Learning to Guide a Swarm of Robots

Alexander Hendrich
TU Darmstadt

Daniel Kauth
TU Darmstadt

Gregor Gebhardt
TU Darmstadt

Abstract

In this work we want to learn a policy which is able to push an object through a maze by guiding a swarm of robots. This can be achieved by applying a light following behavior called *Phototaxis* to the robots and controlling a single light source using the learned policy. We will show that it is useful to split up this policy into two parts. The first policy solves the maze while the second policy is able to push the object in a single direction. We will combine these policies to achieve the original goal in a simulation. Finally we will also show that the learned policy can be used to guide a different number of kilobots than was used to learn it.

1 INTRODUCTION

The recently developed kilobots provide lots of new applications for swarm-intelligence and collective behavior algorithms. By applying *Phototaxis* to them, they provide an easy method to move and control large quantities at the same time. By interacting with a light source a human can control a kilobot swarm to fulfill certain tasks, for example pushing an object through a maze[2]. In this project we want to learn to control the light instead of having to control it manually.

Our task is to learn to push an object through a maze with a swarm of kilobots by moving a single light source. Because this task is quite complex and difficult to learn, we split it up into two separate problems with two separate policies.

The first problem is to find an optimal path through the labyrinth. The policy should return a movement direction for each possible object position. By constantly following these movements, the object should reach the goal position independent of its starting position.

Because we cannot directly control the object's movement we need a method to interact with the light source, so that the kilobots push the object in the desired direction. This will be the focus of the second problem. The policy should return a new light position depending on the output of the first policy (the desired movement direction of the object) and the relative position of the kilobots towards the object.

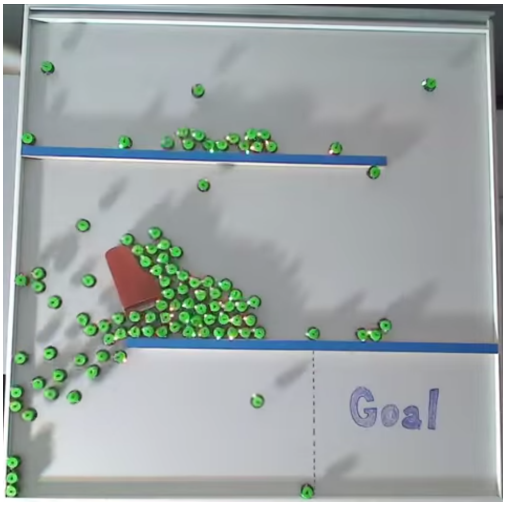


Figure 1: Kilobots pushing a small object through a maze by human controlled light sources[2]

2 PRELIMINARIES

In the following sections we introduce the equipment (kilobots[7]) and methods (Actor Critic Relative Entropy Policy Search)[9] we used.

2.1 Kilobots

Kilobots[7] are open-source, low cost robots specially designed to explore collective algorithms and decentralized cooperation. With previous robots these

Preliminary work. Under review by AISTATS 2015. Do not distribute.

methods could only be tested in a simulator or in small quantities of robots because acquiring and controlling hundreds or even thousands of them is very cost-intensive, time consuming and complex. To eliminate this issue, Harvard University developed Kilobots. Each robot is made with only 14$ worth of parts and large collectives can easily be controlled by a single user.

The low cost however results in limited capabilities of each individual robot. For example, the kilobot doesn't use complex steering or wheels for locomotion. Instead it stands on three solid legs and moves by activating two small vibrating motors placed on each side of the kilobot. This form of movement only allows for very low speeds of up to 1cm per second and can only be performed on special surfaces (e.g. glass).

The kilobot possesses an ambient light sensor to determine the light intensity at its current location. Depending on the current and previously measured light intensity the robot alternates between left and right turns to move towards the light source. This behavior is called *Phototaxis* and is used to control a whole swarm of robots at once. For more details see [2]. While performing left or right turns the robot only activates one of its two motors and the robot doesn't move in a straight line, which further reduces its movement speed.

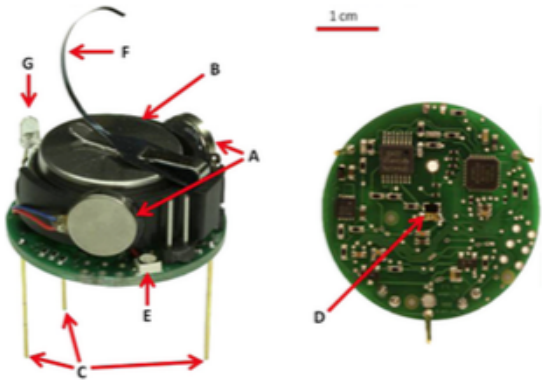


Figure 2: Isometric (left) and bottom (right) views of a Kilobot. Some key features are: (A) vibration motors, (B) lithium-ion battery, (C) rigid supporting legs, (D) infrared transmitter/receiver, (E) three-color (RGB) LED, (F) charging tab, and (G) ambient light sensor. Note the 1 cm line for scale.[7]

2.2 Actor Critic Relative Entropy Policy Search

Actor critic relative entropy policy search (AC-REPS) as introduced by [9] is a method for policy improvement based on *relative entropy policy search* (REPS) [5] which allows for model-free learning with non-parametric continuous policies and random exploration. It consists of the three steps

1. Estimating the Q-Function
2. Actor Critic REPS
3. Obtaining a new Policy

which will be discussed in he following sections.

2.2.1 Estimating the Q-Function

We assume samples of the form (s_i, a_i, r_i, s_i'), where s_i is the current state, a_i is the action taken in this state to get to the next state s_i' and r_i is the reward. We also have a feature-based Q-function $Q(s,a) = \phi(s,a)^T\theta$ with feature function $\phi(s,a)$ and parameter vector θ. Details of the state and action representations and rewards and features we used are discussed in section 3.2.

To estimate θ we use *Least-squares temporal difference learning* (LSTD) with $L_{22} - Regularization$[3] as follows:

The feature matrix for the current state Φ, the expected feature matrix for the next state Φ' and the reward vector R are defined as

$$\Phi = [\phi(s_1, a_1), \ldots, \phi(s_n, a_n)]^T,$$

$$\Phi' = \begin{pmatrix} \mathbf{E}_{\pi(a_1'|s_1')}[\phi(s_1', a_1')|s_1']^T \\ \mathbf{E}_{\pi(a_2'|s_2')}[\phi(s_2', a_2')|s_2']^T \\ \ldots \\ \mathbf{E}_{\pi(a_n'|s_n')}[\phi(s_n', a_n')|s_n']^T \end{pmatrix}$$

$$R = [r(s_1), \ldots, r(s_n)]^T.$$

To compute Φ' the expected value

$$\mathbf{E}_{\pi(a_i'|s_i')}[\phi(s_i', a_i')|s_i']$$

is approximated by taking the average over m samples $a_i' \sim \pi(a_i'|s_i')$, where π is the current policy.

Using the derivations of [4] and [3] one can easily compute θ as

$$\theta = (X^T X + \beta' I)^{-1} X^T y$$

with

$$X = C(A + \beta I) \qquad y = Cb$$
$$A = \Phi^T(\phi - \gamma\Phi') \qquad b = \Phi^T R$$
$$C = \Phi(\Phi^T\Phi + \beta I)^{-1},$$

where β and β' are regularization parameters for the projection and fixed-point step, respectively, which make the estimation numerically more stable and improve noise tolerance.

2.2.2 Actor Critic REPS

In this step we find a new policy π' which optimizes the expected Q-value as given by the estimated Q-function, but has a limited Kullback-Leibner (KL) divergence to the old policy π.

To achieve this we optimize over the joint state action distribution $p(s,a) = p(s)\pi'(a|s)$ and require that the estimated state distribution $p(s)$ is the same as in the given state sample distribution $\mu(s)$. This is implemented by matching feature averages of $p(s)$ and $\mu(s)$ and leads to the following optimization problem:

$$\begin{aligned} \max_p & \int p(s,a)Q(s,a)\, ds\, da\,, \\ \text{s.t. } & \mathrm{KL}(p(s,a)||\mu(s)\pi(a|s)) \leq \epsilon, \\ & \int p(s)\varphi(s)\, ds = \hat{\phi}, \\ & \int p(s,a)\, ds\, da = 1, \end{aligned}$$

where $\hat{\phi} = \int p(s)\varphi(s)\, ds$ is the average feature vector of all state samples, $\varphi(s)$ is another feature function and ϵ is an upper bound on the KL divergence.

This problem can be solved using Lagrangian multipliers and has the solution

$$p(s)\pi'(a|s) \propto \mu(s)\pi(a|s)\exp\left(\frac{Q(s,a)-V(s)}{\eta}\right),$$

where $V(s) = v^T\varphi(s)$ and η and v are Lagrangian multipliers which can be obtained by minimizing the dual function on the samples using standard gradient descent.

2.2.3 Obtaining a new Policy

The previous step only gives us desired probabilities $p(s_i,a_i) = p(s_i)\pi'(a_i|s_i)$ for all samples so in this step we need to generalize these by fitting a new policy $\bar{\pi}$.

This is done by minimizing the KL divergence between π' and the new policy $\bar{\pi}$ which is equivalent to computing a weighted maximum likelihood estimate with sample weights

$$w_i = \exp\left(\frac{Q(s_i,a_i)-V(s_i)}{\eta}\right).$$

In this paper we use a weighted Gaussian Process (GP) to obtain the new policy $\bar{\pi}$ by estimating action a_i for state s_i with weight w_i. We also use the sparse version of the GP as introduced in [8] to reduce computation time.

This choice of weights will reproduce actions which lead to a higher reward with higher probability but will also choose suboptimal actions with some probability. This behavior occurs because the new policy can not be too far away in terms of KL divergence from the old policy and we start with a random policy. This achieves a trade-off between exploration and exploitation.

3 LEARNING PROCESS

In this section we present the states, actions, features and rewards we used and the way we implemented the learning process. Additionally we will describe the kilobot simulator that we wrote to generate samples.

3.1 Maze Solving Policy

This policy tells us where the object should move given the current object position and solves a maze (Figure 3) in this way.

Because this policy is relatively easy to learn and there are specialized path planning algorithms like A^* which can easily compute an optimal path through any maze we did not learn this policy and instead focused on the second one, which is much more interesting. Therefore we manually specified this policy by providing a number of waypoints the object should pass and give the next waypoint as the target position. Once we reach a waypoint we move on to the next one.

In our experiments this simple policy is sufficient to move different objects through a simple maze using the object movement policy. For this reason we will only discuss the object movement policy in the following sections.

3.2 Object Movement Policy

For the object movement policy we decided to only learn how to push the object to the right instead of an arbitrary direction. This simplifies and speeds up the learning process because we need to generate fewer samples and can represent the policy using a smaller state subset for the sparse GP and therefore can compute the policy update much faster. As we will discuss in 3.3 we can still use this policy to push the object in an arbitrary direction by rotating the state and action appropriately.

3.2.1 States and Actions

With n kilobots the state tuple is

$$s = (l_x, l_y, k_{1x}, k_{1y}, \ldots, k_{nx}, k_{ny}),$$

where (l_x, l_y) is the light position and (k_{ix}, k_{iy}) is the position of the $i-th$ kilobot. All positions are relative to the object's center position.

The action tuple is given by

$$a = (a_x, a_y),$$

which is the light displacement.

3.2.2 Rewards

The reward is given by

$$r = (c_x - p_x) - 0.5 \cdot |c_y - p_y|,$$

where (c_x, c_y) is the object's current position and (p_x, p_y) is the object's position from the last time step. Therefore it rewards movement to the right and punishes movement to the left and any movement in y direction.

3.2.3 Features and Kernels

For the learning process we need a feature function $\phi(s, a)$ over state-action pairs to represent the Q-function, a feature function $\varphi(s)$ over states for REPS and a kernel function $K(s_1, s_2)$ between states for the GP.

The feature functions are implemented in terms of kernel functions by choosing a random subset of N samples (s_i, a_i) in each iteration and computing the feature vectors for a given state s and action a as

$$\phi(s,a) = \begin{pmatrix} K'((s_1,a_1),(s,a)) \\ K'((s_2,a_2),(s,a)) \\ \dots \\ K'((s_N,a_N),(s,a)) \end{pmatrix},$$

$$\varphi(s) = \begin{pmatrix} K(s_1,s) \\ K(s_2,s) \\ \dots \\ K(s_N,s) \end{pmatrix},$$

where K' is a kernel function similar to K which compares state-action pairs. For our experiments we used $N = 200$.

Let s be a state with n kilobots with positions (k_{ix}, k_{iy}) and s' be a state with m kilobots with positions (k'_{jx}, k'_{jy}) and let a and a' be the corresponding actions. We define $K(s, s')$ and $K'((s, a), (s', a'))$ as

$$\exp(-0.5(w \cdot d_{other} + (1 - w) \cdot d_{kb})),$$

where $w \in [0, 1]$ weighs the importance of the non-kilobot dimensions and d_{kb} and d_{other} are the distances for the kilobot dimensions and all other dimensions, respectively. For K the other dimensions are only the light positions and for K' they are the light positions and actions.

We compute the distance between two kilobot configurations as

$$\begin{aligned} d_{kb} = &\frac{1}{n^2} \sum_{i,j \in \{1,\dots,n\}} \bar{K}((k_{ix}, k_{iy}), (k_{jx}, k_{jy})) \\ &- \frac{2}{n \cdot m} \sum_{\substack{i \in \{1,\dots,n\} \\ j \in \{1,\dots,m\}}} \bar{K}((k_{ix}, k_{iy}), (k'_{jx}, k'_{jy})) \\ &+ \frac{1}{m^2} \sum_{i,j \in \{1,\dots,m\}} \bar{K}((k'_{ix}, k'_{iy}), (k'_{jx}, k'_{jy})), \end{aligned}$$

where $\bar{K}$ is a 2 dimensional squared exponential kernel with

$$\bar{K}(k_1, k_2) = \exp\left(-0.5(k_1 - k_2)^T \operatorname{diag}\left(\frac{1}{\sigma_{kb}^2}\right)(k_1 - k_2)\right)$$

and $\sigma_{kb} \in \mathbb{R}^2$ is the bandwidth for the kilobot dimensions. This allows us to compare states with different numbers of kilobots and the computed distance is independent of the order of the kilobot positions in the state vectors.

The distance for all other dimensions is computed as

$$d_{other} = (x_1 - x_2)^T \operatorname{diag}\left(\frac{1}{\sigma_{other}^2}\right)(x_1 - x_2),$$

where x_1 and x_2 are vectors holding the values for all non-kilobot dimensions and σ_{other} is the non-kilobot bandwidth.

3.2.4 On-Policy Learning

To learn the object movement policy we use on-policy learning. During this process we have matrices S_k, A_k, R_k, S'_k with n rows holding the n current samples which are initially empty and a current policy π_k. The initial policy π_0 predicts random actions (a_x, a_y), where both a_x and a_y are sampled uniformly from $[-0.015, 0.015]$ (1.5 cm).

In each iteration we use the current policy π_k to generate m samples (s_i, a_i, r_i, s'_i). We select the new samples $S_{k+1}, A_{k+1}, R_{k+1}, S'_{k+1}$ by randomly choosing 10000 samples from the current and newly generated samples to keep the computation tractable.

Next we estimate the Q-function as described in 2.2.1, compute a sample weighting as in 2.2.2 and finally fit a new sparse GP as discussed in 2.2.3 to obtain the new policy π_{k+1}.

We repeat these steps a fixed number of times and manually inspect the learned policy. If it is not satisfactory (it can not solve the maze) we continue the learning process.

3.3 Simulator

We did not use a general purpose robot simulator like V-REP[6], because it would take much longer than necessary to generate samples as the kilobots are simulated more accurately than needed for our task. Instead we implemented our own kilobot simulator (Figure 3) in Python based on the 2D physics engine pybox2d[1]. This allows us to simulate collisions between kilobots and other kilobots, the object, and walls respectively.

When requesting new samples from the simulator we specify the current policy π which will be used to generate these samples as well as the sampling parameters such as the shape of the object, the number of kilobots n_{kb}, the number of episodes n_{ep} and the number of steps per episode n_{st}.

Given these parameters the simulator first generates n_{ep} light start positions $start_i$ equally spaced in a circle of radius $1.5 \cdot r$ around the object which starts in the middle of the screen, where r is the half-width of the object's bounding box. Then for $i \in \{1, \ldots, n_{ep}\}$ we do the following:

The object's position is reset to the middle of the screen and its angle is set to 0. After that the light position l_k is set to $start_i$ and the position of the n_{kb} kilobots is set to $start_i + off_j$, where off_j is a small offset for kilobot j. Then n_{st} time steps are simulated where for each an action a_k is chosen for the current state s_k according to π. Subsequently, the light position is changed instantly to the new position $l_{k+1} = l_k + a_k$. Next the kilobots move directly toward the light for a short amount time (1 second for all of our simulations) and finally we record the current state as s'_k, compute the reward r_k and save the tuple (s_k, a_k, r_k, s'_k) as a sample. We also restrict the magnitude of the light and kilobot movements to get a more realistic simulation.

The simulator can also be used to simulate the kilobots pushing an object through a maze. In this mode we do not record any samples. Instead we move the light according to the object movement policy which gets a target position from the maze solving policy. The policies are combined in the following way:

As discussed in 3.2 the object movement policy learns how to move the light to push the object right-wards given the current light and kilobot positions relative to the object's position. The maze solving policy now gives us a target position t from which we compute the target direction as $d = t - o$ where o is the current object position. Now, we compute the angle α between d and $(1, 0)$ (the vector pointing to the right) and rotate the state around α. Then we use the object movement policy to get an action for the rotated state which we rotate by $-\alpha$ to get the final action which we apply just as in the sample generation case.

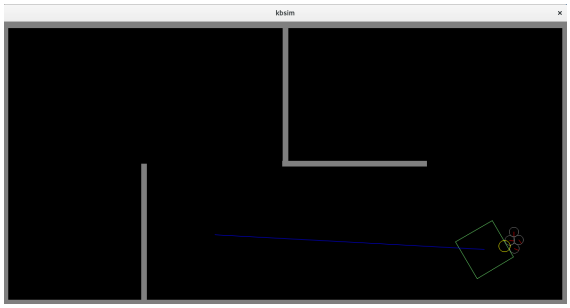


Figure 3: 4 kilobots pushing an object through a maze in the simulator.

4 EXPERIMENTAL RESULTS

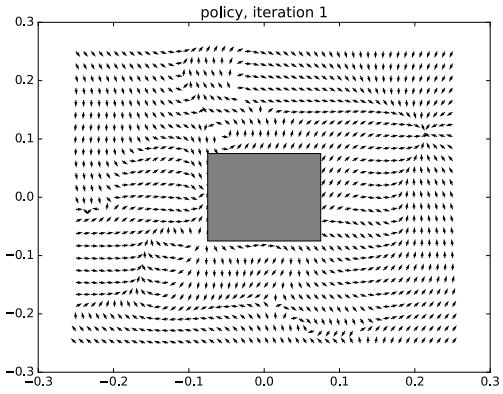
For the final evaluation we made 20 runs of our learning process with 15 iterations each generating $25 \cdot 100 = 2500$ samples per iteration. Using an Haswell Quad Core with 3.4GHz per core each iteration took about 3 minutes, so the total run-time for each policy was about 45 minutes.

Initially the policies are only able to push the object when the kilobots start to the left of the object. After about 5 iterations they can also handle kilobot starting positions above and below the object by first moving the kilobots around to the left side of the object and then pushing the object from the left side. After further iterations the policies are able to push the object to the right even if the kilobots start above and to the right of the object as can be seen in Figure 4. With even more iterations the kilobot starting positions from which the object can be pushed tend to extend even further, but this improvement is inconsistent and the policies may even get worse again. Therefore we manually inspected the policies at each iteration and chose a good policy (which is typically found in the first 15 iterations) for solving the maze.

Figure 5 also shows that kilobot positions left of the object are preferable to positions to the right. Further the value function increases when moving around the object from the right to the left.

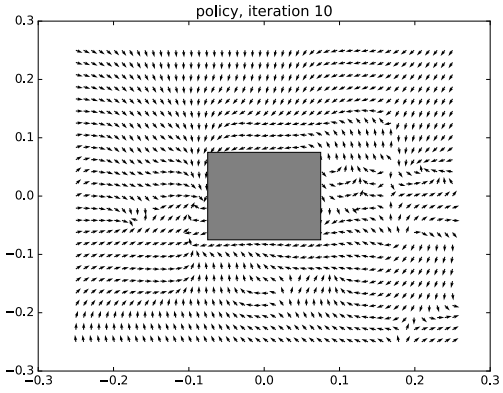
With improving policies the object can be pushed right-wards from a larger number of kilobot starting positions. Additionally the object can be pushed faster and with less vertical movement. Therefore the reward tends to increase as can be seen in Figure 6.

We created the policy plots by getting the mean GP action for states distributed on a grid and plotting it as an arrow indicating the direction the light should move

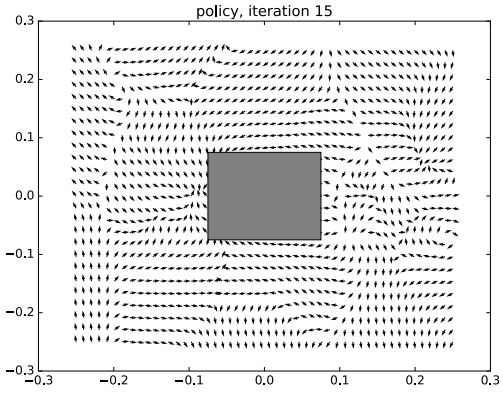


(a) Policy after 1 iteration

(b) Policy after 5 iterations



(c) Policy after 10 iterations



(d) Policy after 15 iterations

Figure 4: Evolution of the learned policy. The mean action given by the Gaussian Process is plotted as an arrow for states distributed on a grid.

in. For this we set the relative light positions to evenly spaced values between -0.25 and 0.25 for the $x-$ and $y-$dimensions, respectively, and set the position for all kilobots to coincide with the light position.

To compute the Value-function we use the same states and uniformly sample a number of actions at random for each state. Then we compute the Q-values for all state-action pairs and maximize over them for each state.

To evaluate the policies we used our simulator to push the object through the maze as described in 3.3. We found that the learned policies are sufficient to achieve this task and are even able to move the object when the kilobots are (almost) completely on the wrong side of the object, which requires moving them around the object. Moreover we can transfer the policy to more or less kilobots and are still able to push the object because of our choice of kernel. We learned the policy with 4 kilobots and tested it with 1,2,3,4,6,8 and 10 kilobots and it was able to push the object through the maze in each case. Finally we were also able to push a circle through the maze even though the policy was learned on a square.

5 CONCLUSION & FUTURE WORK

In our experiments we achieved our goal to learn a policy to push an object using multiple kilobots controlled by a single light and to combine this policy with a maze solving policy to push the object through a maze. The policy is also transferable to a different number of kilobots and may be used to push objects of different simple symmetric shapes which we tested for a circle.

Although this policy works well in the simulator it is

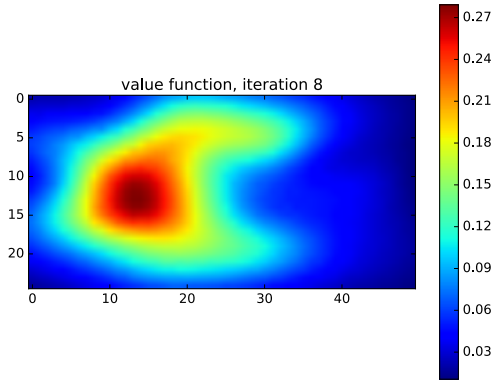


Figure 5: Learned value function after 8 iterations. The object (not shown here) is at the middle of the image.

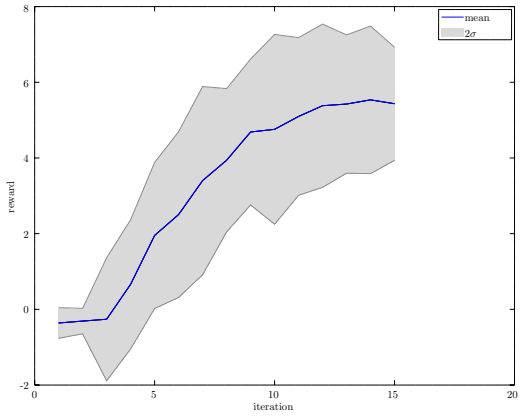


Figure 6: Total reward for each iteration averaged over 20 runs. The shaded area shows twice the standard deviation.

not directly applicable to the real kilobots because our simulator is overly simplistic. It moves the kilobots directly toward the light instead of using Phototaxis and also allows for arbitrary light movements, where in reality the light would be moved by a robot arm which has some constraints on the forces it can apply.

Therefore a future work could simulate the kilobots more realistically and try to transfer the learned policy to the real kilobots by building a maze and controlling a robot arm holding a flashlight using the learned policy.

References

[1] pybox2d, 2016. URL `https://github.com/pybox2d/pybox2d`.

[2] Aaron Becker, Golnaz Habibi, Justin Werfel, Michael Rubenstein, and J. McLurkin. Massive uniform manipulation: Controlling large populations of simple robots with a common input signal. *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS) (Accepted)*, 2013.

[3] Matthew W. Hoffman, Alessandro Lazaric, Mohammad Ghavamzadeh, and R emi Munos. Regularized least squares temporal difference learning with nested l2 and l1 penalization.

[4] Michail G. Lagoudakis and Ronald Parr. Least-squares policy iteration. *Journal of Machine Learning Research 4*, 2003.

[5] Jan Peters, Katharina Mülling, and Yasemin Altun. Relative entropy policy search. In *AAAI*. Atlanta, 2010.

[6] Eric Rohmer, Surya PN Singh, and Marc Freese. V-rep: A versatile and scalable robot simulation framework. In *Intelligent Robots and Systems (IROS), 2013 IEEE/RSJ International Conference on*, pages 1321–1326. IEEE, 2013.

[7] Michael Rubenstein, Christian Ahler, Nick Hoff, Adrian Cabrera, and Radhika Nagpal. Kilobot: A low cost robot with scalable operations designed for collective behaviors. 2014.

[8] Edward Snelson and Zoubin Ghahramani. Sparse gaussian processes using pseudo-inputs. In *Advances in neural information processing systems*, pages 1257–1264, 2005.

[9] Christian Wirth and Gerhard Neumann. Model-free preference-based reinforcement learning. 2016.