Developing Continuous Reinforcement Learning Methods that are Robust to Inconsistent Time Discretization

Gleb Shevchuk and Malayandi Palan

June 2019

Abstract

Standard reinforcement learning (RL) methods operate in discrete-time and assume access to the fixed time discretization parameter. However, in several domains – most notably, robotics – the agent executes actions at inconsistent time intervals in practice. In such settings, RL methods trained with a fixed time discretization may not work well. In this work, we aim to develop RL methods that are robust to inconsistent time discretization. We develop a model-based RL method that learns a policy in continuous-time, independent of the time discretization. We empirically evaluate this method on a set of linear-quadratic regulator (LQR) tasks against an analogous discrete-time RL baseline. We find that the continuous-time method is significantly more robust to inconsistent time discretization than the discrete-time baseline, especially on more complex domains. This demonstrates the potential of using continuous-time tools for developing practical RL methods for the real-world.

1 Introduction

In recent years, reinforcement learning (RL) methods have demonstrated immense potential for solving complex decision problems. However, RL's greatest successes have been restricted to structured, controlled domains that can be easily simulated, such as board games or video games. We have yet to see RL methods achieve much success on practical, real-world systems that cannot be predicted or simulated with ease.

This is largely due to the practical challenges of training RL systems. Due to the poor sample complexity of RL methods [Irp18, LHP+15, MKS+15, SWD+17], we cannot train these systems directly in the real world. Instead, we have to train them in simulation, where we can learn with several magnitudes more data in the same amount of time by distributing the training process across many nodes. This method of training in simulation works extremely well when the simulator accurately models the underlying problem [MKS+13, MKS+15, SHM+16]; however, this is rarely, if ever, the case for real-world systems such as robots interacting with the physical world.

In recent years, there has been much work on the so-called sim-to-real problem: the problem of training an agent in simulation and then successfully deploying that agent in the real world. Most of this work however, has focused on resolving the modelling inaccuracies of the simulator [TFR+17, TBD+18, TPA+18, PAZA18].

In this work, we call attention to another fundamental challenge for sim-to-real and one which has received considerably less-attention: inconsistent time discretization. The physical systems, such as robots, that we hope to deploy RL methods on are fundamentally continuous-time systems. However, almost all RL methods today are discrete-time methods, where we assume that states, controls, and rewards are all observed simultaneously at fixed rates. This may not be an issue if we can discretize time effectively and consistently in the physical system. However, this is rarely, if ever, the case [DAMH19].

These physical systems are almost-always modularized. For example, an intelligent robot has a perception module composed of cameras and sensors, a planning module (which may use RL) that chooses a course of action for the robot, and a control module which executes the control input specified by the planning module. In practice, each of these modules take varying amounts of time to execute their respective roles and communicate information to other modules [DAMH19]. The relative range of these variations increase quite considerably with the average frequency at which each module operates. Thus, an RL agent trained in simulation with a fixed time-discretization may not perform as well in the real-world where there is no longer a stringent notion of time-discretization.

In this work, we attempt to develop a RL method that is robust to inconsistent time discretization. Specifically, we develop a simple model-based RL method that is trained in continuous-time, entirely independent of the notion of time-discretization. We do so by first setting up an ordinary differential equation (ODE) that, when solved, computes the estimated cost of an agent's policy; then, by finding the derivative of the ODE's solution with respect to the policy parameters, we can improve the policy using gradient descent.

We evaluate our continuous-time method against an analogous discrete-time baseline on a set of linear-quadratic regulator (LQR) tasks. We find that, when the time discetization is consistent, our continuous-time method performs comparably to the discrete-time method but takes considerably less time to train, especially when the time-discretization is fine. When the time discretization is inconsistent, as in real problems, our continuous-time method significantly outperforms the discrete-time baseline, especially on more complex problems.

We believe that this demonstrates the potential that continuous-time RL methods have for tackling complex decision problems in the real world.

2 Preliminaries

2.1 Task

We consider a finite-horizon, episodic reinforcement learning task where an agent interacts with an environment [SB+98]. The agent interacts with the environment over a sequence of episodes, each of which have a finite time-horizon. Within each episode, the agent can execute control inputs to influence the evolution of the system to a desirable state. However, the agent does not have perfect knowledge about the environment and thus, the agent must simultaneously learn about the environment while trying to move the system to a desirable state.

2.2 Model

We model the environment as a fully-observable, deterministic dynamical systems with continuous-time dynamics given by

$$\dot{x}(t) = f(x(t), u(t)), \quad t \in [0, T], \tag{1}$$

where $x(t) \in \mathbb{R}^n$ is the state of the system, $u(t) \in \mathbb{R}^m$ is the agent's control input, and $\dot{x}(t) \in \mathbb{R}^n$ is the time-derivative of the state, all at time t. T is the time-horizon of the problem.

We model f as a continuous function in both x and u. Thus, the state $x(t)$ evolves as a continuous trajectory.

A policy $\pi : \mathbb{R}^n \to \mathbb{R}^m$ is a deterministic mapping which dictates what control input the agent will execute at any given state:

$$u(t) = \pi(x(t)), \quad t \in [0, T]. \tag{2}$$

Together, Eq. (1) and (2) define the closed-loop system.

2.3 Objective

The agent's objective is to find a policy that minimizes the episodic cost over a fixed time-horizon, given by

$$J(\pi) = \int_0^T g(x(t), \pi(x((t)))\ dt, \tag{3}$$

where $g : \mathbb{R}^n \times \mathbb{R}^m \to \mathbb{R}$ is the Riemann-integrable stage cost of the system.

For ease of exposition, we assume access to the stage cost function; however, it is straightforward to extend the techniques discussed in this paper to the setting where we do not have access to it.

2.4 Information pattern

We assume that the agent does not have access to the dynamics of the system apriori. Therefore, to find an optimal policy (i.e., one that minimizes (3)), the agent will need to estimate the dynamics (1), either implicitly or explicitly.

When the agent is deployed in the environment, the agent receives observations sampled from the underlying continuous trajectory. Each observation is a state-action-state derivative tuple $(x(t), u(t), \dot{x}(t))$. These observations may be sampled at any, not necessarily constant, rate.

2.5 Time discretization

Standard RL methods do not, in general, attempt to solve (3) directly [SB$^+$98]. Instead, they attempt to optimize an approximate version of (3) by discretizing the entire system according to a fixed time-discretization parameter.

Formally, let dt denote the time discretization parameter. Then, by assuming that the agent's control input is constant between time t and $t + dt$, we can discretize the dynamics (1) as follows:

$$x_{t+dt} = f^{dt}(x_t, u_t) = x_t + \int_0^{dt} f(x(t+\tau), u(t))\ d\tau, \quad t = 0, dt, \ldots, T - dt, \tag{4}$$

where $x_t \in \mathbb{R}^n$ is the state of the system and $u_t \in \mathbb{R}^m$ is the control input to the system, both at time t.

For ease of notation, we let $H = Tdt$ denote the step-horizon of the discretized problem, and with some abuse of notation, rewrite the discrete-time dynamics (4) as

$$x_{t+1} = f(x_t, u_t) \quad t = 0, 1, \ldots, H - 1. \tag{5}$$

It should be clear from context, whether f refers to the continuous-time dynamics (1) or the discretized dynamics (5).

The objective for the discretized problem is to find a policy that minimizes the discretized cost over the fixed step-horizon, given by

$$J(\pi) = \sum_{t=0}^{H-1} g(x_t, u_t). \tag{6}$$

Once again, it should be clear from context, whether J refers to the continuous-time objective (3) or the discretized objective (6).

In this discretized case, the agent receives an observation every time the system steps forward in time. Here, each observation is a state-action-next state tuple (x_t, u_t, x_{t+1}).

3 Method

3.1 Model-based reinforcement learning

In this work, we take a model-based approach to reinforcement learning where we explicitly estimate the system dynamics and use this estimate of the dynamics to find an optimal policy [KS96, KV15].

Formally, we parameterize the dynamics f by θ and the policy π by ϕ. At each step, we rollout the policy in the environment and collect tuples of observations from the system. These observations are then stored in a replay buffer, alongside previous observations from the system.

At the end of each episode, we use the observations stored in the replay buffer to update our estimate of the parameterized dynamics function f_θ using, for example, gradient descent. We then update our estimate of the parameterized policy π_ϕ. We do so by (internally) simulating the system using the estimated dynamics f_θ, to estimate the episodic cost of the policy. We can then take the derivative of this estimated episodic cost with respect to the policy parameters and update the policy accordingly using gradient descent.

See Algorithm 1 for an outline of a generic model-based algorithm and see Algorithm 2 for an outline of a generic method for learning the dynamics. Note that when we roll out the agent in

Algorithm 1 A generic model-based RL algorithm.

Require: Environment `env`, Agent `agent`, Buffer `buffer`

1: **for** $l \in \{1, 2, \ldots\}$ **do**
2: `obs` $\leftarrow$ `env.reset()`
3: **while** `env.terminal` is `False` **do**
4: `action` $\leftarrow$ `agent.act(env.state)`
5: `buffer.append(env.step(action))`
6: **end while**
7: `agent.update_dynamics(buffer)`
8: `agent.update_policy()`
9: **end for**

Algorithm 2 A generic implementation of `learn_dynamics`.

Require: Buffer `buffer`, parameterized dynamics f_θ

Require: Minibatch size M, learning rate α, number of gradient steps N_{steps}

1: **for** $n \in \{1, \ldots, N_{steps}\}$ **do**
2: $(x_i, u_i, \dot{x}_i)_{i=1}^M \leftarrow$ `buffer.sample(M)`
3: $\theta \leftarrow \theta - \alpha \nabla_\theta \left(\sum_{i=1}^M (\dot{x}_i - f_\theta(x_i, u_i)) \right)^2$
4: **end for**

the environment (lines 3-6 in Algorithm 1), we must discretize the time-step in the environment for practical purposes. However, we do not necessarily need to do so when learning the policy (line 8 in Algorithm 1). In the remainder of this section, we discuss our method, which allows us to do exactly that.

3.2 A continuous-time method for policy learning

In order to learn in continuous time, we define the cost-so-far of a policy π_ϕ at time $t \leq T$ as the total cost incurred by rolling out the policy to time t:

$$J_t(\pi_\phi) = \int_0^t g(x(\tau), \pi_\phi(x((\tau)))\ d\tau. \tag{7}$$

Recall that the episodic cost of a policy (in continuous-time) (3) is the total cost incurred by a rolling out a policy in a given episode, which is equivalent to the cost-so-far at time T:

$$J(\pi_\phi) = J_T(\pi_\phi) = \int_0^T g(x(\tau), \pi_\phi(x((\tau)))\ dt. \tag{8}$$

To update the policy in continuous-time, we need to be able to find the derivative of this episodic cost with respect to the policy parameters ϕ.

Given a policy, the state evolves according to the ordinary differential equation (ODE) specified by the dynamics (1) and the policy (2). Thus, the cost-so-far (7), which is a function of the state, must also evolve according to an ODE.

We do not have access to the true dynamics f but have access to the approximate dynamics f_θ; thus, we can set up an ODE which approximately governs the state evolution in the system. It is

Algorithm 3 A continuous-time method for `learn_policy`.

Require: Environment `env`, estimated dynamics f_θ, parameterized policy π_ϕ
Require: Learning rate α
for $n \in \{1, \ldots, N_{steps}\}$ **do**
  $x_0 \leftarrow$ `env.reset()`
  $y_0 \leftarrow (f_\theta(x_0, \pi_\phi(x_0)), 0)$
  $\phi \leftarrow \phi - \alpha \nabla_\phi ODESOLVE(\dot{y}, y_0)[-1]$
end for

then easy to set up another ODE which approximately models the evolution of the cost-so-far, and solve this ODE to find the the total episodic cost (8).

Formally, we set up an ODE with the following augmented dynamics

$$\dot{y}(t) = \begin{pmatrix} \dot{x}(t) \\ g(x(t), \pi_\phi(x(t))) \end{pmatrix} = \begin{pmatrix} f_\theta(x_t, \pi_\phi(t)) \\ g(x(t), \pi_\phi(x(t))) \end{pmatrix}.$$

The initial conditions of this ODE are given by

$$y(0) = \begin{pmatrix} \dot{x}(0) \\ 0 \end{pmatrix} = \begin{pmatrix} f_\theta(x(0), \pi_\phi(x(0))) \\ 0 \end{pmatrix}.$$

We can solve this ODE with the given initial conditions using any numerical ODE solver to find

$$y(T) = ODESOLVE(\dot{y}, y(0)) = \begin{pmatrix} \dot{x}(T) \\ \int_0^T g(x(t), \pi_\phi(x(t)))\ dt \end{pmatrix}.$$

The last entry in $y(T)$, which we denote by $y(T)[-1]$, is exactly the estimated episodic cost of the policy. Using techniques outlined in [CRBD18], we can differentiate through the ODE solver without knowing any of the mechanics of the solver. We can thus update the policy parameters ϕ using gradient descent as follows

$$\phi \leftarrow \phi - \alpha \nabla_\phi ODESOLVE(\dot{y}, y(0))[-1].$$

Our full method is specified in Algorithm 3.

4 Experiments

In this section, we run two sets to experiments to understand how our continuous-time method compares to an analogous discrete-time method. As a sanity-check, we first investigate how the two methods perform on a set of simple LQR experiments where the time discretization is consistent. We then consider the setting we are interested in, where the time discretization is inconsistent; we similarly investigate how the two methods perform in this setting, across a range of problems with varying difficulties. But first, we discuss the experimental setup.

4.1 Experimental Setup

4.1.1 Domain

We run our experiments on the linear-quadratic regulator (LQR), a standard domain for evaluating RL algorithms [Bai94, MGR18]. To vary the difficulty of the domain, we vary the dimension of the

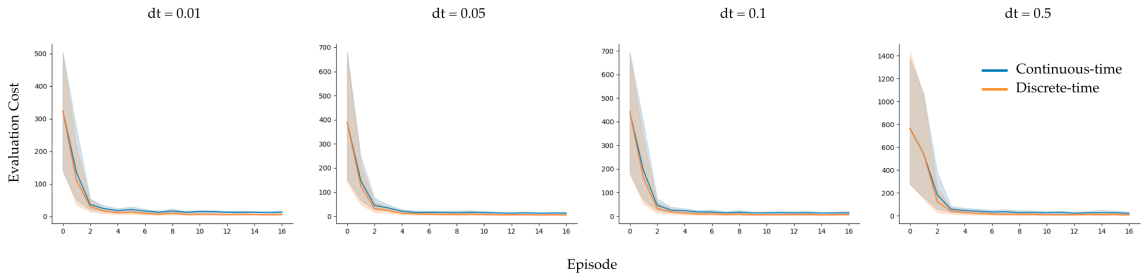

Figure 1: The results of the our first experiment, where we compare how the continuous-time method performs against a discrete-time baseline on a set of LQR environments with different time-discretizations.

state space, n, and the dimension of the control space, m. We use a fixed time-horizon of 1. The remaining parameters of the environment (dynamics and stage cost) are randomly initialized.

4.1.2 A discrete-time baseline

We benchmark our continuous-time method against an analogous discrete-time baseline. Specifically, the discrete-time baseline follows the same model-based structure shown in Algorithms 1 and 2. The only difference between the discrete-time baseline and our continuous-time method is in the policy learning step.

For the discrete-time baseline, we roll out the policy in discrete-time using the estimated dynamics to estimate the discrete-time episodic cost (6); we then take the derivative of this estimated episodic cost with respect to the policy parameters and update the policy accordingly.

4.2 Experiment 1: A sanity check

In this experiment, we investigate how the continuous-time method performs in comparison to the discrete-time method on problems where the time-discretization is perfectly consistent.

We run the experiment on an LQR environment with $(n, m) = (8, 8)$. We vary the time-discretization dt in the environment, using values $0.01, 0.05, 0.1, 0.5$. As a measure of performance, we measure the evaluation cost of the learned policy on the true environment after each episode.

The results of our experiment are shown in Figure 1. We see that the continuous-time method performs just as well as the discrete-time method for all values of dt. This serves as a valuable sanity check for our method, confirming that, even on idealized domains designed for discrete-time methods, our continuous-time method does not perform worse than an analogous discrete-time method.

Additionally, we measured the per episode training time for the discrete-time and the continuous-time policy on each of the domains. The results are shown in Figure 2. Since the continuous-time method is trained independently of the time discretization, the per episode training time is constant across all discretizations. For the discrete-time method however, the time increases exponentially as the discretization reduces; this is because at finer discretizations, the number of time-steps for which the policy is rolled out and backpropagated through increases significantly.

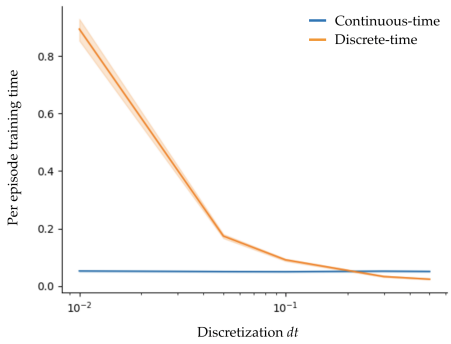

Figure 2: The per episode training time for the discrete-time and the continuous-time policy on a range of domains with varying time discretizations.

4.3 Experiment 2: Inconsistent time-discretization

In this experiment, we investigate how the two methods perform on problems where the time-discretization is inconsistent.

To simulate the behavior of problems with an inconsistent time-discretization, we vary the time-interval between which the agent receives observations and chooses actions. Specifically, this time-interval was uniformly chosen from the interval $[0.01/e, 0.01e]$, where e denotes the time-discretization error rate. In our experiment, we varied this error rate, using values 1, 1.5, and 2. (Note that $e = 1$ corresponds to perfectly consistent time-discretization.)

We additionally vary the complexity of the LQR domains by changing the dimension of the state and control spaces. Specifically, we use a simple domain with $(n, m) = (8, 8)$ and a more complex domain, with $(n, m) = (64, 8)$. The results of these experiments are presented in Figure 3.

We see that on the simpler domain, with $(n, m) = (8, 8)$, both methods perform similarly well even when the inconsistencies in time discretization is large ($e = 2$).

On the more complex domain, with $(n, m) = (64, 8)$, when there is no inconsistency in time-discretization, the discrete-time method does slightly better than the continuous-time methods; however, both methods do converge to reasonably low evaluation losses for the environment.

When the time-discretization is inconsistent however ($e = 1.5, 2$), the discrete-time method performs significantly more poorly than the continuous-time method; on these more realistic problems, the discrete-time methods do not even converge and the evaluation costs in fact increase with the number of training episodes.

5 Discussion

Our experimental results suggest that our continuous-time method is significantly more robust to inconsistent time discretization than the discrete-time baseline on complex problems; on simpler problems, or problems where the time discretization is consistent, the continuous-time method performs comparably to the discrete-time baseline. Additionally, we note that on finely discretized problems, the continuous-time method takes considerably less time to train.

Though these results on the LQR domain show great promise, much work lies ahead. LQR is an

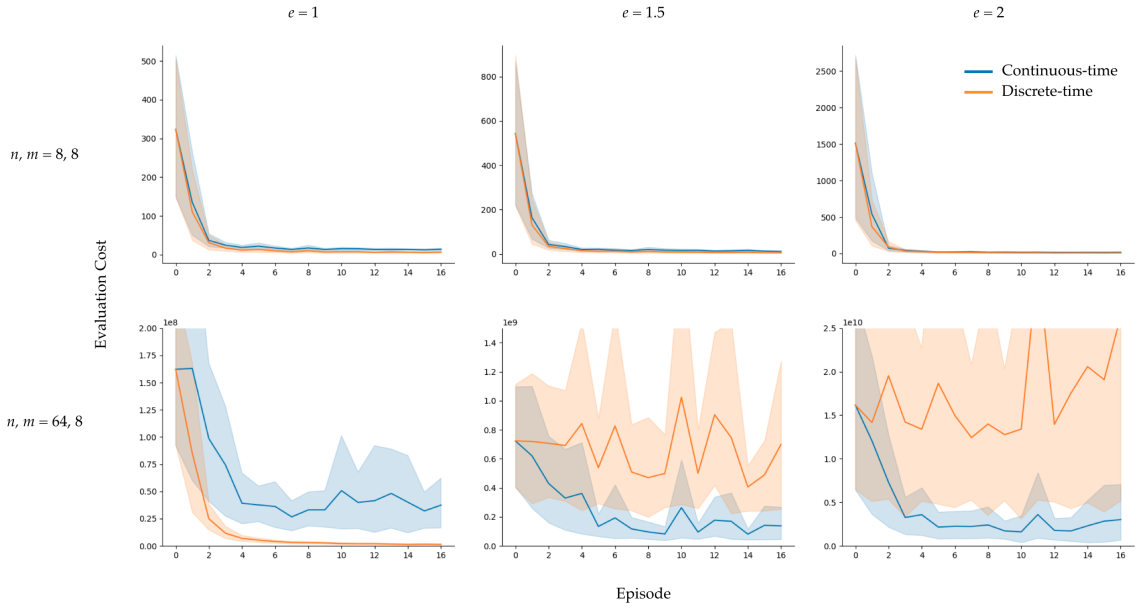

Figure 3: The results of the our second experiment, where we compare how the continuous-time method performs against a discrete-time baseline on a set of LQR environments with inconsistent time-discretizations.

extremely simple benchmark and if we want to convincingly make the claim that continuous-time methods have the potential to work effectively in the real world, we need to test these methods extensively on more realistic domains. We were not able to do so in this work due to (1) time constraints and to (2) challenges with the model-learning formulation we adopted. To be more specific with the latter, on more complex domains, the model-based approach struggled to learn an effective policy due to issues with compounding errors from the learned dynamics. We believe that the development of model-free continuous-time RL methods are an important and promising direction for future work.

Regardless, we do not think of our main contribution as a purely algorithmic one, since the specific algorithm we developed will most likely not see success in complex RL domains for the reasons discussed above. Rather, our goal was to shed light on the potential that continuous-time RL methods have for tackling realistic decision problems.

We believe that, in addition to being more robust to time-discretization issues, continuous-time methods should be significantly more sample efficient than their discrete-time counterparts since they naturally exploit continuous structure in RL problems. It is unclear to use why we did not observe those gains in our simple experiments from the previous section; regardless, the investigation of these claim thoroughly is another promising direction for future work.

In conclusion, we believe that continuous time approaches will be vital to realizing RL that works freely and reliably in the real world; we hope that our work here will convince more people to begin looking at continuous-time RL methods.

Acknowledgements

This work was done jointly with Allan Zhou, who is not enrolled in the course. All three authors contributed equally to this work. The only reason that Allan's name has not been included in the by-line is to avoid confusing the instructors and graders.

We are very grateful for the help and guidance provided by Professor Benjamin Van Roy and Vikranth Dwaracherla. Thank you for organizing a wonderful course. The both of us learned a lot.

References

[Bai94] Leemon C Baird. Reinforcement learning in continuous time: Advantage updating. In *Proceedings of 1994 IEEE International Conference on Neural Networks (ICNN'94)*, volume 4, pages 2448–2453. IEEE, 1994.

[CRBD18] Tian Qi Chen, Yulia Rubanova, Jesse Bettencourt, and David K Duvenaud. Neural ordinary differential equations. In *Advances in Neural Information Processing Systems*, pages 6571–6583, 2018.

[DAMH19] Gabriel Dulac-Arnold, Daniel Mankowitz, and Todd Hester. Challenges of real-world reinforcement learning. *arXiv preprint arXiv:1904.12901*, 2019.

[Irp18] Alex Irpan. Deep reinforcement learning doesn't work yet. https://www.alexirpan.com/2018/02/14/rl-hard.html, 2018.

[KS96] Leonid Kuvayev and Richard S Sutton. Model-based reinforcement learning with an approximate, learned model. In *in Proceedings of the Ninth Yale Workshop on Adaptive and Learning Systems*. Citeseer, 1996.

[KV15] Panqanamala Ramana Kumar and Pravin Varaiya. *Stochastic systems: Estimation, identification, and adaptive control*, volume 75. SIAM, 2015.

[LHP+15] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[MGR18] Horia Mania, Aurelia Guy, and Benjamin Recht. Simple random search provides a competitive approach to reinforcement learning. *arXiv preprint arXiv:1803.07055*, 2018.

[MKS+13] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[MKS+15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529, 2015.

[PAZA18] Xue Bin Peng, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. Sim-to-real transfer of robotic control with dynamics randomization. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1–8. IEEE, 2018.

[SB+98] Richard S Sutton, Andrew G Barto, et al. *Introduction to reinforcement learning*, volume 135. MIT press Cambridge, 1998.

[SHM+16] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. Mastering the game of go with deep neural networks and tree search. *nature*, 529(7587):484, 2016.

[SWD+17] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[TBD+18] Josh Tobin, Lukas Biewald, Rocky Duan, Marcin Andrychowicz, Ankur Handa, Vikash Kumar, Bob McGrew, Alex Ray, Jonas Schneider, Peter Welinder, et al. Domain randomization and generative models for robotic grasping. In *2018 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 3482–3489. IEEE, 2018.

[TFR+17] Josh Tobin, Rachel Fong, Alex Ray, Jonas Schneider, Wojciech Zaremba, and Pieter 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)*, pages 23–30. IEEE, 2017.

[TPA+18] Jonathan Tremblay, Aayush Prakash, David Acuna, Mark Brophy, Varun Jampani, Cem Anil, Thang To, Eric Cameracci, Shaad Boochoon, and Stan Birchfield. Training deep networks with synthetic data: Bridging the reality gap by domain randomization. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 969–977, 2018.