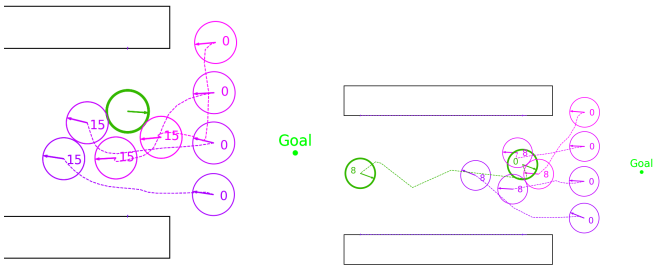
Learning When to Fail: Contingency Plans for Crowd Navigation

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

***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

Pedestrian crowds present a challenging navigation environment for a mobile robot. The robot must comply with latent social rules governing its trajectory while simultaneously reaching its goal in a reasonable amount of time. As robots move into closer contact with humans, advanced methods for navigation in crowds will gain increased importance. Probabilistic planners, which make navigation plans in the belief space, show potential to make headway in this problem and have demonstrated success dealing with uncertainty [1] like that experienced navigating in a crowd. To form their plans, these planners rely on smaller atomic actions which are strung together to form an overall trajectory. Some of these actions are learned by presenting the robot with its starting pose and a waypoint pose and running simulations with pedestrians or other agents [2]. In general, these controllers guiding each action deal with unexpected situations in which their original trajectory to their waypoint becomes infeasible by simply replanning or stopping altogether. Replanning in some cases may take too long to avoid an imminent collision. Additionally, when the waypoint becomes completely unreachable, replanning or stopping may result in a more costly trajectory than simply abandoning it and seeking a safe location from which the higher level planner can generate a new trajectory to the overall goal. We address this issue by developing a failure controller aimed explicitly at motion control for the robot when it has no goal.



(a) (b)

Fig. 1: The green circle represents the robot, the purple circles represent different humans and the black rectangles represent the walls of the corridor. The arrows indicate the heading of each agent. Only the initial and final locations of each object are shown. The numbers indicate the time step at which the snapshot was taken. This is an example of a situation in which the robot stopping (a) will cause more collisions and intrusions than following a failure controller which in this case simply moves the robot along the corridor out of the way of humans (b).

Developing such a controller is useful because it gives long term planners a contingency plan at planning time so they can more efficiently explore the search space. The failure controller itself also helps ensure that the robot obeys a safe policy even when it has no waypoint to navigate to. For example, in some situations like that in Figure 1a, staying still will cause the robot to be a hindrance to the humans and it will likely intrude into their personal space. Moving away to a different area of the scene, as shown in Figure 1b, would give the robot a safer place to replan from.

This work facilitates combining learning with planning for long term crowd navigation by providing a reinforcement learning based failure controller and a supervised learning based classifier for determining when to use it. Specifically, this paper assumes that an existing probabilistic planner uses a set of motion primitives and controllers to create navigation plans. The set of controllers includes some to execute complex behaviors that interact with pedestrians. Each of these controllers has some probability of success which is associated with one goal and trajectory, and some probability of failure which is associated with a different trajectory. Success cannot be guaranteed for the controllers because the surrounding humans may interact with the robot or each other in unexpected ways. Note that the failure controller's objective differs from that of a traditional navigation planner

What do I put here

[1]Patrick Naughton is with the Electrical and Systems Engineering Department at Washington University in St. Louis, St. Louis, MO, USA. patrickrnaughton@wustl.edu

[2]The remaining authors are with the Carnegie Mellon University Robotics Institute, Pittsburgh, PA, USA. {ichatter, tushark, mlikhach}@andrew.cmu.edu

in that the failure controller is not given an explicit goal; rather, it attempts to extract the optimal goal and trajectory from its environment.

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.

The design philosophy taken by this approach separates controllers for individual actions from the long term planner. This means the results presented here could easily be adapted to work with any planner.

The remainder of this paper is organized as follows. Section II describes previous work related to the results presented here. Section III outlines our problem statement in more detail. Section IV describes our approach and Section V describes the experiments we performed to evaluate our design. Section VI describes and discusses the results of these experiments and we conclude with Section VII and discuss future work.

The code of our approach is available here: `https://github.com/patricknaughton01/LearnController`.

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 [3]. 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 [4]. 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 [4].

Another approach uses inverse reinforcement learning to learn latent, possibly stochastic social rules humans observe when navigating in crowds [5]. 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 [6], [2]. Reinforcement learning is particularly suited to this application as noted in [6] 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 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. To characterize the state of the robot, an initial matrix is constructed with the same number of rows as there are agents and obstacles in the scene (including the robot). Each row of the matrix takes

the form $(d_x, d_y, s_{\text{pref}}, \theta, r, v_x, v_y, p'_x, p'_y, v'_x, v'_y, r', d, r_{\text{sum}})$ where

- d_x and d_y are the x and y distances from the robot to the goal.
- s_{pref} is the preferred speed of the robot.
- θ is the direction of motion of the robot w.r.t. its heading.
- r is the radius of the robot.
- v_x and v_y are the x and y velocities of the robot w.r.t. the robot's coordinate frame.
- p'_x and p'_y are the x and y coordinates of the other agent w.r.t. the robot's coordinate frame.
- v'_x and v'_y are the x and y velocities of the other agent w.r.t. the robot's coordinate frame (note, this is not relative to the robot's velocity, just relative to its rotation).
- r' is the radius of the other agent.
- d is the Euclidean distance between the robot and the other agent.
- r_{sum} is the sum of the radii of the robot and the other agent.

The first five components of each row are referred to as the robot's `self_state` because they pertain specifically to the robot.

This is augmented by a series of occupancy maps, one for each human and obstacle in the scene and one for the robot itself. Each occupancy map is centered on and aligned with its respective agent and their heading and is discretized into a number of squares. Each square indicates in a binary fashion whether or not it contains an object. Each square also contains the average velocity of all the objects within that square (simply set to 0 if the square is unoccupied) in the x and y directions. In our case, each occupancy map is discretized into 16 (4x4) 1 unit squares. This collection of occupancy maps and information about the robot's dynamics form the robot's *state*. Note that the dimensions of the state matrix will vary depending on the number of humans and obstacles in the scene.

This combined state is used as the input to a neural network. We model our architecture off of that presented in [2] in order to handle the variable input sizes, as the input matrix changes size based on the number of people and obstacles in the scene. This architecture was shown to produce good results for robotic navigation in the presence of pedestrians in [2]. In addition, we include an LSTM at the end of our network so that the robot can learn relationships between successive states. Figure 2 shows the overall architecture of the model. 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 trained this neural network on 1000 successful trajectories in which the robot can feasibly reach its goal. These trajectories were generated using the RVO2 library [7] [8].

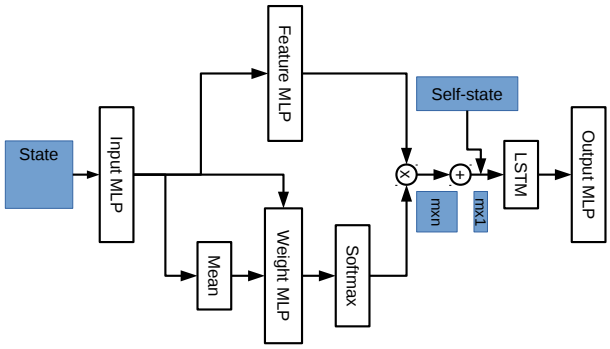


Fig. 2: Model architecture of the neural network used to control the robot. Variable sized inputs are handled by summing a weighted vector across the people and obstacles.

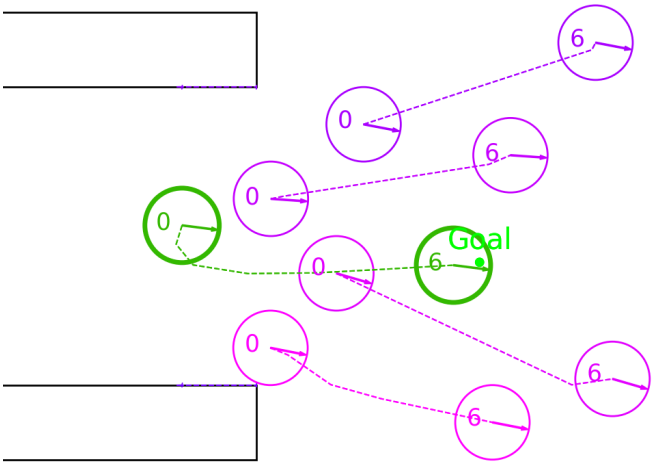


Fig. 3: Example of a successful trajectory. When executing the barge in success controller, the robot expects the humans in the scene to disperse to let it through.

We trained using an Adam optimizer with a learning rate of 0.005 and the negative 2D log-likelihood loss for 500 epochs. This controller was trained with a very high dropout rate of 0.5 in each MLP. We do this as a regularization technique and so that at execution time we can sample our network several times to estimate the epistemic uncertainty of our model. Our input and feature MLPs had 80 output units each, our weight MLP had two hidden layers of 32 units each and 1 output unit, our LSTM had 256 hidden units and the output MLP had two hidden layers with 128 and 64 units respectively and an output layer with 5 units. Because the success controller is not the focus of this work, we only used scenes containing four people and two obstacles. Figure 3 shows an example of a trajectory generated by the final controller.

We split the remaining 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 success controller. 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*. 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. We trained this network with the Adam optimizer with a learning rate of 0.005 and the negative 2D log-likelihood loss. Just as with the success controller, we train with a high dropout likelihood of 0.5. Our input and feature MLPs had 50 output units each, our weight MLP had two hidden layers of 32 and 16 units respectively and 1 output unit, our LSTM had 32 hidden units and the output MLP had two hidden layers with 64 and 32 units and an output layer with 5 units. These 5 output units represent the same things they did for the success controller, but predict the distribution of the robot's previous locations given its current state rather than next locations.

At execution time, we utilize this network's prediction to perform a statistical p-test with some user-specified α value. 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, and check to see if the previous robot's center is inside this ellipse. We perform this same test on the robot's current position given the distribution predicted by the success controller in the previous time step. By default, we assume that the trajectory will be successful and switch to the failure controller only if one of these tests fails. Figure 4 shows an example of such error ellipses for three successive positions of the robot.

We employ this strategy so that we can train the reverse predictor by showing it only successful trajectories. This is desirable for two reasons. First, there are generally many fewer ways for a robot to successfully reach its goal when interacting with pedestrians than for it to fail. By only training on success trajectories, we do not have to find or generate training data for all of these potential methods of failure. This reduces the potential for biasing the classifier to only identify very specific types of failure. Second, it is much easier to find examples of actual humans successfully barging into a group than examples of them failing to do this. This means that only using successful examples as training data will make this system better able to utilize real world human trajectories in the future.

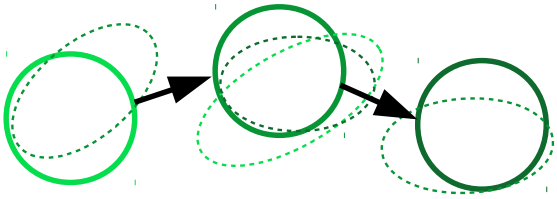


Fig. 4: Example of the error ellipses used to classify trajectories as successes or failures. The robot is represented by a solid circle in each time step and the error ellipses are dashed. Each shade of green is a different time step and the ellipses associated with a given time step are the same color as the robot in that step. Note that in the first and last states there are no previous or next states so these two positions only need to satisfy one test.

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 $\gamma \in (0, 1)$ and model the robot as a Markovian agent [9]. Rather than define a set of states constituting a goal, we define a reward function that punishes actions we determine to be undesirable.

We discretized the action space of the robot into 35 different actions. It can move at two different speeds (half speed and full speed) in 16 evenly spaced directions about its heading. It can also change its heading by $\frac{s_{\max}\pi}{16}$ ($s_{\max}$ is the maximum speed of the robot) in the positive or negative direction. Finally, the robot can choose to remain still.

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 3 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.

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

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

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

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 [10] to obtain an estimate of the optimal Q function. The state input is the same as that of the success controller and reverse predictor except that d_x and d_y are set to 0 because at this point we have given up on reaching the goal. We utilize an ϵ-greedy policy with an ϵ that linearly decays from 1.0 to 0.1 over $7,500$ transitions. We set $\gamma = 0.9$ and train using the Adam optimizer with learning rate 0.001. Our input and feature MLPs had 80 and 120 output units respectively, our weight MLP had two hidden layers of 64 and 32 units respectively and 1 output unit, our LSTM had 128 hidden units and the output MLP had two hidden layers with 128 and 64 units and an output layer with 35 units. Each unit of the output is the estimated Q value of the associated action.

To train our agent, we ran 500 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 failure classifier detects failure (or the success controller has executed for 15 time steps), the robot begins executing an ϵ-greedy policy and stores transitions in its replay buffer. We used a confidence value of 0.95 to detect failure during training. At each time step, the reward achieved by the robot is computed and its policy is updated. After 15 time steps, the current episode ends and another one is started. The failure controller moves for 15 time steps regardless of how long the success controller moved or whether or not a failure was actually detected.

Algorithm 1 summarizes the overall navigation policy of the robot. C_s, P_r, and C_f are the success controller, reverse predictor, and failure controllers respectively. t_s is the maximum number of time steps the success controller can run for, and t_f is the same for the failure controller. Note that dropout remains on for both the success controller and reverse predictor at inference time. This is so that we can sample the networks' outputs multiple time to find the variance of their predictions and thereby estimate the network's epistemic uncertainty. This is then added to the data uncertainty which the model directly outputs in order to find the total uncertainty in the robot's predictions [11]. S is the number of samples to draw from these networks. c is the confidence value to use when performing the p-test to determine whether or not the robot has failed. Here, $G^{(c)}(\mu, \Sigma)$ denotes an error ellipse that contains c fraction of the probability mass of $\mathcal{N}(\mu, \Sigma)$. During execution, we set the ϵ of the failure controller to 0 so that it behaves purely greedily. Additionally, after the robot (and humans in the scene) declare their goal velocities, our simulator uses the ORCA algorithm to adjust these velocities to attempt to make them safe (i.e., prevent collisions in the next time step) [12]. We use this functionality so that humans in the scene respond to the presence of the robot (although not entirely realistically).

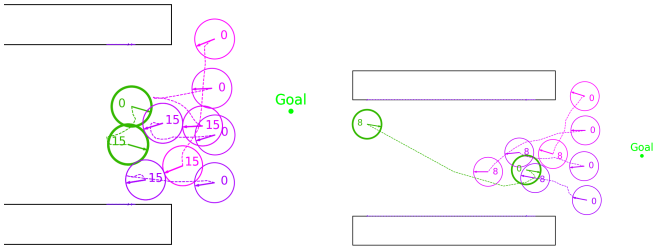
Algorithm 1 Detect And Handle Failure

function EXECUTE(t_s, t_f, S, c)
  Initialize C_s, P_r, and C_f
  Initialize $F \leftarrow false$
  for $t = 1..t_m$ **do**
    Observe s_t
    Sample S times from C_s, compute μ_s and Σ_s
    $G_s = \mathcal{N}(\mu_s, \Sigma_s)$
    Move towards μ_s, observe s_{t+1}
    Sample S times from P_r, compute μ_r and Σ_r
    $G_r = \mathcal{N}(\mu_r, \Sigma_r)$
    if $s_{t+1} \notin G_s^{(c)} \wedge s_t \notin G_r^{(c)}$ **then**
      $F \leftarrow true$
      break
    end if
  end for
  if F **then**
    for $t = 1..t_f$ **do**
      Observe s_t
      Compute μ_f and Σ_f from $C_f(s_t)$
      Move towards μ_f
    end for
  end if
end function

V. EXPERIMENTS

We began by testing the failure detection system to determine if we could accurately distinguish failure scenarios from successful ones. We ran Algorithm 1 on 1000 new success scenarios similar to the ones the success controller learned from (we will refer to these as *success scenes*) with $t_s = 15, t_f = 0, S = 20, c = 0.95$, where $t_f = 0$ because we are only examining the performance of the failure detection and do not care what the robot does after this point in this experiment.

We then ran a similar experiment to determine how often the failure detector would correctly identify scenes in which the robot cannot reach its goal. To do this, we utilized the same basic scene setup that was used to train the failure controller which we will refer to as *failure scenes*. We ran



(a) Trajectory in which the robot continues to execute its success controller even though it incurs a very high cost to reach its goal.

(b) Trajectory in which the robot realizes it cannot reach its goal and so begins executing its failure controller.

Fig. 5: Comparison between trajectories generated by simply executing the success controller (a) versus using the failure controller when a failure is detected (b).

Algorithm 1 on 1000 such scenes using the same parameters as the corresponding experiment with success scenes.

To evaluate the effectiveness of our overall framework, we ran two sets of experiments. In one, the failure classifier is completely disabled so that the robot continues to execute its success controller regardless of what the people in the scene do. We set $t_s = 15, t_f = 0, S = 20, c = 0.95$. The episode ends after 15 time steps or after the robot reaches its goal, whichever comes first. We set the parameters of Algorithm 1 to the same values that were used in our tests of the failure detection system and ran this experiment on 1000 failure scenes.

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. For these episodes, we set $t_s = 15, t_f = 5, S = 20, c = 0.95$ so that the failure controller has some time to improve its state after the detection system reports a failure. Note that if the failure detection system does not report a failure after t_s time steps, the episode simply ends. Just as with the previous experiment, we ran this one on 1000 failure scenes. Figure 5 shows a visual comparison between trajectories generated by the robot in the two experiments.

VI. RESULTS

We found that the failure detection system reported a spurious failure in just 4 out of the 1000 runs on a success scene, or 0.4% of the time. This result is even better than expected since we ran with $c = 0.95$, meaning we would expect up to 5% of these runs to be reported as failures. In failure scenes, we found that the failure detection system reported a failure in 960 out of 1000 runs, or 96.0% of the time. This was a surprisingly positive result because neither the reverse predictor nor the success controller were ever trained on a failure trajectory. The overall accuracy of the failure detection system was 97.8%.

To compare the overall effectiveness of the two navigation frameworks, we used versions of the metrics presented in [13] which were designed to evaluate robot motion in crowds. These metrics are the length of the trajectory, time elapsed, angular distance traveled (computed as total change in direction of motion 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. The time a run takes is the time from the start of the episode to when the robot last changes position. This is why not all success runs take the full 15 seconds and not all failure runs take 20 seconds. Table I summarizes these results where boldface indicates the better result.

TABLE I: Metrics comparing trajectories generated by a robot with no failure controller to a robot using the presented controller. The data here are averages across 1000 runs for both cases.

	No Failure Controller	Failure Controller (Ours)
Length	**3.92**	4.64
Angle	**4.80**	6.01
Time	13.50	**8.40**
Collisions	0.86	**0.21**
Intrusions	15.21	**6.08**

We note that the experiments without the failure controller actually achieved shorter path lengths with smaller overall changes in heading. However, as can be seen in Figure 5a, this is likely because the robot simply continues to butt up against pedestrians without making much progress. Additionally, the values achieved using the failure controller are not much larger, meaning the advantage of the success controller on these metrics is relatively insignificant. The robot also reaches its final position faster when employing the failure controller despite traveling farther. This could potentially allow it to execute its long term navigation plans more quickly because it can begin replanning to its overall goal sooner. More importantly, using the failure controller reduces collisions by over four times and intrusions by over two times. This means that the robot behaves much more safely when it can employ the failure controller presented here rather than relying on just its success controller.

VII. CONCLUSIONS

This work augments the integration of learning with planning for long-term autonomous navigation. We presented a failure detection system and failure controller that allow a robot to detect when its current action is likely to fail and provide a way to safely navigate in the absence of a goal. Our failure detection system achieved a very high accuracy in discriminating between success and failure scenarios which was much better than what was expected. The use of the failure controller with the detection system also substantially reduced the number of collisions and intrusions the robot incurred when navigating in dangerous failure scenes. These contributions can allow a robot to more safely and efficiently

navigate in the presence of humans by giving the robot a contingency plan in cases where it cannot execute its planned controller.

In the future, we would like to extend this work to controllers for other types of situations such as crossing in front of or overtaking a pedestrian. We would also like to collect real pedestrian data from densely crowded areas such as subways and use these recorded trajectories to learn a success controller (and subsequently a reverse predictor and failure controller). These data would allow the robot to better learn how humans actually move in a given scene compared to using the RVO2-generated trajectories. This would hopefully yield navigation that is safer and more reliable than that which is currently achieved.

ACKNOWLEDGMENT

What do i put here?

REFERENCES

[1] M. Likhachev and A. Stentz, "Probabilistic planning with clear preferences on missing information," *Lab Papers (GRASP)*, p. 25, 2008.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

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

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

[10] 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.

[11] Y. Gal, "Uncertainty in deep learning," *University of Cambridge*, vol. 1, p. 3, 2016.

[12] J. Van Den Berg, S. J. Guy, M. Lin, and D. Manocha, "Reciprocal n-body collision avoidance," in *Robotics research*. Springer, 2011, pp. 3–19.

[13] 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.