Multi-Robot Navigation: Guided Cost Learning in Multi-Agent Dynamic Games

Varun Vejalla, John Zoscak, Kyle McDonald

***Abstract*— In this paper[1], we aim to learn the behavior of multiple boundedly rational stochastic agents interacting in a dynamic game. In the setting of this game, we create a new model that learns a control function, cost function, a private cost embedding for each agent, and the dynamics of the system. We demonstrate the effectiveness of our algorithm in a simulated multi-agent collision avoidance scenario. We show that the learned functions lead to reasonable trajectories and controls.**

I. Introduction

Understanding the behavior of intelligent agents interacting within a shared dynamic environment is a central problem in robotics and autonomous systems. Multi-agent dynamic games provide a principled framework for modeling such interactions by defining agents' decision-making processes through cost functions and solution concepts such as Nash or Quantal Response Equilibria. Traditional formulations of these games assume known cost functions, enabling forward solutions to predict agent trajectories. However, in many real-world applications, these cost functions are not directly observable and must instead be inferred from behavioral data.

A. Problem statement

We frame our problem in the following way. We represent the state x_t of the whole system at time t as an element of $\mathbb{R}^n$, and assume there are k agents. At each timestep, each agent i chooses a control u_t^i from the action space $\mathbb{R}^{d_i}$. Write the full action as $\mathbf{u}_t = (u_t^1, \cdots, u_t^k)$. The update is then of the form

$$x_{t+1} = f(x_t, \mathbf{u}_t) + \omega_t$$

where ω_t is a noise term.

This is the general formulation of the problem. However, we impose additional structure on it. We suppose each agent has a state $x_t^i \in \mathbb{R}^m$ associated with it, and that the total state x_t is $(x_t^1, \cdots, x_t^k)$. Note that this implies $n = mk$. We also assume a cost function exists describing how agents act. Agents would act to reduce this cost function. Moreover, each agent's cost function is parametrized by a private cost embedding.

The objective is to learn the cost function and the associated parameters for each agent.

[1]Code is publicly available at https://github.com/VarunVejalla/cost-learning-dynamic

II. Related Work

A. MaxEnt-MADG

The Maximum Entropy Multi-Agent Dynamic Games (MaxEnt-MADG) framework proposed by Mehr et al. [1] offers a compelling probabilistic approach to both forward and inverse problems in multi-agent settings. The framework extends the principle of maximum entropy to multi-agent Markov decision processes (MDPs), capturing bounded rationality and noisy decision-making. In other words, this system is capable of solving the outcomes of many noisy interactive agents by considering their entropic cost equilibrium (ECE), as an extension of game-theoretic Nash equilibrium, to have an expected value equal to the the optimal / most rational policy or control. This helps to make the system responsive to unexpected environmental changes and allows the system to consider the possibility of irrational actions in the case of less predictable agents (such as humans).

They assume agents all have quadratic cost functions, where each agent only differs in how much they weigh each cost function. Although this approach captures a wide range of functions, it would fail to capture a simple function like the distance to a goal, where the goal is also a parameter that must be learned. That is, if the cost function for each agent i was

$$\sqrt{(x - x_{goal})^2 + (y - y_{goal})^2}$$

the goal for each agent would have to be known in order to successfully use their approach.

The biggest drawback is that the cost functions must be known before we can learn the private coefficients on them. We would like to learn these functions instead. On the other hand, because this model has known quadratic cost functions, and it assumes that the system is linear, it can implement a linear controller for estimating the weights of the cost embeddings called a linear-quadratic-regulator (LQR) under the context of a Linear-Quadratic game.

Using the LQR, linear control trajectories can be generated for a number of simulations. Using this work, we can store trajectories and produce a dataset of simulated trajectories that are boundedly-rational and can be used for training neural networks.

B. Guided Cost Learning

Our multi-robot navigational system, which leverages ECE and quantal response equilibria (QRE) to model bounded-rational agents in continuous-space scenarios, draws inspiration from inverse optimal control (IOC) and inverse

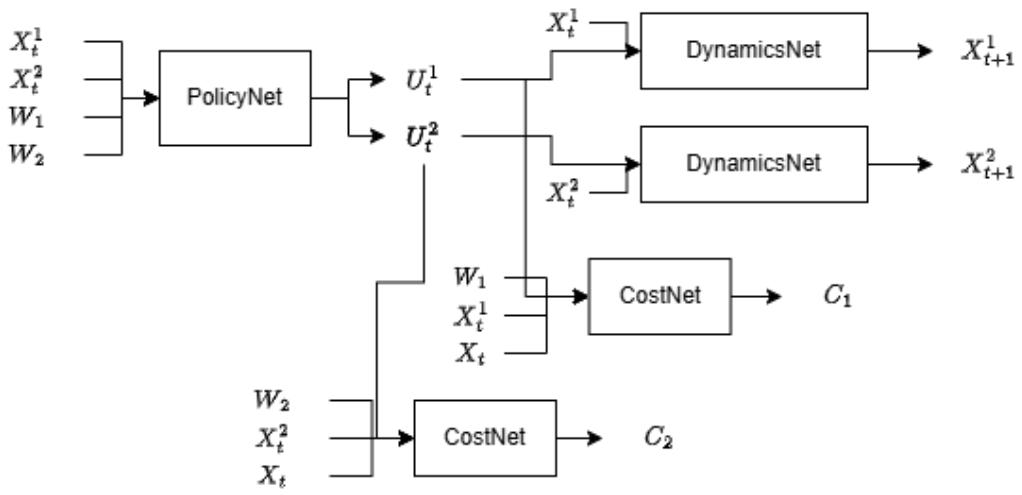

Fig. 1. Diagram of how inputs are handed to each of our networks. PolicyNet takes in the state of both agents, as well as the weights and produces controls U_t^1 and U_t^2. DynamicsNet then takes in those control outputs as well as the state for each agent to calculate X_{t+1}^1 and X_{t+1}^2. CostNet takes in the entire state, as well as the state for the agent it is estimating, the cost features for that agent, and the control output from PolicyNet in order to produce the cost functions for each agent.

reinforcement learning (IRL) frameworks that learn cost functions from demonstrations. We build on the maximum-entropy IOC model, which assumes near-optimal demonstrations with noise, extending it to high-dimensional, continuous spaces using sample-based approximations. The guided cost learning algorithm is a key influence, as it combines sample-based IOC with policy optimization to learn expressive neural network cost functions for systems with unknown dynamics, a capability we adapt by training cost, dynamics, and policy networks jointly for multi-agent navigation.[2] Unlike prior multi-agent IOC methods that assume deterministic behavior, our use of QRE captures noisy interactions, while our nonlinear cost representations help to eliminate manual feature engineering. Additionally, we incorporate joint policy and dynamics learning and employ regularization techniques inspired by QRE's stochasticity.

III. Methods

We have three models that all train in parallel along with learning the private cost features:

- A policy network (PolicyNet) takes in the state and cost embeddings of all the agents, and outputs the actions that each agent will take. We chose this design as it allows the model to learn interactive behaviors between the agents that it observes from interactive trajectories.
- A cost network (CostNet) takes in the ego state, full system state, actions, and cost embeddings, and outputs a cost. *Note: this network is the same for all agents, but the input cost embeddings will differ*
- A dynamics network (DynamicsNet) takes in the state and actions, and outputs a new state. This model tries to understand what the actions actually mean and how they change the state. *Note: Similar to the CostNet, this network is the same for all agents, but the input cost embeddings will differ.*

These models are simply chained together. The input is the state and cost embeddings, and the output is the new state.

There are multiple associated loss functions.

- The dynamics loss is the difference between the predicted state (given the previous state and actions) and the true state.
- The acceleration loss is the sum of the acceleration's magnitude. This is needed so that DynamicsNet doesn't learn unrealistic dynamics.
- The embedding loss is the difference between the magnitude of the embeddings and 1. This makes it so that the embeddings lie on or near the unit hypersphere.
- The entropic cost equilibrium loss incorporates work from [1]. We take the cost and add a term corresponding to the log of the determinant of the Hessian of the cost function (which is given by CostNet). This additional term encourages robust equilibria - changing the action by a small amount shouldn't change the cost much either.

For computational reasons, instead of using the determinant of the Hessian, we use the product of the diagonals of the Hessian. This is a significantly faster calculation, while preserving most of the relevant information.

The total cost function is simply a linear combination of these terms. We backpropagate through the network in order to update the network's parameters in the standard way.

IV. Experiments

In the original MaxEnt-MADG implementations, we were able to validate the prior work by running the MA-IRL algortithm implementations. In the software, the LQR software was validated. Drawing from the linear quadratic game software, we informed our own implementation of the linear

quadratic game, and determined appropriate goal weight vectors for specified cost functions.

A. Dataset Generation

We used the algorithm from [1] in order to simulate trajectories for two agents. We created a true cost function and many random initial states. Using the algorithm, the generated trajectories for each of the agents acts in accordance with their given cost functions. We then split these trajectories into training and testing sets.

The cost function and distribution of initial states was chosen so that agents had a high likelihood of navigating around each other. A sample trajectory is shown in Fig. 2.

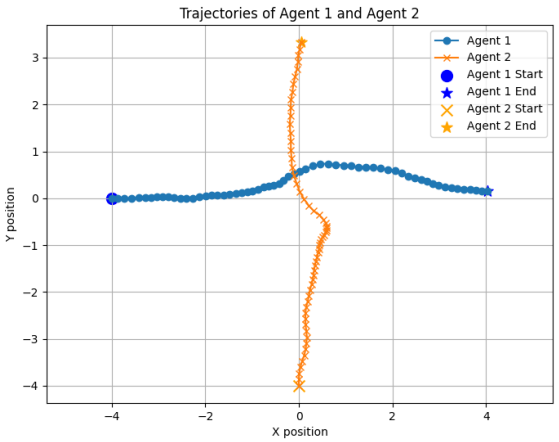

Fig. 2. Example trajectory used in training.

In order to produce enough data for our model to train on we initially produced five hundred simulated trajectories, shown below in Fig. 3.

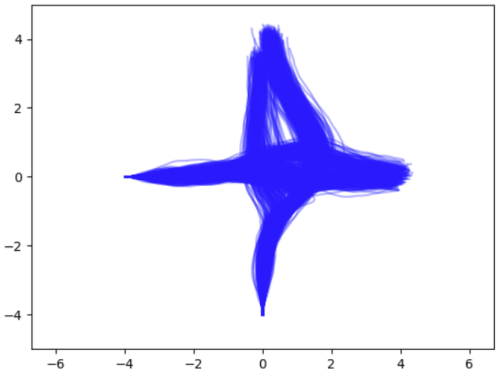

Fig. 3. 500 simulated trajectories using MaxEnt-MADG

B. Costs for Linear Quadratic Game:

When testing the algorithms in Python, we selected several cost functions (for a 2 agent scenario) that can be used for benchmarking against our solution:

- **Cost Function 1**: Measures the distance of the first agent's position ($\mathbf{p}_0 = [x_0, y_0]$) to its goal ($\mathbf{g}_0$) as $\sqrt{\|\mathbf{p}_0 - \mathbf{g}_0\|_2^2 + 10^{-3}}$.
- **Cost Function 2**: Encourages the first agent to stay away from the second agent by computing the inverse distance $\frac{1}{\|\mathbf{p}_0 - \mathbf{p}_1\|_2 + 10^{-6}}$, where $\mathbf{p}_0 = [x_0, y_0]$ and $\mathbf{p}_1 = [x_1, y_1]$.
- **Cost Function 3**: Penalizes the first agent's acceleration ($\mathbf{a}_0 = [a_{x0}, a_{y0}]$) as $\sqrt{\sum a_{0i}^2 + 10^{-6}}$, returning 0 if no action is provided.
- **Cost Function 4**: Measures the distance of the second agent's position ($\mathbf{p}_1 = [x_1, y_1]$) to its goal ($\mathbf{g}_1$) as $\sqrt{\|\mathbf{p}_1 - \mathbf{g}_1\|_2^2 + 10^{-3}}$.
- **Cost Function 5**: Encourages the second agent to stay away from the first agent by computing the inverse distance $\frac{1}{\|\mathbf{p}_1 - \mathbf{p}_0\|_2 + 10^{-6}}$.
- **Cost Function 6**: Penalizes the second agent's acceleration ($\mathbf{a}_1 = [a_{x1}, a_{y1}]$) as $\sqrt{\sum a_{1i}^2 + 10^{-6}}$, returning 0 if no action is provided.

C. Training the Cost, Policy and Dynamics Network:

Using the two methods described above IV-A IV-B, we were able to produce our dataset and then train our cost, policy, and dynamics network by using loss described in III. We found that the most correlated component of our system was the Dynamics network, and thus selected weights that reduced the contributions of the other component's weights to improve the qualitative performance of the network.

V. Results

Unfortunately, due to time constraints, our current results should be interpreted as qualitative rather than quantitative. While they demonstrate promising trends and validate the feasibility of our approach, we were not able to conduct the extensive hyperparameter tuning, large-scale ablation studies, or statistical evaluations typically required for robust quantitative analysis. As such, our findings serve as a proof of concept and a foundation for more comprehensive empirical evaluation in future work.

We used the following weights (for the training loss functions):

- Dynamics weight: 1.0
- Entropy weight: 0.1
- ECE weight: 0.1
- Acceleration weight: 0.1
- Embeddings weight: 0.1

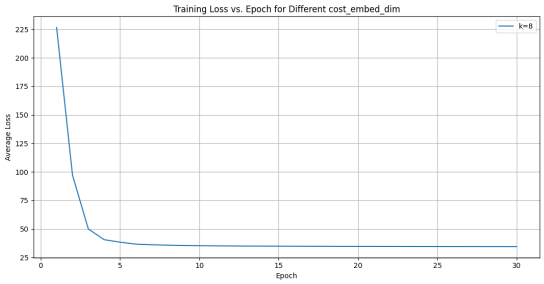

Fig. 4. Loss function over number of epochs.

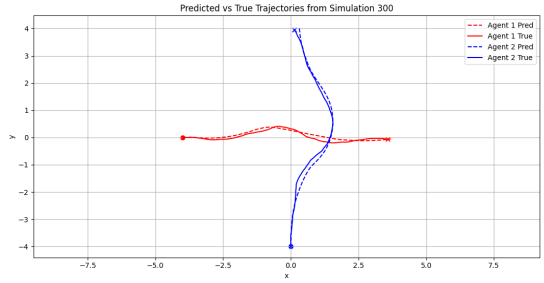

Fig. 5. Example predicted trajectory vs true trajectory.

Looking at the predicted trajectory and the true trajectory pictured in 5, the predicted trajectory tracks closely to the true trajectory when using the weights above in the training process. These qualitative results indicate that the use of ECE loss and other features seems to be less important for the training of a system of MLPs than simply the ability to propagate from the current state to the next state accurately. However, without time to do ablations and perform further testing of what may have caused these features to not contribute, we cannot rule out the potential benefit of considering these loss measurements in the training of our system.

VI. Discussion

Implementing algorithms from the MaxEnt-MADG literature posed several technical and practical challenges throughout our research. While the theoretical foundations of these algorithms are well-established, translating them into a working codebase, particularly in a multi-agent and dynamic context, proved to be significantly more difficult than anticipated.

One of the primary difficulties stemmed from ambiguities or missing implementation details in the original papers. The original codebase for MaxEnt-MADG was written in Julia and has native support for multiple dispatch and differentiable programming which allowed for the original implementation to be concise and highly performant, especially for custom dynamics and control logic. In contrast, translating this functionality into Python required careful re-engineering using frameworks like PyTorch, which came with its own paradigms and constraints. This often meant rewriting core components to fit the expectations of these libraries, which was not always straightforward.

Despite these challenges, converting the codebase to Python ultimately enabled smoother integration of neural networks into the original maximum-entropy framework, leveraging Python's rich ecosystem of deep learning libraries.

There are three main benefits to our system and setup as opposed to prior work:

1) Cost functions can be learned.
2) Different components can be removed/manually implemented. If the dynamics are known, we can plug in a known dynamics network instead of learning it.
3) Once the cost function is learned, we can learn embeddings online for different agents. We can freeze all the networks and only update the embeddings.

VII. Conclusion

This project aimed to model boundlessly rational stochastic agents in multi-agent dynamic games but faced significant hurdles. While we developed a model to learn policy, cost, and dynamics functions alongside private cost embeddings, the results in our simulated collision avoidance scenario were only marginally satisfactory. Inferring unobservable cost functions proved challenging, with learned trajectories and controls often deviating from optimal behavior. The integration of Quantal Response Equilibria and entropic cost equilibrium losses, intended to capture noisy decision-making, added a complexity that made it harder to learn what actions were taken. Translating the MaxEnt-MADG framework from Julia to Python also came with many difficulties, as ambiguities in prior work and missing implementation details led to persistent integration issues. Our dataset of five hundred simulated trajectories, while useful, exposed limitations in generalizing to diverse scenarios. The model's performance, driven by cautiously tuned loss weights, underscored its fragility and limited scalability. Moving forward, substantial improvements are needed to handle more agents, refine cost representations, and overcome computational inefficiencies.

References

[1] N. Mehr, M. Wang, M. Bhatt, and M. Schwager, "Maximum-entropy multi-agent dynamic games: Forward and inverse solutions," *IEEE Transactions on Robotics*, vol. 39, no. 3, pp. 1801–1815, 2023.

[2] C. Finn, S. Levine, and P. Abbeel, "Guided cost learning: Deep inverse optimal control via policy optimization," *CoRR*, vol. abs/1603.00448, 2016. [Online]. Available: http://arxiv.org/abs/1603.00448