DeepWalker: Teaching a Quadruped to Walk

~~Sean Fitzpatrick~~ Robin Yohannan

1. Problem

At a high level, the goal for our project was to produce a control policy that would command a quadruped robot to walk in a straight line. The approach we chose, based on prior experience, was to use reinforcement learning to train a walking policy in simulation and deploy the resulting policy on the physical robot. We envisioned an iterative process in which the experience gathered while running the trained policy on the physical robot would be collected and fed back into subsequent training runs in simulation to improve the policy further. As a point of comparison, we planned to also implement a simple walking gait algorithm that would move each leg in a defined pattern to have the robot crawl forward. The two approaches would be compared on different terrain: flat ground and rough terrain. Time permitting, we planned to try other RL algorithms to compare performance with the original two approaches. The following diagram shows an overview of what we planned to accomplish with the D'Kitty quadruped robot.

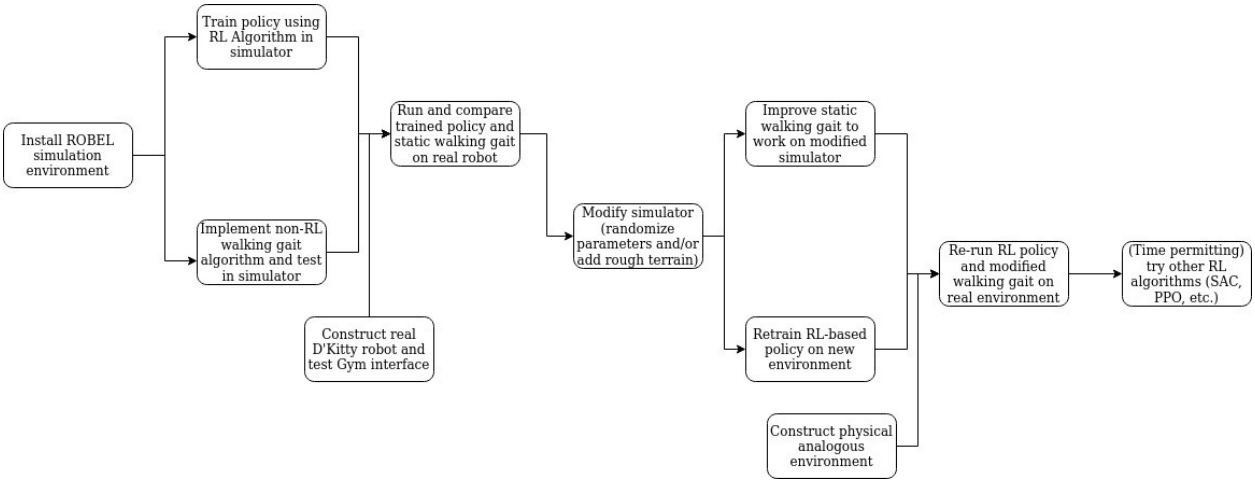

Figure 1: Original roadmap for final project

After the course switched into remote learning mode, we decided to pare down the list of tasks we planned in order to focus mostly on the simulator. The following diagram shows a reduced set of tasks that we chose to pursue using the ROBEL D'Kitty MuJoCo simulator.

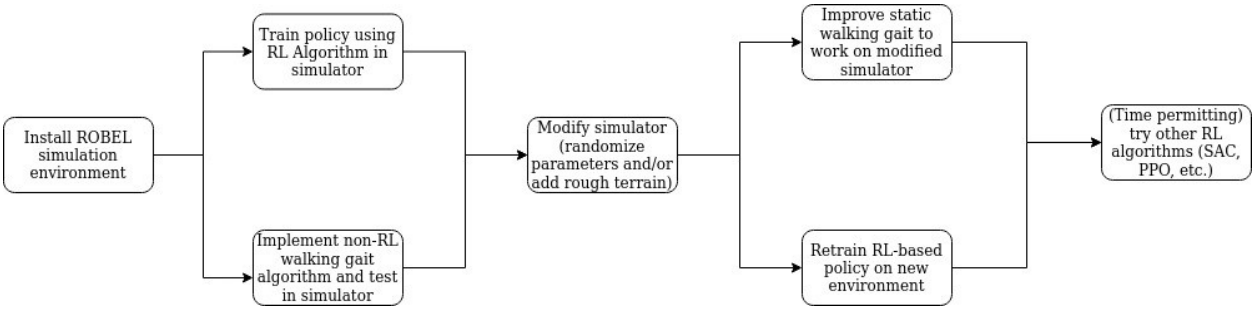

Figure 2: Reduced-scope roadmap for final project

From the subtasks that we narrowed down, we were able to get the simulator up and running, implement a simple non-RL walking gait algorithm and implement a popular off-policy reinforcement learning algorithm, DDPG. We leveraged what was included with the simulator to test both algorithms on flat terrain and on rough terrain with randomized dynamics enabled. Both algorithms shared a common input in the OpenAI Gym API that the D'Kitty simulator supported. The walking task that the simulator implements is modeled as an MDP with a 52-dimensional observation space that consists of the 6DOF pose of the robot, along with the position and velocities of the 12 leg joints, the previous action, range and bearing to the goal, and a "uprightness" metric of the overall robot along the vertical axis. The action space is 12-dimensional and represents the desired position of each of the 12 leg joints at the current timestep. The reward function in the environment encourages the robot to stay upright while moving towards a goal location. Using the Gym API, the non-RL algorithm was designed to be a simple open-loop policy that would produce a set of actions that would result in a periodic gait. After implementing the RL algorithm, the end result would be a trained neural network that would produce the appropriate action to take for the current state of the robot in service of the ultimate goal of walking forward towards a goal location.

2. Process

The first task that we tackled was to get the D'Kitty simulator up and running. Following the instructions in the README file associated with the GitHub repo, we ran into a few roadblocks after discovering missing prerequisite packages that we needed to install after some investigation. The simulator required us to install the MuJoCo physics engine, which carries a free student license that we used to register our installation. After setting up the prerequisites, we were able to build the ROBEL benchmark, which included the D'Kitty and the D'Claw simulator, without further hiccups. After installing the benchmark environment, we wrote a simple Python script to input actions based on a random policy using the Gym API and confirmed that the simulator worked on our laptops.

After getting the simulator working, we started writing a non-RL walking algorithm for the quadruped. We familiarized ourselves with the concept of stability margin, which defines the minimum distance of the projected center of mass from the support pattern made by at least 3 of the legs [2]. We tried to implement a "creeping" gait, which results in the robot lifting and planting one leg at a time while keeping the 3 other legs on the ground for stability. When we tried to implement a wave gait policy, which gives the robot the maximum longitudinal stability margin, we ran into a problem with the way that the quadruped agent was defined in the

simulator. We discovered that the range of the servos was limited, -60 to 60 degrees, and the neutral, zero action translated in a crouched pose for the robot. We had trouble producing a set of join angles that would result in a leg move that can reach far forward. Without the ability to reach forward to shift the center-of-mass, we started to try different options to produce a non-RL gait generator. Eventually, we implemented a simple approach in which the algorithm alternates setting the position on the bottom-most 4 servos from -60 degrees and 0 degrees. The algorithm sets the joint position for one leg at a time and cycles between each leg to obtain a periodic gait cycle. We collected results in the simulator under normal, flat conditions and then ran the same algorithm in an environment with random dynamics and rough terrain enabled.

After collecting data with a non-RL walking algorithm, we set upon the task of implementing the off-policy reinforcement learning algorithm, DDPG, and testing it in our simulator [3]. We started with an implementation of DQN, written in PyTorch, and used the OpenAI Gym Baselines, written in Tensorflow, to modify the DQN implementation to add actor neural networks, clean up the codebase, and add the ability to easily set different hyperparameters in one place [8]. Eventually, we got the code to work on a simpler Gym environment, the 'Ant' walker environment, After that, we tried to start training runs for the D'Kitty environment. The following table shows the hyperparameters chosen for 2 of these simulation training runs. While we continued to work on our DDPG implementation, we discovered that increasing the replay buffer size and the exploration noise improved the performance of the trained policy, but we could not see further improvement beyond the parameters we chose in training run #2.

	Training Run #1	Training Run #2
Max Timesteps	1500000	1500000
Exploration Noise	0.1	0.25
Batch Size	256	256
Replay Buffer Size	1000	15000

Table 1: DDPG parameters chosen in two training runs we performed with the D'Kitty simulator

One of the tasks that we planned in our roadmap was to modify the environment to add rough terrain and randomized dynamics. While working with the simulator, we discovered that the original authors added a *DKittyWalkRandomDynamics* environment to add uncertainty in the parameters of the robot and the surface that it walks on. The specific modifications that were made to the environment were to randomize the joint gains, damping, friction loss, geometry friction coefficient, and masses. The rough terrain was produced by creating a field and randomizing the height of the field up to 0.05 meters. After making the discovery of the alternative environment, we incorporated it into our testing.

3. Results

Flat Terrain, Fixed Dynamics

The following is a summary of what we found when we tried running the simple non-RL algorithm on the fixed dynamics simulation environment.

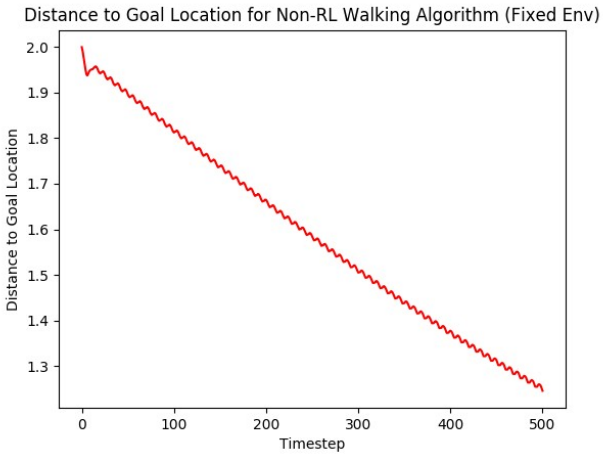

Figure 3: Distance to goal (in meters) of robot using non-RL algorithm to walk in fixed environment

In the plot above, with perfect conditions, the simple periodic walking gait can slowly approach the goal location without too many problems. However, because the control policy was open-loop, the pose of the robot slowly drifted off course and the distance-to-goal began to flatten out towards the end of the run.

While we trained a policy using our DDPG implementation on the fixed environment, we collected returns after each episode, which usually ended in either the robot falling over or after the maximum episode length was reached. The following plot presents the returns per episode alongside the episode lengths and mean-squared error between the values of the current states and their TD-targets during the training session.

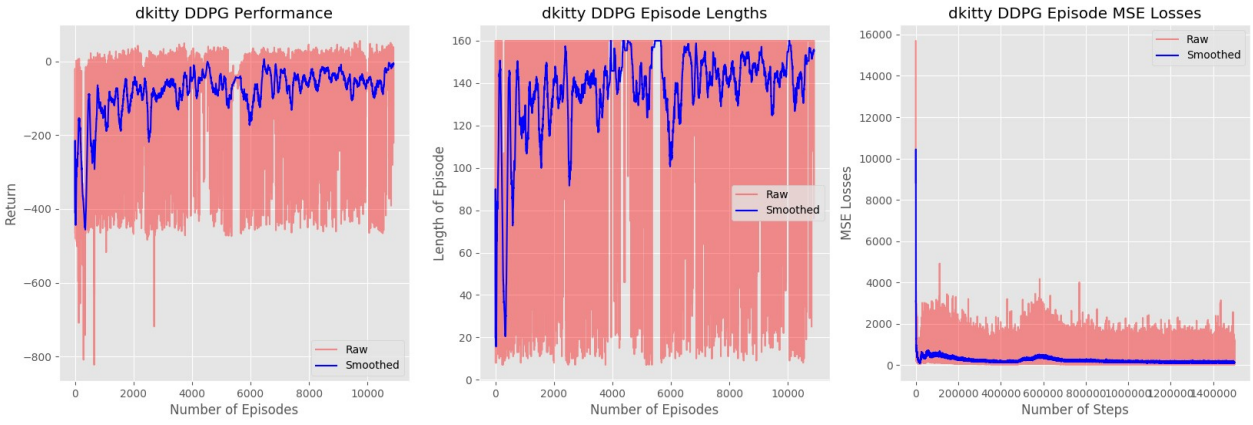

Figure 4: Training data collected on fixed environment

After completing the training, we ran the final policy on the fixed environment and found the following results after 500 timesteps:

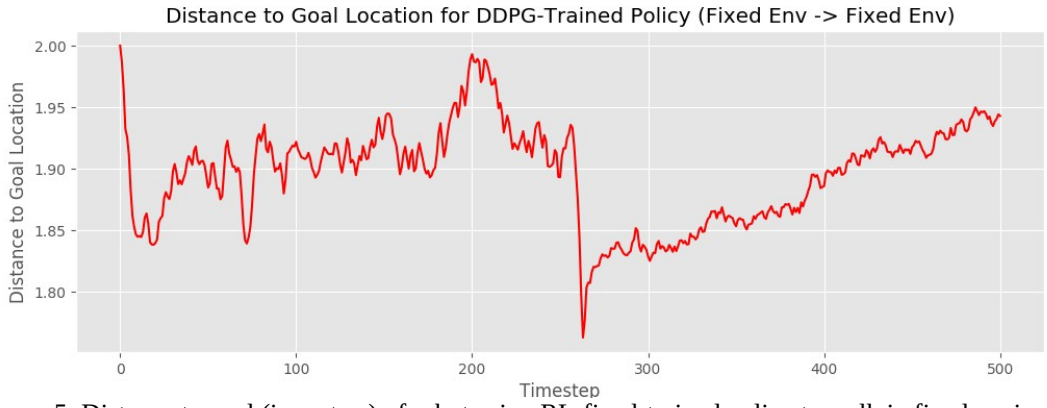

Figure 5: Distance to goal (in meters) of robot using RL-fixed-trained policy to walk in fixed environment

The trained policy did not work well compared to the simple non-RL algorithm for a number of reasons. The main reason was that the robot commanded larger joints movements which resulted in the robot trying primarily to maintain stability over the additional objective of moving towards the goal location. Also, since most of the episodes during training ended in the robot falling down, there was little experience that could have used to optimize the learned policy during training.

We then initiated another training run with random dynamics and rough terrain enabled and got the following data during the run:

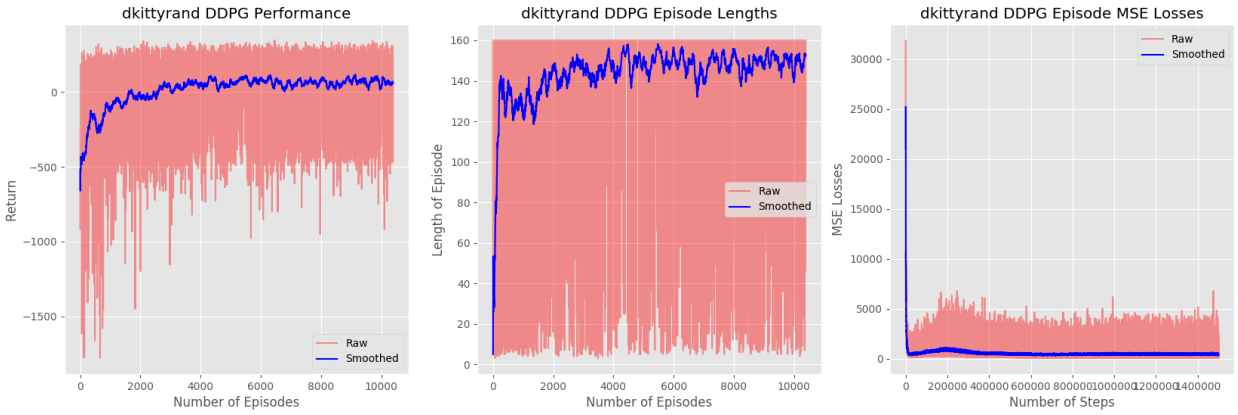

Figure 6: Training data collected on random environment

During the training run, the average return was higher than the training run in the fixed-dynamics simulator but the average episode length was shorter. After the training run was completed, we ran the final policy in the fixed-dynamics environment and observed the following results.

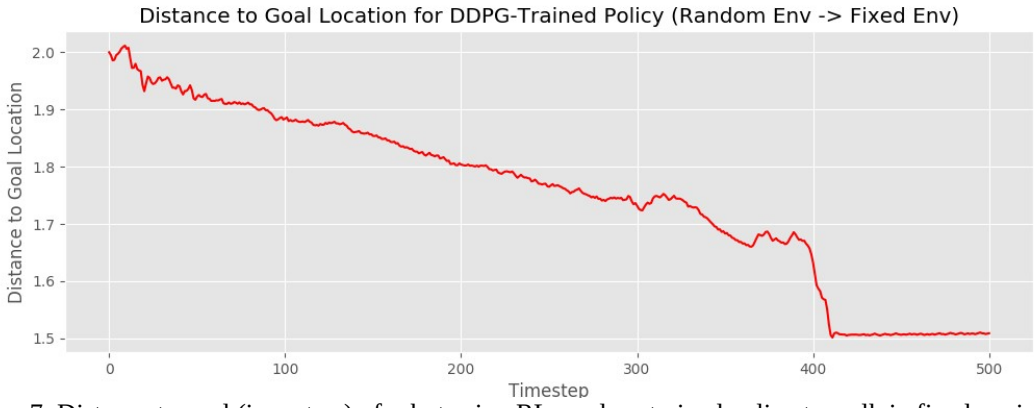

Figure 7: Distance to goal (in meters) of robot using RL-random-trained policy to walk in fixed environment

With the trained policy generated from the randomized simulator, the robot splayed out into a stable configuration at first and started moving slowly toward the goal location, which was an improvement over the fixed-simulator trained policy. Eventually, though, the center-of-mass moved outside the support polygon and the robot tipped over. After that point, the distance-to-goal plateaued for the last 100 timesteps.

Rough Terrain, Random Dynamics

After switching to the randomized simulator with rough terrain for testing final policies, we first tried the non-RL walking algorithm and observed the following results.

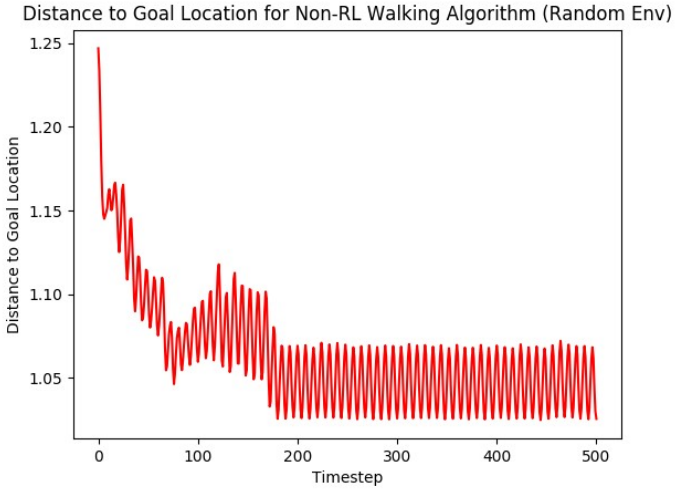

Figure 8: Distance to goal (in meters) of robot using non-RL algorithm to walk in random environment

The robot quickly got stuck in multiple divots in the simulated ground surface and made no progress towards the goal. In subsequent runs with the non-RL policy, we observed two common outcomes: either the robot got stuck or fell over within the first few timesteps of each run.

We then tried running the policy trained in the fixed-dynamics simulator in the randomized environment and observed the following results.

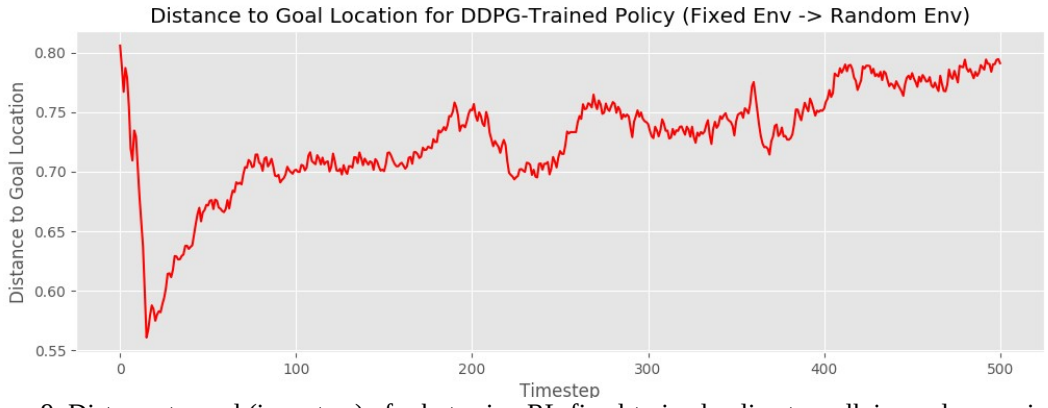

Figure 9: Distance to goal (in meters) of robot using RL-fixed-trained policy to walk in random environment

In the randomized environment, the robot, using the control policy trained in the fixed environment, quickly lost its balance and struggled to regain stability while moving farther away from the goal location. We expected this result as we speculated that the trained policy overfit to the fixed environment and therefore would perform poorly in the uncertain environment.

Finally, we tried running the randomized-trained policy on the randomized environment and observed the following results.

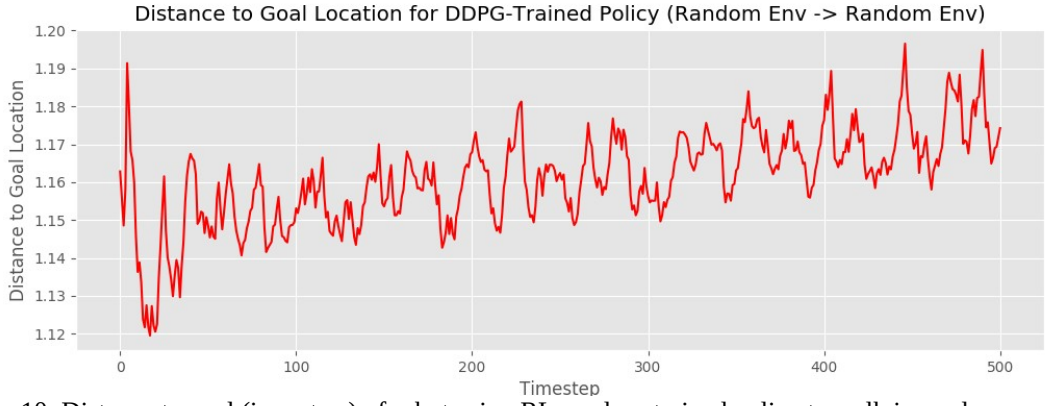

Figure 10: Distance to goal (in meters) of robot using RL-random-trained policy to walk in random environment

The random-trained robot performed similarly to the robot using the fixed-trained policy in that the robot quickly lost stability and spent the rest of the run trying to regain stability. Compared to the fixed-trained policy, the random-trained policy caused the robot to thrash harder to regain stability.

4. Resources

For this project, we made heavy use of the ROBEL benchmark environment, specifically the D'Kitty quadruped simulator [1]. The DDPG code and the non-RL walking gait algorithm can be found on GitHub[1]. The README file provides instructions and pointers for installing the ROBEL benchmark environment and running the training scripts. The repository also contains saved models for trained actors that can be run using the scripts located in the repository.

1 https://github.com/rvy11/ddpg_impl

5. Reflection/Conclusion

Given the difficulty in tuning hyperparameters for stable results using the DDPG algorithm, it is difficult to make a firm conclusion about the value of using DDPG for teaching a quadruped robot to walk. Based on the results we found, we can say that training in a simulator with randomized dynamics produced a policy that worked reasonably well in a fixed-dynamical environment. On the other hand, we did not have time to rewrite the simple non-RL algorithm to work better in the randomized environment, so we could not compare the RL-trained policies with a non-RL algorithm that was complex enough to handle the uncertainty in the new environment. For future work, it would make sense to try other families of RL algorithms such as on-policy algorithms like Proximal Policy Optimization (PPO) or Trust Region Policy Optimization (TRPO), which have been used by other researchers to train quadruped walking gaits [4][5]. We also think that an algorithm like Soft-Actor Critic (SAC) would work better than DDPG because SAC rewards actions with a dual objective to maximize the expected return as well as maximizing the entropy in the policy selection [6]. We believe than an algorithm that rewards exploration could result in more stable performance than what we've observed with DDPG. One more avenue for future work would be to try other non-RL learning methods such as imitation learning to train the agent with better data for at least the beginning of the training run [7]. At the end, we can safely say that quadruped control for walking is a hard enough problem to warrant machine learning techniques. We found evidence that generating a robust walking gait controller is not a solved problem but using a randomized simulator is a promising direction that we would like to pursue in the future with real hardware in a real-world environment.

References

[1] M. Ahn *et al.*, “ROBEL: Robotics Benchmarks for Learning with Low-Cost Robots,” *arXiv:1909.11639 [cs, stat]*, Dec. 2019, Accessed: Apr. 24, 2020. [Online]. Available: http://arxiv.org/abs/1909.11639.

[2] S. Kajita and B. Espiau, “Legged Robots,” in *Springer Handbook of Robotics*, B. Siciliano and O. Khatib, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 361–389.

[3] T. P. Lillicrap *et al.*, “Continuous control with deep reinforcement learning,” *arXiv:1509.02971 [cs, stat]*, Jul. 2019, Accessed: Apr. 24, 2020. [Online]. Available: http://arxiv.org/abs/1509.02971.

[4] J. Tan *et al.*, “Sim-to-Real: Learning Agile Locomotion For Quadruped Robots,” *arXiv:1804.10332 [cs]*, May 2018, Accessed: Apr. 24, 2020. [Online]. Available: http://arxiv.org/abs/1804.10332.

[5] J. Hwangbo *et al.*, “Learning agile and dynamic motor skills for legged robots,” *Sci. Robot.*, vol. 4, no. 26, p. eaau5872, Jan. 2019, doi: 10.1126/scirobotics.aau5872.

[6] T. Haarnoja, S. Ha, A. Zhou, J. Tan, G. Tucker, and S. Levine, “Learning to Walk via Deep Reinforcement Learning,” *arXiv:1812.11103 [cs, stat]*, Jun. 2019, Accessed: Apr. 24, 2020. [Online]. Available: http://arxiv.org/abs/1812.11103.

[7] X. B. Peng, E. Coumans, T. Zhang, T.-W. Lee, J. Tan, and S. Levine, “Learning Agile Robotic Locomotion Skills by Imitating Animals,” *arXiv:2004.00784 [cs]*, Apr. 2020, Accessed: Apr. 24, 2020. [Online]. Available: http://arxiv.org/abs/2004.00784.

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