Improving Mobile Robot Navigation with Deep Neural Control and Search-based Motion Planning

Max Ferguson
Department of Civil and Environmental Engineering
Stanford University
Stanford, CA, United States
Email: maxferg@stanford.edu

***Abstract*—In recent times, there has been a large amount of excitement surrounding the use of reinforcement learning systems for robot control and navigation. A significant research effort has been directed at the problem of visual navigation. However, most of these algorithms encounter difficulty navigating complex trajectories, especially when the target is far from the start position. We propose a new method which combines reinforcement learning with traditional search-based path-planning, to solve long navigation problems. This approach, referred to as Deep Neural Robot Control, performs exceptionally well on several robot navigation tasks, often learning a human-like policies for navigation and collision avoidance. We also introduce framework for zero-shot transfer of learned policies from simulation to the physical world. Using this framework, we demonstrate that a trained DNRC agent can provide a good control policy for a real mobile robot.**

Code: https://github.com/maxkferg/dbm
Video: https://youtu.be/cZWFUaci8jc

1. Introduction

Autonomous control and path-planning have a number of real-world applications, especially in the areas of autonomous driving and robotics. Traditional control algorithms, such as the linear-quadratic regulator (LQR) require that all of the states of the system are measurable. Modern reinforcement learning (RL) techniques provide the opportunity to directly learn from the environment, allowing many of the traditional assumptions to be relaxed. Many state-of-the-art systems leverage an RL algorithm such as A3C to navigate mazes, using only image pixels [1]. While functional, these systems encounter difficulty navigating complex trajectories, especially when the target is far from the starting position. To overcome this problem, we propose a new end-to-end learning algorithm, Deep Neural Robot Control (DNRC), that combines search-based path-planning with an off-policy model-free RL controller. Our preliminary experiments show that this system performs well when navigating complex environments containing one or more moving robots.

The overarching goal of this work is to develop a control and navigation system that performs well in both simulated and physical environments, as illustrated in Figure 1. We have already addressed some problems related to simulation-real transfer and robot perception in related works [2, 3]. For the purpose of this paper, a good navigation algorithm should ensure the robot moves to the target position quickly and safely. Specifically, a good mobile robot control system should:

- Avoid colliding with walls and stationary objects
- Anticipate the movement of people and other robots, and act to avoid collision wherever possible
- Minimize sudden acceleration and rotation of the robot, so as to conserve battery resources
- Reach the target destination as quickly as possible, given the above conditions

The remainder of this document is organized as follows: In Section 2, we describe related works in both the machine learning and traditional robotics domains. In Section 3, provide the mathematical background and notation used in this paper. In Section 4, we briefly introduce deep neural robot control. Finally, we explain how the learned policy was transferred to a real robot, using a simulated environment as an intermediate representation.

2. Related Work

Recent work in visual navigation has focused on navigating in both indoor [1, 4] and outdoor [5] environments using deep reinforcement learning. Researchers have proposed methods that leverage 2D maps [6] as well as methods that navigate on visual input alone [7]. In recent times, a large research effort has been directed at the problem of visual navigation [1, 4, 7]. A common benchmark for maze-solving algorithms is provided in [8]. Our approach is most closely related to hierarchical RL methods, which tend to perform well on maze environments [9]. A related but slightly different problem is that RL agents often have difficulty exploring large state spaces, which has been addressed in a recent work [10]. Finally, in DNRC we use intrinsic motivation in the form of checkpoints to guide the agent to the goal, as presented in [11].

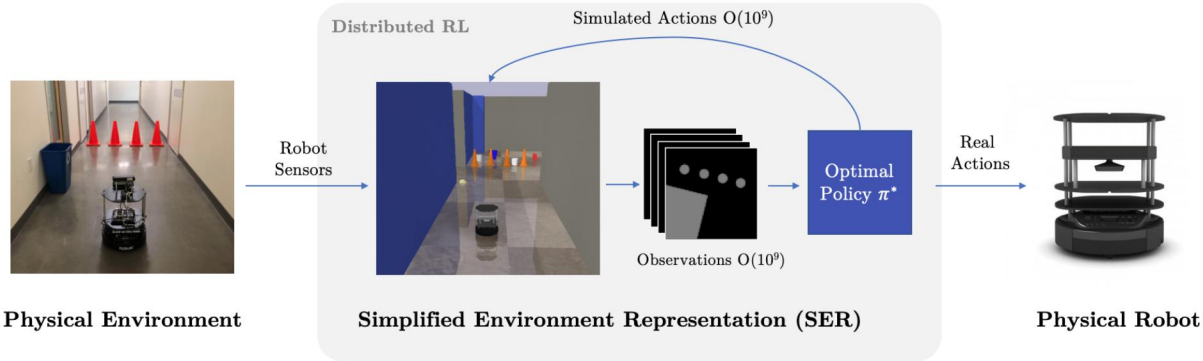


Figure 1. Simplified Environment Representation (SER) framework for zero-shot transfer of control policies from simulation to the real world. Sensors on the mobile robot are used to generate a simplified model of the environment. Reinforcement learning is used to find a good policy in this environment. Actions from this policy are sent directly to the physical robot.

In traditional robotics, the navigation problem is broken into two parts, namely motion planning and control. In motion planning, a search algorithm is used to find the optimum path from the target to the goal. A control algorithm, such as a proportional-integral-derivative controller (PID), is then used to guide the agent along the path. This approach has shown to be effective in a number of real world scenarios [12]. However, it suffers from several limitations. Firstly, the path-planning step is often too computationally demanding to be performed in real-time [13]. Secondly, this approach is difficult to apply in environments which are stochastic. Finally, this approach is difficult to apply when the agent must also avoid other moving objects. An exhaustive classification of traditional path-planning approaches can be found in a survey by Hwang and Ahuja [13].

3. Background

In reinforcement learning (RL), we generally consider an agent interacting with an environment through a sequence of observations, actions and rewards. The goal of the agent is to select actions so as to maximize the cumulative future reward. For robot navigation, the environment is generally modeled as a Partially Observed Markov Decision Process (POMDP). At each time period, the environment is in some state $s \in S$. The robot takes an action $a \in A$, which causes the environment to transition to state s' with probability $T(s'|s,a)$. At the same time, the agent receives an observation $o \in \Omega$. Finally, the agent receives a reward r equal to $R(s,a)$. The process then process repeats. The goal is for the robot to choose actions at each time step that maximize its expected future discounted reward: $E\left[\sum_{t=0}^{\infty} \gamma^t r_t\right]$, where r_t is the reward earned at time t. The discount factor γ determines how much immediate rewards are favored over more distant rewards.

A model-free RL algorithm develops a policy π, for acting in the environment, without explicitly learning a model of the environment [14]. In this work, we use a distributed implementation of the Deep Deterministic Policy Gradient (DDPG) algorithm to learn a robot control policy [15]. DDPG is a model-free, off-policy actor-critic algorithm. For continuous control, parametrized policies π_ϕ can be updated by taking the gradient of the expected return $\nabla_\phi J(\phi)$. In actor-critic methods, the policy, known as the actor, can be updated through the deterministic policy gradient algorithm. [16]. Recently, a number of changes have been proposed to improve the stability of the original DDPG algorithm [17], all of which are included in this work.

4. Neural Robot Controller

In this section we develop a hybrid approach for mobile robot control and navigation, namely Deep Neural Robot Control (DNRC). Modern reinforcement learning algorithms are particularly effective at solving continuous control problems with a short horizon. This makes reinforcement learning ideal for completing short robotic maneuvers such as avoiding moving objects. In DNRC, we find a path from the current position to the goal, by searching over a coarsely discretized representation of the state space, using the A* search algorithm. To reduce the size of the state space, we introduce a heuristic function $h : S \rightarrow \bar{S}$ which maps points from the complex, potentially continuous, state space S, to a simpler discrete representation $\bar{S}$. In practice, this heuristic function just discards inessential dimensions of S, such as robot velocity. Once a path through S has been established, a model-free RL algorithm is used to follow this path.

Assume an agent exists at point p in an environment with state s and target location t. We use the DDPG algorithm to learn the action-value function $Q_\pi(s,a,t)$ and policy $\pi_\phi(s,t)$ for navigating from p to t. We simultaneously learn another function $\bar{V}(h(p),h(t))$ that estimates the cost of moving from p to t. Intuitively, $\bar{V}(h(p),h(t)) \approx Q_\pi(s,a,t)$ for any action a that does not significantly change the robot state. Hence, a single neural network with two separate heads is used to predict both $\bar{V}(h(p),h(t))$ and $Q_\pi(s,a,t)$.

At the start of each episode, the A* search algorithm is used to find the shortest distance from $h(p)$ to $h(t)$, relying on the $\bar{V}_\pi(h(p),h(t))$ as a distance metric. This path is mapped back to a path in Euclidean space. Finally, we place

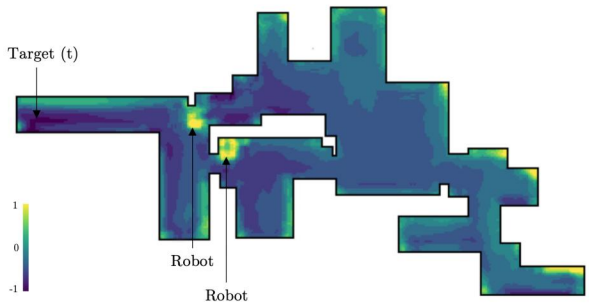


Figure 2. Descretized state space showing expected value of each node, in the house environment

checkpoints along the shortest path and execute $\pi_\phi(s,t)$ repeatedly to move between each checkpoint towards the goal.

Our chosen heuristic function, $h(p_i)$ maps the full environment state to a simple vector describing the position of the robot and nearby obstacles. Figure 2 shows the Value function projected into Euclidean coordinates, providing some intuition for why this method works. Specifically, we can imagine using A* to find the optimum path across the value surface shown in the Figure.

4.1. Mobile Robot Control

We test DNRC in a number of challenging 3D navigation environments, each simulated with the PyBullet physics engine [18]. DNRC must learn to control a two-wheeled mobile robot, based on the TurtleBot platform. The robot is controlled by specifying the forward velocity $v \in [-1, 1]$ and the rotational velocity $r \in [-1, 1]$. At each timestep, the agent must choose an action $a = [v, r]$, based on an observation $o(s)$ of the current state. Many recent works have focused on using raw pixel input from a robot-mounted camera as the input state. However, we decided against this approach as it (1) does not transfer well from simulation to the real world, and (2) ignores additional information about the building such as a plan view map. Instead, we provide the agent with raw sensor data and a set of bitmap tiles describing the surroundings:

$$o(s) = \begin{bmatrix} \dot{x}_{robot} + \epsilon_4 \\ \dot{y}_{robot} + \epsilon_5 \\ \dot{\theta}_{robot} + \epsilon_5 \\ x_{target} + \epsilon_6 \\ y_{target} + \epsilon_7 \\ \mathcal{I}_{walls} \\ \mathcal{I}_{objects} \\ \mathcal{I}_{robots} \\ \mathcal{I}_{checkpoints} \\ \mathcal{I}_{targets} \end{bmatrix} \tag{1}$$

where $(\dot{x}_{robot},\ \dot{y}_{robot},\ \dot{\theta}_{robot})$ is the robot velocity, and (x_{target}, y_{target}) is the position of the target relative to the robot. $\mathcal{I}_{walls}$ is a 64×64 matrix describing the position of walls near the robot. Specifically, the $\mathcal{I}_{walls}$ can be thought of a plan-view image of the environment, centered on the robot, where walls are colored white and the floor is colored black. $\mathcal{I}_{objects}$, $\mathcal{I}_{robots}$, $\mathcal{I}_{checkpoints}$, and $\mathcal{I}_{targets}$ are similar images describing the position of objects, other robots, checkpoints and targets, respectively.

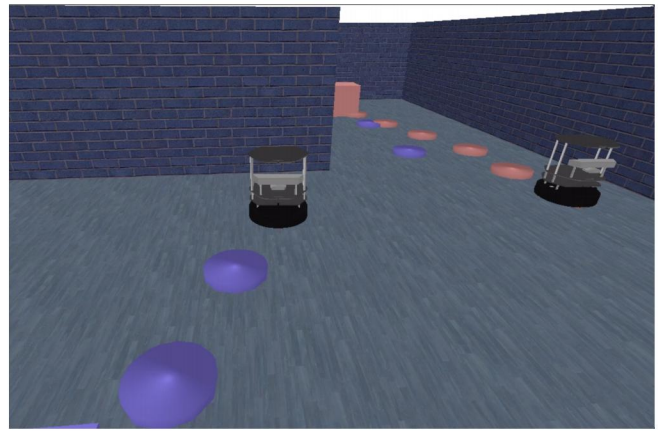


Figure 3. The playground environment with two mobile robots. The pink box is a target for one of the robots.

The reward function is chosen to prioritize the controller objectives described in the introduction. We penalize any control actions with a magnitude greater than 0.6. In a previous work, we discovered that this penalty greatly increases training stability in some discounted MDP's [2]. Without a penalty on action magnitude, the optimizer will choose increasingly large actions to reach the goal faster. This will often cause the gradients to explode, especially when actions are restricted to $[-1, 1]$ by the $tanh$ activation function. Additionally, we choose to penalize rotational velocity, to prevent the robot rotating aimlessly. The battery $R_{batt}(s,a)$, and spin $R_{spin}(s,a)$, penalties are defined as follows:

$$R_{batt}(s,a) = -0.001 \left\| \max(|a| - 0.6, 0) \right\|^2 \tag{2}$$

$$R_{spin}(s,a) = -0.001 \left\| \dot{\theta}_{robot} \right\|^2 \tag{3}$$

The total reward is found by summing these values:

$$\begin{aligned} R(s,a) = R_{target}(s,a) + R_{collision}(s,a) \\ + R_{batt}(s,a) + R_{spin}(s,a) \end{aligned} \tag{4}$$

where $R_{target}(s,a)$ is 1 if the agent reaches the target, otherwise it is 0. $R_{collision}(s,a)$ is a function that penalizes the robot for getting too close to other objects:

$$R_{collision}(s,a) = \max(-1, -0.3e^{20(0.4-d)}) \tag{5}$$

where d is the distance between the robot and any other object, measured in meters. The episode is terminated if the robot reaches the target, or after 100 timesteps.

4.2. Environments

We define multiple testing environments, which are now described.

Playground Environment: This is a simple environment with 3 connected rooms. Robots and targets are placed randomly. On average, the target can be reached within 14 steps. A screenshot from the Playground environment is shown in Figure 3.

House Environment: This is a more complex environment with 9 connected rooms and multiple long corridors. Robots and targets are placed randomly. On average, the target can be reached within 43 steps. This environment was automatically generated from a Lidar scan of a real house.

Building Environment This is a very large environment with 16 rooms and long corridors. Robots and targets are placed randomly. On average, the target can be reached within 82 steps. This environment was automatically generated from a Lidar scan of the Y2E2 building on Stanford campus.

Each of the environments contain at least two other robots. The robots share the same policy, but each make actions and observe rewards independently. The episode is terminated once every robot has reached the `done` state.

4.3. Training

The DNRC agent is trained in a variety of different navigation environments, using the Deep Deterministic Policy Gradient Algorithm (DDPG) with TD3 extensions [17]. The training process is distributed across a cluster of virtual machines using the Apex training algorithm [19]. The training cluster consists of a single master node and four worker nodes. The master node has 64 CPU cores and a single NVIDIA V100 GPU. Each worker node has 96 CPU cores and 384 GB of memory. Trajectories are generated on the worker nodes by executing the policy π_ϕ on each robot. Each trajectory is sent to the master node which stores it in a replay buffer. The learner continuously optimizes the current policy using DDPG with prioritized experience replay. The updated policy is sent to the worker nodes at regular intervals (every few seconds). In our implementation, the worker nodes generated approximately 20,000 trajectory steps per second, and the learner sampled approximately 80,000 trajectory steps/second from the replay buffers. Training took 4-48 hours based on the complexity of the environment.

4.4. Results

The average episode reward is compared to two other modern reinforcement learning algorithms, Twin Delayed Deep Deterministic policy gradient (TD3) [17] and Proximal Policy Optimization (PPO) [20] in Table 1.

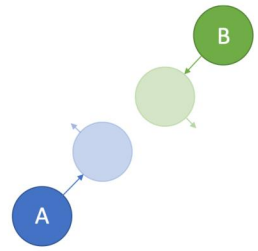
4.5. Human-level Control

After extensive training, the DNRC agents begin to exhibit some human-like control policies. The interaction between multiple robots is particularly interesting: Initially, the agents learn to slow down when another robot is nearby so as to avoid collision, as shown in Figure 4a. However, this policy is sub-optimal in the discounted Markov setting, as it prolongs the time until the goal is reached. After much more training, the agents learn that they can safely pass each other at speed, so long as they both turn the same way to avoid collision. For example, we observed that both robots would always pass on the right when travelling directly towards each other, as shown in Figure 4b. As both robots execute the same policy, each agent learns to anticipate the behaviour of the other robot and acts accordingly.

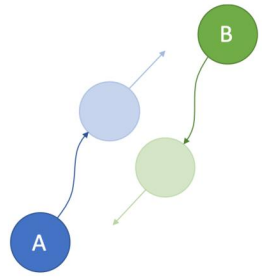
In some cases, we observed the development of non-greedy control policies, which tend to be quite atypical of reinforcement learning approaches. One example, is the interaction of two robots in a narrow hallway, as shown in Figure 5. In this scenario, `robot A` aims to reach target T_A, but `robot B` is blocking the path. After 20 million training steps, this results in a deadlock where `robot A` never reaches the target. However, after 80 million training steps `robot A` first moves to the side of the hallway to let `robot B` passed. It then proceeds to the goal.

TABLE 1. MEAN EPISODE REWARD FOR EACH ENVIRONMENT. THE PPO AND TD3 ALGORITHMS (BASELINES) ARE COMPARED TO OUR DEEP NEURAL ROBOT CONTROL (DNRC) ALGORITHM

Environment	PPO	TD3	DNRC (ours)
Playground (3 Robots)	0.91	0.93	**0.96**
House (3 Robots)	0.44	0.45	**0.91**
Building (5 Robots)	0.03	0.05	**0.34**

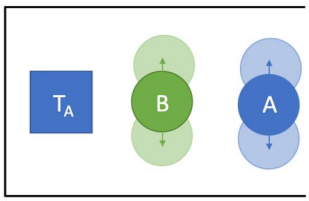


(a) 100 million training steps, 2 hours

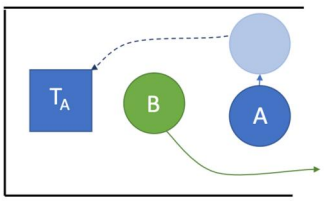


(b) 800 million training steps, 16 hours

Figure 4. Human-level collision avoidance. (a) Faced with a potential collision, the agents initially learn to slow down and slowly pass each other. (b) After extensive training, robots learn to both move to the right and pass each other at full speed.



(a) 100 million training steps, 2 hours



(b) 800 million training steps, 16 hours

Figure 5. Non-greedy strategic control. (a) Agent A can not pass agent B to reach target T_A, as both agents move greedily and somewhat unpredictably. (b) After extensive training, agent A learns to move to the wall, letting agent B past and allowing it to reach the target.

5. Physical Robot Tests

For the proposed algorithm to have practical use, it must also perform well when controlling a real robot. In this section, we demonstrate that DNRC transfers well from a simulated environment to a real environment. Physical tests are conducted in a three bedroom house. The robot (a TurtleBot 2) must navigate to random target locations in the house, whilst avoiding walls and furniture. Given the large number of samples required for training, it is not feasible to run the training algorithm using the real robot. Instead, we train the algorithm is simulation using stochastic environment dynamics, and then evaluate the trained algorithm directly, using zero-shot transfer.

During simulation, we randomly choose the TurtleBot dynamics from a Gaussian prior over the real-world dynamics. This ensures that the learned control policy is robust to variations in TurtleBot dynamics. To formalize this approach, we adopt the transfer learning framework from Sutton and Barto [21]. We aim to train a policy π_Ω that performs well on tasks $M \in \mathcal{M}$ in set of tasks (each task is represented as an MDP). We assume that there is a distribution over tasks, such that $M \sim \Omega_M$. Next, we assume that the state-space, action-space and reward function is constant across all tasks in the task space M. We assume that there is some distribution over the task dynamics, such that $T(s, a, s') \sim \Omega_T$. Therefore, the distribution over tasks Ω_M is solely characterized by the distribution over dynamics Ω_T. It follows, that a policy π_Ω that is able to achieve good performance on a finite number of source tasks drawn from Ω_M, will generalize well across target tasks drawn from Ω_M [21]. Therefore, we expect that the policy will also perform well in the physical system if we choose Ω_T based on our prior knowledge about the physical-world dynamics.

It is also important to ensure that the distribution over the state space in simulation is similar to the distribution of over state-space in the physical world. We address this as follows: Sensors on the robot are used to scan the real-world environment and develop a simplified representation of the real house that is consistent with the training environment. At each timestep, we copy the position of walls and objects from the physical world, over to our simulator. We then generate observations directly from the simulator, and pass those to the trained DNRC agent. The resulting physical-world policy is near-optimal, so long as we correctly map the physical world to our simulator. The DNRC agent transfers well to the physical environment achieving an average reward of **0.58**.

6. Discussion

A number of challenges and interesting observations were encountered in this work, that would go unpublished if not mentioned here:

- Orthogonalizing the action space (into linear and angular velocity) greatly improved training stability
- Convergence was very poor if the environment was not reset after the target was reached. It is thought that resetting the environment frequently helps to reduce variance in the $Q(s, a)$, making learning more stable.
- The Proximal Policy (PPO) algorithm was tested extensively, but the minimum batch size seemingly scales with task difficulty. Hence, applying this algorithm to complex tasks seems problematic.
- Network bandwidth was the limiting factor in our distributed training setup, with each node producing 25 GB/s of observation data.

7. Conclusion

We proposed a new method, Deep Neural Robot Control (DNRC), to solve long navigation problems with reinforcement learning. This approach, performs exceptionally well on several robot navigation tasks, learning a human-like policy for navigation and collision avoidance. We demonstrated that the DNRC policy can be transferred to a physical robot, using a simulated environment model as an abstraction layer over the physical world.

Acknowledgments

We thank our colleagues from CS 230 who provided suggestions, insight and expertise that greatly assisted the research. We acknowledge the support of Professor Kincho Law and the Stanford Engineering Informatics Group for providing assistance and laboratory resources for the physical tests. We thank Professor Ng, Ahmadreza Momeni and Kian Katanforoosh for their guidance throughout the project.

References

[1] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International Conference on Machine Learning (ICML)*, 2016, pp. 1928–1937.

[2] M. Ferguson and K. Law, "Learning robust and adaptive real-world continuous control using simulation and transfer learning," *arXiv preprint arXiv:1802.04520*, 2018.

[3] ——, "A 2D-3D object detection system for updating building information models with mobile robots," in *IEEE Winter Conference on Applications of Computer Vision (WACV)*, 2019.

[4] Y. Zhu, R. Mottaghi, E. Kolve, J. J. Lim, A. Gupta, L. Fei-Fei, and A. Farhadi, "Target-driven visual navigation in indoor scenes using deep reinforcement learning," in *IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 2017, pp. 3357–3364.

[5] P. Mirowski, M. K. Grimes, M. Malinowski, K. M. Hermann, K. Anderson, D. Teplyashin, K. Simonyan, K. Kavukcuoglu, A. Zisserman, and R. Hadsell, "Learning to navigate in cities without a map," *arXiv preprint arXiv:1804.00168*, 2018.

[6] G. Brunner, O. Richter, Y. Wang, and R. Wattenhofer, "Teaching a machine to read maps with deep reinforcement learning," in *AAAI Conference on Artificial Intelligence*, 2018.

[7] M. Pfeiffer, S. Shukla, M. Turchetta, C. Cadena, A. Krause, R. Siegwart, and J. Nieto, "Reinforced imitation: Sample efficient deep reinforcement learning for map-less navigation by leveraging prior demonstrations," *arXiv preprint arXiv:1805.07095*, 2018.

[8] M. Jaderberg, V. Mnih, W. M. Czarnecki, T. Schaul, J. Z. Leibo, D. Silver, and K. Kavukcuoglu, "Reinforcement learning with unsupervised auxiliary tasks," *arXiv preprint arXiv:1611.05397*, 2016.

[9] K. Frans, J. Ho, X. Chen, P. Abbeel, and J. Schulman, "Meta learning shared hierarchies," *arXiv preprint arXiv:1710.09767*, 2017.

[10] A. Ecoffet, J. Huizinga, J. Lehman, K. O. Stanley, and J. Clune, "Go-explore: a new approach for hard-exploration problems," *arXiv preprint arXiv:1901.10995*, 2019.

[11] M. Bellemare, S. Srinivasan, G. Ostrovski, T. Schaul, D. Saxton, and R. Munos, "Unifying count-based exploration and intrinsic motivation," in *Advances in Neural Information Processing Systems*, 2016, pp. 1471–1479.

[12] K. J. Åström and T. Hägglund, *PID controllers: theory, design, and tuning*. Instrument society of America Research Triangle Park, NC, 1995, vol. 2.

[13] Y. K. Hwang and N. Ahuja, "Gross motion planninga survey," *ACM Computing Surveys (CSUR)*, vol. 24, no. 3, pp. 219–291, 1992.

[14] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press Cambridge, 1998, vol. 1, no. 1.

[15] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[16] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *ICML*, 2014.

[17] S. Fujimoto, H. van Hoof, and D. Meger, "Addressing function approximation error in actor-critic methods," *arXiv preprint arXiv:1802.09477*, 2018.

[18] E. Coumans and Y. Bai, "Pybullet, a python module for physics simulation for games, robotics and machine learning," *GitHub repository*, 2016.

[19] E. Liang, R. Liaw, P. Moritz, R. Nishihara, R. Fox, K. Goldberg, J. E. Gonzalez, M. I. Jordan, and I. Stoica, "Rllib: Abstractions for distributed reinforcement learning," *arXiv preprint arXiv:1712.09381*, 2017.

[20] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[21] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction*. MIT press, 2018.