Learning to Fail: Failure Plans and Predictions for Crowd Navigation

Patrick Naughton[1], Ishani Chatterjee[2], Tushar Kusnur[3], and Maxim Likhachev[4]

***Abstract*— When navigating in the presence of humans, mobile robots need robust, efficient ways of dealing with highly stochastic dynamic obstacles. This problem is often addressed by building a (implicit or explicit) model of possible human trajectories and planning the robot's motion based on this model to avoid collisions. We call this a *success controller*. However, such controllers typically assume that their goal is feasible, without considering cases in which humans may move to block the robot's path. This can lead to the robot simply stopping, continuously replanning to an infeasible goal, or, in the worst case, colliding with a pedestrian. This work seeks to explicitly address this deficiency by creating a classifier that examines the robot's trajectory to determine whether or not it believes the robot can efficiently achieve its goal point. Additionally, we develop a *failure controller* that the robot executes if the classifier determines that it cannot reach its goal. This controller safely guides the robot to a different location from which it can replan. We show that this failure controller results in fewer intrusions near people in a crowded scene than simply relying on the success controller.**

I. INTRODUCTION

The contributions of this work are:

1) A classifier that determines whether or not a robot is likely to reach its goal in a given scene.
2) A failure controller that safely guides the robot to a stable position from which it can initiate replanning.

II. RELATED WORK

Several methods use explicit models of human behavior to achieve smooth, predictable robot navigation among pedestrians. Trautman et al. model the interaction between the robot and pedestrians as an extension of an interactive Gaussian process that accommodates multiple goals [1]. The social forces model treats humans and the robot in question as masses subject to Newtonian dynamics and applies fictitious forces to them to predict and plan trajectories [2]. It recomputes these forces and their effects on robot motion at each time step to determine how the robot should move. These techniques however rely on hand-crafted models of human behavior to achieve their results and handle unexpected or uncooperative human actions by simply replanning using the same model. The social forces model in particular does not demonstrate robust navigation plans and will sometimes exhibit oscillatory behavior in more crowded or narrow areas [2].

Another approach uses inverse reinforcement learning to learn latent, possibly stochastic social rules humans observe when navigating in crowds [3]. This method uses example trajectories recorded from humans or gathered from teleoperated runs. This approach however is extremely unlikely to observe failed trajectories where a human attempts to execute some navigation plan and is forced to completely abort their initial goal. If a human attempts to overtake someone else, for example, they have many contingency options in the case where the other person is either intentionally or unintentionally uncooperative. For example, they could use verbal communication or body language to more explicitly communicate their intentions, options which are not available to many mobile robots. For this reason, inverse reinforcement learning will likely be unable to formulate a useful model for navigation when situations such as these occur.

Reinforcement learning has successfully been applied to the social robot navigation problem using a variety of different models [4], [5]. Reinforcement learning is particularly suited to this application as noted in [4] because it is extremely difficult to specify what the optimal action for a robot to take is, but it is comparatively easy to alert the robot when it performs a socially unacceptable or unsafe action. Previous work has focused on using reinforcement learning to develop policies that generate optimal (in terms of time) paths to a robot's goal in the presence of humans or other autonomous agents. These policies however generally assume the goal is reachable and do not make contingency plans if that assumption turns out to be incorrect. Additionally, the agent is explicitly given a goal to reach by the experimenters; we wish to navigate in the case of failure at which point there is no obvious goal.

The above methods all either deal with failure at execution time by simply replanning or do not consider failure to reach the goal at all. We depart from this paradigm by designing a controller specifically targeted at producing trajectories when the robot's original goal is no longer reachable.

III. PROBLEM STATEMENT

We consider a robot that strings together different motion primitives and controllers to generate a navigation policy to reach some overall goal. Motion primitives are basic actions the robot can take which are guaranteed to succeed, for example, drive forward one meter. Controllers are more complicated actions that may fail, for example, barging past a group of pedestrians. These controllers can be used by the robot in specific situations to navigate in a scene. We refer to the controller that guides the robot a *success controller*. This work is concerned with detecting and handling the failure of these controllers. Specifically, we develop a *failure controller* that corresponds to a given success controller. This failure

Some text
[1]Notes patrickrnaughton@wustl.edu
[2]Ishani notes ichatter@andrew.cmu.edu

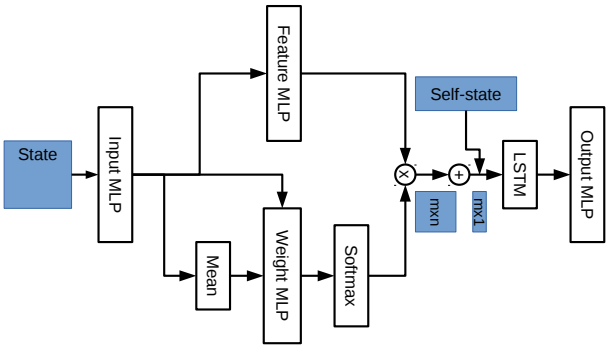

Fig. 1: Model architecture

controller directs the robot if it is determined at execution time that the success controller is unlikely to achieve its goal.

Additionally, we would like to rigorously determine at execution time whether or not the success controller is likely to succeed. In this framework, the robot can decide at each time step whether it should begin using the failure controller. Once it switches to the failure controller, it cannot switch back to the success controller until the planner generates a new plan.

IV. APPROACH

We begin with a fixed success controller. In this work, we consider one specific controller for barging into a group of people at the end of a corridor. While the experiments and results here only concern this controller, the framework can be extended to include other controllers as well, for example, to overtake or cross in front of pedestrians. In order to train this controller, we first generate example runs in which the robot can reach its goal using the RVO2 simulator [6], [7]. We employ a neural network architecture based on the one presented in [5] to allow the network to handle variable input sizes. The state of the robot is characterized by ... In addition, we include an LSTM at the end of our network so that the robot can learn the sequence of actions. Figure 1 shows the overall architecture of the network. The output of the network has five nodes which are interpreted as the x and y coordinates of the mean, the x and y standard deviations, and the correlation between x and y of a Gaussian distribution of the next location of the robot conditioned on its current state. At execution time, we then feed this network its current state and attempt to move to the mean of this distribution in the next time step.

We split the approach into two distinct parts: First, we consider just determining whether or not the robot should switch to its failure controller. Given this classifier, we then develop a failure controller that, starting from the state at which the classifier reports we have failed, attempts to learn a policy that guides the robot to a safe location from which to replan.

A. Determining When To Switch

In order to rigorously determine whether or not the success controller is likely to succeed (and thus, whether or not we should switch to the failure controller), we train an additional neural network that has the same architecture as the original. However, it attempts to learn a target function that predicts the distribution of previous locations of the robot given the robot's current state and the fact that the robot is on a trajectory that leads to its goal. It is trained on $10,000$ trajectories generated by the success controller in scenes where the success controller can reach its goal. This network however does not receive the robot's full state, rather, it only observes the robot's *self state*, that is, its ... This was done so that we could train the network by only showing it successful trajectories without the uncertainty in its prediction exploding when it observes unsuccessful ones.

At execution time, we utilize this network's prediction to perform a statistical p-test with an α value of 0.02. At each time step we construct the error ellipse given by the Gaussian distribution represented by the reverse predictor that contains $1-\alpha$ of the distribution's probability mass. By default, we assume that the trajectory will be successful and switch to the failure controller only if the robot's previous position falls outside the predicted error ellipse.

B. Failure Controller

For the failure controller, we need to address a somewhat novel formulation of the navigation problem in that the robot now has no specific goal it is trying to reach. Once the failure controller gets invoked, the robot has determined that it is unlikely to reach its goal. Because it is impossible to explicitly define optimality in this framework, we formulate this as a reinforcement learning problem $\langle \mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{P}, \gamma \rangle$ with an infinite set of states $\mathcal{S}$, a finite set of actions $\mathcal{A}$, state transition matrix $\mathcal{P}(s_t, a_t, s_{t+1})$, reward function $\mathcal{R}(s_t, a_t, s_{t+1})$ and discount factor $\gamma \in (0, 1)$ [8] and model the robot as a Markovian agent. Rather than define a set of states constituting a goal, we define a reward function that punishes actions we determine to be undesirable.

Our reward function is split into three parts: $\mathcal{R}_{\text{collision}}$, $\mathcal{R}_{\text{movement}}$, and $\mathcal{R}_{\text{smoothness}}$. The overall reward the robot observes is simply the sum of these three components. Equations 1, 2, and 2 show how these values are calculated. Note that each term is computed (where appropriate) with respect ot each agent and obstacle in the scene. This means that an action that causes a collision with two agents receives a collision reward of -2, not just -1. We apply the $\mathcal{R}_{\text{movement}}$ and $\mathcal{R}_{\text{smoothness}}$ rewards so that, other things equal, the robot will prefer to remain stationary and move in straight lines at consistent velocities that are more predictable for other humans in the scene.

$$\mathcal{R}_{\text{collision}} = \begin{cases} -1 & d < 0 \\ -0.25 & 0 \leq d < 0.2 \\ 0 & else \end{cases} \tag{1}$$

$$\mathcal{R}_{\text{movement}} = \begin{cases} 0 & a_t = 0 \quad (remain\ still) \\ -0.01 & else \end{cases} \tag{2}$$

$$\mathcal{R}_{\text{smoothness}} = \begin{cases} 0 & a_t = a_{t-1} \vee a_{t-1} = 0 \\ -0.01 & else \end{cases} \tag{3}$$

We then define a $Q_\pi(a_t, s_t)$ function that represents the total expected discounted return achieved by executing action a_t from state s_t and thereafter following policy π.

$$Q_\pi = \sum_{k=0}^{\infty} E[\gamma^k \mathcal{R}(s_{t+k}, \pi(a_{t+k} \mid s_{t+k}), s_{t+k+1})] \tag{4}$$

We represent this function with a deep neural network and use the Deep Q-learning algorithm with experience replay developed in [9] to obtain an estimate of the optimal Q function. We utilize an ϵ-greedy policy with an ϵ that linearly decays from _ to _ over _ episodes. We set $\gamma = 0.9$ and train using the Adam optimizer with learning rate 0.001.

To train our agent, we ran 1000 episodes in which the robot begins by executing its success controller. However, the humans in the scene move in such a way to prevent the robot from barging in. Namely, they move into the corridor towards the robot rather than dispersing to allow it through. Once the reverse predictor determines that the robot has failed, the robot begins executing an ϵ-greedy policy and stores transitions in its replay buffer. At each time step, the reward achieved by the robot is computed and its policy is updated. After 10 time steps, the current episode ends and another one is started. The failure controller moves for 10 time steps regardless of how long the success controller moved.

V. EXPERIMENTS

To evaluate the effectiveness of our framework, we ran two sets of experiments. In one, the reverse predictor is completely disabled so that the robot continues to execute its success controller regardless of what the people in the scene do. The episode ends after 15 time steps or after the robot reaches its goal, whichever comes first.

In the other set, the robot begins by executing its success controller and only switches to the failure controller when the p-test performed by the reverse predictor fails. If the robot executes the success controller for 15 time steps without triggering the reverse predictor's failure condition, it is cut off and the failure controller is allowed to run. If the robot reaches its goal, this episode is simply thrown out. After the failure controller initiates, it executes for 10 time steps before the episode ends.

VI. RESULTS

We used versions of the metrics presented in [10], which were designed to evaluate robot motion in crowds, to compare the two frameworks' effectiveness. These metrics are the length of the trajectory, time elapsed, angular distance traveled (computed as total change in heading between time steps), the number of collisions incurred and the number of intrusions incurred. Here, the robot is considered intruding on a human if it comes within $0.2m$ of the human. Similarly to how the reward function is computed, we count collisions (and intrusions) in each time step according to the number of humans or obstacles the robot is colliding with in that step.

VII. CONCLUSIONS

ACKNOWLEDGMENT

REFERENCES

[1] P. Trautman, J. Ma, R. Murray, and A. Krause, "Robot navigation in dense human crowds: the case for cooperation," in *2013 IEEE International Conference On Robotics and Automation*, 2013, pp. 2153–2160.

[2] G. Ferrer, A. Garrell, and A. Sanfeliu, "Social-aware robot navigation in urban environments," in *2013 European Conference on Mobile Robots*. IEEE, 2013, pp. 331–336.

[3] H. Kretzschmar, M. Spies, C. Sprunk, and W. Burgard, "Socially compliant mobile robot navigation via inverse reinforcement learning," *The International Journal of Robotics Research*, vol. 35, no. 11, pp. 1289–1307, 2016.

[4] Y. F. Chen, M. Everett, M. Liu, and J. P. How, "Socially aware motion planning with deep reinforcement learning," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*. IEEE, 2017, pp. 1343–1350.

[5] C. Chen, Y. Liu, S. Kreiss, and A. Alahi, "Crowd-robot interaction: Crowd-aware robot navigation with attention-based deep reinforcement learning," *arXiv preprint arXiv:1809.08835*, 2018.

[6] J. van den Berg, S. J. Guy, J. Snape, M. C. Lin, and D. Manocha, "Rvo2 library: Reciprocal collision avoidance for real-time multi-agent simulation," 2011.

[7] S. Stvel, "Python rvo2," https://github.com/sybrenstuvel/Python-RVO2/, 2017.

[8] R. S. Sutton and A. G. Barto, *Introduction to Reinforcement Learning*, 1st ed. Cambridge, MA, USA: MIT Press, 1998.

[9] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[10] D. Vasquez, B. Okal, and K. O. Arras, "Inverse reinforcement learning algorithms and features for robot navigation in crowds: an experimental comparison," in *2014 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2014, pp. 1341–1346.