Hopper: Reinforcement Learning-Based Control of a Wheeled Bipedal Robot

Parthiv Krishna
parthiv@stanford.edu

Aaron Schultz
azs@stanford.edu

***Abstract*—This project involves the application of Deep Deterministic Policy Gradient (DDPG) and Soft Actor-Critic (SAC) reinforcement learning algorithms to the control of Hopper, a small, wheeled biped robot. We train and test these algorithms in a physics simulator that models the dynamics of the robot. These tests consist of balancing, velocity control, and jumping, which are all scenarios that we envision being used frequently for Hopper. We compare the performance of DDPG and SAC against a traditional LQR controller as a baseline.**

I. Introduction

In this project we apply reinforcement learning techniques to control Hopper, a small robot that the Extreme Mobility team within Stanford Student Robotics has been developing over the past several months. As both authors are part of the Extreme Mobility team, this robot is of particular interest. A CAD render of Hopper can be seen in Fig. 1.

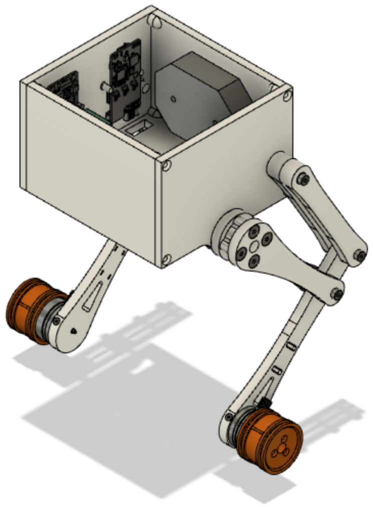


Fig. 1. Hopper

Hopper is a wheeled bipedal robot; it moves by making use of driven wheels that are attached to the end of two actuated legs. To move effectively, Hopper must maintain its balance while also responding to the desired control input to reach a target position or velocity. Additionally, Hopper can theoretically jump by rapidly extending its legs, and land after jumping by controlling the wheels to maintain stability.

Traditional control methods like PID or optimal control methods like LQR tend to work fairly well for balancing problems, they become significantly less effective under conditions where traction is lost. In the context of Hopper, when the robot jumps, it loses the ability to drive its wheels with high traction and minimum slipping. We demonstrate that a reinforcement learning-based controller for Hopper could provide a more reliable and robust approach to balancing under driving and jumping conditions.

In this project we apply a reinforcement learning-based control approach to the robot. Hopper is intended to be a platform to test and benchmark different control algorithms, so we also compare the results obtained by reinforcement learning with those from LQR, an optimal control method.

II. Related Work

Over the past two decades, reinforcement learning has continually been applied to robotics control problems [1]. The ability of machine learning to model complex and poorly understood systems makes it a compelling way to work with the highly non-linear dynamics and complex real-world interactions of robots. Reinforcement learning allows a robot to learn an optimal behavior through these dynamics via trial and error without having to understand all the minutiae. To this end, most reinforcement learning for robotic applications is done using policy gradient methods, where an optimal policy is learned directly without first modeling the system (this is known as model-free reinforcement learning). In lieu of learning the dynamics model these algorithms instead learn a value function which aids in the optimization of the policy, this method is known as the actor-critic method [2] [3].

In 2015, Lillicrap et al. published their paper "Continuous Control with Deep Reinforcement Learning" which introduced an actor-critic model-free algorithm for problems with a continuous action space [4]. Their algorithm, DDPG, adapted the success of Deep Q-Learning, which was shown to be capable of human-level performance in Atari video games [5], and extended it to the be capable of outputting continuous actions. In their paper they demonstrated that this approach was able to robustly solve a wide range of classical control problems and other simulated physical tasks. And it has since been applied to real-world robot systems as well [6].

In 2018, Haarnoja et al. attempted to improve upon the results of DDPG with their own algorithm Soft Actor-Critic (SAC) [7]. Similar to DDPG it is an off-policy model-free actor-critic algorithm, however, it follows a maximum entropy framework instead of the standard maximum reward framework. This maximum entropy framework leads to better exploration and attempts to overcome some of the brittleness and hyperparameter sensitivity challenges that DDPG has. In their paper they showed that SAC gave comparable results to DDPG on simple simulated robotics tasks and greatly exceeded DDPGs performance on more complex systems. SAC has now also been applied to real-world mobile robot control problems [8].

III. Methods

For this project we applied both of the reinforcement learning algorithms discussed above to control Hopper: Deep Deterministic Policy Gradient (DDPG) and Soft Actor-Critic (SAC). Both of these algorithms are off-policy actor-critic algorithms for working in a continuous space, making them a good choice for robotics applications. The algorithms being off-policy means that they can train on data regardless of the policy used to generate it, which greatly increases the efficiency by allowing for training across many uncorrelated episodes.

The DDPG algorithm was published by Lillicrap et al. in 2015, where they applied it to simulated robotic systems [4]. DDPG consists of a critic network $Q(s,a|\theta^Q)$ and actor network $\mu(s|\theta^\mu)$. A copy of both these networks, $Q'(s,a|\theta^{Q'})$ and $\mu'(s|\theta^{\mu'})$, are employed to calculate target values, and their weights are updated to slowly track those of the learned networks in order to increase stability. At every time step during training an action is selected by $a_t = \mu(s_t|\theta^\mu) + \mathcal{N}_t$, where $\mathcal{N}_t$ is noise added for exploration. The action is executed and the resulting reward r_t and state s_{t+1} are observed. The transition (s_t, a_t, r_t, s_{t+1}) is then stored in a replay buffer. To update the network weights a random minibatch of N transitions (s_i, a_i, r_i, s_{i+1}) are selected from the replay buffer. The target value for each transition is computed:

$$y_i = r_i + \gamma Q'(s_{i+1}, \mu'(s_{i+1}|\theta^{\mu'})|\theta^{Q'})$$

And the critic is updated to minimize the loss:

$$L = \frac{1}{N}\sum_i (y_i - Q(s_i, a_i|\theta^Q))^2$$

The actor is then updated using the sampled policy gradient:

$$\nabla_{\theta^\mu} J \approx \frac{1}{N}\sum_i \nabla_a Q(s,a|\theta^Q)|_{s=s_i, a=\mu(s_i)} \nabla_{\theta^\mu}\mu(s|\theta^\mu)|_{si}$$

Finally, the target networks are updated with $\tau \ll 1$:

$$\theta^{Q'} := \tau\theta^Q + (1-\tau)\theta^{Q'}$$
$$\theta^{\mu'} := \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$

DDPG has been shown to work well in practice, but there is some brittleness around the fact that exploration is done by adding noise to a deterministic action. This can mean that exploration is only done in a narrow range around the current policy.

The SAC algorithm was published by Haarnoja et al. in 2018, who also applied it to simulated control problems [7]. In contrast to the deterministic actions selected by DDPG, SAC selects actions probabilistically. The goal of this algorithm is to maximize not only the reward but also the entropy of the policy in order to encourage exploration. This maximum entropy objective attempts to find a policy that maximizes:

$$J(\pi) = \sum_{t=0}^{T} \mathbb{E}_{(s_t,a_t)\sim\rho_\pi}[r(s_t,a_t) + \alpha\mathcal{H}(\pi(\dot{|}s_t))]$$

where $\mathcal{H}$ is the entropy measure and α is the temperature parameter that determines how important the entropy is. To do this the algorithm alternates optimizing three different networks: a state value function $V_\psi(s_t)$, a soft Q-function $Q_\theta(s_t,a_t)$, and a policy $\pi_\phi(a_t|s_t)$. The weights for these networks are ψ, θ, and ϕ respectively. The algorithm also makes use of a slow tracking target value network, like used in DDPG, parameterized by $\bar{\psi}$. The updates for these parameters are computed as follows. The soft value function is trained to minimize the squared residual error:

$$J_V(\psi) = \mathbb{E}_{s_t\sim\mathcal{D}}\left[\tfrac{1}{2}(V_\psi(s_t) - \mathbb{E}_{a_t\sim\pi_\phi}[Q_\theta(s_t,a_t) - \log\pi_\phi(a_t|s_t)])^2\right]$$

where $\mathcal{D}$ is the replay buffer. The soft Q-function parameters are trained to minimize the soft Bellman residual:

$$J_Q(\theta) = \mathbb{E}_{(s_t,a_t)\sim\mathcal{D}}\left[\frac{1}{2}\left(Q_\theta(s_t,a_t) - \hat{Q}(s_t,a_t)\right)^2\right],$$

with

$$\hat{Q}(s_t,a_t) = r(s_t,a_t) + \gamma\mathbb{E}_{s_{t+1}\sim p}[V_{\bar{\psi}}(s_{t+1})]$$

Finally, the policy parameter is learned by minimizing the expected KL-divergence:

$$J_\pi(\phi) = \mathbb{E}_{s_t\sim\mathcal{D}}\left[\mathrm{D}_{\mathrm{KL}}\left(\pi_\phi(\dot{|}s_t)\middle\|\frac{\exp(Q_\theta(s_t,\dot{)}))}{Z_\theta(s_t)}\right)\right]$$

Gradients for all these minimizations are provided in the paper, but are not listed here for brevity [7].

The experiments and testing for this project were conducted in a physics simulator built with PyBullet and OpenAI Gym (see below). We modeled Hopper using a Unified Robot Description Format (URDF) file. The URDF describes the inertia of the various components (links), the connections between the links (joints), and the torque ranges of the motors that drive each joint. Each link also has an associated state which evolves over time based on gravity and control actions. The state for each link consists of:

- Position [x, y, z]
- Linear velocities [$\dot{x}$, $\dot{y}$, $\dot{z}$]
- Orientation [ϕ, θ, ψ] (pitch, roll, yaw)
- Angular velocities [$\dot{\phi}$, $\dot{\theta}$, $\dot{\psi}$]

As seen in Fig. 2, the simulator's renderer displays a visual representation of Hopper's state, with the location and orientation of the various links and joints visible.

As a modeling decision, we have simplified the legs to be linear sliders rather than the multi-bar linkages seen on the real robot. The hierarchical nature of the URDF format makes representing closed-loop linkages impossible. On a mechanical level, the leg linkage was designed to convert rotation at the hip from the motor into vertical linear motion at the foot. As

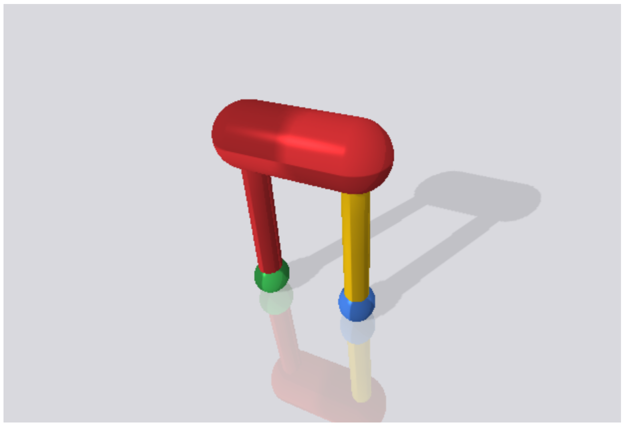


Fig. 2. Hopper PyBullet Simulation

a result, modeling the leg as a linear slider is a reasonably accurate simplification.

By using a physics simulator, the state of the components can be simulated without an explicit dynamics model. This is beneficial because the dynamics of Hopper are quite complicated, especially because Hopper can lose contact with the ground through jumping.

As a comparison for these reinforcement learning methods we also used an LQR controller to control a simplified Hopper model. LQR is an optimal control method which involves linearizing the system dynamics around the goal position and computing an optimal set of gains to minimize a given quadratic cost function. The specifics are beyond the scope of this paper, but the controller ultimately takes the following form:

$$\mathbf{u} = -L(\mathbf{x} - \mathbf{x}^*)$$

Where L is the computed optimal gains, $\mathbf{u}$ is the control inputs (in Hopper's case wheel torque), $\mathbf{x}$ is the current state of the robot, and $\mathbf{x}^*$ is the goal state. These gains are mathematically proven to be optimal around the goal state which the dynamics are linearized around, but the farther the state is from the goal the fewer guarantees can be made. In particular this means that the LQR controller is incapable of controlling Hopper through jumps, as the dynamics change entirely when the wheels lose contact.

IV. Experiments and Results

For our experiments, we tested three important situations in which a controller for Hopper would operate. These are stable balancing under noise, velocity control to match a commanded velocity, and jumping onto elevated surfaces.

For the first experiment, balancing under noise, we applied noise to the LQR and RL simulators and attempted to have both algorithms balance the robots. For the RL algorithms, this involved designing a reward function with the following characteristics:

- Penalize nonzero $\mathbf{s} = [x, \dot{x}, \theta, \dot{\theta}]^T$ (of the main body) to encourage the agent to keep the robot's x position and pitch close to 0.
- Reward staying alive ($|x| < 1$, $|\theta| < 1$) to encourage the agent to stay balanced for as long as possible.

The resulting reward function $R_1(\mathbf{s})$ is as follows:

$$R_1(\mathbf{s}) = \mathbf{1}\{|x| < 1\} * \mathbf{1}\{|\theta| < 1\} - ||K_1\mathbf{s}||_2^2$$

where K_1 is a diagonal gains matrix. The position x and linear velocity $\dot{x}$ is determined by the center of the robot's body.

The LQR controller is setup similarly. The robot dynamics are linearized around $\mathbf{s}^* = \mathbf{0}$ and the gains matrix L is computed to minimize the cost function:

$$J = \sum_t (\mathbf{s}_t - \mathbf{s}^*)^T Q(\mathbf{s}_t - \mathbf{s}^*) + \rho u_t^2 = \sum_t \mathbf{s}_t^T Q \mathbf{s}_t + \rho u_t^2$$

where Q is a matrix that weights the state variables and ρ weights a penalty for control effort . For these tests, we set $K_1 = Q = I$.

Both RL models were trained for 75000 steps, which was much longer than needed to converge. After every 1000 training steps, 10 trials of 1000 steps were run using the model trained so far, and the average reward was recorded along with the model parameters. Results from the training of the two RL models can be seen in Fig. 3. The ideal total reward would be holding $\mathbf{s} = \mathbf{0}$ for all 1000 steps of the trial simulation, for a reward of 1000. In reality, this is not possible due to simulator noise. The DDPG model seems to arrive at a better solution, with reward reaching approximately 875, compared to SAC which reached an average of approximately 525.

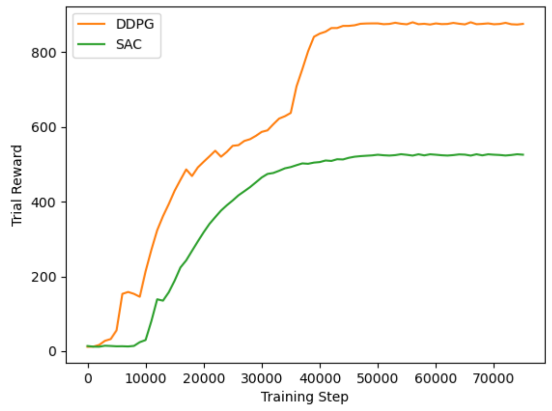


Fig. 3. Noise Rejection Training Reward

The models for each algorithm with the best reward (based on the average of the 10 trials) were selected. At test time, the robots were perturbed with noise applied to the wheel and leg angular velocities. The two values of noise, one for the wheel and one for the leg, were sampled randomly from a Gaussian distribution with variance $\sigma^2 = 0.01$.

Figure 4 shows a plot comparing the stability of the three different algorithms by showing the how much hopper deviates from vertical over time. All of the control techniques perform very well keeping within $5°$ of vertical throughout the test. Notably, both RL algorithms outperform LQR, with DDPG

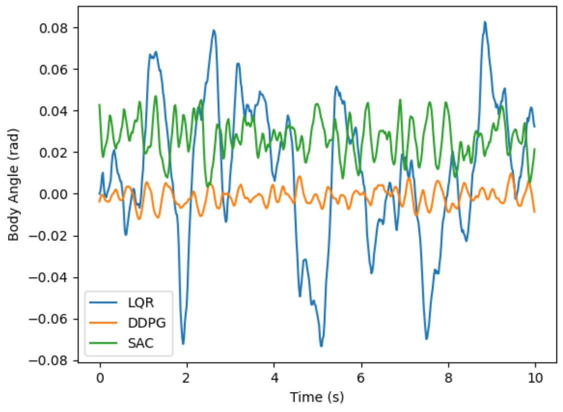


Fig. 4. Noise rejection of the different control schemes

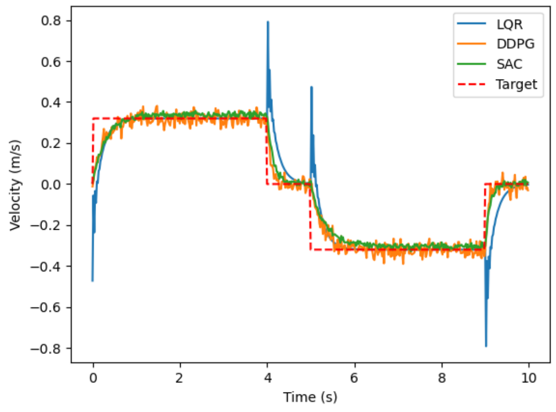


Fig. 5. Velocity over time of the different control schemes tracking a velocity control trajectory

appearing to perform the best. DDPG has both the smallest ripples as well as a low average offset from 0.

For the second experiment, we added an additional value to the state, describing a target velocity v for a velocity controller. We created the velocity controller by developing a new reward function and training the RL models on the updated reward. For this function, we would still like Hopper to stay balanced in an angular sense, but now don't care about the x position of the robot, just the difference between between the measured velocity $\dot{x}$ and the desired velocity v. Thus, for any given state $\mathbf{s} = [x, \dot{x}, \theta, \dot{\theta}, v]^T$, we can define an error vector $\epsilon = [\theta, \dot{\theta}, v - \dot{x}]^T$, which we want to bring to $\mathbf{0}$. The resulting reward function $R_2(\mathbf{s})$ is:

$$R_2(\mathbf{s}) = \mathbf{1}(|\theta| < 1) - ||K_2\epsilon||_2^2$$

Once again, the agents were trained over 75000 steps, and then tests were run using the model with the best average reward performance. The trajectory was defined to drive at 0.32 m/s for 4 seconds, then stop for 1 second, then drive at -0.32 m/s for 4 seconds, then stop for 1 second. The results of these velocity tests can be seen in Figs. 5 and 6.

All the controllers tracked the target trajectory quite well. Qualitatively, the RL controllers once again outperformed the LQR controller, which exhibited extraneous velocity spikes when the commanded velocity changed. The superior of the two RL controllers is less clear. The SAC controller achieved a much smaller velocity ripple than DDPG, but DDPG's average velocity was closer to the desired.

The final experiment involved jumping onto a raised surface (box), which is a task that traditional control methods would struggle with. As such, we focused solely on developing an RL controller to handle this task. The goal for this controller is to reduce distance to a specific location; specifically, the middle of the step. Additionally, we rewarded any state that had Hopper's wheels contacting the top surface of the box.

We added three variables to the state to represent the target position, r, as well as a variable c which is 1 when the robot is touching the top of the box and 0 otherwise. The state is

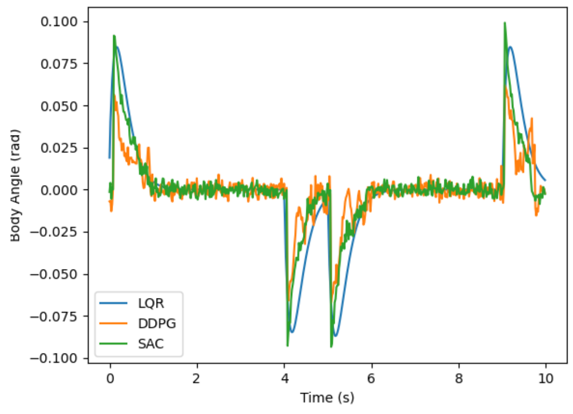


Fig. 6. Angle from vertical over time of the different control schemes tracking a velocity control trajectory

then $\mathbf{s} = [x, \dot{x}, \theta, \dot{\theta}, r, c]^T$. Again, we have an error vector ϵ, which is now $\epsilon = [\theta, \dot{\theta}, r - x]^T$. The reward function $R_3(\mathbf{s})$ is then:

$$R_3(\mathbf{s}) = \mathbf{1}(|\theta| < 1) + c - ||K_3\epsilon||_2^2$$

Both algorithms were trained with the same process as before. The training rewards can be seen in Fig. 7. In this case, the optimal reward would be holding $\epsilon = 0$ on top of the box for the entire 1000 steps of the simulation, which would correspond to a reward of 2000. Of course, a reward of 2000 is not reasonably possible in this experiment. We see that DDPG reaches a reward of approximately 432, while SAC reaches a reward of 485. Both algorithms succeeded in learning a controller to jump up onto the platform.

V. Conclusions and Future Work

Both reinforcement learning algorithms performed exceptionally well on all the different tests we set. To our surprise the RL algorithms not only met but in some metrics exceeded the performance of our LQR optimal control comparison.

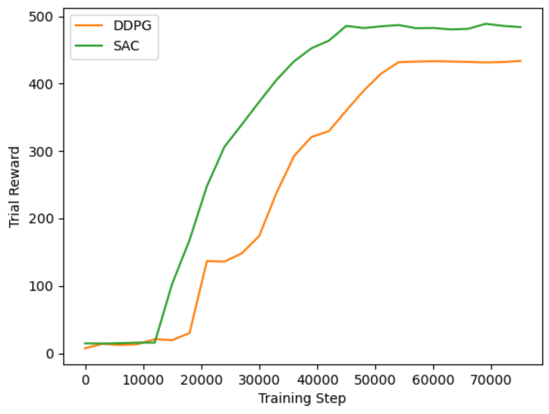
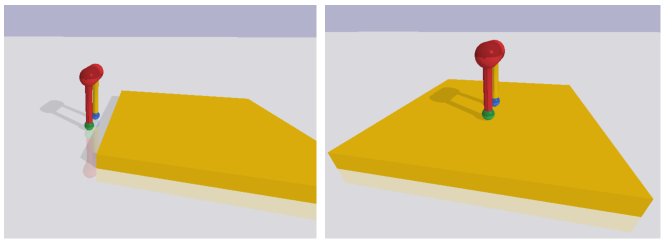


Fig. 7. Box Jump Training Reward



(a) Before (b) After

Fig. 8. Jump Experiment

The simple balancing and velocity control tests are tasks that LQR should be extremely well suited for, making the results of the RL algorithms particularly impressive. The box jump task, which would be quite difficult for traditional control algorithms, was successfully performed by the controllers learned by both DDPG and SAC.

When comparing the two algorithms, it is difficult to identify one as monolithically better than the other. In some experiments, DDPG performed better, and in others, SAC performed better. Given more computing resources and time, we would perform more tests, and potentially look at training models with different values in the $K_1, K_2,$ and K_3 diagonal weight matrices. Though the algorithms performed well with little tuning, there could be further performance gains to be realized by changing the weights.

In the future, we would love to apply these learned control policies on a real-world hardware version of Hopper. Our simulation results suggest that we could achieve good performance with them. However, making the step from simulation to real-world is not without its challenges. In a hardware version of the robot we would have to deal the problems like sensor and torque noise that are not currently present in our simulation. An extension we could start with is to add these kind of real-world uncertainties to the simulation and ensure that or RL algorithms are still able to find robust policies.

Another consideration that needs to be made when running these controllers on real hardware is safety. The black-box nature of these techniques that employ neural networks makes it hard to make guarantees about how the robot will act in all situations. As a result it is common to put a second controller in between the machine learning controller and the actual hardware in order to filter the outputs and ensure safety [6] [8]. Hopper is a fairly small and low power robot, so it should not pose much danger to people working on it, but these kind of safety checks are also important to ensure the robot does not damage itself.

Although more work would be needed to deploy these reinforcement learning controllers on a physical version of Hopper, their performance in simulation makes them a compelling avenue for both simple and more complex control tasks.

CONTRIBUTIONS

Aaron's Contributions:

- Calculated and coded up the dynamics model for the wheeled inverted pendulum.
- Linearized the dynamics for use in LQR.
- Programmed the LQR controller.
- Ran LQR experiments and collected data.

Parthiv's Contributions:

- Developed physics simulator for Hopper using PyBullet.
- Created associated Gym environments for the three experiments.
- Implemented RL algorithms and reward functions using PyTorch and NumPy
- Trained DDPG and SAC reinforcement learning algorithms on Hopper simulation, ran RL experiments, and collected data.

REFERENCES

[1] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1238–1274, 2013.

[2] O. Kroemer, S. Niekum, and G. Konidaris, "A review of robot learning for manipulation: Challenges, representations, and algorithms." *J. Mach. Learn. Res.*, vol. 22, pp. 30–1, 2021.

[3] L. Weng, "Policy gradient algorithms," *lilianweng.github.io/lil-log*, 2018. [Online]. Available: https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html

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

[5] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[6] M. Vecerik, T. Hester, J. Scholz, F. Wang, O. Pietquin, B. Piot, N. Heess, T. Rothörl, T. Lampe, and M. Riedmiller, "Leveraging demonstrations for deep reinforcement learning on robotics problems with sparse rewards," *arXiv preprint arXiv:1707.08817*, 2017.

[7] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, "Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor," 2018.

[8] X. Yu, Y. Fan, S. Xu, and L. Ou, "A self-adaptive sac-pid control approach based on reinforcement learning for mobile robots," 2021.