A Small Step Towards Self-Driving Scooters Using State-of-the-Art Deep Reinforcement Learning Algorithms

Steve Han
stevehan2001@utexas.edu
The University of Texas at Austin
CS394R, Final Project
May 2022

Abstract

The goal of this project is to create electric scooters that can steer themselves to target locations while keeping themselves balanced. After training simulated scooters with the PPO algorithm, this goal is mostly achieved in simulation. A video demonstration is found here. After careful consideration of Professor Stone's advice, I've decided not to release the codebase. Just kidding it's right here.

1 Introduction and Motivation

Electric scooters are rising drastically in popularity. I would cite the abundant sources online to justify this statement, but the growing number of rental scooters haphazardly lying on sidewalks and the ubiquitous presence of personal scooters around campus should make this fact self-evident. The truth is, electric scooters (or e-scooters for short) are some of the most efficient vehicles for micro-mobility. Its simple structure and foldable design make it more portable than bikes, and its larger wheels and better maneuverability make it safer than skateboards. Because of its energy efficiency compared to a car, it could become the future of micro-mobility.

However, it is not without its problems. First, misplaced rental scooters is a huge issue for sidewalks in cities. Second, rider injuries are growing faster than the popularity of e-scooters (Namiri et al., 2020). Self-driving scooters would be perfect to address these problems: scooters can drive themselves away to charge, and riders can let the AI take the wheel. There have already been developments to address the first issue: Go X and Tortoise have created scooters using training wheels to balance itself and a human teleoperator to navigate (GoX, 2020). However, the usage of training wheel reduces the speed and maneuverability that are so quintessential of the scooter. The first step towards autonomous scooters is to solve the balancing problem that's inherent to two-wheeled vehicles, and this project attempts to do precisely this.

Creating a self-steering scooter is a perfect reinforcement learning task. The task is sequential - setting the steering servo to one position will affect the tilt angle of the scooter in future time steps, and negative reward is given when the scooter falls down. The task is also Markovian given sufficient state. Even though self-balancing essentially boils down to an inverted pendulum problem, the highly non-linear system of a scooter and the diverse physical conditions in the real world don't allow for a 5-line solution like CartPole does (Jian, 2021). In the case of bicycles, efforts at developing bicycle controllers based on control theory and bicycle dynamics "only work properly in simulation environments and fails to apply to the real world due to disturbances present in a real environment (Choi et al., 2019)." To satiate modern deep RL algorithms' voracious appetite for data, I first create the scooter in simulation. Then, I train the virtual scooter on tasks that get progressively harder. In the beginning, all it needs to do is to balance itself and not fall down. Second, it needs to steer to a random goal without falling down. Third, it needs to do the above tasks even on rough terrain. Fourth, it must be adaptable to changes in its physical properties (center of mass, motor torque etc.). This ensures that when I build the physical scooter, inconsistencies from the simulated version don't cause everything to fail. Due to the timeline of the project, I was only able to fully accomplish the first two tasks.

2 Related Works

There hasn't been any research on scooter balancing to my knowledge, but there have been advances

in bike balancing to make Google's April Fools video (Google, 2016) come true. The most recent research I found (Choi et al., 2019) approaches this issue using state-of-the-art deep reinforcement learning algorithms, which I take inspiration from. In their paper, they create a simulation using equations for bicycle dynamics, and they train the policy using the DDPG (Deep Deterministic Policy Gradient) algorithm. The trained bicycle can not only balance itself but also go to any location. To make this possible, the agent can control the steering torque, the displacement of its center of mass, and the velocity of the bicycle. My project will be different from this in the following ways:

1. I use a fully-fledged physical simulation (Mujoco) instead of a set of equations for bicycle dynamics, which could be inaccurate. Mujoco simulation is more nuanced (taking into account even the angular momentum of the handlebars and the frictions in joints) and easier to include real-world elements (such as bumpy roads and obstacles). This facilitates the Sim2Real process.

2. I remove the ability for the agent to arbitrarily displace its center of mass, which seems to not be grounded in reality. This makes the balancing task harder, but it enables real-life scooters to drive with or without a human rider.

3. As suggested in the future works section of the paper, I use PPO instead of DDPG to hopefully achieve a more stable controller.

With these modifications, however, I'm not able to match all the results in the bike paper due to increased computational demand and my limited time. Specifically, my agent is not able to vary its velocity to achieve sharper turns.

Randløv and Alstrøm worked with Andrew G. Barto to achieve bicycle control with the Sarsa(λ) algorithm with replacing trace (Randløv and Alstrøm, 1998). They allowed the algorithm to displace the center of mass of the bike (by either -2, 0 or 2 cm) to achieve stability, and they use a shaping algorithm to teach the bike to navigate to a goal. In fact, the cover of our RL textbook (Sutton and Barto, 2018) shows the trajectories from the simulated bike. Isn't that cool? There is also an attempt at bicycle balancing without using RL (Deng et al., 2018). In their work, they use a PID controller and a balancing equation to steer an actual moving bike. They have to tune the PID parameters for different forward velocities, and their bike does not have a target direction to steer to. If I were to construct the scooter physically, I would follow their paper.

3 Background

3.1 Scooter Balancing

There are 3 main methods that I'm considering for balancing my scooter: steering control, mass balancing, and Control Moment Gyroscope (CMG). According to (Deng et al., 2018), steering control has low energy consumption but fails at low speeds while CMG has high energy consumption but can work in low speeds. Since steering control is the way that humans balance scooters, I choose to use steering control for high speeds with the intention of adding CMG in the future for lower speeds. I've also experimented with balancing by shifting the mass of the steering pole left and right using a servo as shown in figure 1, but I don't see this as a viable hardware implementation.

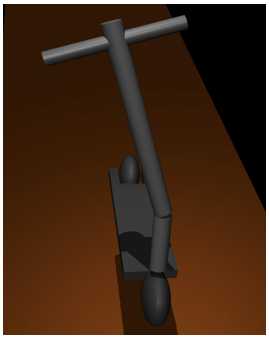

Figure 1: By adding a joint controlled by a servo towards the bottom of the steering pole, we can balance the scooter by shifting its mass. I'm not using this setup in this project, but it's a potential solution to balancing at low velocities.

So far, I've only been citing bike balancing research. However, is scooter balancing essentially the same as bike balancing? Is steering enough to keep a scooter balanced, or is the shifting of the rider's weight also essential? I struggled with these questions for a long time. After reading sources such as this summary report (Papadopoulos, 1987), I believe that the scooter does not have the same self-balancing abilities as a bike. Specifically, a bike can self-steer: when it leans to one side, it can automatically steer in the same direction to prevent itself from falling. A scooter is different from a bike since its front wheel is not large enough for the contact point between the ground and the wheel

to sit behind the steer axis (this scooter is designed this way, but it's not a popular design), so it cannot self-steer. In addition, the center of mass of a scooter relative to its wheels is way higher than that of a bike. However, this doesn't mean that we cannot manually achieve the self-steering effect. The algorithm can simply manually steer to the direction of leaning when it detects a tilt. Therefore, I believe that the results from bike balancing should be applicable to scooter-balancing.

My hypothesis for a potential balancing algorithm is like this: when the scooter leans to the right, it should steer to the right to prevent falling down. Then, to correct its path, it must first counter-steer to the right to create a lean to the left, then it can steer to the left to return to going straight. My hope is for the RL algorithm to pick up on this complex counter-steering maneuver. In the end of this project, the algorithm seems to be doing something like this in a microscopic scale, but the overall effect is too chaotic to demonstrate the counter-steering strategy clearly.

3.2 Reinforcement Learning

In case you don't know yet, Reinforcemnt Learning (RL) (Sutton and Barto, 2018) is a subfield of machine learning that involves an agent learning from interactions with an environment. At each timestep, the agent receives the state of the environment, chooses the next action to take, and receives a reward from the environment for taking that action. In addition, the environment is governed by a transition function that takes the state and action and returns a distribution over the next state and reward. The goal of the agent is to find a policy, or a probability distribution of actions given a state, to maximize its rewards.

One method of finding the optimal policy is the Policy Gradient method. The Policy Gradient theorem states gives a way of computing the derivative of a policy's value with respect to its parameters:

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla \pi(a|s,\theta),$$

where π is the policy parameterized by θ, and the distribution μ is the on-policy distribution under π (Sutton and Barto, 2018). This is the foundation of Actor-Critic methods, which also learn a value function to improve the stability of learning.

3.3 Proximal Policy Optimization

Proximal Policy Optimization, or PPO for short, is an improvement over the vanilla Actor-Critic method. Specifically, I'll be using PPO_Clip, which attempts to prevent overstepping during optimization by enforcing that the new policy be close to the old policy. Its update equation is (Achiam, 2018):

$$\theta_{k+1} = \arg\max_\theta \mathrm{E}_{s,a\sim\pi_{\theta_k}} \left[L(s,a,\theta_k,\theta) \right],$$

where L is

$$L(s,a,\theta_k,\theta) = \min \left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)} A^{\pi_{\theta_k}}(s,a), \right.$$
$$\left. \mathrm{clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}, 1-\epsilon, 1+\epsilon \right) A^{\pi_{\theta_k}}(s,a) \right).$$

For more information, please read the PPO article in the spinning-up page (Achiam, 2018).

4 Proposed Method

First, I create a physical simulation of the scooter using Mujoco and its Python wrapper in the dm_control library (Tunyasuvunakool et al., 2020). I added a velocity servo on the front wheel, a position servo on the handle bar, and some friction, armature, and mass for each component. The wheels of the scooter are modeled as ellipsoids, which can easily lose their balance even from numerical errors in the simulator. It's not a physically accurate simulation, but it's good enough for my purpose. Then, I wrapped it into an OpenAI Gym environment (Brockman et al., 2016), and started running Stable Baselines' implementation of PPO (Raffin et al., 2021) on it. At each 1/30th of a second, the agent receives the following state: the angle ω and angular velocity $\dot{\omega}$ of the scooter's tilt, as well as the current position θ and target position θ_t of the steering servo. The agent can choose to change the servo target position by .1, .01, 0, -.01, or -.1. The scooter is always in forward motion at around 12 miles per hour. The physical simulation is run as much as possible in-between each timestep.

In the balance-only scenario, the agent receives a reward of 1 for every time step that it's not falling. In the navigate-to-goal task, the state additionally includes the signed angle (ψ) between the direction that the scooter is facing and the goal, as (poorly) illustrated in 3. The reward function is also changed to

$$-(\omega^2 + 0.1\dot{\omega}^2) - 2\psi^2,$$

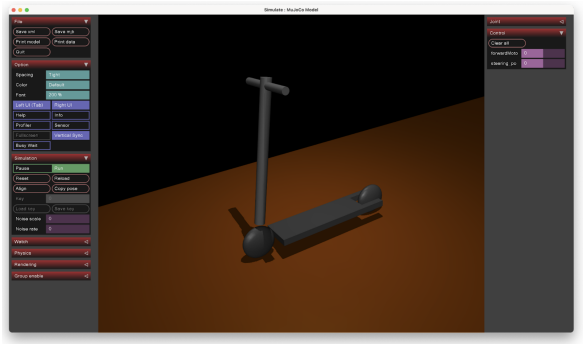

Figure 2: My Mujoco model for the scooter

with the reward being 40000 when the goal is reached, which happens when the agent is 4 meters away from the goal (I made this condition loose since it's difficult to control the precise location when the scooter is going 12 mph). The reason for these rewards will be described in the experiments section. Also, to discourage the scooter from constantly choosing $\pm.1$ options and swinging side-to-side, I give a $-.05$ reward for choosing them, a $-.02$ reward for choosing $\pm.01$ options, and a 0 reward for choosing to do nothing.

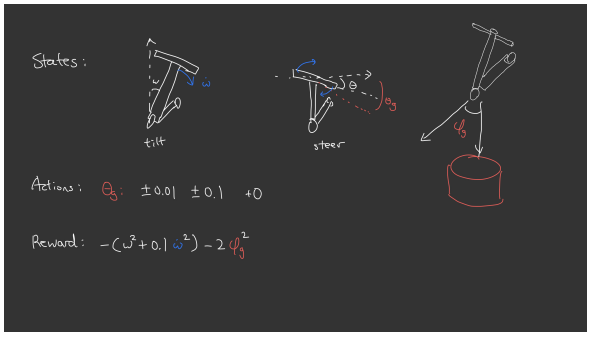

Figure 3: My attempt at explaining the variables present in the gym environment.

5 Experiments and Results

5.1 Balance Only

The first task is to simply keep itself balanced after starting with a random tilt of $[-.1, .1]$. As a comparison, I tried to use both the DQN and PPO algorithms to solve this gym environment. All the hyperparameters are kept the same as the default ones in Stable Baselines (clip_range = 0.2 for PPO and buffer_size = 1000000 for DQN). I ran both algorithm a couple of times, and these training curves in Figure 4 are representative of the runs I got for each algorithm.

It's curious that the off-policy DQN algorithm actu-

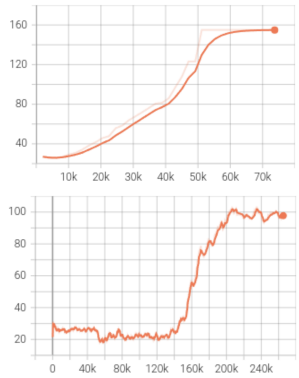

Figure 4: Training curves for PPO (up) and DQN (down). The x-axis is the total timestep and the y-axis is the reward. This is not a scientific comparison, but I've run each algorithm a couple times and the results are consistent with the plot. As shown in the plot, PPO is able to achieve a higher reward with less timesteps than DQN, both of which are using default hyperparameters from Stable Baselines.

ally took more timesteps to converge than PPO. Since this is a practical project, I'll be content with not knowing the reason and move on with PPO. Pretty soon the scooter learns to balance itself indefinitely, and this task is completely solved. The scooter essentially goes around in a big circle, which is uninteresting yet effective.

5.2 Navigate to Goal

The second task is to navigate to a fixed goal after starting at a fixed position facing a random direction. Specifically, the agent starts at $[0, 0]$ facing a random direction $[0, 2\pi]$, while the goal is at $[-50, 0]$. This one turned out to be a lot more difficult than the first one, since the agent needs to learn how to turn. I first wasted a lot of time by calculating the unsigned angle between the scooter and the goal, which is only in the range of $[0, \pi]$ and fails to capture the full picture. Then, I realized that the default network size in Stable Baselines was unable to capture the complexity of the problem, so I used hidden layers of 300 and 400 nodes for the policy network and 200 and 200 nodes for the value network, as the bike balancing paper (Choi et al., 2019) did. I initially used a reward function of $1 + \cos(\psi_g)$ per step, 1000 when goal is reached, and 0 when falling. However, the trained scooter never discovered the goal, and was instead content with spinning around in a circle collecting the positive reward that it was able to get. So, I switched over to the reward function used in the aforementioned paper, which was much crueler (having all

negative rewards) and was better tuned to encourage the agent to go towards the goal.

The training process was also challenging. With the environment randomizing the starting angle at the beginning of each episode, the agent seemed to be torn between policies for different starting angles and never learned anything useful. I tried making the scooter always face in the opposite direction from the goal, but after days of training the policy still wouldn't converge. So, I instead set the angle to be the constant $\pi/2$ to help the agent learn how to make a simpler 90° turn. After the agent is trained, I then set the angle to be a constant 3.14, which proved feasible for the agent to learn. Perhaps surprisingly, all other angles are also taken care of by this agent trained on only two angles, since the learned policy covers all possible angles towards the goal. After some more training with random staring angles, the agent is able to solve this task almost perfectly. Figure 5 shows the training curve.

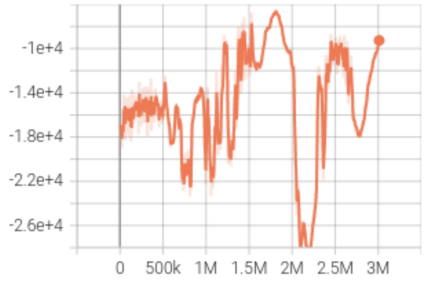

Figure 5: The training curve for the navigate-to-goal agent, with timesteps plotted against the average reward. Note that I changed the starting angle from $\pi/2$ to 3.14 around 2M timesteps.

Empirically, as shown in Figure 6, only 1 out of 200 trajectories failed.

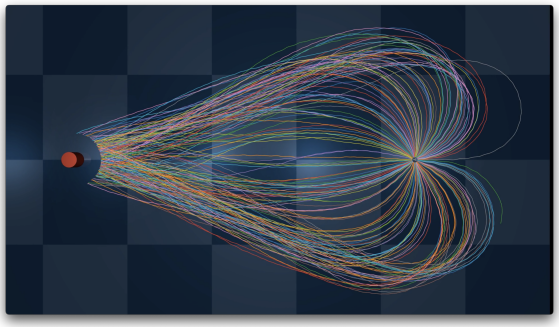

Figure 6: Out of 200 trajectories, only 1 of them (the green line towards the bottom right) failed to reach the goal.

I also tested the same trained network on other domains. The first one is to navigate from a random starting location and direction to a random goal. The random starting location is within the rectangle centered at origin of length 40 and width 10. The random goal is a random point within the annulus formed by concentric circles centered on the scooter with radii 50 and 100 (I chose these numbers because of the limitations of the state representation, as will be discussed in the future works section). In 200 trials, there were no failures (Figure 7).

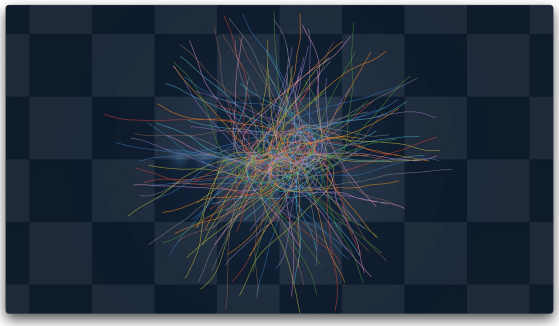

Figure 7: The scooter is able to navigate from a random starting location and direction to a random goal within 50 to 100 meters.

Finally, instead of resetting the simulation at the end of each episode, I let the scooter continue steering towards a different goal. Similar to the previous task, the goal is chosen randomly within 50 to 100 meters away from the scooter. This leads to the scooter being able to follow any path specified by the goals, as shown in Figure 8.

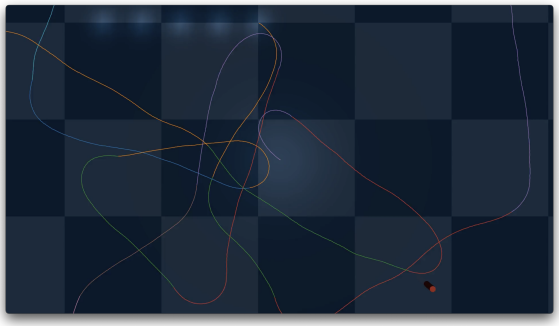

Figure 8: The scooter is able to follow a sequence of random goals. Each segment of different color is a different episode.

6 Future Works

6.1 Improvements to Existing Methods

The current simulation environment is not efficient. The usage of a single core in the CPU and 10% VRAM in the GPU during training means that the environment can benefit hugely from parallelization. Whether it is stuffing multiple scooters in the same Mujoco environment like the training of ANYmal (Hwangbo et al., 2019) or simply spawning multiple Mujoco instances, there is room for improving the performance.

The state representation is not complete. In other words, the environment is partially observable and not fully Markov. The missing angular acceleration for the tilt of the scooter does not appear to be important, but the lack of distance to goal is pivotal. A goal that is 100 meters away and 50 degrees away from the scooter and a goal that is 10 meters away and 50 degrees away from it are very different. Following this same policy, if it can reach the goal from 100 meters, it might end up spinning around the goal indefinitely when it's close. This is why I didn't allow goals to be less than 50 meters away from the scooter - sometimes the scooter would run in circles around the goal. I tried adding the distance to goal as a state, but the training process was taking long and I gave up (having already kept my PC running constantly for 2 weeks as well as having wasted way too much time on the animations).

Another partial solution to that infinite loop around goal problem is to let the agent vary its velocity, which I didn't have time to test. Varying velocity will let the agent make sharper turns to reach goals otherwise unreachable with constant velocity. With this mechanism, we can also test the practical problem of starting a scooter up from rest, which is usually done by humans by kicking it off. Other ways of balancing such as CMG and mass balancing might be needed during the low-velocity period.

Adding bumpy roads to the simulation is also important. I've tried adding a few bumps on the floor, but since the steering joint of the scooter has high friction, the bumps didn't cause the scooter to change its steering direction. The scooter just jumped over them as if nothing happened. I need more testing to reach any conclusions.

6.2 Sim2Real

This is arguably the most difficult part of this project. First, Mujoco simulation needs to be tuned to match the physical mechanics as much as possible. Second, I can try Professor Stone's method for creating a "grounded simulator" based on real-world trajectories (Hanna et al., 2021). Third, I'm thinking of making the agent adaptable to different parameters, as small as a slightly different mass for the handlebar and as large as adding the mass of a rider. This would allow the scooter to not only work on real-world machines but also accommodate riders of different weights. To do this, the agent should be able to experiment with the unknown environment for a few timesteps in the beginning of the episode to learn these parameters (for example, by accelerating and seeing how fast the scooter actually moves). This idea is more fleshed out in the recent work on training a legged robot to be able to quickly adapt to different environments (Kumar et al., 2021), which I'm keen to try on the scooter environment. Their method should allow the scooter to adapt to different road conditions.

Also, to improve the training efficiency and reduce the wobbliness of the scooter, I plan to use imitation learning to teach the scooter how to counter-steer efficiently and the options framework to build up low-level skills such as going straight and turning for specific degrees.

7 Conclusions

As promised in the abstract, the simulated scooter is successful in all 4 domains: simply balancing, steering to a fixed goal, steering to random goals, and following a sequence of goals.

This has been a very interesting project, and I'll be thrilled if I can continue with those future works during my Master's thesis. Regardless, I've had a lot of fun with both this project and the class, and I don't regret a single second I spent.

References

Joshua Achiam. 2018. Spinning Up in Deep Reinforcement Learning.

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

SeungYoon Choi, Tuyen P. Le, Quang D. Nguyen,

Md Abu Layek, SeungGwan Lee, and TaeChoong Chung. 2019. Toward self-driving bicycles using state-of-the-art deep reinforcement learning algorithms. *Symmetry*, 11(2).

Wenhao Deng, Skyler Moore, Jonathan Bush, Miles Mabey, and Wenlong Zhang. 2018. Towards automated bicycles: Achieving self-balance using steering control. V002T24A012.

Nederland Google. 2016. Introducing the self-driving bicycle in the netherlands.

GoX. 2020. Worlds 1st fleet of 100x self driving scooters in peachtree corners - go x apollo.

Josiah P. Hanna, Siddharth Desai, Haresh Karnan, Garrett Warnell, and Peter Stone. 2021. Grounded action transformation for sim-to-real reinforcement learning. *Special Issue on Reinforcement Learning for Real Life, Machine Learning, 2021*.

Jemin Hwangbo, Joonho Lee, Alexey Dosovitskiy, Dario Bellicoso, Vassilios Tsounis, Vladlen Koltun, and Marco Hutter. 2019. Learning agile and dynamic motor skills for legged robots. *Science Robotics*, 4(26):eaau5872.

Xu Jian. 2021. How to beat the cartpole game in 5 lines.

Ashish Kumar, Zipeng Fu, Deepak Pathak, and Jitendra Malik. 2021. Rma: Rapid motor adaptation for legged robots.

Nikan K. Namiri, Hansen Lui, Thomas Tangney, Isabel E. Allen, Andrew J. Cohen, and Benjamin N. Breyer. 2020. Electric Scooter Injuries and Hospital Admissions in the United States, 2014-2018. *JAMA Surgery*, 155(4):357–359.

Jim Papadopoulos. 1987. Bicycle steering dynamics and self-stability: A summary report on work in progress.

Antonin Raffin, Ashley Hill, Adam Gleave, Anssi Kanervisto, Maximilian Ernestus, and Noah Dormann. 2021. Stable-baselines3: Reliable reinforcement learning implementations. *Journal of Machine Learning Research*, 22(268):1–8.

Jette Randløv and Preben Alstrøm. 1998. Learning to drive a bicycle using reinforcement learning and shaping. In *Proceedings of the Fifteenth International Conference on Machine Learning*, ICML '98, page 463–471, San Francisco, CA, USA. Morgan Kaufmann Publishers Inc.

Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.

Saran Tunyasuvunakool, Alistair Muldal, Yotam Doron, Siqi Liu, Steven Bohez, Josh Merel, Tom Erez, Timothy Lillicrap, Nicolas Heess, and Yuval Tassa. 2020. dm_control: Software and tasks for continuous control. *Software Impacts*, 6:100022.