Deep Reactive Planning in Dynamic Environments

Kei Ota[1*] **Devesh K. Jha**[2] **Tadashi Onishi**[1] **Asako Kanezaki**[3]
Yusuke Yoshiyasu[4] **Yoko Sasaki**[4] **Toshisada Mariyama**[1] **Daniel Nikovski**[2]
[1] Mitsubishi Electric [2]MERL [3]Tokyo Institute of Technology [4]AIST

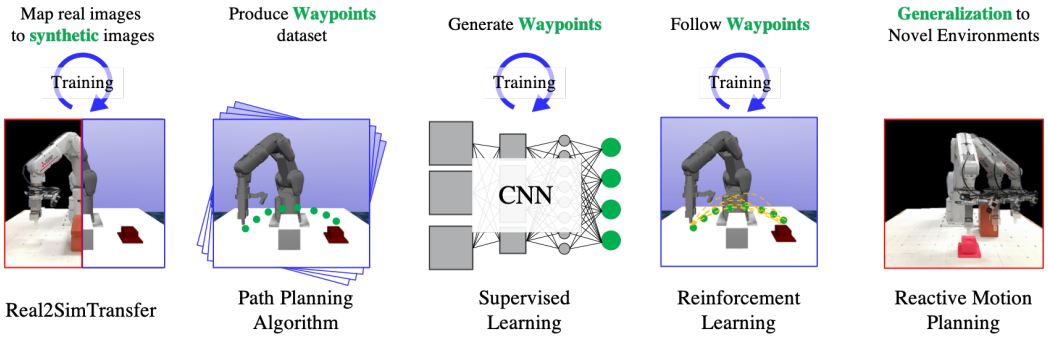


Figure 1: Our proposed agent learns an end-to-end reactive planning technique by combining traditional path planning algorithms, supervised learning (SL) and reinforcement learning (RL) algorithms in a synergistic way. A deep CNN is used to learn the sequence of waypoints obtained from a kinematic planning algorithm (e.g., a Bidirectional RRT*) given a depth image of the environment. The agent learns to follow arbitrary waypoints using path-conditioned RL, thus resulting in efficient exploration. We show that our trained agent can achieve good sample efficiency, as well as generalization to novel environments in simulation as well as real environments. The whole learning process is done in the simulator by learning a Real2Sim transfer function to make the training process efficient and suitable for robotic systems.

Abstract: The main novelty of the proposed approach is that it allows a robot to learn an end-to-end policy which can adapt to changes in the environment during execution. While goal conditioning of policies has been studied in the RL literature, such approaches are not easily extended to cases where the robot's goal can change during execution. This is something that humans are naturally able to do. However, it is difficult for robots to learn such *reflexes* (i.e., to naturally respond to dynamic environments), especially when the goal location is not explicitly provided to the robot, and instead needs to be perceived through a vision sensor. In the current work, we present a method that can achieve such behavior by combining traditional kinematic planning, deep learning, and deep reinforcement learning in a synergistic fashion to generalize to arbitrary environments. We demonstrate the proposed approach for several reaching and pick-and-place tasks in simulation, as well as on a real system of a 6-DoF industrial manipulator.

Keywords: Reactive Planning, Trajectory Optimization, Deep RL

1 Introduction

Deciding how to reach a goal state by executing a long sequence of actions in robotics and AI applications has traditionally been in the domain of automated planning, which is typically a slow, deliberative process that makes extensive use of knowledge about how the world works and how the agent can operate in it. Yet, humans and other living organisms demonstrate amazing ability to

*Correspondence to: Ota.Kei@ds.MistsubishiElectric.co.jp and jha@merl.com

4th Conference on Robot Learning (CoRL 2020), Cambridge MA, USA.

replan very quickly when the goal is moving, for example, when a predator is chasing prey or when a baseball player is trying to hit a fast-moving ball. This ability is likely to result not from just very fast execution of a basic planning procedure, but from skillful generalization over many previously computed and suitably cached solutions to instances of planning problems with variable goals. For a baseball player, these would be the many successful swings performed during target practice, over many variations of the trajectory of the incoming ball. While recently we have seen tremendous progress in reinforcement learning and deep learning, designing agents which can demonstrate such behavior still remains elusive. Our work is motivated by designing agents that can demonstrate such intelligent behavior by reacting to changes in an environment during execution.

We propose a learning-based approach to reactive planning, inspired by the way humans and animals perform it, reacting to change in environments effortlessly due to their highly responsive reflexes. As long as the planning environment does not involve tight, complex obstacle-cluttered environment, most humans can swiftly change their plan and still perform a trajectory-centric task without having to replan the original trajectory. Within this approach, a central question is how generalization should take place, and what the output of the generalizing module should be. While it is tempting to output and generalize directly over motor commands, this might make the learning problem unnecessarily hard, requiring too many training samples. Instead, we propose to decouple the problem into a trajectory (re)-planning component, subject to generalization, and a trajectory following component, which remains constant and goal-independent (see Fig. 1).

The main contribution of the proposed work is to present a technique for training agents that can perform reactive tasks in dynamic environments in an end-to-end fashion. We synergistically bring together planning, supervised learning, and reinforcement learning to design controllers for robotic systems, which can use low-cost vision systems to perform trajectory-centric tasks. Furthermore, we use a real2sim transfer technique, which allows us to design policies only in simulation by learning a function that maps real images to simulated ones, and thus makes training more efficient. On an intuitive note, the proposed technique decouples the problem of trajectory planning and trajectory-centric control. More specifically, the trajectory planner is a neural network that takes as input an image of the environment and outputs a collision-free trajectory defined by a sequence of relatively few waypoints. This neural network is trained in a supervised learning fashion using kinematic trajectories generated in different environments using an off-line deliberative planner such as RRT. The trajectory-centric controller is a deep RL policy that takes as an input an arbitrary trajectory and allows the robot to follow it (see Fig. 1). The training is done in a simulation environment by performing real2sim transfer for efficient training and the trained policy is transferred to the real system without further fine-tuning.

2 Related Work

In this section, we review related work in open literature. Motion planning is one of the core and widely studied topics in robotics. The most widely used algorithms are sampling-based methods, such as rapidly exploring random trees (RRT) [1] and probabilistic roadmaps (PRM) [2]. There are several other optimization-based techniques such as CHOMP and STOMP [3, 4]. While RRT-like approaches are not reactive per se, there are several approaches which use sampling-based algorithms for reactive planning [5, 6]. However, all these traditional approaches require explicit detection and tracking of obstacles [7, 8].

The combination of reference paths and RL has been widely researched [9, 10, 11, 12]. In [9, 11], Probabilistic Roadmaps (PRM) [2] are used to find reference paths, and RL is used for point-to-point navigation as a local planner for PRM. In [10], a model-based RL agent is learned for autonomous robotic assembly by exploiting the prior knowledge in the form of CAD data, using a Guided Policy Search (GPS) approach to learn a trajectory-tracking controller. However, the GPS algorithm still produces very local policies and cannot generalize to changing environments. Another relevant set of ideas could be found in [13], where a model-based approach is used to predict movement of obstacles in the environment of the robot and an MPC approach is used to replan. This is done in our proposed technique, too, in a more implicit manner. Another related and widely successful algorithm that learns a visuomotor policy from several example trajectories could be found in [14, 15]. However, it is, by its nature, goal-centric, and cannot be easily extended for reactive planning.

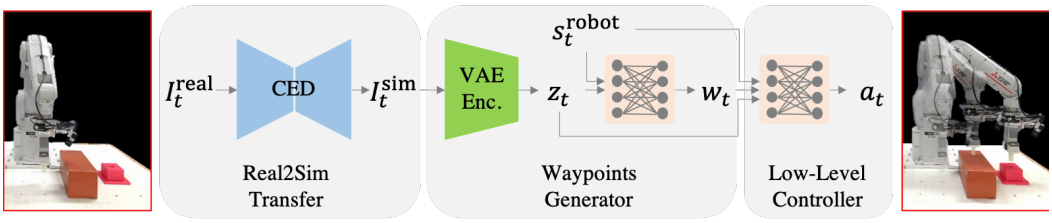


Figure 2: Architecture of the proposed method. The whole architecture consists of three modules: a Real2Sim transfer module converts the real depth image to the simulated one using a Convolutional Encoder-Decoder (CED). The simulated images are then compressed to latent variables by using a Variational Auto Encoder (VAE), and the waypoints generator takes the latent variables of the simulated image and robot's state, and generates the waypoints that roughly guide the agent to the desired goal. The input to the RL agent is a concatenation of the waypoints, the latent variables, and the internal state of robot, and outputs a low-level action to control the robot.

Goal parameterization in RL has been studied extensively to design policies which can generalize to different goals [16, 17]. One of the main differences that we would like to point out is that goal parameterization in RL cannot achieve reactive planning as it would require an auxiliary function that implicitly or explicitly updates the goal states from an observation. In our proposed work, this is achieved as the path encodes the goal and automatically tries to avoid obstacles in the environment, too. The closest work to ours is [12, 18]. [12] learns an RL agent that optimizes trajectory for a 6-DoF manipulator arm. Our approach, however, can deal with changes in the environment by conditioning an RL agent with a reference path which is also generated based on an observation of the current environment. [18] considers a similar setting for 2-dimensional robot navigation tasks only in simulation. Our method, however, considers higher-dimensional path planning, and is evaluated on real systems as well as simulated ones. Moreover, we try to minimize data collection in the real systems by additionally introducing the Real2Sim transfer pipeline, which is learned in a efficient way by using a common approach for domain adaptation [19, 20, 21, 22], and thus becomes more challenging than [18].

3 Method

The main idea of the proposed method is to encode a robot trajectory by a fixed number of waypoints in Cartesian or joint space, to be followed in a set order, and have a deep neural network produce the correct waypoints for a novel goal state and/or changed environment. To this end, the goal state and current environment are presented as inputs to a convolutional neural network (CNN), in the form of an image. Correspondingly, the coordinates (Cartesian or joint angles) of the waypoints are presented as outputs of the CNN. A training set for the CNN is produced by running an offline planner such as RRT, for a large number of environments, respectively input images. The CNN is trained over this data set, and upon completion of training, is used to produce suitable waypoints for novel inputs that are observed in novel environments. The trajectory defined by the waypoints is then followed by a low-level action controller obtained by means of RL. The architecture of the end-to-end training method is shown in Fig. 2.

3.1 Waypoints Generator

Learning an optimal policy from high-dimensional inputs, such as images, generally requires a huge number of interactions in the environment, and also bigger neural networks, and is thus not very suitable for using the policy on a real system. Although learning to produce an optimal action needs to solve an optimization problem, and thus requires significant computational time, generating an optimal path (i.e., the sequence of waypoints) is relatively easy using a prior path planning method, and using DNNs in a supervised learning mode is also easy to implement. Those waypoints can then be used to guide the agent to a goal position [12, 18]. We use this idea to first train a module for the agent that can predict an optimal path using information about an environment encoded via a depth image and the robot's state.

We follow the same procedure as in [18]: we first generate a data set that consists of pairs of depth images in a simulator I^{sim}, and optimal paths $\boldsymbol{w}$, which are generated by using Bi-directional RRT* [23] with a random start, goal, and obstacle position. Then, the waypoints generator learns to output the waypoints by minimizing the following objective function:

$$\mathcal{L}_{\text{waypoints}} = \mathbb{E}_{I^{\text{sim}} \sim p_{\text{sim}}} \left[\| G(I^{\text{sim}}, \boldsymbol{s}^{\text{robot}}) - \boldsymbol{w} \|_2^2 \right]. \tag{1}$$

The waypoints generator, G in Eq. (1), consists of a CNN that takes a depth image I^{sim} and a robot's state $\boldsymbol{s}^{\text{robot}}$, and generates a short horizon path $\boldsymbol{w}$ that consists of 5 waypoints, each of which has 3- or 6-dimensional relative position or angle with respect to the current state of the agent. Thus, the SL module provides the agent with a rough, collision-free path that it can follow to reach the desired goal state.

3.2 Path-Conditioned RL

We utilize the short horizon optimal paths, which are generated by the waypoints generator, to improve the sample efficiency of training an RL agent and its generalization capability over novel environments. In order to do that, we exploit the waypoints in two ways: reward shaping and curriculum learning.

Reward Shaping Since the depth image of the environment does not directly provide an agent with the information to accomplish a task, it generally needs a huge amount of interaction with environments to learn an optimal policy. We mitigate this problem by utilizing waypoints. Although the waypoints do not have dynamical information that can directly control a robot, it can roughly guide the agent to move to the goal state by using the waypoints for reward shaping. We follow a formulation similar to the one proposed in [12, 18], which we call *path-conditioned RL*. It defines a reward function by combining the original RL formulation and reference path (waypoints) $\boldsymbol{w}$ based function as

$$r(\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{w}_t) = f(\boldsymbol{s}_t, \boldsymbol{a}_t) + h(\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{w}_t), \tag{2}$$

where $f(\boldsymbol{s}_t, \boldsymbol{a}_t)$ comes from the original RL formulation, and $h(\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{w}_t)$ is waypoints based function. We define $f(\boldsymbol{s}_t, \boldsymbol{a}_t)$ as:

$$f(\boldsymbol{s}_t, \boldsymbol{a}_t) = \lambda_1 \mathbb{I}_{\text{collision}} + \lambda_2 \mathbb{I}_{\text{goal}} + \lambda_3 \|\ddot{\boldsymbol{\theta}}\|, \tag{3}$$

where $\lambda_1 \mathbb{I}_{\text{collision}}$ penalizes collision with obstacles, and $\lambda_2 \mathbb{I}_{\text{goal}}$ encourage the agent to achieve a reach task, and $\lambda_3 \|\ddot{\boldsymbol{\theta}}\|$ penalizes angular accelerations to generate a smoother trajectory. We then define $h(\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{w}_t)$ as

$$h(\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{w}_t) = \lambda_4 d_{\text{path}} + \lambda_5 n_{\text{progress}}, \tag{4}$$

where d_{path} is the distance to the reference path (waypoints) and n_{progress} is the progress along the path. The first term limits the exploration area by penalizing the distance to the waypoints, whereas the second term encourages the agent to move along the waypoints towards the goal state. More details can be found in [18] and the Appendix B.4.

Curriculum Learning It is important to note that the waypoint generator is only based on kinematic planning for the robot in the given environment. Consequently, it is possible that the RL agent starts deviating from the sequence of waypoints as it tries to follow these waypoints using feasible control actions, and thus may start diverging if the episode length is long. This could degrade (or even destabilize) the whole training process as it can guide the waypoint generator to domains where it does not generalize well. In these situations, the RL agent cannot achieve the original task of reaching a goal state because the waypoints generator is not able to accurately guide the agent to the goal state. To make sure that the RL agent stays close to the sequence of waypoints generated by the waypoint generator, we initialize the robot state to waypoints which are closer to the goal state of the robot, instead of starting at the initial state of the robot. We iterate over a sequence of such random initialization of RL episodes which makes RL training easier. This allows the RL agent to collect informative samples from different parts of the trajectory and thus allows stable learning (as the RL agent does not diverge from the planned sequence of waypoints).

3.3 Real2Sim Transfer

In general, training a deep RL agent could be very sample inefficient, and we would like to minimize data collection on the real system when training the RL agent. In order to do that, we train a style transfer model that maps images taken in the real system to a simulator using a Convolutional Encoder-Decoder (CED). The CED learns a Real2Sim generative model $F : I^{\text{real}} \rightarrow I^{\text{sim}}$ by minimizing the following objective function:

$$\mathcal{L}_{\text{real2sim}}(F) = \mathbb{E}_{I^{\text{real}} \sim p_{\text{real}}} \left[\| F(I^{\text{real}}) - I^{\text{sim}} \|_1 \right]. \tag{5}$$

Our aim is to utilize the Real2Sim generator F to convert the real images I^{real} to simulated ones I^{sim}, so that we can optimize the policy only in the simulator. To do so, we first generate pairs of simulated depth images and waypoints as described in Sec. 3.1, and reproduce the same scenario in the real system: locate the obstacles and the goal to the same location, and control the robot to imitate the angles to make the pose look similar in simulation. Then, we train the CED by minimizing the objective function of (5).

4 Experimental Settings

This section briefly describes the experimental settings including the environments, hardware, and tasks used in experiments to validate the proposed method (more details in Appendix).

4.1 MDP

States The states of the system consist of current joint angles $\boldsymbol{\theta}_t \in \mathbb{R}^6$, angular velocities $\dot{\boldsymbol{\theta}}_t \in \mathbb{R}^6$ in configuration space, and waypoints $\boldsymbol{w}_t \in \mathbb{R}^{5\times 6}$ that lead the agent to the goal state. Since the robot state and waypoints do not contain spatial information about an environment, we also add latent variables $\boldsymbol{z}_t \in \mathbb{R}^{64}$ of the environment to the state, which is produced by using Variational Auto Encoder (VAE) [24], instead of using raw pixels to make the problem easier to solve [25, 26]. If the agent moves in Cartesian space, then we use the position and velocities of the end tool of the robot, instead of using joint angles. See Fig. 2 for more details of the network used for training.

Actions The action of the agent $\boldsymbol{a}_t$ is the vector of angular velocities $\dot{\boldsymbol{\theta}}$ for the next step. We define a time step described as Δt. Therefore, the angles of the next step $\boldsymbol{\theta}_{t+1}$ can be calculated as

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \boldsymbol{a}_t \Delta t. \tag{6}$$

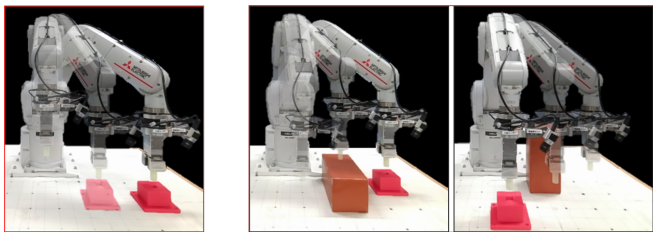
Termination Condition An episode terminates on the following three conditions: the joint angles of the agent $\boldsymbol{\theta}_t$ are sufficiently close to the goal state, or the number of steps of an episode is over a specified threshold, or the robot violates the known maximum joint angles.

4.2 Hardware

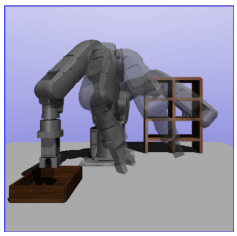
We use a MELFA RV-4FR robot, which is an industrial robot that has 6 degrees of freedom [27]. We operate the robot in a position control mode where position commands are sent to the robot every $\Delta t = 0.0035$ seconds, which is determined by the minimum operational time of the industrial robot we used in a real setting. The generated actions must ensure that joint angular accelerations are within a known specified range. Since lower acceleration is a desirable feature for a lot of industrial manipulators where direct torque control is not accessible, we penalize the angular acceleration when training our agent (see Sec. 3.2). For the robot to perceive the environment, we use Azure Kinect, which is an inexpensive, commercial RGBD camera. Since the maximum frame rate of the camera is 30 FPS, which is much slower than the control step of the robot's controller, we implement a distributed system using ROS, so that the RL component does not need to wait for images.

4.3 Tasks

In order to evaluate the proposed method, we prepare two different tasks: *Peg-Insertion* and *Pick-and-Place* . We use the MuJoCo [28] physics engine to simulate both of these tasks.



(a) *Peg-Insertion* environment.



(b) *Pick-and-Place* environment.

Figure 3: The trajectories obtained by the proposed method for the two different settings.

The *Peg-Insertion* task is for simulating peg insertion, navigating a robotic hand which holds a white peg, and inserting it into a red hole located on top of the table, while avoiding collision with a number of N^{obs} brown obstacles as illustrated in Fig. 3a. Since the grasping and insertion is not the focus of this work, an episode starts from a random initial position with the robot holding the peg, with the goal of reaching a random hole position, without collision with randomly located obstacles. A video of the implementation of the algorithm on the real system is provided in the following link: `https://youtu.be/hE-Ew59GRPQ`. The generated trajectories must ensure that joint angles and angular velocities that describe the trajectories are within a known specified range. We, however, fix the orientation of the robot hand, since we assume the hole is located on top of the horizontal table. As a result, the control input is the velocity in Cartesian space.

The *Pick-and-Place* task simulates a pick-and-place task, where the robot picks an object located in a two-row, three-column bookcase, and puts it into a box located on top of a table as depicted in Fig. 3b. Each of the cubes in the bookshelf is 150 mm deep, 150 mm high and 200 mm wide. The manipulator starts from an initial pose denoted by $\boldsymbol{\theta}_{\text{start}}$, and has to reach the box. The start position is sampled randomly from the center of each cube of the bookshelf, in order to imitate a grasping motion needed to grab an item.

For both tasks, we determine the success of an episode based on the following conditions: 1) the path does not collide with obstacles, and 2) the robot arm reaches the goal position closely enough, specifically we define the threshold to be $d_{\text{goal}} = 50$ [mm], and 3) the episode length does not exceed a pre-defined length of $T = 300$ steps.

5 Experimental Results

In this section, we try to quantify generalization and performance of the proposed technique.

5.1 Performance of the Waypoints Generator on the Real System

To evaluate the quality of the waypoints generation module, we use two different metrics – 1) success rate : success is declared in Sec. 4.3 and 2) waypoint error : the average error in reproducing the waypoints generated by the Bi-directional RRT*.

As described in Sec. 3.3, our aim is to train our policy in simulation, and we achieve this by learning a function that maps real images to simulated ones using CED to avoid exploration on the real system. Thus, we measure the success rate on different numbers of training data collected in the real system and compare the results of supervised learning of only real data (Real-Only) and our approach of learning the waypoints generator only in simulation (CED + Sim). We also tested other approaches, such as supervised learning of only simulated data, CycleGAN [29], unsupervised domain adaptation [30], fine-tuning the simulation-based model with real data [31], and mixing the simulated and real images. We found that mixing the simulation and real data works best among these approaches. Therefore, we also compare it, denoting it as "Real + Sim" in the experiment. We generate $50k$ images in simulation, as it is relatively inexpensive to generate images in simulation. More details about the evaluation experiments are available in the Appendix A.2.

Table 1 shows the comparison between different approaches using the two different metrics. We can clearly see that the combination of CED and waypoints generator trained in simulation successfully

Table 1: Success rate and average waypoints errors of the *Peg-Insertion* task in the real system. The bold numbers show the highest success rate or minimum waypoints error. The percentages refer to percent of training data, not success rate or waypoints error.

		Success rate			Waypoints error [mm]		
		10%	50%	100%	10%	50%	100%
$N^{\text{obs}} = 0$	Real-Only	0.08	0.06	0.10	14.4	16.7	13.1
	Real + Sim	**1.00**	**1.00**	**1.00**	**3.2**	**2.8**	**2.5**
	CED + Sim (Ours)	0.80	**1.00**	**1.00**	5.7	3.3	3.4
$N^{\text{obs}} = 1$	Real-Only	0.00	0.00	0.00	12.2	12.8	13.8
	Real + Sim	0.42	0.90	0.71	13.8	**3.6**	5.1
	CED + Sim (Ours)	**0.80**	**0.98**	**0.94**	**9.3**	7.9	**2.8**

Table 2: Performance on generalization task in the real system. The results are averaged over 50 trials.

	Success rate	
	Random obstacles	Moving goal
Real-Only	0.00	0.00
CED + Sim (Ours)	**0.90**	**1.00**

generalizes to a new environment, even if the training data decreases to 10%. Thus, we can train policies in simulation with very little data required from the real system.

5.2 Generalization to Novel Environments

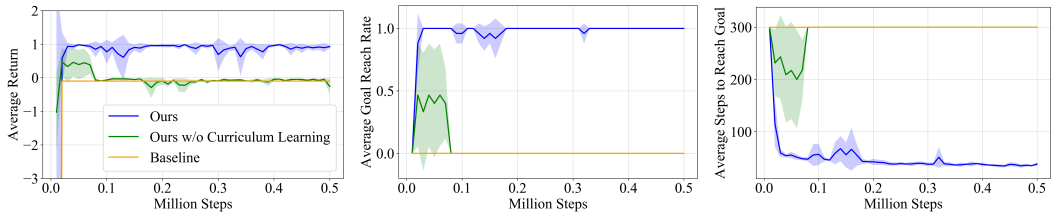
Next, we evaluated the generalization capability of our method with respect to novel environments with our end-to-end policy, i.e., including a low-level action generation module using RL. As for the test environment, we prepared two different settings and tested over 50 trials on each setting in the real system. First, we tested generalization on a moving goal on the $N^{\text{obs}} = 0$ environment, which changes its goal position during execution of the robot, before the robot has reached the previous position. Second, we changed the positions of the obstacles and the goal during execution by stopping the robot in order to ensure the safety of the operator. The definition of success is whether the agent reaches within d_{goal} of the goal position. We compared our method of CED + Sim against the Real-Only model, both of which use 100% of data as in the previous experiment.

Table 2 shows the result of the experiments. It suggests our method can reactively track the moving goal positions, as depicted in the Fig. 3a, and also reactively change the path with more complex settings of changing both obstacle and goal positions and/or orientation during an episode. Videos that show this behavior on the real system are provided in the `https://youtu.be/hE-Ew59GRPQ`.

5.3 Improving Sample Efficiency and Final Performance using Path-Conditioned RL

As described in Sec. 3.2, we believe that path-conditioned RL lets the agent explore more efficiently, and thus leads to better sample efficiency. Moreover, we think the RL-based low-level controller can generate optimal trajectories in terms of minimizing angular accelerations and time to reach a goal state. To verify these, we conducted the following experiments on the *Pick-and-Place* environment in simulation.

Sample Efficiency We tested whether the proposed method can improve the sample efficiency compared with a baseline RL agent, which does not contain waypoints in the state. Therefore, the state of the baseline agent will be the concatenation of the latent variables $\boldsymbol{z}_t$ obtained by using VAE, and robot's state $\boldsymbol{s}_t^{\text{robot}}$, and trained only from the reward function that removes waypoints-related terms, i.e., removed $h(\boldsymbol{s}_t, \boldsymbol{a}_t, \boldsymbol{w}_t)$ from Eq. (2). Both agents were trained with the Soft Actor Critic



(a) Episodic return (b) Goal reach rate (c) Steps to Reach Goal

Figure 4: Training curves on each training type. The solid lines represent average returns over five instances with different random seeds. The shaded region represents the standard deviation of the five instances. Our approach outperforms baseline on both sample efficiency and final performance.

Table 3: Time [sec] to reach goal, and average angular acceleration [rad/sec^2] during execution.

		Pick-and-Place					
		1	2	3	4	5	6
Time [sec]	Baseline	0.89	0.92	0.80	0.96	1.25	1.26
	Ours	**0.48**	**0.51**	**0.59**	**0.52**	**0.57**	**0.54**
Angular Accelerations [rad/sec^2]	Baseline	696	672	687	702	4365	4539
	Ours	**124**	**125**	**163**	**135**	**153**	**136**

(SAC) algorithm [32] with the same hyperparameters as in the paper. For a fair comparison, we evaluated both agents with the same reward function as the baseline agent.

Figure 4 demonstrates the training curves of resulting episodic returns, goal reach rate, and number of steps needed to reach a goal state. Comparing our method with the baseline, we see that solving the reaching task only from robot's state and latent variables of the depth image is difficult, and the use of the waypoints can make the original task substantially easier. It also improves the sample efficiency and achieves better final performance. Next, we compared the result of our path-conditioned RL with and without curriculum learning. The result suggests that just using waypoints does not successfully converge to achieve the desired task, but curriculum learning plays an important role to achieve the task. We believe that using curriculum learning makes training more stable, thus leading to faster convergence (see Sec. 3.2).

Quality of Generated Trajectories Finally, we evaluated the quality of the generated trajectories in terms of average angular accelerations during an episode, and time to reach a goal state. We compared our method against a combination of Bi-directional RRT* and a PID controller to follow the generated path as a baseline. We use the PID controller in the officially provided simulator of the robot used in the real experiment [33]. We evaluated 6 different scenarios, each of which starts from imitating to pick an object in a cube (1 to 6 corresponds to from top left to bottom right in Fig. 3b), and reach a box located in the same position for fair comparison.

Table 3 shows the results. It shows that the use of low-level RL-based controller does improve the quality of the generated path in terms of both time to reach the goal state and in minimizing angular accelerations during an episode.

6 Conclusion

In this paper, we presented an end-to-end policy that can perform reactive planning for robotic systems in dynamic environments. The proposed method uses kinematic planning, supervised learning, and RL to train an agent that can do reactive planning using input from a depth camera. To minimize training with the real system, we use a real2sim transfer function, and train the agent entirely in simulation with very few real data samples. We show that the proposed algorithm outperforms several baseline methods, and allows a 6-DoF manipulator arm to perform reactive planning in complex dynamic environments in an end-to-end fashion.

References

[1] S. M. LaValle. Planning algorithms. Cambridge university press, 2006.

[2] L. E. Kavralu, P. Svestka, J.-C. Latombe, and M. H. Overmars. Probabilistic roadmaps for path planning in high-dimensional configuration spaces. IEEE Trans. Robotics and Automation, 12 (4), 1996.

[3] N. Ratliff, M. Zucker, J. A. Bagnell, and S. Srinivasa. Chomp: Gradient optimization techniques for efficient motion planning. In 2009 IEEE International Conference on Robotics and Automation, pages 489–494, May 2009. doi:10.1109/ROBOT.2009.5152817.

[4] M. Kalakrishnan, S. Chitta, E. Theodorou, P. Pastor, and S. Schaal. Stomp: Stochastic trajectory optimization for motion planning. In 2011 IEEE International Conference on Robotics and Automation, pages 4569–4574, May 2011. doi:10.1109/ICRA.2011.5980280.

[5] J. Bruce and M. M. Veloso. Real-time randomized path planning for robot navigation. In Robot Soccer World Cup, pages 288–295. Springer, 2002.

[6] R. Gayle, A. Sud, M. C. Lin, and D. Manocha. Reactive deformation roadmaps: motion planning of multiple robots in dynamic environments. In 2007 IEEE/RSJ International Conference on Intelligent Robots and Systems, pages 3777–3783. IEEE.

[7] D. Kappler, F. Meier, J. Issac, J. Mainprice, C. G. Cifuentes, M. Wüthrich, V. Berenz, S. Schaal, N. Ratliff, and J. Bohg. Real-time perception meets reactive motion generation. IEEE Robotics and Automation Letters, 3(3):1864–1871, 2018.

[8] D. Morrison, P. Corke, and J. Leitner. Learning robust, real-time, reactive robotic grasping. The International Journal of Robotics Research, 39(2-3):183–201, 2020. doi:10.1177/0278364919859066. URL https://doi.org/10.1177/0278364919859066.

[9] A. Faust, K. Oslund, O. Ramirez, A. Francis, L. Tapia, M. Fiser, and J. Davidson. Prm-rl: Long-range robotic navigation tasks by combining reinforcement learning and sampling-based planning. In Proceedings of IEEE International Conference on Robotics and Automation (ICRA), 2018.

[10] G. Thomas, M. Chien, A. Tamar, J. A. Ojea, and P. Abbeel. Learning robotic assembly from cad. In Proceedings of IEEE International Conference on Robotics and Automation (ICRA). IEEE, May 2018. ISBN 9781538630815. doi:10.1109/icra.2018.8460696. URL http://dx.doi.org/10.1109/icra.2018.8460696.

[11] H.-T. L. Chiang, A. Faust, M. Fiser, and A. Francis. Learning navigation behaviors end-to-end with autorl. IEEE Robotics and Automation Letters, 4(2):2007–2014, 2019.

[12] K. Ota, D. K. Jha, T. Oiki, M. Miura, T. Nammoto, D. Nikovski, and T. Mariyama. Trajectory optimization for unknown constrained systems using reinforcement learning. In 2019 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pages 3487–3494, Nov 2019. doi:10.1109/IROS40897.2019.8968010.

[13] B. Lötjens, M. Everett, and J. P. How. Safe reinforcement learning with model uncertainty estimates. In 2019 International Conference on Robotics and Automation (ICRA), pages 8662–8668. IEEE, 2019.

[14] S. Levine, C. Finn, T. Darrell, and P. Abbeel. End-to-end training of deep visuomotor policies. J. Mach. Learn. Res., 17(1):1334–1373, Jan. 2016. ISSN 1532-4435.

[15] S. Levine and V. Koltun. Guided policy search. In Proceedings of International Conference on Machine Learning (ICML), pages 1–9, 2013.

[16] S. Nasiriany, V. Pong, S. Lin, and S. Levine. Planning with goal-conditioned policies. In Proceedings of Advances in Neural Information Processing Systems (NIPS), pages 14843–14854, 2019.

[17] J. Chang, N. Kumar, S. Hastings, A. Gokaslan, D. Romeres, D. Jha, D. Nikovski, G. Konidaris, and S. Tellex. Learning deep parameterized skills from demonstration for re-targetable visuomotor control. arXiv preprint arXiv:1910.10628, 2019.

[18] K. Ota, Y. Sasaki, D. K. Jha, Y. Yoshiyasu, and A. Kanezaki. Efficient exploration in constrained environments with goal-oriented reference path. In Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), 2020.

[19] A. Shrivastava, T. Pfister, O. Tuzel, J. Susskind, W. Wang, and R. Webb. Learning from simulated and unsupervised images through adversarial training. In Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pages 2107–2116, 2017.

[20] J. Hoffman, E. Tzeng, T. Park, J.-Y. Zhu, P. Isola, K. Saenko, A. Efros, and T. Darrell. Cycada: Cycle-consistent adversarial domain adaptation. In Proceedings of International Conference on Machine Learning (ICML), pages 1989–1998, 2018.

[21] K. Bousmalis, N. Silberman, D. Dohan, D. Erhan, and D. Krishnan. Unsupervised pixel-level domain adaptation with generative adversarial networks. In Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pages 3722–3731, 2017.

[22] K. Rao, C. Harris, A. Irpan, S. Levine, J. Ibarz, and M. Khansari. Rl-cyclegan: Reinforcement learning aware simulation-to-real. In Proceedings of IEEE Conference on Computer Vision and Pattern Recognition (CVPR), pages 11157–11166, 2020.

[23] M. Jordan and A. Perez. Optimal bidirectional rapidly-exploring random trees. , CSAIL, MIT, Cambridge, MA, 2013.

[24] D. P. Kingma and M. Welling. Auto-encoding variational bayes. In Proceedings of International Conference on Learning Representations (ICLR), 2013.

[25] D. Yarats, A. Zhang, I. Kostrikov, B. Amos, J. Pineau, and R. Fergus. Improving sample efficiency in model-free reinforcement learning from images. arXiv preprint arXiv:1910.01741, 2019.

[26] A. Kendall, J. Hawke, D. Janz, P. Mazur, D. Reda, J.-M. Allen, V.-D. Lam, A. Bewley, and A. Shah. Learning to drive in a day. In 2019 International Conference on Robotics and Automation (ICRA), pages 8248–8254. IEEE, 2019.

[27] Melfa rv-fr. http://www.mitsubishielectric.com/fa/products/rbt/robot/. Accessed: 2020-07-15.

[28] E. Todorov, T. Erez, and Y. Tassa. Mujoco: A physics engine for model-based control. In Proceedings of IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS), pages 5026–5033. IEEE, 2012.

[29] J.-Y. Zhu, T. Park, P. Isola, and A. A. Efros. Unpaired image-to-image translation using cycle-consistent adversarial networks. In Proceedings of International Conference on Computer Vision (ICCV), pages 2223–2232, 2017.

[30] Y. Ganin and V. Lempitsky. Unsupervised domain adaptation by backpropagation. In Proceedings of International Conference on Machine Learning (ICML), pages 1180–1189, 2015.

[31] S. R. Richter, V. Vineet, S. Roth, and V. Koltun. Playing for data: Ground truth from computer games. In Proceedings of European Conference on Computer Vision (ECCV), pages 102–118, 2016.

[32] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine. Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor. In Proceedings of International Conference on Machine Learning (ICML), pages 1861–1870, 2018.

[33] Rt toolbox3. https://eu3a.mitsubishielectric.com/fa/en/products/rbt/robot/rt_toolbox3. Accessed: 2020-07-26.

[34] Deepmind control suite. https://github.com/deepmind/dm_control/blob/master/dm_control/mujoco/engine.py#L794-L801. Accessed: 2020-05-11.

[35] J. Johnson, A. Alahi, and L. Fei-Fei. Perceptual losses for real-time style transfer and super-resolution. In European Conference on Computer Vision, 2016.

A Experimental Details

A.1 Real System

In this section, we provide more details of the real system which we use for experiments in Sec. 5.1 and Sec. 5.2.

Depth Sensor As described in Sec. 4.2, we use Azure Kinect for the real experiments, and reproduce the same setup in the simulator using MuJoCo. For the MuJoCo simulator, the raw pixel values $I^{\text{sim-raw}}$ are first converted to real distance values. Referring to [34], the conversion can be computed as:

$$I^{\text{sim-dist}}_{i,j} = p^{\text{near}}/(1 - I^{\text{sim-raw}}_{i,j} \times (1 - p^{\text{near}}/p^{\text{far}})), \tag{7}$$

where $I^{\text{sim-raw}} \in [0,1]^{W \times H}$ is the depth image whose width and height is W and H, and the p^{near} and p^{far} are camera parameters. Then, we clip only regions of interest distance $d_{\text{min}}, d_{\text{max}}$ as:

$$I^{\text{sim}}_{i,j} = \max(\min(I^{\text{sim-dist}}_{i,k}, d_{\text{max}}), d_{\text{min}}), \tag{8}$$

where we specifically use $(d_{\text{min}}, d_{\text{max}}) = (0.5, 1.5)$ [m] for our setting. Finally, the pixel values are normalized so that the minimum value becomes 0 and the maximum value becomes 1.

Obstacles We use the same shape obstacle over all experiments whose size is $(W \times H \times D) = (122, 248, 114)$ [mm].

Peg and Hole The peg and hole that we used for the real experiments are printed using a 3D-printer with the diameter of $\phi^{\text{peg}} = 20$ [mm], and $\phi^{\text{hole}} = 21$ [mm].

A.2 Peg Insertion Experiments

Robot's State Space The start, goal, obstacle's positions are randomly sampled from a position defined in Table 4 when an episode starts. The randomization area for generating an optimal path using Bi-directional RRT* is also defined in Table 4.

Table 4: The randomization area from which start, goal, obstacle positions, and robot state are sampled in the *Peg-Insertion* task. Note that the orientation of the obstacles have three choices (rotate $\pi/2$ along Y or Z axes), and the height (Z-axis value) of the obstacles and the goal does not change due to a geometrical constraint.

Randomization Target	Axis	Min. [m]	Max. [m]
Start, Goal, Obstacle, Robot's state	X	0.20	0.40
	Y	-0.40	0.40
	Z	0.05	0.30

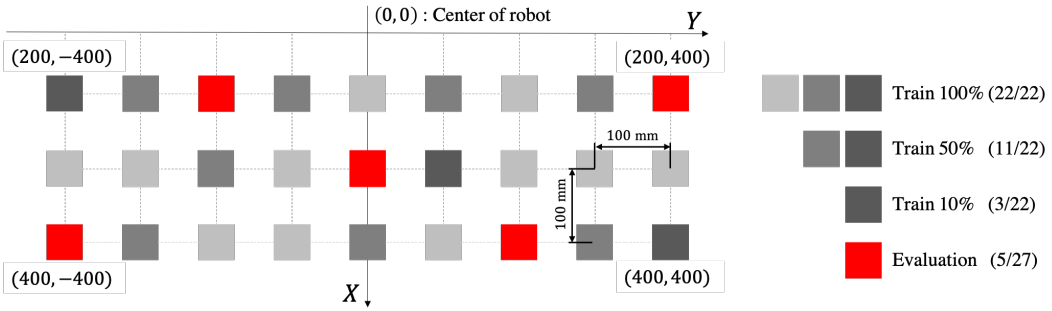


Figure 5: Goal positions [mm] where each training and evaluation data are sampled. The position is sampled from a grid of $9 \times 3 = 27$ with each grid size is 100×100 [mm].

Settings for the Experiment in Sec. 5.1 For the experiment in Sec. 5.1, where we evaluate the performance of the waypoints generator with respect to the number of data collected in the real system, we carefully divide the dataset into training and evaluation while not allowing the each data to overlap each other. In order to do that, we sample goal positions for each dataset from discrete positions of a grid of $9 \times 3 = 27$ with each grid size is 100×100 [mm], and results in the total grid size is 200×800 [mm]. From the grid, we assigned $22/27$ of goal positions for training, and the others for evaluation so that the distribution of the goal positions in the two different dataset do not overlap. Furthermore, we divided the training dataset of $22/27$ goal positions into three subsets, which roughly contains 10%, 50%, 100% of data sampled from different goal positions again, whose data size is roughly 10K data for 100% dataset. It is noted that such a discrete grid is only created for evaluation purposes.

Since the experiment in Sec. 5.1 does not include subsequent low-level action module to evaluate the waypoints generator, we move the robot to the closest waypoint by sending a position command to the real system (without using the RL trained policy).

Settings for the Experiment in Sec. 5.2 As for the CDE model used in the generalization experiments in Sec. 5.2, we use the same model with the one trained using 100% as defined in the previous section.

The distribution of the goal position, however, is different from the previous experiment: the goal is sampled continuously from the randomization area defined in Table 4, i.e., not categorically from the grid in Fig. 5.

A.3 Pick-and-Place Experiments

Robot's State Space The goal position is randomly sampled from a range defined in Table 5 when an episode starts. The start position is randomly sampled from categories of 6 different cube location, where 1 to 6 corresponds to from top left to bottom right in Fig. 3b as described in Sec. 5.3. The randomization area for generating an optimal path using Bi-directional RRT* is also defined in Table 4.

Settings for the Experiment in Sec. 5.3 In order to compare the generated trajectories of ours against the PID controller in the officially provided simulator, we fix a goal position to be $\boldsymbol{x}^{\text{goal}} = (0.4, -0.2)$.

B Training Details

As depicted in Fig. 2, we train four neural networks models to accomplish the task of reactive planning in the real system: 1) the Convolutional Encoder-Decoder (CDE) for Real2Sim transfer, 2) the VAE model that extracts latent variables of an environment, 3) the waypoints generator which generates waypoints $\boldsymbol{w}$ that roughly guides the agent to a goal state, and 4) the RL-based low-level controller that produces an optimal action in terms of time to reach goal and generating a smoother

Table 5: The randomization area from which start, goal, and robot's state are sampled. Note that the height (Z-axis value) of the goal does not change due to a geometrical constraint.

Randomization Target	Axis	Min. [deg]	Max. [deg]
Robot's state	θ_1	-240	240
	θ_2	-120	120
	θ_3	0	164
	θ_4	-200	200
	θ_5	-120	120
	θ_6	-360	360
Goal	X	0.40	0.45
	Y	-0.20	0.30

Table 6: Architecture of VAE.

	Shape	Layer size	Stride	Type
Encoder	$64 \times 64 \times 1$	-	-	Input: depth image I^{sim}
	$32 \times 32 \times 16$	$3 \times 3 \times 16$	2	Convolution + ReLU
	$16 \times 16 \times 32$	$3 \times 3 \times 32$	2	Convolution + ReLU
	$8 \times 8 \times 64$	$3 \times 3 \times 64$	2	Convolution + ReLU
	4096	-	-	Flatten
	64×2	-	-	Fully Connected: output means and variances
Decoder	64	-	-	Latent variables $\boldsymbol{z}$
	4096	-	-	Fully Connected + ReLU
	$8 \times 8 \times 64$	-	-	Reshape
	$16 \times 16 \times 64$	$3 \times 3 \times 64$	2	Deconvolution + ReLU
	$32 \times 32 \times 32$	$3 \times 3 \times 32$	2	Deconvolution + ReLU
	$64 \times 64 \times 16$	$3 \times 3 \times 16$	2	Deconvolution + ReLU
	$64 \times 64 \times 1$	$3 \times 3 \times 1$	1	Convolution: output reconstructed image $\hat{I}^{\text{sim}}$

trajectory. The four modules are trained separately, and we train each module while freezing the parameters of the other modules. In this section, we provide details about these models.

B.1 Real2Sim Transfer

We adopt our architecture of the Real2Sim transfer CDE model $F : I^{\text{real}} \to I^{\text{sim}}$ from [35] referring to [29] with the 2 residual blocks instead of 9 for speeding up computation. We train the CDE model by minimizing Eq. (5) with Adam optimizer for 1000 epochs with a learning rate of 0.0002 as in [29].

B.2 VAE

The VAE model we train for extracting latent variables $\boldsymbol{z}_t$ of an environment consists of a convolutional encoder-decoder network, i.e., a convolutional encoder g_ϕ maps a depth image of the simulator I_t^{sim} to a low-dimensional latent variables $\boldsymbol{z}_t$, and a deconvolutional decoder f_θ reconstructs $\boldsymbol{z}_t$ back to the original image I_t^{sim}. Table 6 show the architecture of the VAE.

The VAE model is trained by maximizing the following objective:

$$J_{\text{VAE}} = \mathbb{E}_{I_t^{\text{sim}} \sim p^{\text{sim}}} \left[\mathbb{E}_{\mathbf{z}_t \sim q_\phi\left(\mathbf{z}_t \mid I_t^{\text{sim}}\right)} \left[\log p_\theta \left(I_t^{\text{sim}} \mid \mathbf{z}_t \right) \right] - \beta D_{\text{KL}} \left(q_\phi \left(\mathbf{z}_t \mid I_t^{\text{sim}} \right) \| p \left(\mathbf{z}_t \right) \right) \right] \tag{9}$$

with the $50K$ size of dataset collected in the simulator as described in Sec. 5.1, and Adam optimizer for 1000 epochs with a learning rate of 0.0001. For more details of VAE, interested readers are referred to [24].

Table 7: Architecture of the waypoints generator. The $N^{\text{state}} = 3$ if the robot moves in Cartesian space, or $N^{\text{state}} = 6$ if in configuration space.

Shape	Layer size	Stride	Type
$64 + N^{\text{state}}$	-	-	Concatenate latent variables $\boldsymbol{z}$ and robot's state $\boldsymbol{s}^{\text{robot}}$
256	-	-	Fully Connected + ReLU
$N^{\text{waypoints}} \times N^{\text{state}}$	-	-	Fully Connected: output waypoints $\boldsymbol{w}$

Table 8: Coefficients of each reward term λ_i used in our experiments.

Term	Value	Description
$\lambda_1 \mathbb{I}_{\text{collision}}$	-0.1	Obstacle collision penalty
$\lambda_2 \mathbb{I}_{\text{goal}}$	1.0	Goal reach reward
$\lambda_3 \|\ddot{\boldsymbol{\theta}}\|$	-0.0001	Angular acceleration penalty
$\lambda_4 d_{\text{path}}$	0.1	Distance penalty to the closest waypoints
$\lambda_5 n_{\text{progress}}$	-1.0	Distance reward going toward the goal state

B.3 Waypoints Generator

As described in Sec. 3.1, the waypoints generator consists of a CNN that takes a depth image I^{sim} and a robot's state $\boldsymbol{s}^{\text{robot}}$, and generates a short horizon path $\boldsymbol{w}$ that consists of $N^{\text{waypoints}} = 5$ waypoints, each of which has 3- or 6-dimensional relative position or angle with respect to the current state of the agent. Table 7 shows the architecture of the waypoints generator. The depth image is converted into latent variables using the VAE model described in the previous section, and the concatenation of the latent variables $\boldsymbol{z}$ and robot's state $\boldsymbol{s}^{\text{robot}}$ is inputted into the waypoints generator model to produce the waypoints.

In order to train the model, we first collect $50K$ pairs of depth images, robot's states, and waypoints generated by using Bi-directional RRT* with random start, goal, and obstacles position. We then train the model with Adam optimizer for 500 epochs with a learning rate of 0.001.

B.4 Reinforcement Learning

This section summarizes extra details of the reward function and the curriculum learning setting we used in Sec. 5.3.

Reward Table 8 summarizes the coefficients of the reward terms defined in (3), and (4).

Curriculum Learning Here we explain the curriculum learning setting we used in Sec. 5.3. First, we randomly sample the number of steps $N^{\text{waypoints-step}}$, which specifies how many steps we rollout using the waypoints generators from an initial state. Then, we iteratively produce the waypoints for $N^{\text{waypoints-step}}$-steps by making use of the waypoints $\boldsymbol{w}_t$ to evolves the environment at each time steps t. More specifically, we compute the next joint angles of the robot as:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \boldsymbol{w}_t^{\text{closest}}, \tag{10}$$

where the $\boldsymbol{w}^{\text{closest}} \in \mathbb{R}^6$ is the closest waypoints, and then we set the robot's state to the internal property of the simulator. Algorithms 1 shows the procedure of the curriculum learning setting we used in .

Algorithm 1 Curriculum Learning

Randomly sample number of steps to use waypoints generator $N^{\text{waypoints-step}}$ from $\{0, ...15\}$
for $t = 0$ to $N^{\text{waypoints-step}} - 1$ **do**
    Generate waypoints using the waypoints generator as: $\boldsymbol{w}_t = G(I_t^{\text{sim}}, \boldsymbol{s}_t^{\text{robot}})$
    Reset robot state to the closest waypoints $\boldsymbol{w}_t$
end for
Start an episode with an initial state of $\boldsymbol{s}_{\text{waypoints-step}}^{\text{robot}}$