Probabilistic Robotics Final Report

Sawyer Paccione, Auguste Brown, Stephanie Bentley

Mechanical Engineering
Tufts University
Medford MA, USA
May 10, 2022

Contents

1 Introduction

As robots develop towards more complex applications and become more integrated into our lives, they, by necessity, interact more frequently with humans. For many autonomous applications, robots act in unknown environments with dynamic and static obstacles, including humans with often erratic and varying movement paths, making detecting and predicting their motion challenging. Applications from a Roomba-like cleaning robot to an autonomous delivery robot rely on moving through a dynamic environment with humans acting independently and with unknown goals. Not only do robots in these situations need to plan their navigation around human and non-human obstacles in the environment and follow human social norms, but preserving human safety must be a top priority. Thus, social navigation is a crucial and challenging area of robotic research. Since training reinforcement learning algorithms on physical robots is expensive, time-consuming, and prone to confounding factors, many algorithmic solutions to reinforcement learning are trained and tested in simulations. However, these simulations can vary from real-world application in terms of available state information or unexpected dynamics, making the sim-to-real transition challenging.

This project seeks to determine improvements to social navigation algorithms developed and tested in simulation to mitigate the sim-to-real gap. This paper builds on the Decentralized Structural-Recurrent Neural Network developed by Liu et al for social navigation, which aims to address problems with previous algorithms in dense or only partially observable environments that cause the *freezing robot problem* [1]. The original algorithm relies on sensing the positions of both the ego robot and the observable human agents exactly, which is unrealistic in a real environment. Thus, our paper seeks to reintroduce uncertainty into the simulation sensing as a form of domain randomization and determine the resulting impact of handling the uncertainty with a Kalman Filter on the algorithm's effectiveness in simulation. The paper compares three conditions, the baseline algorithm from Liu et al. [1], a noisy algorithm with introduced sensor noise, and a filtered algorithm using a Kalman Filter to estimate robot state, all tested in simulation.

This paper found that the baseline algorithm had superior performance in simulation with a 89% success rate as expected for an idealized algorithm, while the noisy algorithm had a decreased success rate of 75%. However, the filtered algorithm was successfully able to account for some of the noise, improving the success rate to 81%. Thus, the Kalman Filter in addition to the DS-RNN algorithm may perform better in a physical environment, improving the transfer from sim to real.

2 Background

Social navigation is a key feature in many robot applications, enabling increased automation in environments designed for humans. However, this area is still one of ongoing research. Social navigation is most commonly implemented through either model-based

approaches, which model social constraints, or learning-based approaches, which use imitation or reinforcement learning. Each of these approaches involve various benefits and drawbacks [2]. Work has been done on new solutions for social navigation, such as the approach seen in *Iterative Program Synthesis for Adaptable Social Navigation*[2], but these novel approaches are less common. Progress has also been made in understanding human-human interaction and predicting trajectories of multiple pedestrians [3], which inspired new contributions in crowd navigation. Another recent social navigation algorithm developed by Cui et al. represents navigating a dynamic pedestrian area as a Partially-Observable Markov Decision Process, training the navigation reinforcement learning on both real interaction data and a virtual transition model to learn the world transition model [4].

As reinforcement learning grows in popularity in robotics research, the sim-to-real problem of transferring models trained in simulation onto real robots has gained more attention. Compared to training reinforcement learning algorithms on physical robots, simulations allow algorithms to be trained faster, more scalably, with lower costs, and less safety risk [5]–[8]. However, an algorithm trained in simulation is often significantly less effective when tested on a physical robot, labelled the *sim-to-real problem* or *reality gap* [8]. Much research around reinforcement learning algorithms in robotics has centered around bridging this gap. Some efforts center around building more realistic simulators, while another popular technique, *domain adaption*, involves attempting to unify the source and target domain spaces to improve the algorithm transferability [7]. One of the most popular methods, *domain randomization*, involves training the simulation on highly random data in order to cover the variation between the simulation and reality [7]. Tobin et al. had success using domain randomization on RGB image sensor inputs for object detection [5], while Vacaro et al. effectively used domain randomization in an end-to-end flow from sensor input to object manipulation on a robotic arm [6]. As part of the comprehensive domain randomization, Surmann et al. added Gaussian noise to sensor data and train on varying randomized environments [9]. Similarly, an algorithm for robotic control in Peng et al. was shown to transfer to a physical system with a similar level of performance to simulation by training the simulation with randomized dynamics models and introducing observation noise [7]. To our knowledge, the sim-to-real problem has not been explicitly studied in the context of social navigation algorithms, so this paper aims to take initial steps at domain randomization by adding sensor noise for an existing social navigation algorithm.

The current study is based on the social navigation algorithm developed by Liu et al. [1]. This algorithm, labeled a Decentralized Structural Recurrent Neural Network (DS-RNN), models social navigation as an MDP with the relationships between the ego robot and human agents in the environment as a spatio-temporal graph, in which each agent is a node connected by edges representing both spatial relationships and temporal relationships across each timestep. The algorithm then utilizes a Recurrent Neural Network for each factor in the graph, connecting layers to obtain a value and policy [1]. While the DS-RNN algorithm highlighted in this paper was tested on a physical robot, Liu et al. do not detail effectiveness on the physical robot and provide

no discussion of consideration for the transfer [1]. Therefore, this paper aims to consider the sim-to-real gap in the context of the DS-RNN algorithm by adding sensor noise to determine the effect of initial domain randomization in the simulation performance.

3 First Attempt

Originally, we designed a project based on the social navigation algorithm developed by Cui et al [4]. This paper develops an algorithm for navigating a dynamic pedestrian area using a Partially-Observable Markov Decision Process. The model is trained by utilizing kinematics to estimate future transitions, weighting the reward by distance to the goal and penalizing collisions, and determining the error in predicting future obstacle maps. The algorithm takes LIDAR data, processed by created stacked obstacle maps, as well as relative goal position and current velocity to sample a next action with linear and angular velocities from the policy [4]. Additionally, previous studies have proposed integrating haptic feedback into a social-force based model of navigation to increase communication and collaboration with surrounding humans [10]. Thus, our study originally intended to integrate haptic feedback into the social navigation algorithm to communicate the intentions of the navigating robot to the surrounding humans, in addition to an obstacle avoidance algorithm.

Our goal was to implement the social navigation algorithm on a Turtlebot2 using input of the robot's state and the RGBD output from an Astra camera and outputting steering control plus a direction indicator for the haptic feedback. This haptic feedback would be sent to a Sphero to provide feedback to humans in the environment.

However, the process of implementing the original social navigation algorithm as a baseline was extremely time-intensive and bug-prone, leaving little possibility of extending the algorithm further. Thus, our project pivoted to focus on the sim-to-real application of the DS-RNN algorithm.

4 Methodology

Building on the baseline DS-RNN algorithm presented in Liu et al.[1], we developed additional variations of a noisy algorithm and a filtered algorithm. The noisy algorithm adds Gaussian noise to the computation of each agent's position, including both the ego robot and humans in the environment. This noise represents a combination of sensor and environmental noise that would be present in a physical system in order to make the simulation more realistic. The filtered algorithm uses a Kalman Filter on the robot's ego state in order to compensate for some of the noise introduced into the algorithm, to determine whether the trained algorithm sufficiently accounts for the noise or whether manually accounting for the noise is more effective. Each of these variations, including the baseline from Liu et al.[1], were tested in the Gym simulation environment. We predicted the noisy variation would have significantly decreased performance in simulation compared to the baseline algorithm, while the

filtered algorithm would improve performance over the noisy but not reach performance levels of the baselines algorithm, since the baseline is idealized.

4.1 Algorithms

4.1.1 Social Navigation Algorithm

The main social navigation algorithm, as extended from Liu et al, models the interaction between the robot and humans in the environment as an MDP, represented by the tuple $\langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma, \mathcal{S}_0 \rangle$ [1]. The robot state w^t includes the robot's x and y positions and velocities, the goal x and y position, the maximum speed, heading angle, and radius. There are a finite number, n, of humans present in the simulation, with the state of the i^{th} human represented as u_i^t and including human x and y position. The overall state $s_t = [\mathbf{w}^t, u_1^t, ...u_n^t]$. For each episode, the robot starts at initial state $s_0 \in \mathcal{S}_0$, and at each timestep t takes action $a_t \in \mathcal{A}$ according to the current policy $\pi(a_t, s_t)$. The robot receives reward r_t and transitions to state s_{t+1}. Humans take action according to their own policies and move to the next states. Episodes continue until t reaches the maximum episode length T (timeout), the robot reaches its goal (success), or a robot collides with a human (collision). The reward has a discount factor of $\gamma \in (0, 1]$ and thus the total reward, $R_t = \sum_{k=0}^{\infty} \gamma^k * r_{t+k}$, is accumulated at each time step [1].

The scenario is represented as a decentralized spatio-temporal graph, $\mathcal{G} = (\mathcal{V}, \mathcal{E}_S, \mathcal{E}_T)$. Each node in the set $\mathcal{V}$ is an agent in the scenario, the set of spatial edges $\mathcal{E}_S$ represent the spatial relationships between two agents during one time step, and the set of temporal edges, $\mathcal{E}_T$, represent the transition for the same agent over adjacent time steps [1].

A network architecture is developed from the graph representation with a RNN layer for each factor of the graph, with additional layers that ultimately result in an output of the value $\mathcal{V}$ and the policy $\pi(a_t, s_t)$ [1].

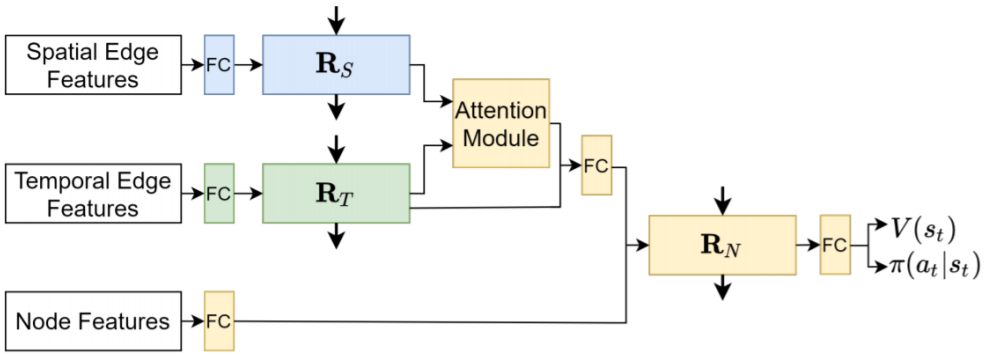

Figure 1: DS-RNN network architecture from Liu et al. [1]

4.1.2 Kalman Filter Algorithm

The Kalman Filter was implemented to provide a better estimate of the robot's ego state given the additional noise. The filter was developed with the *pyfilter* library's KalmanFilter, which implements a linear Kalman Filter. The filter object was constructed with a 4x1 state vector of $\bar{x}_t = \begin{bmatrix} x_t \\ \dot{x}_t \\ y_t \\ \dot{y}_t \end{bmatrix}$, representing the position and velocity in the Cartesian coordinate system. The kinematics of the robot moving are defined by the following equations:

$$x_t = x_{t-1} + \dot{x}_{t-1} * dt + \frac{1}{2} * \ddot{x}_{t-1} * dt^2 \tag{1}$$

$$\dot{x}_t = \dot{x}_{t-1} + \ddot{x}_{t-1} * dt \tag{2}$$

$$y_t = y_{t-1} + \dot{y}_{t-1} * dt + \frac{1}{2} * \ddot{y}_{t-1} * dt^2 \tag{3}$$

$$\dot{y}_t = \dot{y}_{t-1} + \ddot{y}_{t-1} * dt \tag{4}$$

From these equations, the state transition matrix was defined as

$$A = \begin{bmatrix} 1 & dt & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & dt \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

To account for noise in the acceleration, the covariance matrix was defined as

$$P = \begin{bmatrix} \sigma^2_{noise} & 0 & 0 & 0 \\ 0 & \sigma^2_{noise} & 0 & 0 \\ 0 & 0 & \sigma^2_{noise} & 0 \\ 0 & 0 & 0 & \sigma^2_{noise} \end{bmatrix}$$

The process noise in the system is estimated using *filterpy*'s discrete white noise function with a variance of σ^2_{noise}.

The measurement vector at each step was defined as the recorded x and y position of the robot with added noise, or $\bar{z}_t = \begin{bmatrix} x_t \\ y_t \end{bmatrix}$. Thus, the measurement function was defined as $C = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$. To account for the noise added to sensor data, the measurement noise matrix was $\begin{bmatrix} \sigma^2_{noise} & 0 \\ 0 & \sigma^2_{noise} \end{bmatrix}$, where σ^2_{noise} was the standard deviation of the noise set in the configurations. The models in this paper used a value of $\sigma_{noise} = 0.2$.

Our control vector at each step of the Kalman Filter was defined as the difference between the determined action velocity and the previous velocity, or $\bar{u}_t = \begin{bmatrix} a\dot{x}_t - \dot{x}_{t-1} \\ a\dot{y}_t - \dot{y}_{t-1} \end{bmatrix}$.

Thus, the control transition matrix was $B = \begin{bmatrix} 0 & 0 \\ 1 & 0 \\ 0 & 0 \\ 0 & 1 \end{bmatrix}$. At each step of an episode, the robot uses the Kalman Filter to predict its state given the latest control vector, and then takes a measurement to update the prediction. The resulting state belief is used for the step in the environment as input for the reinforcement algorithm.

4.2 Simulation

Our work made use of an OpenAI Gym environment. The simulation contains a robot and a list of humans, initialized to random positions. Each agent has a goal position, and uses a policy to reach this goal. The humans randomly select either the ORCA or Social Force policies (discussed in Liu, et al.), while the robot uses the learned DS-RNN policy. In navigating towards their goals, the robot is invisible to the humans, simulating a robot's need to navigate without help from humans. At each timestep, all agents select an action based on their policy, and move accordingly. The simulation ends when the robot either experiences a collision, reaches its goal, or the maximum simulation time is reached. While training, each episode returns a reward value.

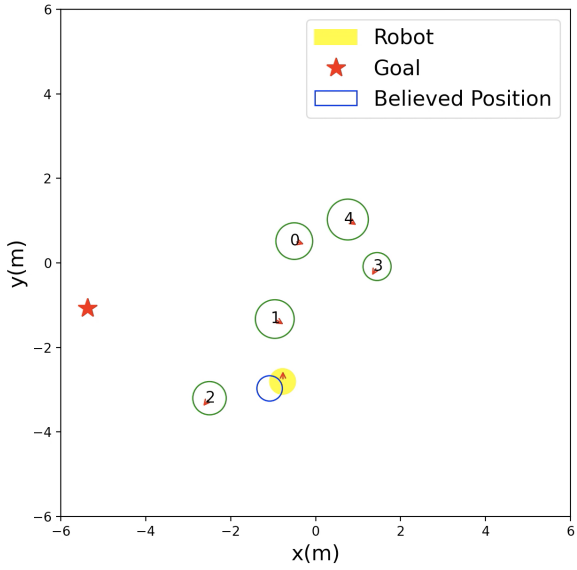

Figure 2: Visualization of the simulation environment

In Liu, et al.'s implementation[1], the robot has perfect knowledge of its own state as well as the position and velocity of all humans. Since this assumption is inaccurate when dealing with real robots, we sought to make a more realistic simulation.

4.3 Experimental Design

In order to increase the realism of the simulation, and thus more rigorously test the DS-RNN model, we designed an experiment to compare several cases. We used Liu et al.'s model as the control case to compare our results. To better simulate the noise of the real world, we also made a baseline noisy case, which uses a noisy version of the robot position. Finally, we implemented a Kalman Filter case, which uses a Kalman Filter to fuse the noisy action velocity with a noisy position "sensor reading" and produce a better state estimation.

We ran our experiment in a simulation with 5 humans and one robot. The robot has a radius of 0.3 meters, and the radius of the humans is randomized. The maximum simulation time was set to 50 seconds, and the timestep length was set to 0.25 seconds. We selected the standard deviation of the added noisy to be 0.2 meters.

Each case was implemented in the simulation environment, and then trained with the DS-RNN algorithm. After each training episode, a reward value of 20 was returned for reaching the goal, -10 for a collision, and 0 for a simulation timeout. We began by training the control model on 10e6 time steps.

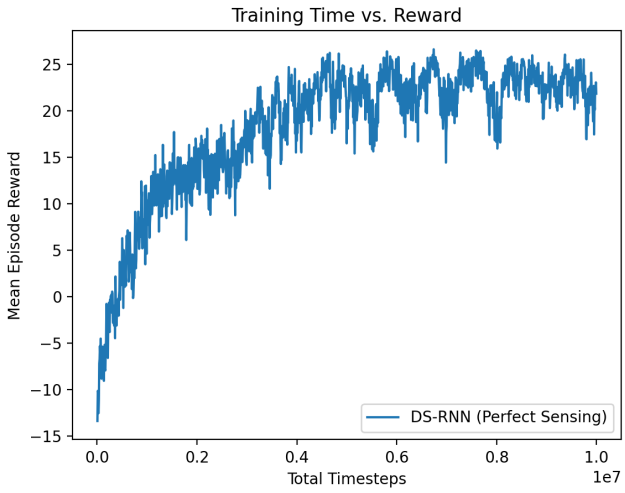

Figure 3: Training curve for the control model

This training process took roughly 10 hours. We noted that the training seemed to reach diminishing returns at roughly halfway through, so we trained the other two cases with just 5e6 time steps.

4.4 Measures

After training the three models, we tested each case on 2500 episodes. Metrics were recorded from each case, including success rate, collision rate, timeout rate, average

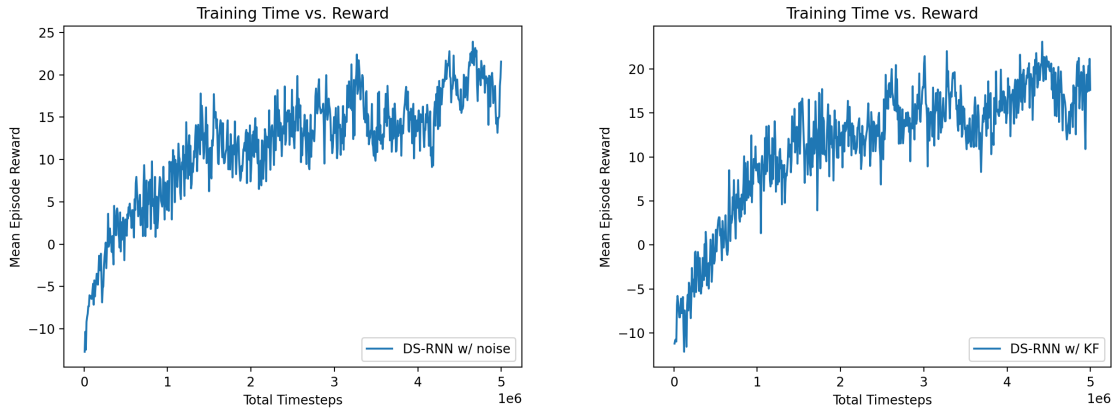

Figure 4: Training curve for the two additional cases

navigation time, and average path length. Success rate, collision rate, and timeout rate were calculated as the number of episodes where the robot reached the goal, collided with a human agent, or exceeded the maximum simulation time out of the total of 2500 episodes. The average navigation time was calculated as the average amount of time taken for the robot to navigate to the goal across all the success cases. The average path length was calculated as the average distance covered by the robot in each episode, regardless of the episode outcome. The first three of these metrics describe how successful the learned policy was, while the latter two describe how efficient it was.

5 Results

The baseline model implemented by Liu, et al. saw the best performance of the three cases. This was expected, as their model assumed perfect knowledge of all actors' position and velocity. Over the 2500 testing episodes, the baseline model succeeded in 89% of the episodes, collided with humans in 11%, and never timed out. It had an average navigation time of 10.31 seconds, and an average path length of 16.27 meters. These metrics served as our baseline which we compared our models to.

Next, the "noisy" model was tested, which uses dead reckoning based on a noisy velocity signal to compute its position. This model served as a more realistic depiction of the noise seen in the real world. The noisy model achieved a success rate of 75%, with a collision rate of 21% and a timeout rate of 4%. Additionally, the noisy model produced an average navigation time of 12.34 seconds and an average path length of 19.50. Our goal was to use a Kalman Filter to improve upon this performance without access to unrealistic information (i.e. perfect state knowledge).

Finally, we tested our model which used a Kalman Filter to combine a noisy velocity signal with a noisy position "sensor" to achieve a better state estimate. The policy trained with a Kalman Filter succeeded in 81% of the testing episodes, while it collided

with humans in 19%, and never timed out. This model resulted in an average navigation time of 11.07 seconds and an average path length of 17.15 meters.

Examples videos can be viewed here. Results are summarized in Table 1.

	Success	Collision	Timeout	Avg. Nav Time	Avg. Path Len
Baseline	89 %	11 %	0 %	10.31 s	16.27 m
Noisy	75 %	21 %	4 %	12.34 s	19.50 m
Kalman	81 %	19 %	0 %	11.07 s	17.15 m

Table 1: Summarized Results

6 Discussion

Our analysis shows that while adding noise to the robots sensors and movement clearly increases the robot's Collision rate and decreases it's Success rate, these unwanted changes can be mitigated by using a Kalman Filter. The resulting percentages offered by the Kalman Filter approach are clearly not as optimal as the Baseline percentages, but they offer a better view of what this algorithm would look like on a real robotic system. Additionally, the performance of the Kalman Filter model was substantially better than that of the naive noisy model.

Our findings also suggest that the use of a Kalman Filter greatly improves the efficiency of navigation algorithms when subjected to noise. The Kalman Filter successfully reduced the average episode path length from 19.50 to 17.15 meters. This is likely because the noisy policy was trained to stay further away from humans, because more collisions resulted when it got too close.

We cannot say for certain that either of these approaches, when implemented on a real robot, would result in satisfying success and collision percentages. This study suggests that the use of a Kalman Filter in the training and rollout of the policy improves these percentages from their naive baseline while accounting for some variation from a real environment, but does not necessarily provide optimal results. Next steps include implementing both the baseline approach of assuming there is no noise and the Kalman filter approach on a real robot, for instance the Turtlebot2. In order to do so, the depth camera would be used to detect the positions of people and each detected person would be tracked with an ID to feed the position of the tracked people into the observation of the RL network. Furthermore, the robot state can be obtained by running SLAM. Lastly, since the Turtlebots cannot move holonomically, and instead move in the direction of their orientation, this would need to be modelled differently in the training of the policy.

7 Impact and Conclusion

By testing the effect of implementing a Kalman Filter on a Crowd Navigation Algorithm, this work established that the Kalman Filter can be used successfully with building a policy using Reinforcement Learning. This work represents a more accurate model than the original solution which had perfect sensing, providing a strategy for the DS-RNN Network to get closer to bridging the gap between simulation and real-life robotics. Future research should focus on establishing the best methods for determining the state vectors, sensing data, and the variances associated with them for the DS-RNN Network to train it's policy, as well as testing the results on physical robots. All code used can be found on Github.

Bibliography

[1] S. Liu, P. Chang, W. Liang, N. Chakraborty, and K. Driggs-Campbell, "Decentralized structural-RNN for robot crowd navigation with deep reinforcement learning," pp. 3517–3524, May 2021, ISSN: 2577-087X. DOI: 10.1109/ICRA48506.2021.9561595.

[2] J. Holtz, S. Andrews, A. Guha, and J. Biswas, "Iterative program synthesis for adaptable social navigation," in *2021 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, ISSN: 2153-0866, Sep. 2021, pp. 6256–6261. DOI: 10.1109/IROS51168.2021.9636540.

[3] A. Alahi, K. Goel, V. Ramanathan, A. Robicquet, L. Fei-Fei, and S. Savarese, "Social lstm: Human trajectory prediction in crowded spaces," in *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016, pp. 961–971. DOI: 10.1109/CVPR.2016.110.

[4] Y. Cui, H. Zhang, Y. Wang, and R. Xiong, "Learning world transition model for socially aware robot navigation," *arXiv:2011.03922 [cs]*, Nov. 8, 2020, version: 1. arXiv: 2011.03922. [Online]. Available: http://arxiv.org/abs/2011.03922 (visited on 03/13/2022).

[5] J. Tobin, R. Fong, A. Ray, J. Schneider, W. Zaremba, and P. Abbeel, "Domain randomization for transferring deep neural networks from simulation to the real world," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2017, pp. 23–30. DOI: 10.1109/IROS.2017.8202133.

[6] J. Vacaro, G. Marques, B. Oliveira, *et al.*, "Sim-to-real in reinforcement learning for everyone," in *2019 Latin American Robotics Symposium (LARS), 2019 Brazilian Symposium on Robotics (SBR) and 2019 Workshop on Robotics in Education (WRE)*, 2019, pp. 305–310. DOI: 10.1109/LARS-SBR-WRE48964.2019.00060.

[7] W. Zhao, J. P. Queralta, and T. Westerlund, "Sim-to-real transfer in deep reinforcement learning for robotics: A survey," *CoRR*, vol. abs/2009.13303, 2020. [Online]. Available: https://arxiv.org/abs/2009.13303.

[8] X. B. Peng, M. Andrychowicz, W. Zaremba, and P. Abbeel, "Sim-to-real transfer of robotic control with dynamics randomization," *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pp. 1–8, 2018. DOI: 10.1109/ICRA.2018.8460528.

[9] H. Surmann, C. Jestel, R. Marchel, F. Musberg, H. Elhadj, and M. Ardani, "Deep reinforcement learning for real autonomous mobile robot navigation in indoor environments," arXiv, 2020. DOI: 10.48550/ARXIV.2005.13857. [Online]. Available: https://arxiv.org/abs/2005.13857.

[10] Y. Che, C. T. Sun, and A. M. Okamura, "Avoiding human-robot collisions using haptic communication," in *2018 IEEE International Conference on Robotics and Automation (ICRA)*, ISSN: 2577-087X, May 2018, pp. 5828–5834. DOI: 10.1109/ICRA.2018.8460946.