Hierarchical Reinforcement Learning for Manipulation Proposal

Lucas Pauker
Department of Computer Science
Stanford University
Stanford, California 94305
Email: lpauker@stanford.edu

***Abstract*—In this research proposal, we propose a procedure to effectively compare reinforcement learning (RL) and hierarchical reinforcement learning (HRL) algorithms. Specifically, we compare three different algorithms: end-to-end RL, HRL with preset sub-policies, and HRL with learned sub-policies. For these algorithms, we leverage existing knowledge in the literature and use deep RL and HRL methods with proximal policy optimization for training. We propose metrics for both simulated and real robot experiments centered around training time, accuracy, and reward to compare the three different algorithms. The task we propose to use is to rotate a cube to a desired face with the TriFinger robot. This task is simple and has a clear hierarchy, making it effective for this research. Furthermore, we use the TriFinger setup since it is cheap, open source, and has proven sim-to-real transferability. Lastly, we present some initial results that inform our experimental procedure and show that our research is viable.**

I. Introduction

Hierarchical reinforcement learning (HRL) is a reinforcement learning (RL) scheme that learns on multiple time horizons. Instead of learning a single policy to solve a task, HRL seeks to learn a high-level policy that chooses optimal sub-policies as its actions. Each sub-policy may also be learned with RL. This has the effect of learning on multiple time horizons: the high-level policy learns on a longer time horizon than the sub-policies. Formally, the theory of HRL extends the traditional framework of RL by introducing closed-loop sub-policies. Then the sub-policies can be added to the set of actions that can be chosen by a high-level policy. Sutton et al. [14] introduce such a framework, where the closed-loop sub-policies are called options. The framework uses the RL notions of states $\mathcal{S}$ and primitive actions $\mathcal{A}$. $\mathcal{O}$ denotes the set of all available options. Each option consists of a policy, termination condition, and an initiation set. The high-level policy is then $\mu : \mathcal{S} \times \mathcal{O} \to [0, 1]$ selects an option $o \in \mathcal{O}$ that begins in state s_t according to the probability distribution $\mu(s_t, \cdot)$. This is analogous to the policy $\pi : S \times A \to [0, 1]$ in typical RL. For the rest of the paper, we will refer to options as sub-policies.

For this paper, we focus on the task of rotating a cube to a desired face while the cube stays in a robot hand. This task lends itself to HRL since there are intuitive sub-policies. Namely, these sub-policies are rotating the cube forwards, backwards, left, and right. The robot we use in simulation and for real experiments is the TriFinger robot [16]. This task is difficult because the state and action spaces are continuous. The action space is nine-dimensional and consists of the torque for each of the nine degrees of freedom for the robot. The state space is six-dimensional and consists of the rotation and spatial position of the cube.

We seek to compare the effectiveness of three approaches to solving this task. The three approaches we consider are end-to-end RL, HRL with preset sub-policies, and HRL with learned sub-policies. By investigating the tradeoffs between these three methods, we seek to inform future efforts to build dexterous robots. Furthermore, there is a need in the current literature to compare these different types of algorithms head-to-head: tested on the same systems and trained in a similar fashion.

The 2020 Real Robot Challenge (RRC) was a competition where the challenge is to design manipulation agents using the TriFinger robot. One task for the competition was to move a cube from an initial location to a final location. This task was divided into four difficulty levels; the highest difficulty level considers the goal orientation of the cube while the others are only focused on the spatial position of the cube. Since the highest difficulty level is cube rotation, this shows that rotating a cube is a difficult task. Funk et al. [10] present various baselines for the RRC. Their baselines include three methods: using grasp heuristics and control strategies, implementing Bayesian optimization, and using residual policy learning. Notably, we see that the baselines do not include HRL. We seek to find if HRL improves accuracy for the rotation task, which would provide evidence that it should be considered as part of an algorithm for the competition.

The task we are considering falls under the topic of in-hand manipulation, which is the ability to reposition an object while the object stays in a robot hand. In-hand manipulation is useful for building robust and adaptable robots. If the orientation of the object in a robot hand is not right for the task at hand, in-hand manipulation is essential. Chavan-Dafle et al. [4] discuss different ways for a robot hand to regrasp objects by leveraging outside forces such as gravity and normal forces from surfaces such as tables and walls. This paper hard codes different regrasp procedures for a robot hand with various objects. This is useful, however, if a new object is added to the environment or if a new robot hand is introduced, creating the regrasp procedures is difficult and time-consuming and must be done manually. In short, the methods in this paper are not easily

transferable to general objects and robot hands. As far as we know, there is no such regrasping system in the literature that is proven to work on real robotic systems. Therefore, there is a need for object and hand agnostic methods of building regrasp procedures. RL is a natural choice to solve this problem. This paper is a first step in the direction of building effective in-hand manipulation procedures that do not depend on a specific robot setup and that can be learned with only knowledge of the goal state.

II. Related Work

A. HRL Frameworks

There are various HRL frameworks that seek to use sub-policies to speed up the learning process and to easily learn new tasks. The most well-known framework is the options framework developed by Sutton et al. [14]. Other frameworks include the hierarchies of abstract machines approach developed by Parr and Russell and the MAXQ framework developed by Dietterich [2]. These approaches all use semi-Markov decision processes (SMDPs) to model sub-policies. SMDPs extend MDPs by allowing actions to take variable amounts of time. This quality makes SMDPs effective for modeling continuous-time discrete-event systems [14]. In this paper, we use the options approach developed by Sutton et al. [14] to implement the HRL policy with preset sub-policies because their approach is the most commonly used in HRL literature.

B. Task-Insensitive HRL

There is an abundance of work focused on learning sub-policies autonomously. This work can be divided into two classes: *discovering* sub-policies and *learning* sub-policies. Discovering sub-policies generally consists of finding useful states that the agent should reach and then learning policies to achieve them. Learning sub-policies consists of finding optimal policies without identifying specific subgoal states.

Discovering sub-policies using subgoal states is well-known and well-researched. Mcgovern and Barto [12] use the notion of bottleneck states to identify states that should be used as subgoals. Bottleneck states are states that all the successful trajectories pass through. One example of a bottleneck state is a doorway connecting two rooms in a multi-room navigation task. The authors identify these states using diverse density and use typical RL methods to create sub-policies based on the bottleneck states. Şimşek and Barto [5] also use the notion of bottleneck states to identify subgoals and sub-policies. However, this paper uses an intuitive measure of centrality of graphs to find bottleneck states. Konidaris and Barto [11] extend the previous approaches to be used with continuous state and action spaces. This is an important contribution because most robotics problems have continuous state and/or action spaces, and discretizing these spaces is not desirable. For our task, we have both continuous state and action spaces. Daniel et al. [6] develop an approach to find sub-policies on continuous and discrete state and action spaces. The authors use a probabilistic formulation to infer all the relevant components from data. The authors notably show that their approach applies well to both discrete and continuous domains.

These papers all use the notion of bottleneck states to identify subgoals and then learn policies based on these subgoals. The papers show that the policies identified are effective in learning and make intuitive sense. However, these papers lack effective comparison to typical RL methods, especially on complex robotics tasks. Furthermore, the subgoal approach does not scale well as tasks become more complex [1].

Other more modern work uses objective functions to learn sub-policies. Bacon et al. [1] extends the options framework to an option-critic architecture that learns options automatically. This approach is successfully applied to various tasks. Notably, the authors show that the option-critic framework outperforms a standard actor-critic agent and a SARSA agent when the goal changes mid-training. Florensa et al. [8] similarly present a method to learn hierarchical structure automatically. This paper uses Stochastic Neural Networks (SNN) to learn useful skills then trains a high-level policy on top of these skills. The authors show that the SNN approach outperforms baselines but they do not use typical RL baselines or other HRL approaches as baselines. Frans et al. [9] also seeks to learn sub-policies automatically. This paper uses neural networks to learn sub-policies and a high-level policy. The authors show that their HRL approach outperforms baselines on certain tasks, learns meaningful sub-policies, and can learn well on sparse environments.

These three papers all learn sub-policies automatically without any domain knowledge. This is useful for many tasks that may not have obvious sub-policies. Furthermore, the papers highlight the success of these learned sub-policy methods when applied to new tasks as well as when the goal is altered mid-training. The HRL methods also outperform non-hierarchical in certain tasks, however these tasks tend to be specific and it is not clear that the HRL methods would perform as well in other settings. Furthermore, there is little work comparing task-insensitive HRL to HRL policies with domain-specific sub-policies.

C. Robot Hardware

TriFinger is an open-source robotic platform that is developed by Wüthrich et al. [16]. TriFinger is unique because the hardware for it is inexpensive and it performs safety checks to assert that the hardware will not break, meaning it is accessible to a general audience. TriFinger is easy to program and can be used to test control algorithms for a variety of tasks.

In this paper, we propose to use the TriFinger robot for both simulated and real robot tests. We choose to use this robot because it is cheap, easy to use, has an existing simulation environment, and because the sim-to-real transfer is effective. Furthermore, we use this robot because it was used in the 2020 Real Robot Challenge and therefore there are existing performance results on similar tasks to the one we are considering.

D. Further Research

The current state-of-the-art lacks comprehensive comparisons between RL and HRL policies. Our proposed research is to conduct a thorough comparison of end-to-end learning, HRL with preset sub-policies, and HRL with learned sub-policies. This research is important for practitioners looking to implement algorithms and deciding between RL and HRL.

Currently, HRL methods require the number of sub-policies to be specified before training. There is still work to be done to learn this automatically and create a truly end-to-end HRL system. This is a particularly difficult problem since it is desirable to have sub-policies with physical significance and therefore one must be careful not to have too many sub-policies. Furthermore, there is work to be done to learn multi-layered hierarchies which is a novel topic of research.

III. Proposed Research

A. Overview

We propose to investigate the efficacy of HRL compared to end-to-end learning on the cube rotation task. Specifically, we seek to compare the following three methods:

1) End-to-end learning: training an end-to-end RL policy with no hierarchical components on the cube rotation task.
2) HRL with predefined sub-policies: training an HRL policy on the cube rotation task with the preset sub-policies of rotating the cube forwards, backwards, left, and right by 90 degrees.
3) HRL with learned sub-policies: training an HRL policies with four sub-policies learned by neural networks.

We seek to compare these three methods by running simulated and real robot testing. We compare the methods by analyzing the time to train the policies, the accuracy, and the rewards.

B. Reward Function

The reward function is an essential aspect of successfully training agents to perform well on the task at hand. In order to formulate the reward function, we must first develop some notation. We define $\vec{n}_g$ to be the vector normal to the goal cube face. This vector always points in the direction up from the palm. We define $\vec{n}_r$ to be the vector normal to the cube face corresponding to $\vec{n}_g$. For example, if the goal state is the cube with the letter "E" on top, then $\vec{n}_r$ is the vector normal to the "E" face on the cube being manipulated. Lastly, d represents the distance from the center of the cube being manipulated to the center of the hand. Figure 1 is a diagram of $\vec{n}_g$, $\vec{n}_r$, and d.

Intuitively, we want the reward function to include some notion of rotational distance from the goal state and some notion of the distance to the center of the hand. The reason we include the distance to the center of the hand in the reward function is because we want the cube to be in a stable place in the hand and do not want it to fall out of the hand. Furthermore, by keeping the cube in a constant spatial location, we can string together multiple rotations to perform complex manipulations. The angle between $\vec{n}_g$ and $\vec{n}_r$ is

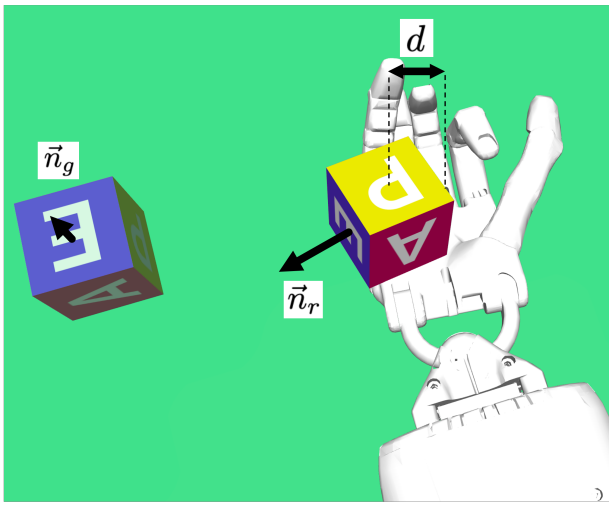

Fig. 1. Diagram showing $\vec{n}_g$, $\vec{n}_r$, and d for a simulated cube manipulation task.

$$\theta = \arccos\left(\frac{\vec{n}_g \cdot \vec{n}_r}{\|\vec{n}_g\|\|\vec{n}_r\|}\right). \tag{1}$$

We define the loss function as

$$\mathcal{L} = \theta + \alpha d \tag{2}$$

and the reward function as

$$\mathcal{R} = -\mathcal{L}. \tag{3}$$

We can see that the loss and reward functions includes a parameter α. α must be tuned depending on the units of θ and d. Tuning α is important for training policies that are effective. To tune α, we will use a grid search approach with end-to-end learning trained on a range of values for α. We will then select α based on the value that leads to the best reward after a number of training steps.

Something to note is that the reward function is dense. An alternative choice for the reward function is a sparse function where a reward is given when the cube is in the right orientation and a penalty is given otherwise. We use the dense reward function in this research since it performed better than the sparse reward function in our preliminary tests.

C. Training

We seek to train the algorithms on a simulated TriFinger robot. For the simulation, we use OpenAI Gym with Mujoco for the physics simulation [3][15].

For training the algorithms, we use the proximal policy optimization (PPO) algorithm. PPO extends traditional policy gradient methods by using minibatch updates. PPO has high sample efficiency, is easy to understand, and performs well on baselines compared to A2C and ACER [13]. We use the OpenAI Baselines library for implementation of the PPO algorithm [7].

Training will be done on Google Cloud Compute with GPUs to speed up training. For the preliminary results, we used a NVIDIA Tesla T4 GPU on Google Cloud. However, we will likely scale this up and use use multiple GPUs for training.

For end-to-end training, we train using PPO with a 2 layer MLP network with a hidden size of 64. We also use a batch size of 2048 timesteps. The initial state of the cube is a fixed orientation. The goal state of the cube is the cube oriented to a random face. In total, there are 24 goal states: six possible faces and each face has four orientations.

For HRL with preset sub-policies, we define four sub-policies: rotate by 90 degrees forwards, backwards, left, and right. For each sub-policy, we have a preset goal state corresponding to the desired rotation. The high-level policy is trained similarly to the end-to-end training, however the policy can only use the sub-policies learned. During training, we train one epoch of each sub-policy then one epoch of the high-level policy. Therefore, each sub-policy and the high-level policy get trained for one epoch every five epochs of total training time. To train each sub-policy and the master policy, we use PPO with a 2 layer MLP network with a hidden size of 64. We also use a batch size of 2048 timesteps.

For training the HRL algorithm with learned sub-policies, we draw inspiration from Frans et al. [9]. We use 2 layer MLPs with a hidden size of 64 for both the master and sub-policies. We also use 4 sub-policies to be able to compare directly to the HRL algorithm with preset sub-policies. We use a batch size of 2048 timesteps. The initial and goal states are identical to the end-to-end learning.

Note that we use a similar architecture across all three approaches: we have the same reward function, same optimization algorithm and the same parameters. This will allow us to directly compare these different methods.

D. Metrics

To compare the various learning methods, we propose to use training time, accuracy, and reward metrics. Combined, these metrics describe the sample efficiency of each of the algorithms. We will perform these tests on both the simulated TriFinger robot as well as a real robot. TriFinger can be implemented in OpenAI Gym and has easy sim-to-real transferability [16].

For the training metric, the goal is to see how fast each method converges on an optimal policy. We compare the number of timesteps each method takes to reach a certain average reward threshold. This threshold will be set close to the optimal average reward.

We will also use accuracy and reward metrics to compare the three algorithms. For these tests, each algorithm is trained with the same number of training steps. The number of training steps will be chosen based on the number of steps required for the algorithms to converge to an optimal policy. We define success as a binary variable indicating whether the robot successfully turned the cube to the desired face. The accuracy is the number of successes compared to the number of trials for many trials. We compare the accuracy of the three algorithms by running 100 trials for each algorithm. Furthermore, we compare the average reward for each algorithm across 100 trials. For the trials, each algorithm has the same set of 100 goal states to reduce randomness.

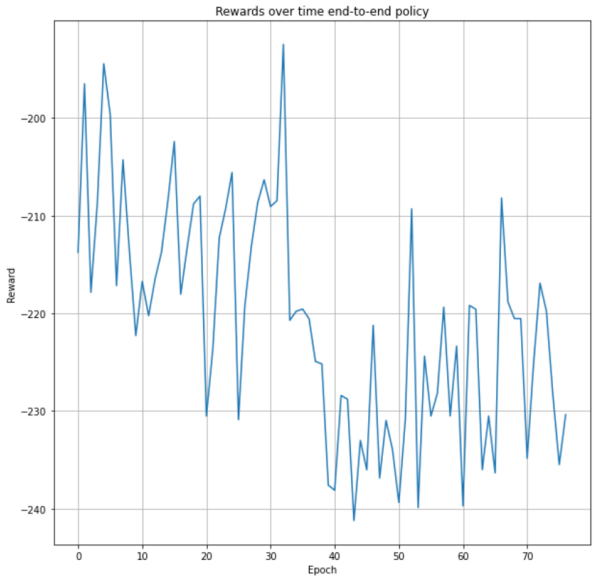

Fig. 2. End-to-end PPO algorithm trained for five million timesteps.

E. Timeline

This research will take a total of 16 weeks. The first 2 weeks will be spent setting up the TriFinger environment in simulation on the Google Cloud Compute server with appropriate GPUs. The next 6 weeks will be spent building the three algorithms in simulation, tuning the algorithms, and collecting results on the simulated environment. The next 6 weeks will be spent transferring the simulation to a real TriFinger robot and performing experiments on the robot. The last 4 weeks will be spent synthesizing results and writing a paper based on these results.

IV. Preliminary Research

The preliminary research we have conducted consists of training the end-to-end learning algorithm and the sub-policies for HRL with preset sub-policies. The preliminary research was done using the OpenAI hand_env environment. For future experiments, we will use the TriFinger simulation. All training was done on Google Cloud Compute with an NVIDIA Tesla T4 GPU.

For the end-to-end learning algorithm, we trained PPO for five million timesteps. The reward over time for training is shown in Figure 2. We can see that the rewards over time have high variance and in general do not have a strong upward trend. In fact, the rewards decrease significantly around epoch 35. This shows that training for five million timesteps is not enough timesteps for the end-to-end RL algorithm to converge on an optimal policy. Therefore, for future experiments, training time will be longer and performed with more compute power in order to successfully converge to an optimal policy.

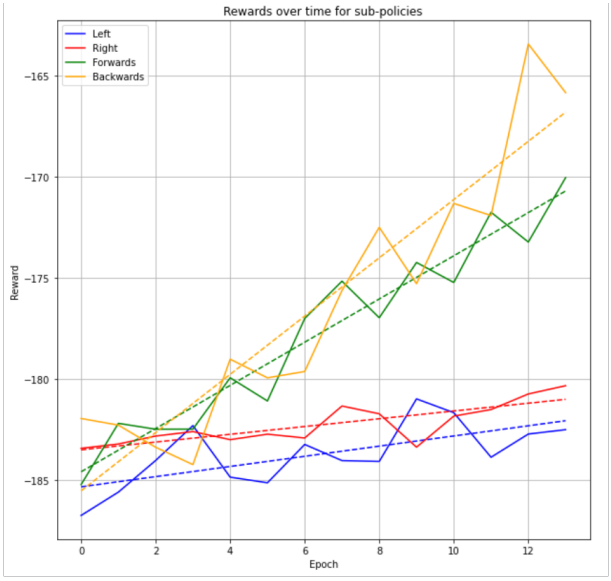

Fig. 3. Sub-policies (left, right, forwards, backwards) trained for one million timesteps.

For the sub-policy training, we trained each sub-policy for one million timesteps. The reward over time for training is shown in Figure 3. The solid lines in the figure are the rewards over time and the dotted lines in the figure are the linear fits for each sub-policy rewards over time. For the sub-policy training, we see strong evidence of successful training since each training curve has a clear upward trend. Notably, we see that the forwards and backwards sub-policies improve faster than the left and right sub-policies. This is due to the way that the OpenAI hand_env hand is configured. The hand is able to move its fingers forwards and backwards easier than left and right. Furthermore, it is clear that the training curves do not flatten out. Each curve is still on an upwards trajectory at the end of training. Therefore, since the training does not converge to an optimal policy, we likely need more training time to fully train the sub-policies. However, overall, since the rewards increases over time, it is clear that the training method is successful.

Our preliminary experiments show us two insights that will be applied to our future research. First, we need more compute power to effectively train the policies. Training a nine degree of freedom robot is expensive and takes many iterations. Therefore, for our research in the future we plan on using more GPUs or more powerful GPUs to accelerate training. Second, we see that our methods are feasible since we successfully trained the sub-policies. We believe that training on a simulated TriFinger robot will be as successful as training on the OpenAI hand_env.

Acknowledgments

I would like to thank Professor Jeannette Bohg and Claire Chen for supporting this work.

References

[1] Pierre-Luc Bacon, Jean Harb, and Doina Precup. The option-critic architecture, 2016.

[2] Andrew G. Barto and Dridhar Mahadevan. Recent advances in hierarchical reinforcement learning. *Discrete Event Dynamic Systems: Theory and Applications,*, 13: 343–379, 2003.

[3] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym, 2016.

[4] Nikhil Chavan-Dafle, Alberto Rodriguez, Robert Paolini, Bowei Tang, Siddhartha Srinivasa, Michael Erdmann, Matthew T. Mason, Ivan Lundberg, Harald Staab, and Thomas Fuhlbrigge. Extrinsic dexterity: In-hand manipulation with external forces. In *Proceedings of (ICRA) International Conference on Robotics and Automation*, pages 1578 – 1585, May 2014.

[5] Özgür Şimşek and Andrew Barto. Skill characterization based on betweenness. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, editors, *Advances in Neural Information Processing Systems*, volume 21. Curran Associates, Inc., 2009. URL https://proceedings.neurips.cc/paper/2008/file/934815ad542a4a7c5e8a2dfa04fea9f5-Paper.pdf.

[6] Christian Daniel, Herke van Hoof, Jan Peters, and Gerhard Neumann. Probabilistic inference for determining options in reinforcement learning. *Machine Learning*, 104, 09 2016. doi: 10.1007/s10994-016-5580-x.

[7] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. Openai baselines. https://github.com/openai/baselines, 2017.

[8] Carlos Florensa, Yan Duan, and Pieter Abbeel. Stochastic neural networks for hierarchical reinforcement learning. *CoRR*, abs/1704.03012, 2017. URL http://arxiv.org/abs/1704.03012.

[9] Kevin Frans, Jonathan Ho, Xi Chen, Pieter Abbeel, and John Schulman. Meta learning shared hierarchies, 2017.

[10] Niklas Funk, Charles Schaff, Rishabh Madan, Takuma Yoneda, Julen Urain De Jesus, Joe Watson, Ethan K. Gordon, Felix Widmaier, Stefan Bauer, Siddhartha S. Srinivasa, Tapomayukh Bhattacharjee, Matthew R. Walter, and Jan Peters. Benchmarking structured policies and policy optimization for real-world dexterous object manipulation, 2021.

[11] George Konidaris and Andrew Barto. Skill discovery in continuous reinforcement learning domains using skill chaining. In Y. Bengio, D. Schuurmans, J. Lafferty, C. Williams, and A. Culotta, editors, *Advances in Neural Information Processing Systems*, volume 22. Curran Associates, Inc., 2009. URL https://proceedings.neurips.cc/paper/2009/file/e0cf1f47118daebc5b16269099ad7347-Paper.pdf.

[12] Amy Mcgovern and Andrew G. Barto. Automatic discovery of subgoals in reinforcement learning using diverse

density. In *In Proceedings of the eighteenth international conference on machine learning*, pages 361–368. Morgan Kaufmann, 2001.

[13] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017. URL http://arxiv.org/abs/1707.06347.

[14] Richard S. Sutton, Doina Precup, and Satinder Singh. Between mdps and semi-mdps: A framework for temporal abstraction in reinforcement learning. *Artificial Intelligence*, 112:181–211, 1999.

[15] Emanuel Todorov, Tom Erez, and Yuval Tassa. Mujoco: A physics engine for model-based control. In *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pages 5026–5033, 2012. doi: 10.1109/IROS.2012.6386109.

[16] Manuel Wüthrich, Felix Widmaier, Felix Grimminger, Joel Akpo, Shruti Joshi, Vaibhav Agrawal, Bilal Hammoud, Majid Khadiv, Miroslav Bogdanovic, Vincent Berenz, Julian Viereck, Maximilien Naveau, Ludovic Righetti, Bernhard Schölkopf, and Stefan Bauer. Trifinger: An open-source robot for learning dexterity. *CoRR*, abs/2008.03596, 2020. URL https://arxiv.org/abs/2008.03596.