADRL agents reach the goal more quickly than GA3C-CADRL agents, but such multiagent behavior is a result of GA3C-CADRL agents having the capacity to observe all of the other 5 agents each time step. In other scenarios, GA3C-CADRL agents successfully navigate the 10- and 20-agent circles, whereas some SA-CADRL agents collide (near $(-1, -1)$ and $(0, 0)$ in Fig. 6e and $(0, 0)$ in Fig. 6f).

cases with a collision, percent of cases where an agent gets stuck and doesn't reach the goal, and the remaining cases where the algorithm was successful, the average extra time to goal, $\bar{t}_g^e$ beyond a straight path at v_{pref}. These metrics provide measures of efficiency and safety.

Aggregated results in Fig. 7 compare a model-based algorithm, ORCA [5], SA-CADRL [16], and several variants of the new GA3C-CADRL algorithm. With $n \leq 4$ agents in the environment (a), SA-CADRL has the lowest $\bar{t}_g^e$, and the agents rarely fail in these relatively simple scenarios.

3) $n \leq 4$ AGENTS: ABLATION STUDY

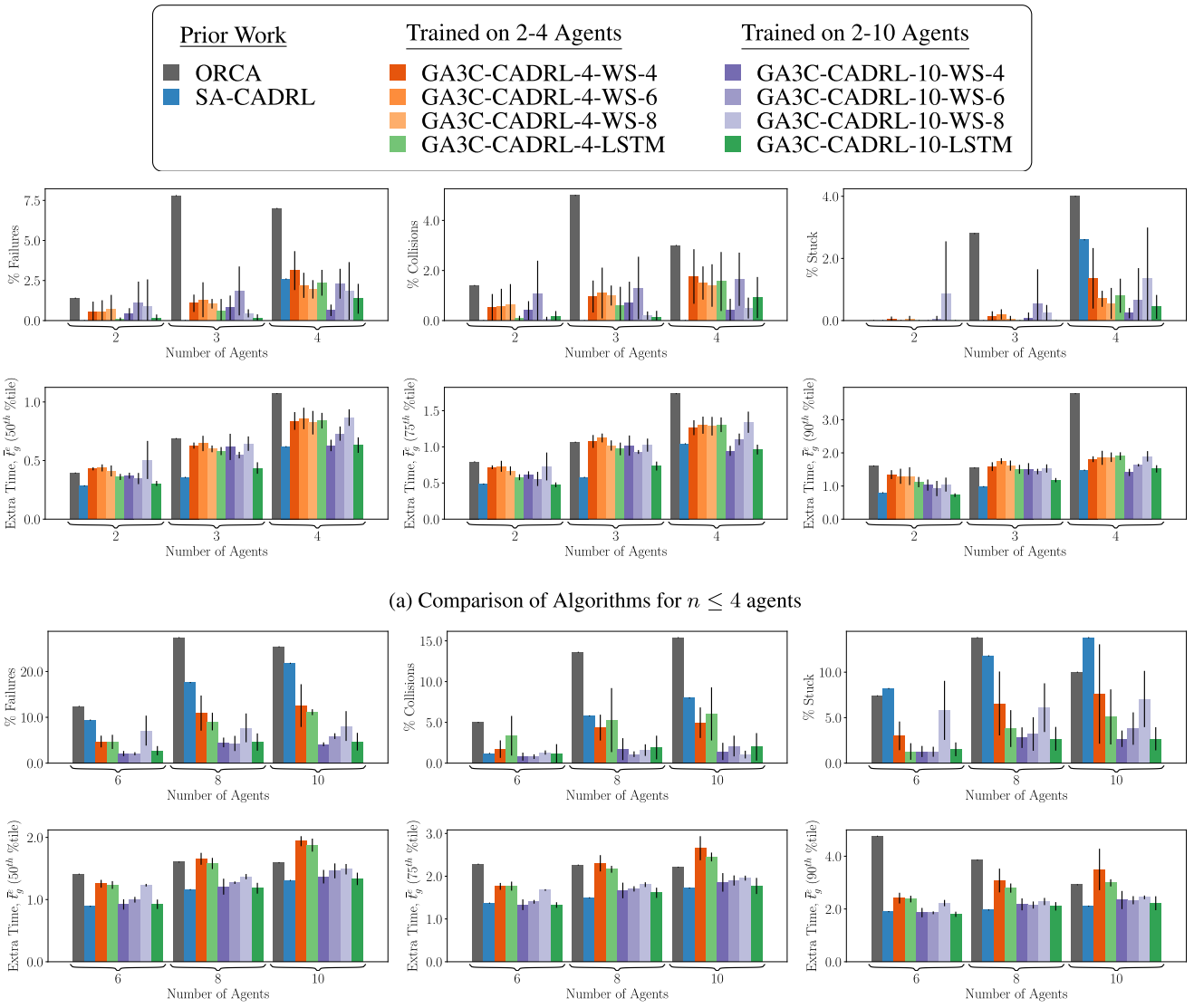
There are several algorithmic differences between SA-CADRL and GA3C-CADRL: we compare each ablation one-by-one. With the same network architecture (pre-defined number of agents with weights shared (WS) for all agents), GA3C-CADRL-4-WS-4 loses some performance versus SA-CADRL, since GA3C-CADRL must learn the notion of a collision, whereas SA-CADRL's constant-velocity collision checking may be reasonable for small n. Replacing the WS network head with LSTM improves the performance when the number of agents is below network capacity, potentially because the LSTM eliminates the need to pass "dummy" states to fill the network input vector. Lastly, the second training phase (2-10 agents) improves policy performance even for small numbers of agents.

Overall, the GA3C-CADRL-10-LSTM variant performs comparably, though slightly worse, than SA-CADRL for small numbers of agents, and outperforms the model-based ORCA algorithm.

4) $n > 4$ AGENTS: NUMERICAL COMPARISON TO BASELINES

A real robot will likely encounter more than 3 pedestrians at a time in a busy environment. However, experiments with the SA-CADRL algorithm suggest that increasing the network capacity beyond 4 total agents causes convergence issues. Thus, the approach taken here for SA-CADRL is to supply only the closest 3 agents' states in crowded scenarios. The GA3C-CADRL policy's convergence is not as sensitive to the maximum numbers of agents, allowing an evaluation of whether simply expanding the network input size improves performance in crowded scenarios. Moreover, the LSTM variant of GA3C-CADRL relaxes the need to pre-define a maximum number of agents, as any number of agents can be fed into the LSTM and the final hidden state can still be taken as a representation of the world configuration.

Even in $n > 4$-agent environments, interactions still often only involve a couple of agents at a time. Some specific cases where there truly are many-agent interactions are visualized in Fig. 6. In the 6-agent swap (left), GA3C-CADRL agents exhibit interesting multiagent behavior: the bottom-left and middle-left agents form a pair while passing the top-right and



(a) Comparison of Algorithms for $n \leq 4$ agents

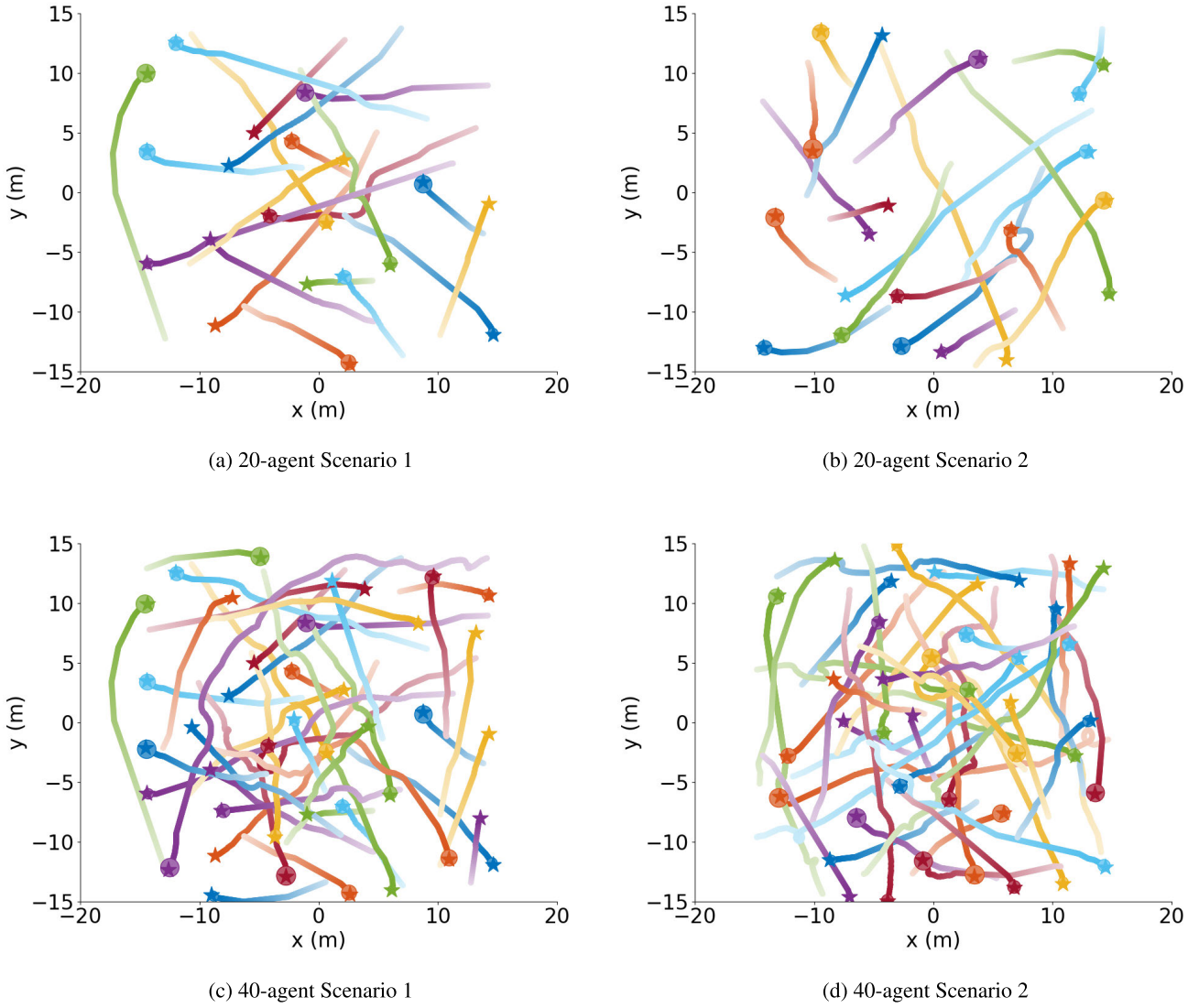
(b) Comparison of Algorithms for $n > 4$ agents

FIGURE 7. Numerical comparison on the same 500 random test cases (lower is better). The GA3C-CADRL-10-LSTM network shows comparable performance to SA-CADRL for small *n* (a), much better performance for large *n* (b), and better performance than model-based ORCA for all *n*. Several ablations highlight SA-CADRL and GA3C-CADRL differences. With the same architecture (SA-CADRL & GA3C-CADRL-4), the GA3C policy performs better for large *n* (b), but worsens performance for small *n* (a). Adding a second phase of training with up to 10 agents (GA3C-CADRL-10-4) improves performance for all *n* tested. Adding additional pre-defined agent capacity to the network (GA3C-CADRL-10-6, GA3C-CADRL-10-8) can degrade performance. The LSTM (GA3C-CADRL-10-LSTM) adds flexibility in prior knowledge on number of agents, maintaining similar performance to the WS approaches for large *n* and recovering comparable performance to SA-CADRL for small *n*.

middle-right agents. This phenomenon leads to a particularly long path for bottom-left and top-right agents, but also allows the top-left and bottom-right agents to not deviate much from a straight line. In contrast, in SA-CADRL the top-left agent starts moving right and downward, until the middle-right agent becomes one of the closest 3 neighbors. The top-left agent then makes an escape maneuver and passes the top-right on the outside. In this case, SA-CADRL agents reach the goal more quickly than GA3C-CADRL agents, but the interesting multiagent behavior is a result of GA3C-CADRL agents having the capacity to observe all of the other 5 agents each time step, rather than SA-CADRL which just uses the nearest 3 neighbors. GA3C-CADRL agents successfully navigate the 10- and 20-agent circles (antipodal swaps), whereas several SA-CADRL agents get stuck or collide.[7]

Statistics across 500 random cases of 6, 8, and 10 agents are shown in Fig. 7b. The performance gain by using GA3C-CADRL becomes larger as the number of agents

[7]Note there is not perfect symmetry in these SA-CADRL cases: small numerical fluctuations affect the choice of the closest agents, leading to slightly different actions for each agent. And after a collision occurs with a pair of agents, symmetry will certainly be broken for future time steps.



(a) 20-agent Scenario 1

(b) 20-agent Scenario 2

(c) 40-agent Scenario 1

(d) 40-agent Scenario 2

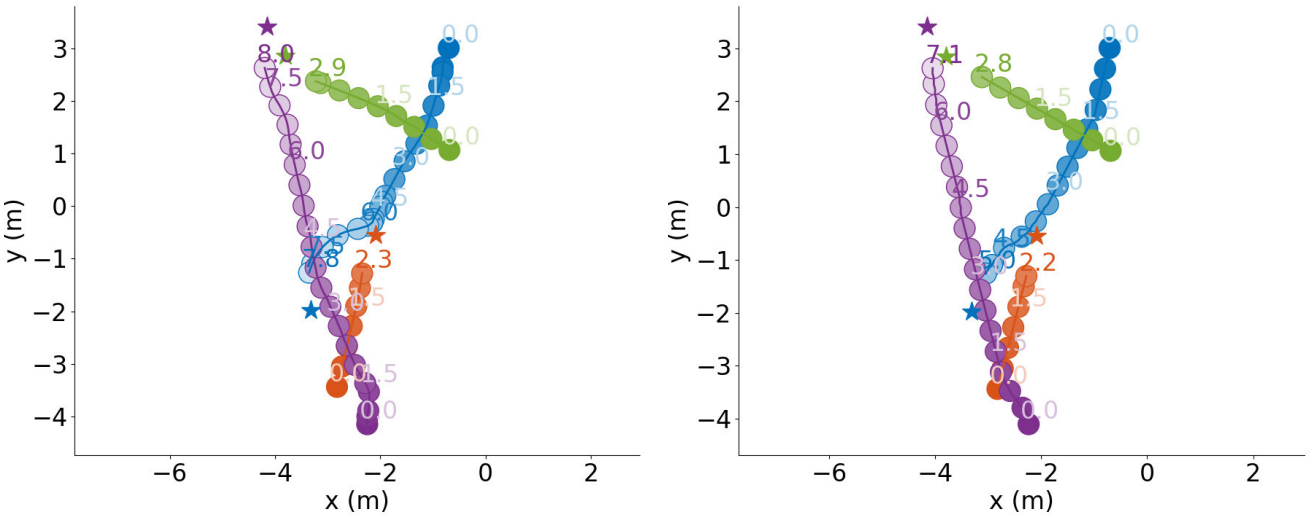
FIGURE 8. Random 20- and 40-agent scenarios. These figures highlight the learned policy's ability to handle large numbers of agents with a single LSTM-based representation (GA3C-CADRL-10). All agents reach their goals without collision in (a)-(d).

in the environment increases. For $n = 6, 8, 10$, GA3C-CADRL-10-LSTM shows a 3-4x reduction in failed cases with similar $\bar{t}_g^e$ compared to SA-CADRL. GA3C-CADRL-10-LSTM's percent of success remains above 95% across any $n \leq 10$, whereas SA-CADRL drops to under 80%. It is worth noting that SA-CADRL agents' failures are more often a result of getting stuck rather than colliding with others, however neither outcomes are desirable. The GA3C-CADRL variants outperform model-based ORCA for large n as well. The domain size of $n = 10$ agent scenarios is set to be larger (12×12 vs. $8 \times 8m$) than cases with smaller n to demonstrate cases where n is large but the world is not necessarily more densely populated with agents.

Furthermore, Fig. 8 shows that the learned policy generalizes beyond the 2-10 agent scenarios it was trained on. In particular, Figs. 8a and 8b show 2 random 20-agent scenarios and Figs. 8c and 8d show 2 random 40-agent scenarios. In addition to the examples shown, across 10 random trials of 20-agent scenarios, all agents reached their goals successfully (0 collisions or stuck agents). Across 10 random trials of 40-agent scenarios, only 4/400 trajectories ended in collisions, and 1 agent became stuck, with 395/400 agents reaching their goals successfully.

5) $n > 4$ AGENTS: ABLATION STUDY

We now discuss the GA3C-CADRL variants. For large n, GA3C-CADRL-WS-4 strongly outperforms SA-CADRL. Since the network architectures and number of agents trained on are the same, the performance difference highlights the benefit of the policy-based learning framework. Particularly for large n, the multiagent interactions cause SA-CADRL's constant velocity assumption about other



(a) DRLMACA trajectory with 4 agents

(b) GA3C-CADRL trajectory with 4 agents

FIGURE 9. GA3C-CADRL and DRLMACA 4-agent trajectories. Both algorithms produce collision-free paths; numbers correspond to the timestamp agents achieved that position. GA3C-CADRL agents are slightly faster; the bottom-most DRLMACA agent (left) slows down near the start of the trajectory, leading to a larger time to goal (8.0 vs. 7.1 seconds), whereas the bottom-most GA3C-CADRL agent (right) cuts behind another agent near its v_{pref}. Similarly, the top-right DRLMACA agent slows down near $(-2, 0)$ (overlapping circles), whereas the top-right GA3C-CADRL agent maintains high speed and reaches the goal faster (7.8 vs. 5.0 seconds). Agents stop moving once within $0.8m$ of their goal position in these comparisons.

agents (to convert the value function to a policy in (9)) to become unrealistic. Replacing the WS network head with LSTM (which accepts observations of any number of agents) causes slight performance improvement for $n = 6, 8$ (GA3C-CADRL-4-WS-4 vs. GA3C-CADRL-4-LSTM). Since GA3C-CADRL-4-WS-6,8 never saw $n > 4$ agents in training, these are omitted from Fig. 7b (as expected, their performance is awful). The second training phase (on up to 10 agents) leads to another big performance improvement (GA3C-CADRL-4-* vs. GA3C-CADRL-10-*). The additional network capacity of the WS approaches (GA3C-CADRL-WS-4,6,8) appears to have some small, in some cases negative, performance impact. That observation suggests that simply increasing the maximum number of agents input to the network does not address the core issues with multiagent collision avoidance. The GA3C-CADRL-10-LSTM variant performs similarly to GA3C-CADRL-10-WS-4, while providing a more flexible network architecture (and better performance for small n, as described earlier).

The ability for GA3C-CADRL to retrain in complex scenarios after convergence in simple scenarios, and yield a significant performance increase, is a key benefit of the new framework. This result suggests there could be other types of complexities in the environment (beyond increasing n) that the general GA3C-CADL framework could also learn about after being initially trained on simple scenarios.

6) COMPARISON TO OTHER RL APPROACH

Table 1 shows a comparison to another deep RL policy, DRLMACA [10]. DRLMACA stacks the latest 3 laserscans as the observation of other agents; other algorithms in the comparisons use the exact other agent states. DRLMACA assumes $v_{pref} = 1m/s$ for all agents, so all test cases used in Table 1 share this setting (v_{pref} is random in Fig. 7, explaining the omission of DRLMACA).

During training, all DRLMACA agents are discs of the same radius, R, and some reported trajectories from [10] suggest the policy can generalize to other agent sizes. However, our experiments with a trained DRLMACA policy [39] suggest the policy does not generalize to other agent radii, as the number of collisions increases with agent radius. In particular, 69% of experiments ended in a collision for 4 agents running DRLMACA with $r = 0.5m$. Moreover, a qualitative look at DRLMACA trajectories in Fig. 9 demonstrates how agents often slow down and stop to wait for other agents, whereas GA3C-CADRL agents often move out of other agents' paths before needing to stop. Even though the implementation in [39] uses the same hyperparameters, training scheme, network architecture, and reward function from the paper, these results are worse than what was reported in [10].

7) FORMATION CONTROL

Formation control is one application of multiagent robotics that requires collision avoidance: recent examples include drone light shows [40], commercial airplane formations [41], robotic soccer [42], and animations [43]. One possible formation objective is to assign a team of agents to goal coordinates, say to spell out letters or make a shape.

Fig. 10 shows 6 agents spelling out the letters in "CADRL". Each agent uses GA3C-CADRL-10-LSTM and knowledge of other agents' current positions, velocities, and

TABLE 1. Performance of ORCA [5], SA-CADRL [16], DRLMACA [10], and GA3C-CADRL (new) algorithms on the same 100 random test cases, for various agent radii, with $v_{pref} = 1.0\ m/s$ for all agents. For both $r = 0.2m$ and $r = 0.5m$, GA3C-CADRL outperforms DRLMACA, and DRLMACA performance drops heavily for $r = 0.5m$, with 69% collisions in random 4-agent scenarios.

	Test Case Setup					
size (m)		8 x 8	8 x 8		8 x 8	8 x 8
# agents		2	4		2	4
	Extra time to goal $\bar{t}_g^e$ (s) (Avg / 75th / 90th percentile) $\Rightarrow$ smaller is better					
ORCA	$r = 0.2m$	0.18 / 0.39 / 0.82	0.4 / 0.62 / 2.4	$r = 0.5m$	0.43 / 1.11 / 1.65	0.95 / 1.22 / 1.86
SA-CADRL		0.2 / 0.29 / 0.43	**0.26 / 0.38 / 0.75**		**0.27 / 0.37 / 0.66**	**0.48 / 1.03 / 1.71**
DRLMACA		0.91 / 1.33 / 4.82	1.46 / 2.16 / 3.35		0.72 / 1.12 / 1.33	1.26 / 2.42 / 2.57
GA3C-CADRL-10-LSTM		0.17 / 0.24 / 0.71	**0.25 / 0.53 / 0.7**		**0.27 / 0.37 / 0.57**	**0.6 / 1.24 / 1.69**
	% failures (% collisions / % stuck) $\Rightarrow$ smaller is better					
ORCA	$r = 0.2m$	4 (4 / 0)	7 (7 / 0)	$r = 0.5m$	4 (4 / 0)	12 (9 / 3)
SA-CADRL		**0 (0 / 0)**	2 (0 / 2)		**0 (0 / 0)**	2 (0 / 2)
DRLMACA		3 (0 / 3)	14 (8 / 6)		23 (23 / 0)	71 (69 / 2)
GA3C-CADRL-10-LSTM		**0 (0 / 0)**	**0 (0 / 0)**		**0 (0 / 0)**	**1 (1 / 0)**

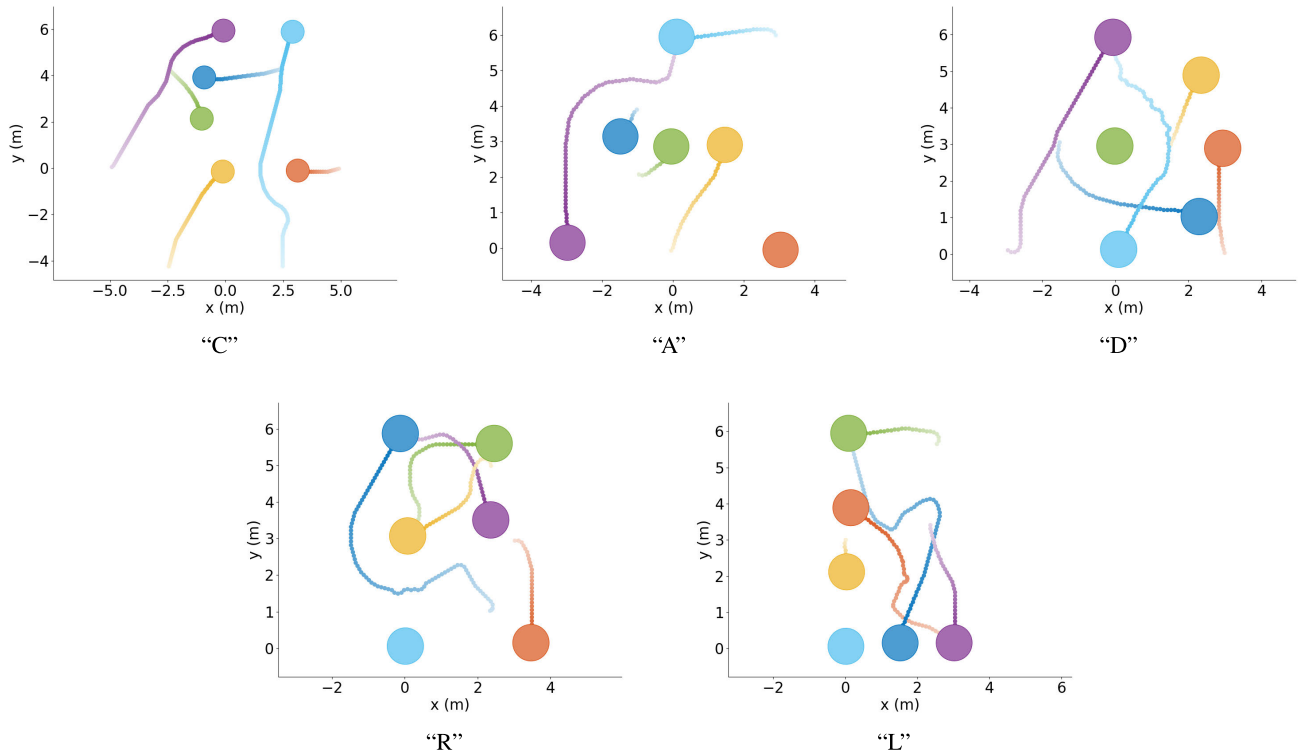


FIGURE 10. 6 agents spelling out "CADRL". Each agent is running the same GA3C-CADRL-10-LSTM policy. A centralized system randomly assigns agents to goal positions (random to ensure interaction), and each agent selects its action in a decentralized manner, using knowledge of other agents' current positions, velocities, and radii. Collision avoidance is an essential aspect of formation control.

radii, to choose a collision-free action toward its own goal position. All goal coordinates lie within a $6 \times 6m$ region, and goal coordinates are randomly assigned to agents. Each agent has a radius of $0.5m$ and a preferred speed of $1.0m/s$. Agents start in random positions before the first letter, "C", then move from "C" to "A", etc. Agent trajectories darken as time increases, and the circles show the final agent positions. Multiple iterations are animated in the video attachment.

C. HARDWARE EXPERIMENTS

This work implements the policy learned in simulation on two different hardware platforms to demonstrate the flexibility in the learned collision avoidance behavior and that the learned policy enables real-time decision-making. The first platform, a fleet of 4 multirotors, highlights the transfer of the learned policy to vehicles with more complicated dynamics than the unicycle kinematic model used in training. The second platform, a ground robot operating among pedestrians, highlights the policy's robustness to both imperfect perception from low-cost, on-board perception, and to heterogeneity in other agent policies, as none of the pedestrians follow one of the policies seen in training.

The hardware experiments were designed to demonstrate that the new algorithm could be deployed using realistic sensors, vehicle dynamics, and computational resources. Combined with the numerical experiments in Fig. 7 and Table 1,

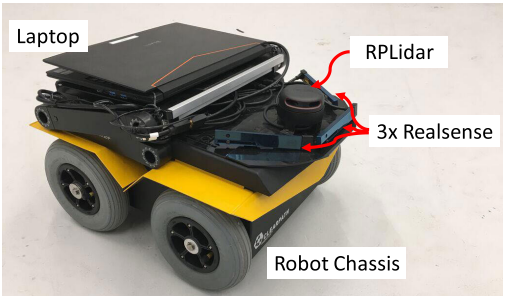


FIGURE 11. Robot hardware. The compact, low-cost (< $1000) sensing package uses a single 2D Lidar and 3 Intel Realsense R200 cameras. The total sensor and computation assembly is less than 3 inches tall, leaving room for cargo.

the hardware experiments provide evidence of an algorithm that exceeds the state of the art and can be deployed on real robots.

1) MULTIPLE MULTIROTORS

A fleet of 4 multirotors with on-board velocity and position controllers resemble the agents in the simulated training environment. These experiments consider the case of multirotors flying within the same plane (roughly 1m above the ground). Each vehicle's planner receives state measurements of the ego vehicle (position, velocity, heading angle) and of the other vehicles (positions, velocities) from a motion capture system at 200Hz [44]. At each planning step (10Hz), the planners build a state vector using other agent states, an assumed agent radius (0.5m), a preferred ego speed (0.5m/s), and knowledge of their own goal position in global coordinates. Each vehicle's planner queries the learned policy and selects the action with highest probability: a desired heading angle change and speed. A desired velocity vector with magnitude equal to the desired speed, and in the direction of the desired heading angle, is sent to the velocity controller. To smooth the near-goal behavior, the speed and heading rates decay linearly with distance to goal within 2m, and agents simply execute position control on the goal position when within 0.3m of the goal. Throughout flight, the multirotors also control to their desired heading angle; this would be necessary with a forward-facing sensor, but is somewhat extraneous given that other agents' state estimates are provided externally.

The experiments included two challenging 4-agent scenarios. In Fig. 12, two pairs of multirotors swap positions in parallel, much like the third column of Fig. 5. This policy was not trained to prefer a particular directionality – the agents demonstrate clockwise/left-handed collision avoidance behavior in the center of the room. In Fig. 13, the 4 vehicles swap positions passing through a common center, like the fourth column of Fig. 5. Unlike in simulation, where the agents' dynamics, observations, and policies (and therefore, actions) are identical, small variations in vehicle states lead to

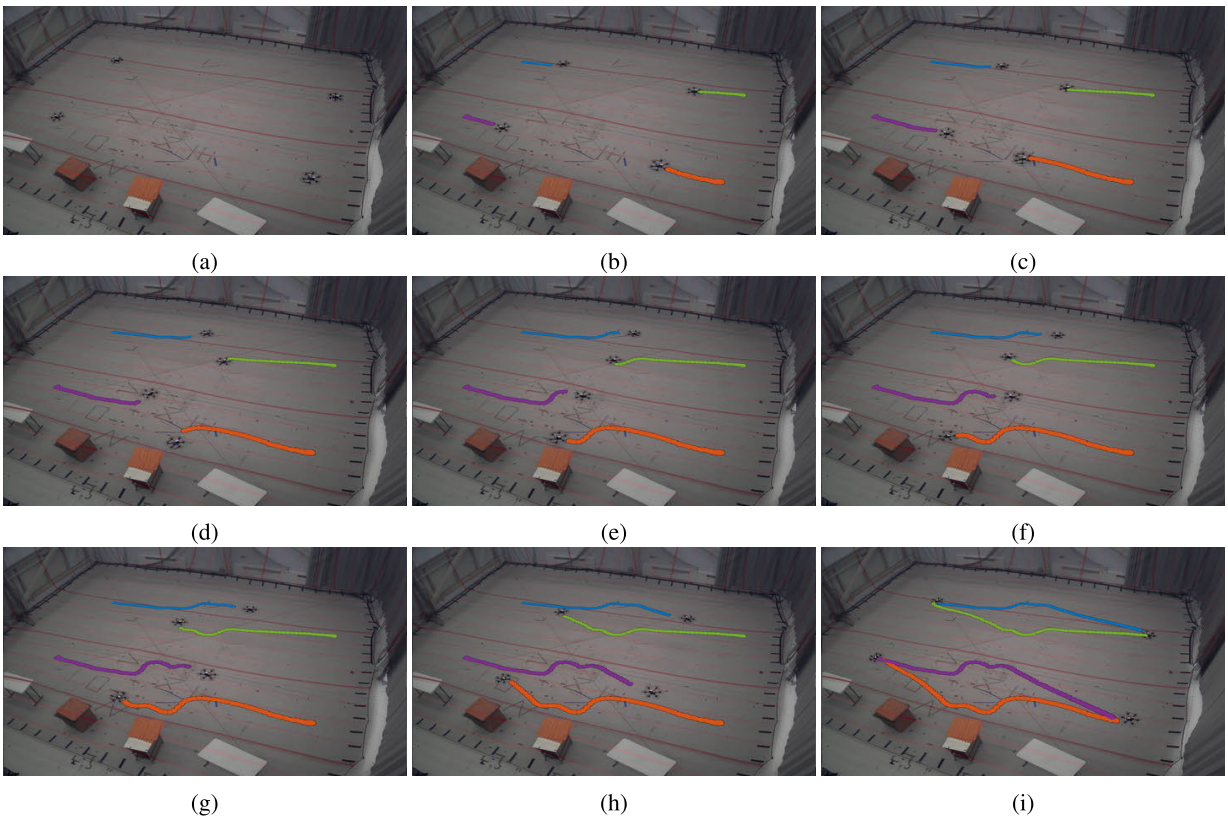


FIGURE 12. 4 Multirotors running GA3C-CADRL: 2 parallel pairs. Each vehicle's on-board controller tracks the desired speed and heading angle produced by each vehicle's GA3C-CADRL-10-LSTM policy.

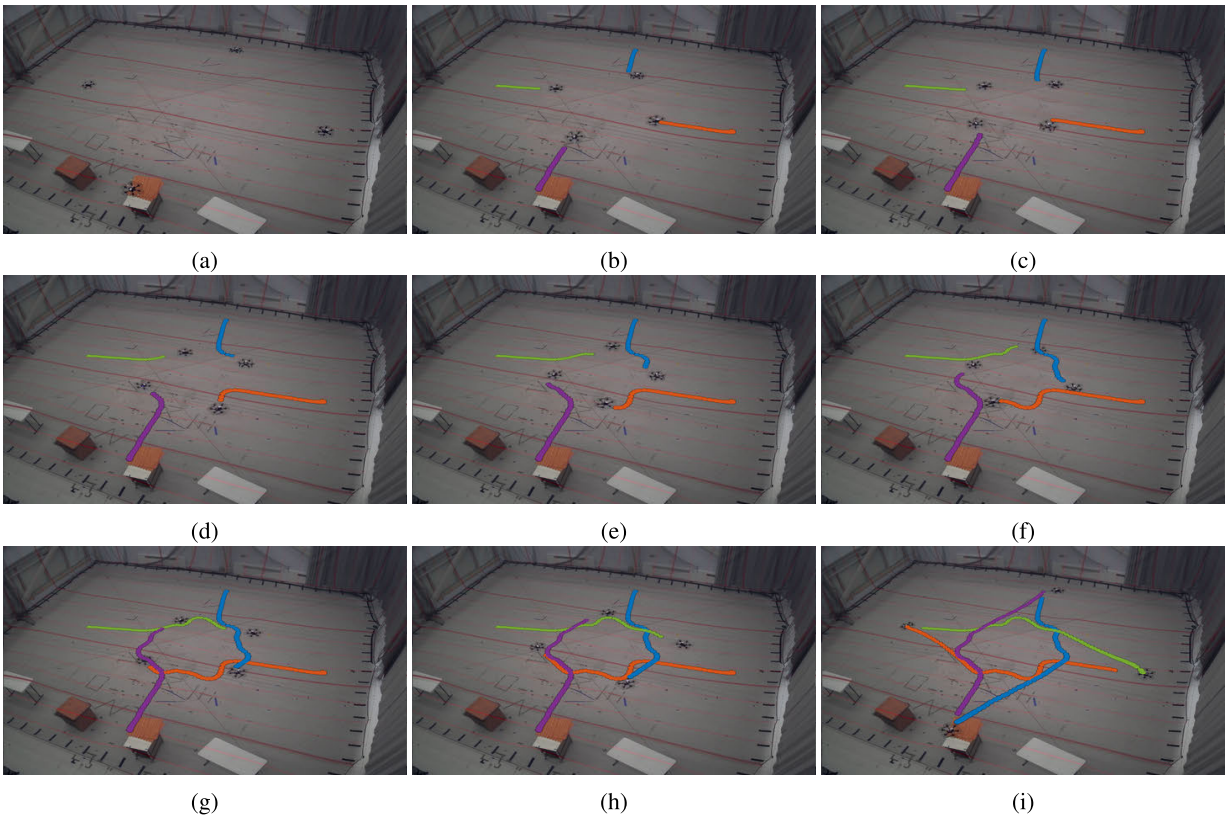


FIGURE 13. 4 Multirotors running GA3C-CADRL: 2 orthogonal pairs. The agents form a symmetric "roundabout" pattern in the center of the room, even though each vehicle has slightly different dynamics and observations of neighbors.

slightly different actions for each agent. However, even with these small fluctuations, the vehicles still perform ''roundabout'' behavior in the center.

Further examples of 2-agent swaps, and multiple repeated trials of the 4-agent scenarios are included in the video attachment.

2) GROUND ROBOT AMONG PEDESTRIANS

A GA3C-CADRL policy implemented on a ground robot demonstrates the algorithm's performance among pedestrians. We designed a compact, low-cost ($< \$1000$) sensing suite with sensors placed as to not limit the robot's cargo-carrying capability (Fig. 11). The sensors are a 2D Lidar (used for localization and obstacle detection), and 3 Intel Realsense R200 cameras (used for pedestrian classification and obstacle detection). Pedestrian positions and velocities are estimated by clustering the 2D Lidar's scan [45], and clusters are labeled as pedestrians using a classifier [46] applied to the cameras' RGB images [47]. A detailed description of the software architecture is in [48]. Despite not having perfect state knowledge, as was available in the simulations, the robot is able to avoid collisions using only on-board sensing.

Snapshots of a particular sequence are shown in Fig. 14: 6 pedestrians move around between the robot's starting position and its goal (large circle) about 6m away. Between the first two frames, 3 of the pedestrians remain stationary, and the other 3 move with varying levels of cooperativeness, but these roles were not assigned and change stochastically throughout the scenario. The robot successfully navigates to its goal in the proximity of many heterogeneous agents. Other examples of safe robot navigation in challenging scenarios are available in the video attachment.

D. LSTM ANALYSIS

This section provides insights into the design and inner workings of the LSTM module in Fig. 3 in two ways: how agent states affect the LSTM gates, and how the ordering of agents affects performance during training.

1) LSTM GATE DYNAMICS

We first analyze the LSTM of the trained GA3C-CADRL-10-LSTM network, building on [26], using notation from [36]. The LSTM weights, $\{W_i, W_f, W_o\}$, and biases, $\{b_i, b_f, b_o\}$ are updated during the training process and fixed during inference.

Recall the LSTM has three inputs: the state of agent j at time t, the previous hidden state, and the previous cell state, which are denoted $\tilde{\mathbf{s}}^o_{j,t}$, $\mathbf{h}_{t-1}$, C_{t-1}, respectively. Of the four gates in an LSTM, we focus on the input gate here. The input gate, $\mathbf{i}_t \in [0, 1]^{n_h}$, is computed as,

$$\mathbf{i}_t = \sigma([W_{i,s},\ W_{i,h},\ W_{i,b}]^T \cdot [\tilde{\mathbf{s}}^o_{j,t},\ \mathbf{h}_t,\ b_i]), \qquad (14)$$

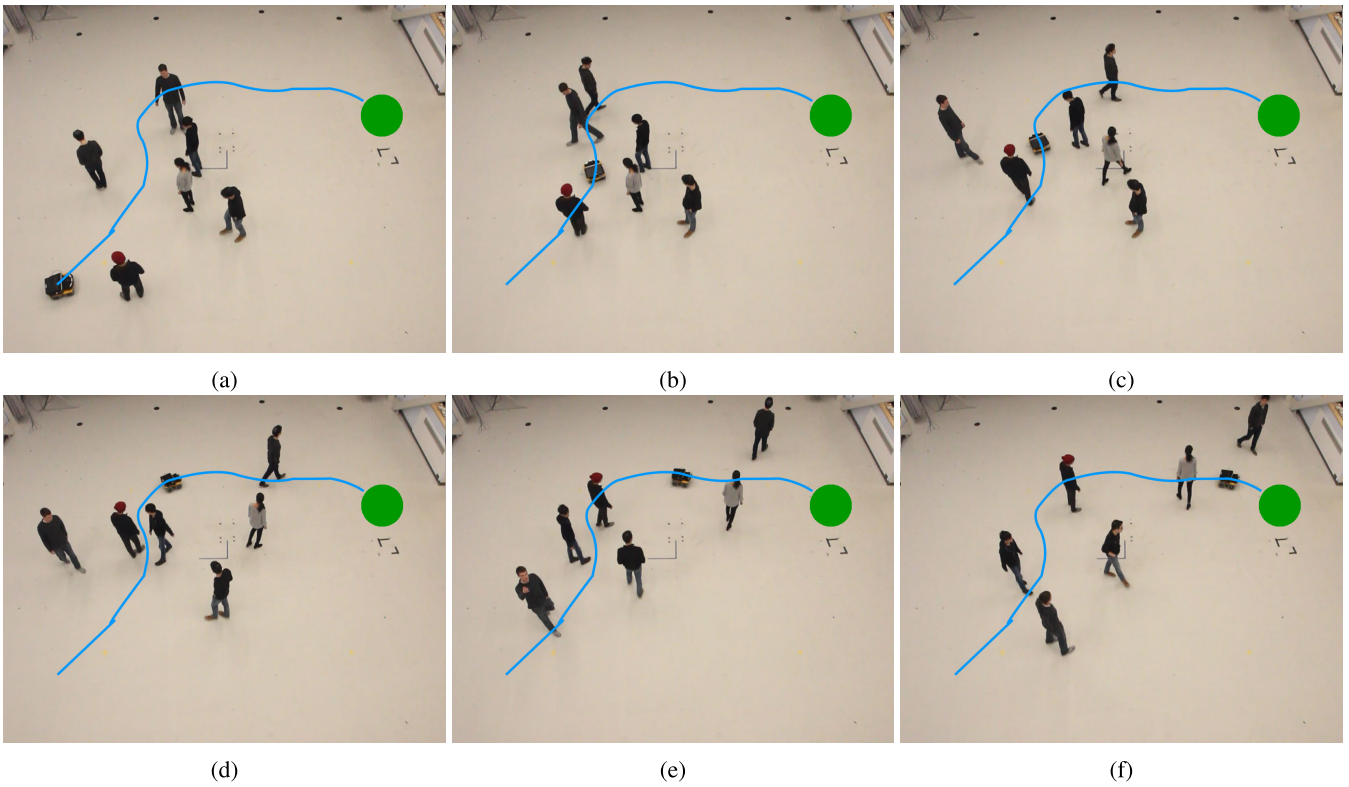


FIGURE 14. Ground robot among pedestrians. The on-board sensors are used to estimate pedestrian positions, velocities, and radii. An on-board controller tracks the GA3C-CADRL-10-LSTM policy output. The vehicle moves at human walking speed (1.2m/s), nominally.

where $W_i = [W_{i,h},\ W_{i,s}]$, $W_{i,b} = \text{diag}(b_i)$, and $n_h = 64$ is the hidden state size.

Thus, $\mathbf{i}_t = [1]^{n_h}$ when the entire *new* candidate cell state, $\tilde{C}_t$, should be used to update the cell state, C_t,

$$\tilde{C}_t = \tanh([W_{C,s}, W_{C,h}, W_{C,b}]^T \cdot [\tilde{\mathbf{s}}^o_{j,t}, \mathbf{h}_{t-1}, b_t]) \quad (15)$$

$$C_t = \mathbf{f}_t * C_{t-1} + \mathbf{i}_t * \tilde{C}_t, \quad (16)$$

where $\mathbf{f}_t$ is the value of the forget gate, computed analgously to $\mathbf{i}_t$. In other words, $\mathbf{i}_t$ with elements near 1 means that agent j is particularly important in the context of agents $[1, \ldots, j-1]$, and it will have a large impact on C_t. Contrarily, $\mathbf{i}_t$ with elements near 0 means very little of the observation about agent j will be added to the hidden state and will have little impact on the downstream decision-making.

Because $\mathbf{i}_t$ is a 64-element vector, we must make some manipulations to visualize it. First, we separate $\mathbf{i}_t$ into quantities that measure how much it is affected by each component, $\{\tilde{\mathbf{s}}^o_{j,t}, \mathbf{h}_{t-1}, b_i\}$:

$$\tilde{i}_{t,s} = ||\mathbf{i}_t - \sigma([W_{i,h}, W_{i,b}]^T \cdot [\mathbf{h}_t, b_i])||_2 \quad (17)$$

$$\tilde{i}_{t,h} = ||\mathbf{i}_t - \sigma([W_{i,s}, W_{i,b}]^T \cdot [\tilde{\mathbf{s}}^o_{j,t}, b_i])||_2 \quad (18)$$

$$\tilde{i}_{t,b} = ||\mathbf{i}_t - \sigma([W_{i,s}, W_{i,h}]^T \cdot [\tilde{\mathbf{s}}^o_{j,t}, \mathbf{h}_t])||_2 \quad (19)$$

$$\bar{\mathbf{i}}_t = \begin{bmatrix} i_{t,s} \\ i_{t,h} \\ i_{t,b} \end{bmatrix} = k \cdot \begin{bmatrix} \tilde{i}_{t,s} \\ \tilde{i}_{t,h} \\ \tilde{i}_{t,b} \end{bmatrix} \quad (20)$$

$$k = \frac{||\mathbf{i}_t||_1}{\tilde{i}_{t,s} + \tilde{i}_{t,h} + \tilde{i}_{t,b}}, \quad (21)$$

where the constant, k, normalizes the sum of the three components contributing to $\bar{\mathbf{i}}_t$, and scales each by the average of all elements in $\mathbf{i}_t$.

An example scenario is shown in Fig. 15. A randomly generated 7-agent scenario is depicted on the left column, where the ego agent is at $(-12,\ 0)$, and its goal is the star at $(0,\ 0)$. The 6 other agents in the neighborhood are added to the LSTM in order of furthest distance to the ego agent, so the tick marks on the x-axis of the right-hand figures correspond to each neighboring agent. That is, the agent at $(-5,\ 3)$ is furthest from the ego agent, so it is agent $j = 0$, and the agent at $(-10,\ 0)$ is closest and is agent $j = 5$.

For this scenario, $\bar{\mathbf{i}}_t$ (top of the stack of three slices) starts about 0.3, and goes up and down (though trending slightly upward) as agents are added. The bottom slice corresponds to $i_{t,s}$, middle to $i_{t,h}$, and top to $i_{t,b}$.

The top and middle slices are tiny compared to the bottom slice for agent 0. This corresponds to the fact that, for $j = 0$, all information about whether that agent is relevant for decision-making is in $\tilde{\mathbf{s}}^o_{0,0}$ (bottom), since the hidden and cell states are initially blank ($h_{-1} = \mathbf{0}$). As more agents are added, the LSTM considers *both* the hidden state and current observation to decide how much of the candidate cell state to pass through – this intuition matches up with the relatively larger middle slices for subsequent agents.

The importance of the contents of $\tilde{\mathbf{s}}^o_{j,t}$ is demonstrated in the bottom row of Fig. 15. It considers the same scenario as the top row, but with the closest agent's velocity vector

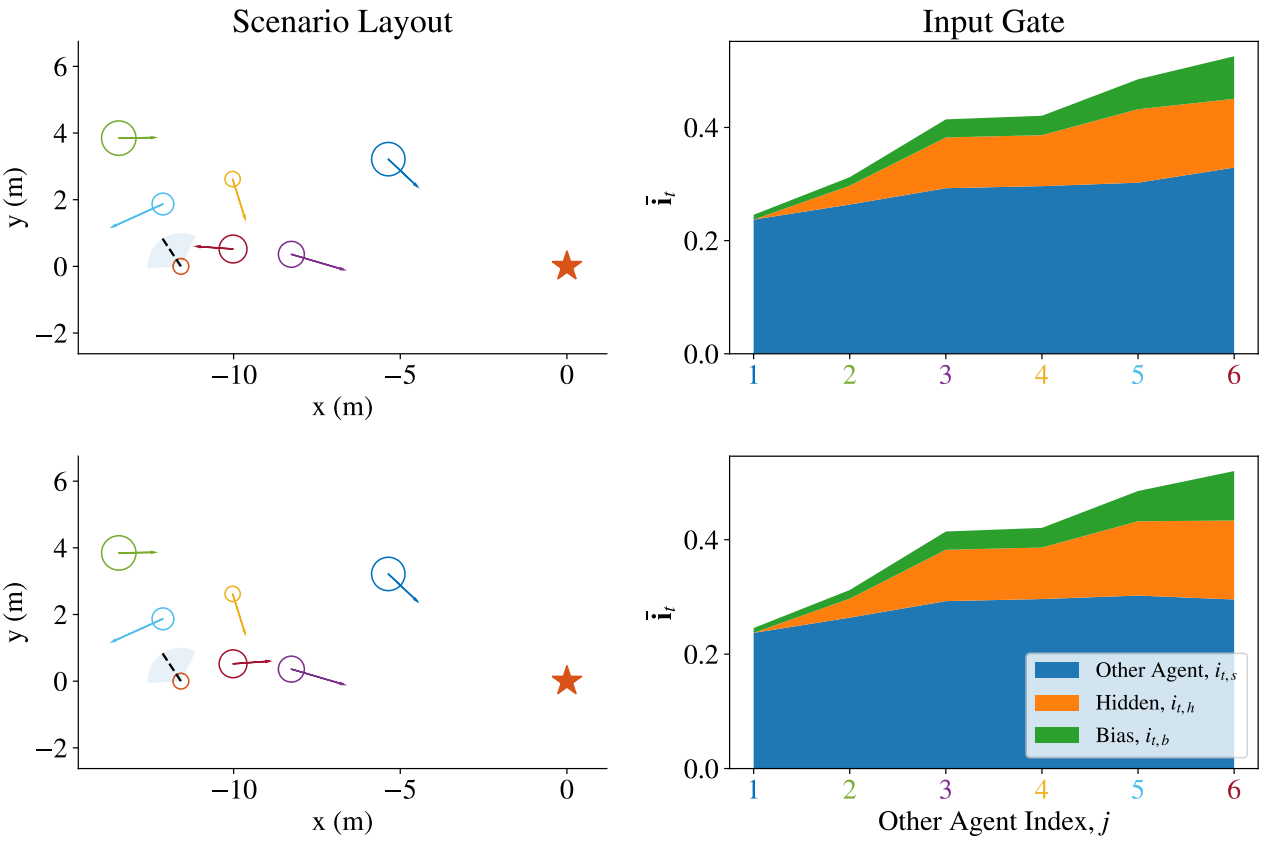


FIGURE 15. Gate Dynamics on Single Timestep. In the top row, one agent, near (-12,0), observes 6 neighboring agents. Other agent states are passed into the LSTM sequentially, starting with the furthest agent, near (-5, 3). The top right plot shows the impact of each LSTM cell input on the input gate as each agent is added to the LSTM: other agent state (bottom slice), previous hidden state (middle slice), and bias (top slice). The bottom row shows the same scenario, but the closest agent, near (-10,0), has a velocity vector away from the ego agent. Accordingly, the bottom right plot's bottom slice slightly declines from $j = 5$ to $j = 6$, but the corresponding slice increases in the top right plot. This suggests the LSTM has learned to put more emphasis on agents heading toward the ego agent, as they are more relevant for collision avoidance.

pointing away from the ego agent, instead of toward. The values of $\bar{\mathbf{i}}_t$ for all previous agents are unchanged, but the value of $\bar{\mathbf{i}}_t$ is larger when the agent is heading toward the ego agent. This is seen as an uptick between $j = 5$ and $j = 6$ in the bottom slice of the top-right figure, and a flat/slightly decreasing segment for the corresponding piece of the bottom-right figure. This observation agrees with the intuition that agents heading toward the ego agent should have a larger impact on the hidden state, and eventually on collision avoidance decision-making.

This same behavior of increased $\bar{\mathbf{i}}_t$ (specifically i_{t_s}) when the last agent was heading roughly toward the ego agent was observed in most randomly generated scenarios. Our software release will include an interactive Jupyter notebook so researchers can analyze other scenarios of interest, or do similar analysis on their networks.

2) AGENT ORDERING STRATEGIES

The preceding discussion on LSTM gate dynamics assumed agents are fed into the LSTM in the order of "closest last." However, there are many ways of ordering the agents.

Fig. 16 compares the performance throughout the training process of networks with three different ordering strategies. "Closest last" is sorted in decreasing order of agent distance to the ego agent, and "closest first" is the reverse of that. "Time to collision" is computed as the minimum time at the current velocities for two agents to collide and is often infinite when agents are not heading toward one another. The secondary ordering criterion of "closest last" was used as a tiebreaker. In all cases, $\tilde{p}_x$ (in the ego frame) was used as a third tiebreaker to preserve symmetry.

The same network architecture, differing only in the LSTM agent ordering, was trained for 1.5M episodes in 2-4 agent scenarios (Phase 1) and 0.5M more episodes in 2-10 agent scenarios (Phase 2). All three strategies yield similar performance over the first 1M training episodes. By the end of phase 1, the "closest first" strategy performs slightly worse than the other two, which are roughly overlapping.

At the change in training phase, the "closest first" performance drops substantially, and the "time to collision" curve has a small dip. This suggests that the first training phase did not produce an LSTM that efficiently combines previous agent summaries with an additional agent for these two heuristics. On the other hand, there is no noticeable dip with the "closest last" strategy. All three strategies converge to a similar final performance.

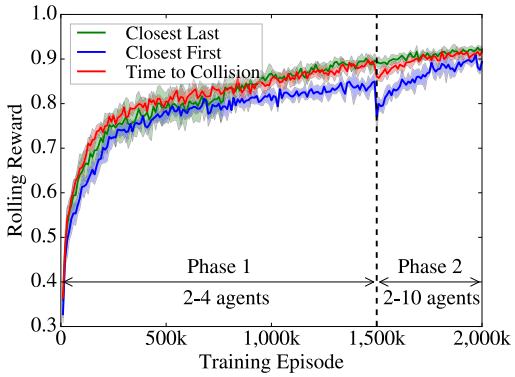


FIGURE 16. Training performance and LSTM ordering effect on training. The first phase of training uses random scenarios with 2-4 agents; the final 500k episodes use random scenarios with 2-10 agents. Three curves corresponding to three heuristics for ordering the agent sequences all converge to a similar reward after 2M episodes. The "closest last" ordering has almost no dropoff between phases and achieves the highest final performance. The "closest first" ordering drops off substantially between phases, suggesting the ordering scheme has a second-order effect on training. Curves show the mean $\pm 1\sigma$ over 5 training runs per ordering.

In conclusion, the choice of ordering has a second order effect on the reward curve during training, and the "closest last" strategy employed throughout this work was better than the tested alternatives. This evidence aligns with the intuition from Section III-B.

V. CONCLUSION

This work presented a collision avoidance algorithm, GA3C-CADRL, that is trained in simulation with deep RL without requiring any knowledge of other agents' dynamics. It also proposed a strategy to enable the algorithm to select actions based on observations of a large (possibly varying) number of nearby agents, using LSTM at the network's input. The new approach was shown to outperform a classical method, another deep RL-based method, and scales better than our previous deep RL-based method as the number of agents in the environment increased. These results support the use of LSTMs to encode a varying number of agent states into a fixed-length representation of the world. Analysis of the trained LSTM provides deeper introspection into the effect of agent observations on the hidden state vector, and quantifies the effect of agent ordering heuristics on performance throughout training. The work provided an application of the algorithm for formation control, and the algorithm was implemented on two hardware platforms: a fleet of 4 fully autonomous multirotors successfully avoided collisions across multiple scenarios, and a small ground robot was shown to navigate at human walking speed among pedestrians. Combined with the numerical comparisons to prior works, the hardware experiments provide evidence of an algorithm that exceeds the state of the art and can be deployed on real robots.

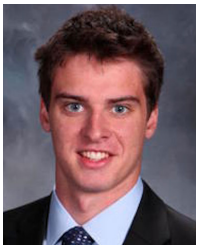
ACKNOWLEDGMENT

Yu Fan Chen was with the Massachusetts Institute of Technology (MIT), Cambridge, MA 02139, USA.

REFERENCES

[1] R. Kummerle, M. Ruhnke, B. Steder, C. Stachniss, and W. Burgard, "A navigation system for robots operating in crowded urban environments," in *Proc. IEEE Int. Conf. Robot. Autom.*, May 2013, pp. 3225–3232.
[2] P. Trautman and A. Krause, "Unfreezing the robot: Navigation in dense, interacting crowds," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst.*, Oct. 2010, pp. 797–803.
[3] J. Snape, J. V. D. Berg, S. J. Guy, and D. Manocha, "The hybrid reciprocal velocity obstacle," *IEEE Trans. Robot.*, vol. 27, no. 4, pp. 696–706, Aug. 2011.
[4] G. Ferrer, A. Garrell, and A. Sanfeliu, "Social-aware robot navigation in urban environments," in *Proc. Eur. Conf. Mobile Robots*, Sep. 2013, pp. 331–336.
[5] J. Van den Berg, S. J. Guy, M. Lin, and D. Manocha, "Reciprocal n-body collision avoidance," in *Robotics Research* (Springer Tracts in Advanced Robotics), no. 70. Berlin, Germany: Springer, 2011, pp. 3–19.
[6] J. Alonso-Mora, A. Breitenmoser, M. Rufli, P. Beardsley, and R. Siegwart, "Optimal reciprocal collision avoidance for multiple non-holonomic robots," in *Distributed Autonomous Robotic Systems*. Berlin, Germany: Springer, 2013, pp. 203–216.
[7] H. Kretzschmar, M. Spies, C. Sprunk, and W. Burgard, "Socially compliant mobile robot navigation via inverse reinforcement learning," *Int. J. Robot. Res.*, vol. 35, pp. 1289–1307, Jan. 2016.
[8] P. Trautman, J. Ma, R. M. Murray, and A. Krause, "Robot navigation in dense human crowds: The case for cooperation," in *Proc. IEEE Int. Conf. Robot. Autom.*, May 2013, pp. 2153–2160.
[9] M. Kuderer, H. Kretzschmar, C. Sprunk, and W. Burgard, "Feature-based prediction of trajectories for socially compliant navigation," in *Robotics, Science and Systems*, 2012. [Online]. Available: http://www.roboticsproceedings.org/rss08/p25.html
[10] P. Long, T. Fanl, X. Liao, W. Liu, H. Zhang, and J. Pan, "Towards optimally decentralized multi-robot collision avoidance via deep reinforcement learning," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, May 2018, pp. 6252–6259.
[11] L. Tai, G. Paolo, and M. Liu, "Virtual-to-real deep reinforcement learning: Continuous control of mobile robots for mapless navigation," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, Sep. 2017, pp. 31–36.
[12] I. Sutskever, O. Vinyals, and Q. V. Le, "Sequence to sequence learning with neural networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2014, pp. 3104–3112.
[13] K. Cho, B. van Merrienboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using RNN encoder-decoder for statistical machine translation," 2014, *arXiv:1406.1078*. [Online]. Available: http://arxiv.org/abs/1406.1078
[14] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.
[15] Y. F. Chen, M. Liu, M. Everett, and J. P. How, "Decentralized non-communicating multiagent collision avoidance with deep reinforcement learning," in *Proc. IEEE Int. Conf. Robot. Autom. (ICRA)*, May 2017, pp. 285–292.
[16] Y. F. Chen, M. Everett, M. Liu, and J. P. How, "Socially aware motion planning with deep reinforcement learning," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, Sep. 2017, pp. 1343–1350.
[17] M. Everett, Y. F. Chen, and J. P. How, "Motion planning among dynamic, decision-making agents with deep reinforcement learning," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, Oct. 2018, pp. 3052–3059.
[18] J. B. Rawlings, "Tutorial overview of model predictive control," *IEEE Control Syst.*, vol. 20, no. 3, pp. 38–52, Jun. 2000.
[19] D. Fox, W. Burgard, and S. Thrun, "The dynamic window approach to collision avoidance," *IEEE Robot. Autom. Mag.*, vol. 4, no. 1, pp. 23–33, Mar. 1997.
[20] M. Phillips and M. Likhachev, "SIPP: Safe interval path planning for dynamic environments," in *Proc. IEEE Int. Conf. Robot. Autom.*, May 2011, pp. 5628–5635.
[21] G. S. Aoude, B. D. Luders, J. M. Joseph, N. Roy, and J. P. How, "Probabilistically safe motion planning to avoid dynamic obstacles with uncertain motion patterns," *Auto. Robots*, vol. 35, no. 1, pp. 51–76, May 2013.
[22] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*, 1st ed. Cambridge, MA, USA: MIT Press, 1998.
[23] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015.

[24] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *Proc. Int. Conf. Mach. Learn.*, 2016, pp. 1928–1937.
[25] M. Babaeizadeh, I. Frosio, S. Tyree, J. Clemons, and J. Kautz, "Reinforcement learning thorugh asynchronous advantage actor-critic on a GPU," in *Proc. ICLR*, 2017, pp. 1–12.
[26] S. Omidshafiei, D.-K. Kim, J. Pazis, and J. P. How, "Crossmodal attentive skill learner," 2017, *arXiv:1711.10314*. [Online]. Available: http://arxiv.org/abs/1711.10314
[27] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," 2017, *arXiv:1707.06347*. [Online]. Available: http://arxiv.org/abs/1707.06347
[28] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," 2018, *arXiv:1802.09477*. [Online]. Available: http://arxiv.org/abs/1802.09477
[29] M. Hessel, J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver, "Rainbow: Combining improvements in deep reinforcement learning," in *Proc. 32nd AAAI Conf. Artif. Intell.*, Jan. 2018, pp. 1–14.
[30] M. Bojarski, D. Del Testa, D. Dworakowski, B. Firner, B. Flepp, P. Goyal, L. D. Jackel, M. Monfort, U. Muller, J. Zhang, X. Zhang, J. Zhao, and K. Zieba, "End to end learning for self-driving cars," 2016, *arXiv:1604.07316*. [Online]. Available: http://arxiv.org/abs/1604.07316
[31] B. Kim and J. Pineau, "Socially adaptive path planning in human environments using inverse reinforcement learning," *Int. J. Social Robot.*, vol. 8, no. 1, pp. 51–66, Jun. 2015.
[32] M. Pfeiffer, U. Schwesinger, H. Sommer, E. Galceran, and R. Siegwart, "Predicting actions to act predictably: Cooperative partial motion planning with maximum entropy models," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, Oct. 2016, pp. 2096–2101.
[33] B. Baker, I. Kanitscheider, T. Markov, Y. Wu, G. Powell, B. McGrew, and I. Mordatch, "Emergent tool use from multi-agent autocurricula," in *Proc. Int. Conf. Learn. Represent.*, 2020. [Online]. Available: https://openreview.net/group?id=ICLR.cc/2020/Conference
[34] J. García and F. Fernández, "A comprehensive survey on safe reinforcement learning," *J. Mach. Learn. Res.*, vol. 16, no. 1, pp. 1437–1480, 2015.
[35] A. Alahi, K. Goel, V. Ramanathan, A. Robicquet, L. Fei-Fei, and S. Savarese, "Social LSTM: Human trajectory prediction in crowded spaces," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 961–971.
[36] C. Olah, "Understanding LSTM networks," in *COURSERA, Neural Networks for Machine Learning*, 2015. [Online]. Available: https://colah.github.io/posts/2015-08-Understanding-LSTMs/
[37] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, and M. Kudlur, "TensorFlow: A system for large-scale machine learning," in *Proc. OSDI*, vol. 16, 2016, pp. 265–283.
[38] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014, *arXiv:1412.6980*. [Online]. Available: http://arxiv.org/abs/1412.6980
[39] U. Lau. (2019). *Rl-Collision-Avoidance*. Accessed: Sep. 10, 2019. [Online]. Available: https://github.com/Acmece/rl-collision-avoidance
[40] Intel. (2019). *Intel Drones Light Up the Sky*. Accessed: Sep. 4, 2019. [Online]. Available: https://www.intel.com/content/www/us/en/technology-innovation/aerial-technology-light-show.html
[41] Airbus. (2019). *Airbus Commercial Aircraft Formation Flight: 50-Year Anniversary*. Accessed: Sep. 4, 2019. [Online]. Available: https://www.youtube.com/watch?v=JS6w-DXiZpk
[42] H. Kitano, M. Asada, Y. Kuniyoshi, I. Noda, and E. Osawa, "Robocup: The robot world cup initiative," in *Proc. 1st Int. Conf. Auto. Agents*, 1997, pp. 340–347.
[43] Pixar. (2003). *Finding Nemo (School of Fish Scene)*. Accessed: Sep. 4, 2019. [Online]. Available: https://www.youtube.com/watch?v=Le13by2WM70
[44] S. Omidshafiei, A.-A. Agha-Mohammadi, Y. F. Chen, N. K. Üre, J. P. How, J. L. Vian, and R. Surati, "MAR-CPS: Measurable augmented reality for prototyping cyber-physical systems," in *Proc. AIAA Infotech Aerosp.*, Jan. 2015, p. 643.
[45] T. Campbell, M. Liu, B. Kulis, J. P. How, and L. Carin, "Dynamic clustering via asymptotics of the dependent Dirichlet process mixture," in *Proc. Adv. Neural Inf. Process. Syst.*, 2013, pp. 449–457.
[46] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "SSD: Single shot multibox detector," in *Proc. Eur. Conf. Comput. Vis.* Cham, Switzerland: Springer, 2016, pp. 21–37.
[47] J. Miller, A. Hasfura, S.-Y. Liu, and J. P. How, "Dynamic arrival rate estimation for campus mobility on demand network graphs," in *Proc. IEEE/RSJ Int. Conf. Intell. Robots Syst. (IROS)*, Oct. 2016, pp. 2285–2292.
[48] M. Everett, "Robot designed for socially acceptable navigation," M.S. thesis, Dept. Mech. Eng., MIT, Cambridge, MA, USA, Jun. 2017.



MICHAEL EVERETT received the S.B., S.M., and Ph.D. degrees in mechanical engineering from MIT, in 2015, 2017, and 2020, respectively. He is currently a Postdoctoral Associate with the Department of Aeronautics and Astronautics, MIT, and conducts research with the Aerospace Controls Laboratory. His research interests include fundamental gaps in the connection of machine learning and real mobile robotics, with recent emphasis on developing the theory of safety/robustness of learned modules. He was the author of works that won the Best Paper Award on Cognitive Robotics at IROS 2019, the Best Student Paper Award and a Finalist for the Best Paper Award on Cognitive Robotics at IROS 2017, and a Finalist for the Best Multi-Robot Systems Paper Award at ICRA 2017. He has been interviewed live on the air by BBC Radio and his team's robots were featured by Today Show and the Boston Globe.

YU FAN (STEVEN) CHEN received the B.A.Sc. degree from the University of Toronto, in 2012, and the S.M. and Ph.D. degrees in aeronautics and astronautics from MIT, in 2014 and 2017, respectively. He is currently the Research Scientist of Facebook Reality Labs, formerly known as Oculus Research. His research interest includes perception and decision-making for robotics and augmented reality applications. His current research interest includes self-supervised learning by aggregating information from multiple views and enforcing geometric consistency. His earlier works on multi-agent collision avoidance have won the Best Student Paper Award at IROS 2017 and a Finalist for the Best Multi-Robot Systems Paper Award at ICRA 2017.

JONATHAN P. HOW (Fellow, IEEE) received the B.A.Sc. degree in aerospace from the University of Toronto, in 1987, and the S.M. and Ph.D. degrees in aeronautics and astronautics from the Massachusetts Institute of Technology (MIT), in 1990 and 1993, respectively. He then studied for 1.5 years at MIT as a Postdoctoral Associate. Prior to joining MIT in 2000, he was an Assistant Professor with the Department of Aeronautics and Astronautics, Stanford University. He is currently the Richard C. Maclaurin Professor of aeronautics and astronautics with MIT. His research interests include robust planning and learning under uncertainty with an emphasis on multiagent systems. He is a Fellow of AIAA. He was elected to the Board of Governors of the IEEE Control System Society (CSS) in 2019 and is a member of the IEEE CSS Technical Committee on Aerospace Control and the Technical Committee on Intelligent Control. He is also the Director of the Ford-MIT Alliance and was a member of the USAF Scientific Advisory Board (SAB) from 2014 to 2017. He was the Planning and Control Lead of the MIT DARPA Urban Challenge Team. His work has been recognized with multiple awards, including the 2020 AIAA Intelligent Systems Award. He was the Area Chair of International Joint Conference on Artificial Intelligence in 2019 and will be the Program Vice-Chair (tutorials) of the Conference on Decision and Control in 2021. He was the Editor-in-Chief of *IEEE Control Systems Magazine* from 2015 to 2019. He is also an Associate Editor of the *Journal of Aerospace Information Systems* (AIAA) and the IEEE TRANSACTIONS ON NEURAL NETWORKS AND LEARNING SYSTEMS.