Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-18)

Learning Environmental Calibration Actions for Policy Self-Evolution*

Chao Zhang, Yang Yu, Zhi-Hua Zhou
National Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210023, China
{zhangc,yuy,zhouzh}@lamda.nju.edu.cn

Abstract

Reinforcement learning in physical world is often expensive. Simulators are commonly employed to train policies. Due to the simulation error, trained-in-simulator policies are hard to be directly deployed in physical world. Therefore, how to efficiently reuse these policies to the real environment is a key issue. To address this issue, this paper presents a policy self-evolution process: in the target environment, the agent firstly executes a few calibration actions to perceive the environment, and then reuses the previous policies according to the observation of the environment. In this way, the mission of policy learning in the target environment is reduced to the task of environment identification through executing the calibration actions, which needs much less samples than learning a policy from scratch. We propose the POSEC (POlicy Self-Evolution by Calibration) approach, which learns the most informative calibration actions for policy self-evolution. Taking three robotic arm controlling tasks as the test beds, we show that the proposed method can learn a fine policy for a new arm with only a few (e.g. five) samples of the target environment.

1 Introduction

Reinforcement learning aims at continually improving the decision-making ability of an agent through autonomous trial-and-errors interactions with the environment [Sutton and Barto, 1998]. Recently, reinforcement learning has shown significant progress in applications [Silver *et al.*, 2016; Mnih *et al.*, 2015]. However, the state-of-the-art reinforcement learning algorithms would still require a lot of environment samples (e.g., millions) in order to learn a good policy [Silver *et al.*, 2016]. Such high sample amount requirement blocks reinforcement learning approaches from many applications in physical world, where every environment sample can be expensive. For examples, a trial of physical robot experiment commonly takes time from minutes to hours, a trial of stock investment additionally takes money, and a trial of medical treatment can even cost life.

To alleviate the high training cost of reinforcement learning in physical world, simulators are commonly employed. Besides commercial simulators for special applications, there have been developed open-source simulators such as in the OpenAI Gym [Brockman *et al.*, 2016], which have significantly facilitated the recent advances of reinforcement learning research. However, as a compromise on the cheap samples, simulators always have simulation errors. Policies trained in a simulator can behave quite differently in the real environment, particularly when the agent takes many steps such that the simulator error accumulates. Therefore, how to rapidly adapt trained-in-simulator policies to the real environment is a key issue for applying reinforcement learning in physical world applications.

Studies on transfer reinforcement learning aims at solving the policy adaptation problem across environments [Mehta *et al.*, 2008]. Transfer reinforcement learning approaches try to reuse the experience from similar tasks, so that a good policy for a new environment can be obtained with a small cost. These approaches can be roughly categorized according to the type of the experience they reuse. Sample-based transfer approaches transfer the source task samples to the target task. While transferring samples directly is often prone to negative transfer [Lazaric *et al.*, 2008], representations and model parameters could be more consistent across domains. Thus a bunch of studies focused on representation transfer and parameter transfer, which generally learn some higher-level characteristics from a set of source tasks and reuse the characteristics during the learning on target tasks. Different transfer reinforcement learning approaches also assume different situations. While, in this work, we focus on adapting trained-in-simulator policies to the real environment, where the simulators have been designed to have the same goal, state space and action spaces with the real environment.

The design of the simulator usually allows a trained-in-simulator policy to be directly executable in the real environment. But the policy may have insufficient performance due to the inaccuracy of the simulated dynamics. If the agent can perceive its environment and adjust the policy according

*This work is supported by National Key R&D Program of China (2018YFB1004300), Jiangsu SF (BK20170013), and Collaborative Innovation Center of Novel Software Technology and Industrialization. Yang Yu is the corresponding author.

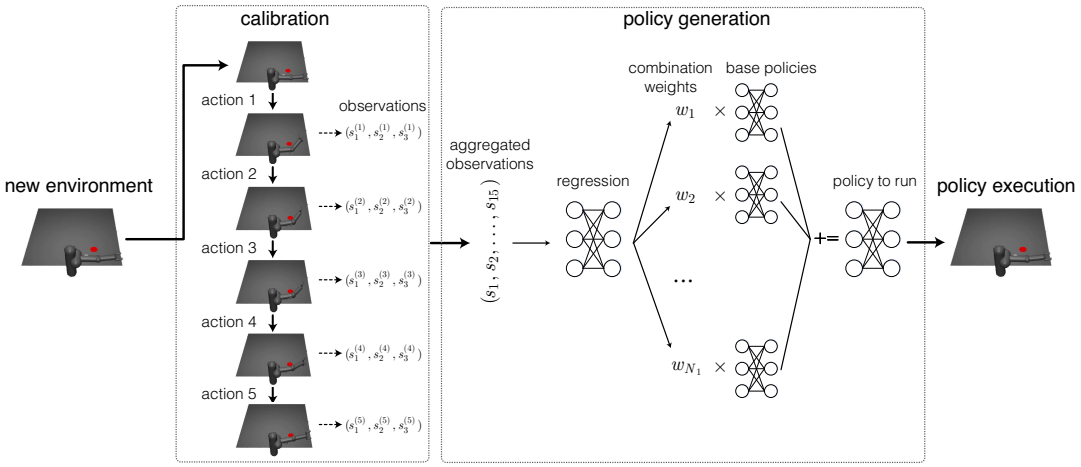


Figure 1: Illustration of policy self-evolution of POSEC using calibration actions. In a new environment, the agent executes the calibration actions to collect observations about the environment, predicts the combination weights of base polices, and obtains the final policy to run in the environment.

to the perceived environment information, we could obtain a good performance policy immediately in the target environment instead of re-training the policy. This observation leads to our idea that, when the agent is deployed in the target environment, it first performs a few calibration actions for perceiving the environment, and then derives a self-evolved policy from a set of pre-trained policies. In this way, the policy learning mission for the target environment is reduced to the environment identification task through executing the calibration actions, which may need much less environment samples than learning a policy from scratch. Recently, [Zhou, 2016] proposed the new concept of *learnware*, with properties of reusability, evolvability and comprehensibility. The evolvability emphasizes the self-evolution ability of a pre-trained model to get accustomed to new environments, for which the pre-trained model should be able to perceive the environment changes and then adapt itself to the new environments. It is evident that the study reported in this paper can be viewed as an effort towards this direction.

In this paper, we propose the POSEC (POlicy Self-Evolution by Calibration) approach to implement this idea. In a new environment, or whenever the policy outcome is found unexpected, the agent can run the self-evolution process illustrated in Figure 1. First, it executes a few calibration actions. From the observations after the actions, a policy is obtained through combining the pre-trained base policies. This policy is the self-evolved policy that is to run in the environment. By POSEC, only a few samples from the target environment are needed to obtain a fine policy. In order to realize this calibration process, the calibration actions, the regression model, and the base policies need to be pre-trained. The training of these components are in the off-line stage with the assistant of a parameterized simulator, and is consists of three steps: in the first step, a set of simulated tasks are drawn with the help of the randomized simulator, and base policies are obtained by a heavy learning in each of the task, which is doable due to the cheap cost of training in the simulator; in the second step, a new batch of simulated tasks is drawn, in each of which the best combination of the base policies is calculated; in the third stage, POSEC searches for the best calibration actions, of which the observed outcome lead to the best prediction of the solved combination weights.

We apply POSEC to three robotic arm controlling tasks. A simulator of an arm with configurable lengths is employed, and the observed outcome of calibration actions is the end effector position of the arm. POSEC is then asked to control new arms with randomly assigned arm lengths. Experiment results show that POSEC can lead to a high reward policy with only a few (1, 5 and 10) calibration actions, which can be even better than reinforcement learning training from scratch using a million samples.

The rest of this paper starts from introducing the related work, then the proposed approach, the experiment results, and the conclusion are in the sections followed.

2 Background

2.1 Reinforcement Learning

Reinforcement learning enables an agent to autonomously discover the optimal policy through autonomous trial-and-error interactions with its environment [Sutton and Barto, 1998]. It is commonly studied through Markov Decision Process (MDP). An MDP consists of state space *S*, action space *A*, reward function *R(s,a)*, transition function $P(s'|s,a)$, and discounted factor γ. The goal of reinforcement learning is to find an optimal policy $\pi^* : S \times A \to R$ that maps states to distribution of actions so as to maximize the total reward. In the discounted setting, the expected value of discounted total rewards J_π starting from an initial state s_0 is:

$$J(\pi) = E_{s_0,a_0,\ldots}[\sum_{t=0}^{\infty} \gamma^t r(s_t)],$$

where $a_t \sim \pi(\cdot|s_t)$ and $s_{t+1} \sim P(\cdot|s_t, a_t)$. Consider the horizon of T steps , this expression can be rewritten as the expectation over all trajectories of length T, as

$$J(\pi) = \int_{\mathcal{T}} P_\pi(\tau) R(\tau) d\tau,$$

where τ is a trajectory in the trajectory space $\mathcal{T}$, $P_\pi(\tau)$ is the probability that executing the policy π will generate the trajectory τ, and $R(\tau)$ is the return of τ. We will rely more on the trajectory-wise form of the expected total reward.

Classical reinforcement learning approaches solve the best policy π that maximizes the expected total reward $J(\pi)$. According to whether the MDP will be estimated, reinforcement learning approaches can be categorized as model-based approaches [Deisenroth and Rasmussen, 2011; Brafman and Tennenholtz, 2002; Jong and Stone, 2007] and model-free approaches [Watkins and Dayan, 1992; Strehl *et al.*, 2006; Sutton *et al.*, 2000]. In this work, we consider model-free reinforcement learning. According to whether the policy is directed learned, the approaches can be categorized as value-based approaches [Watkins and Dayan, 1992; Ratitch and Precup, 2004] and policy search approaches [Sutton *et al.*, 2000]. This work builds on top of reinforcement learning approaches, and both categories can be employed.

Classical reinforcement learning approaches train and use the policy in the same environment. However, the environment often changes in applications. For example, the policy for controlling a robotic arm could be trained in a simulator with some simulation error. More commonly, an aged robotic arm changes its dynamics. Thus, a robust policy should has the ability of adapting to its environment.

2.2 Transfer Reinforcement Learning

Transfer learning reuses the experiences gained from previous tasks to help the learning in the target task [Pan *et al.*, 2011]. For reinforcement learning, reuse of the experience can reduce the sample requirement to the new environments.

According to the type of experience that the target task receives from the source task, transfer reinforcement learning approaches can be roughly divided into sample transfer, representation transfer and parameter transfer. Sample transfer algorithms mainly reuse the samples from the source tasks in the target task. Direct reusing the samples from different tasks has a large risk of negative transfer, and thus the samples need to be carefully handled. Representation and model parameters can better reflect the similarity among tasks in our setting, which is conducive to generalization.

Parameter transfer methods often explicitly define a distribution on the task space and try to learn and adapt policy parameters in order reduce the number of samples required to solve the target task. For examples, [Finn *et al.*, 2017] proposes to train a common model, such that the model has the maximal performance over all tasks while can be fast adapted to a specific task through task-specific gradient updates. [Peng *et al.*, 2017] proposed dynamic randomization of the simulator to train the robot to adapt to the dynamic changes of the object position in the physical world task, by introducing LSTM to extract environment latent variables and adjust its behavior accordingly. Comparing with to the above studies, this work proposes to explicitly learn calibration actions for environment probing, which requires much fewer environment steps compared with [Finn *et al.*, 2017] and the learned actions can be more informative compared with [Peng *et al.*, 2017]. [Pan *et al.*, 2011].

2.3 Derivative-Free Optimization

Derivative-free optimization methods solve an optimization task $\arg\min_x f(x)$ by utilizing only the information of the function values of f on sampled solutions. They perform the optimization in a trial-and-error way that commonly consists of a sampling step generating samples of x from some experience model, and a learning step updating the experience model from the samples. Different derivative-free optimization methods mainly differ in the design of the model and the sampling approach. Representative methods include heuristic approaches such as simulated annealing [Kirkpatrick *et al.*, 1983], CMA-ES [Hansen and Ostermeier, 2001], and more recently theoretical-grounded approaches such as Bayesian optimization [Shahriari *et al.*, 2016], optimistic optimization [Munos, 2014], and RACOS [Yu *et al.*, 2016; Hu *et al.*, 2017].

Since derivative-free optimization methods rely only the function values but not the gradient of the function, these methods can be applied in a wide range of optimization problems, including non-differentiable and non-convex functions. This work involves a task of optimizing the calibration actions with the target that the resulting performance is maximized. This optimization is non-differentiable, thus we will employ the derivative-free method to solve the optimization.

3 The Proposed Method

As illustrated in Figure 1, the self-evolution process of POSEC needs the base policies, the regression model and the calibration actions. These are obtained in the off-line training stage consisted of three parts: training the base policies, optimizing the combination weights, and learning the calibration actions and the regression model.

To support the learning of a set of diverse base policies, we need to generate a range of environments. Therefore, we assume that the simulator is parameterized, and we can sample environments from a distribution over the simulator parameters. A key assumption is, therefore, that the latent parameter of the target environment is covered by the distribution. We assume that the target environment can be realized by some (unknown) parameter of the simulator.

3.1 Training Base Policies

We sample a set of environments by configuring the simulator with a randomly sampled parameter. Each environment corresponds to an MDP. We collect M_1 environments, $\{\text{MDP}_1, \text{MDP}_2, \ldots, \text{MDP}_{M_1}\}$. Any off-the-shelf policy search algorithm can then be employed to train policies, each for an MDP. As the result, M_1 policies $\{\pi_1, \pi_2, \ldots, \pi_{M_1}\}$ have been obtained. These policies will serve as the base policies, and any further policy will be a combination of the base policies with respect to a combination weight vector w,

$$\pi_w(a|s) = \sum\nolimits_{t=1}^{M_1} \frac{w_t}{\sum_{t=1}^{M_1} w_t} \pi_t(a|s).$$

Here, we only consider a linear combination of the base policies. While a nonlinear combination is also feasible, the linear combination needs less samples to be estimated and may be more robust.

3.2 Optimizing Combination Weights

We then draw another set of M_2 environments $\{\text{MDP}'_1, \text{MDP}'_2, \ldots, \text{MDP}'_{M_2}\}$, and solve the combination weights of the base policies on each environment. For the i-th environment, the expected return $J_{MDP'_i}(\cdot)$ is:

$$J_{MDP'_i}(w) = \int_\tau P_{\pi_w}(\tau) R(\tau) d\tau,$$

where $P_{\pi_w}(\tau)$ is the distribution about the combination policies π_w over the trajectory $\tau = (s_0, a_0, s_1, a_1, ...)$.

We try to maximize the expected return to get the optimal combination weights w^* of the policy π_w on this environment:

$$w_i^* = \arg\max_w J_{MDP'_i}(w).$$

The optimization of w can be either by gradient ascent methods since the objective is differentiable, or by derivative-free methods that solve the weights directly.

3.3 Optimizing Calibration Actions

After the above two steps, we have obtained a set of base policies $D = \{\pi_1, \pi_2, \ldots, \pi_{M_1}\}$, and the combination weights w_i^* for each of the environments $\{\text{MDP}'_1, \text{MDP}'_2, \ldots, \text{MDP}'_{M_2}\}$. We are then to find the regression model and the calibration actions.

Given any fixed sequence of calibration actions A (initialized randomly), an agent executes the actions in its environment, and the outcome states after executing each of the action are concatenated as the feature vector of the environment, denoted as $F(\text{MDP}, A)$ where MDP is the environment and A is the calibration actions.

When the environment feature vector is available, we train a regression model to predict the optimal combination weights from the feature vector. Because we learn from small samples of combination weights, we use linear models to predict them. For the given actions A and all the M_2 environments, consider the optimal linear regression as

$$\theta^* = \arg\min_\theta \sum_{i=1}^{M_2} \|w_i^* - \theta^T F(\text{MDP}'_i, A)\|_2, \qquad (1)$$

where $F(\text{MDP}'_i, A)$ denotes the feature vector constructed from the states in MDP'_i after executing every action in A. Using this regression model with the actions A, for any environment MDP, we now can obtain the combination weights $\theta^{*\top} F(\text{MDP}, A)$ of the base policies. The combined policy is denoted as $\pi_{\theta^{*\top} F(\text{MDP}, A)}$.

Note that the calibration actions A has not been optimized yet. The objective for solving A is that A can maximize the total reward of the combined policy on environments. To avoid overfitting, We again draw M_3 environments, $\{\text{MDP}''_1, \text{MDP}''_2, \ldots, \text{MDP}''_{M_3}\}$, for solving A. The total reward summed over these M_3 environments is used for the objective function. Thus the optimal calibration actions are obtained by solving

$$A^* = \arg\max_A \sum_{i=1}^{M_3} \int P_{\pi_{\theta^{*\top} F(\text{MDP}''_i, A)}}(\tau) R(\tau) d\tau \qquad (2)$$

Algorithm 1 POSEC Training Process

Input:

$\{\text{MDP}_i\}_{i=1}^{M_1}$: the first batch of M_1 environments;
$\{\text{MDP}'_i\}_{i=1}^{M_2}$: the second batch of M_2 environments;
$\{\text{MDP}''_i\}_{i=1}^{M_3}$: the third batch of M_3 environments;
L_{rl}: A policy search algorithm;
L_{opt-w}: Optimization algorithm for combination weights w;
L_{opt-A}: Optimization algorithm for calibration actions;
L_{reg}: Optimization algorithm for regression coefficient;
I: Number of iterations.

Output:

A^*: A sequence of the optimized calibration actions;
θ^*: A regression model.

1: $\forall i = 1, 2, \ldots, M_1 : \pi_i \leftarrow$ the policy by running L_{rl} on environment MDP_i with I iterations.
2: $\forall i = 1, 2, \ldots, M_2 : w_i^* \leftarrow$ the weights by running L_{opt-w} on environment MDP'_i.
3: Solve the calibration actions A^* by L_{opt-A} and the regression model θ^* by L_{reg} from the objective function

$$(A^*, \theta^*) = \arg\max_{(A,\theta)} \sum_{i=1}^{M_3} \int P_{\pi_{\theta^\top F(\text{MDP}''_i, A)}}(\tau) R(\tau) d\tau$$

4: **return** A^*, θ^*

Algorithm 2 POSEC Calibration Process

Input:

MDP: A new test environment.

Output:

$\pi_{\text{self-evolved}}$: The combination of the base policies.

1: $F(\text{MDP}, A^*) \leftarrow$ the feature vector of the environment build from the states after every action in A^*.
2: $w^* \leftarrow$ the optimal weights predicted by the regression model θ^* from the features $F(\text{MDP}, A^*)$
3: **return** $\pi_{\text{self-evolved}}(a|s) = \sum_{t=1}^{M_1} w_t^* \pi_t(a|s)$

It is clear now that we can evaluate the objective function given the calibration actions. But the actions may be too complex to solve by gradient ascent. We thus employ the derivative-free optimization algorithm to solve the problem. The derivative-free optimization algorithm uses a trial-and-error process. It samples different actions to try; learns from the objective value of the actions for sampling better actions.

In the general case, the training process of the full algorithm is outlined in Algorithm 1, and the calibration process in Algorithm 2.

4 Experiments

We empirically evaluate POSEC, particularly, answering the following questions:

- ***Q1***: Can the learned calibration actions effectively extract features for the environment, and be better than random actions? Is the number of the calibration actions effect the performance?

- ***Q2***: How do the calibration actions act?
- ***Q3***: Can the self-evolved policy serve as a better initial policy for environment specific-refinement?

4.1 Experiment Settings

We employ three robotic arm controlling tasks that use Mujoco physics simulator from OpenAI Gym (`https://gym.openai.com`). These three tasks are respectively called Pusher, Striker and Thrower, that all their arms have 7 degree of freedom (DOF) and are illustrated in Figure 2. Each of these three tasks has 23 dimensional state space and 7 dimensional action space. Specifically, the 23 dimensional state space consists of the angles and velocities for each of the joint of the robotic arms, the position and velocity of the end effector, and the position of the object being manipulated. The 7 dimensional action space refers to torques of the 7-DoF motor joints. Details of each task are listed below, where $d()$ denotes the euclidean distance.

- **Pusher**. The robotic arm pushes a cylinder onto a coaster. Given the end-effector position w, the object being manipulated position m, the goal position q, and the action a. The reward function is

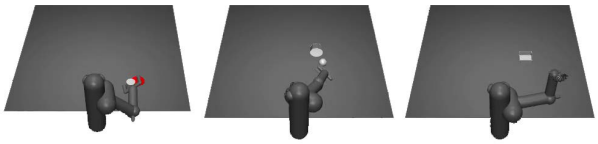
$$R(s,a) = -d(m,q) - 0.5d(m,w) - 0.1a^T a.$$

- **Striker**. The robotic arm hits a ball to a target. Given the end-effector position w, the object being manipulated position m, the goal position q, and the action a. The reward function is

$$R(s,a) = -3d(m,q) - 0.5d(m,w) - 0.1a^T a.$$

- **Thrower**. The robotic arm throws a ball into a box. Given the ball hitting the ground position z, the goal position q, and the action a. The reward function is

$$R(s,a) = -d(z,q) - 0.002a^T a.$$



(a) Pusher (b) Striker (c) Thrower

Figure 2: Experimental tasks, Pusher, Striker and Thrower. We change the length of the robotic arm of these tasks to get different environments.

For each task, we randomly change the robotic arm length parameters to generate $M_1 = 100$ different environments (while the more the better), and then training base policies for these different environments using TRPO [Schulman *et al.*, 2015], one of the best methods for the controlling tasks [Duan *et al.*, 2016]. For the randomization, variables `r_forearm_link` and `r_wrist_flex_link` are sampled from $[0.1, 0.5]$, `r_upper_arm_link` and `r_elbow_flex_link` are sampled from $[0.2, 0.6]$, independently and uniformly at random. For each environment of each task, a base policy is trained, and all these 100 policies are represented as neural networks with the same structure (two hidden layers with 64 nodes). In the TRPO training process, we set the discount factor γ to be 0.99 and the number of iteration to be 250. It needs to be noted that the discount factor will not affect the results, as it is part of the environment, but not the algorithm parameter.

We then generate $M_2 = 20$ different environments for each task. The derivative-free optimization method SRACOS [Hu *et al.*, 2017] is employed to solve the combination weights according to Eq.(1). We use the algorithm implementation from `https://github.com/eyounx/ZOOpt`, with the sample budget 250. Finally, we generate $M_3 = 20$ environments to evaluate the regression model and the calibration actions. We use SRACOS again to optimize the calibration actions according to Eq.(2). The experiment codes are at `https://github.com/eyounx/POSEC`.

4.2 Experiment Results

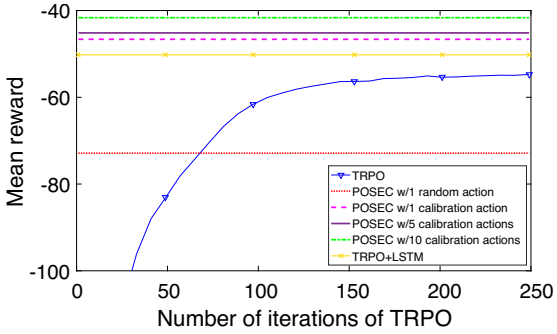
We address each of the three questions.

Q1) In order to investigate the effectiveness of the optimized calibration actions, we compare the mean reward on a new batch of test environments. We run POSEC with 1, 5, and 10 calibration actions, and compare with POSEC using 1 random action. We also compare POSEC with the LSTM approach [Peng *et al.*, 2017] that is trained over the $M_1 + M_2 + M_3$ environments. The learning from scratch method is also included as a reference, which is by training the TRPO with 250 iterations (i.e., 2.5 million samples).

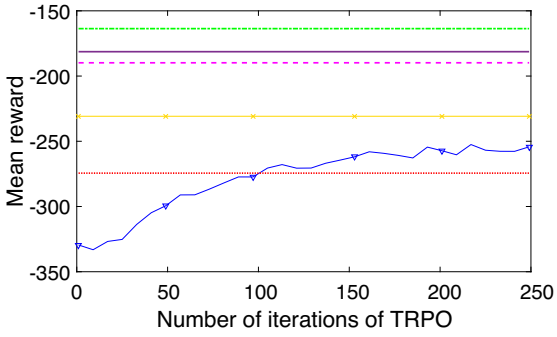
The results are shown in Figure 3. It can be observed that, first, comparing POSEC with different number of calibration actions, more actions lead to better performance consistently in the three tasks. Meanwhile, more calibration actions result in more environmental samples. Thus, the number should be determined in applications. Second, comparing POSEC with 1 calibration action and POSEC with 1 random action, it is clear that random action leads to much worse performance, due to its non-informative outcome. Third, comparing with the LSTM method, on Pusher and Striker methods, POSEC with 1 calibration action has already been superior, and on Thrower, 5 calibration actions is superior. Finally, on Pusher and Striker, POSEC with 1 calibration action (thus 1 environmental sample) is better than TRPO using 250 million samples. On thrower, it is better than TRPO using 50 million samples. This indicates that POSEC can have a strong performance with only a few environment samples.

Q2) We have recorded a demo showing how the calibration actions act. It can be watched at `https://github.com/eyounx/POSEC/raw/master/POSEC.m4v`. We also observe that the learned calibration actions are quite stable across repetitions of our experiments, leading to very small variance. Three repetitions can also been observed in the video.

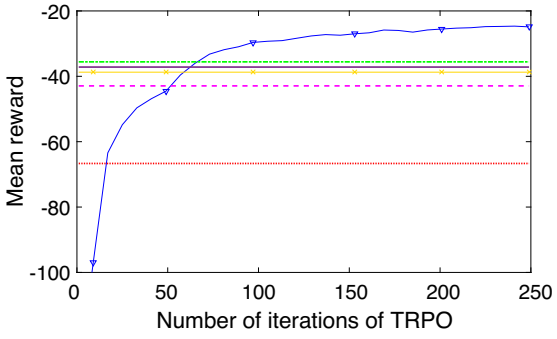
Q3) We investigate further refinement training using TRPO on each of the task, with the POSEC policy (5 calibration actions) as the initial policy. The results are shown in Figure 4. We can observe that POSEC initial policies consistently lead to the best performance in the three tasks. The LSTM method is also effective, comparing with reinforcement learning from



(a) Pusher

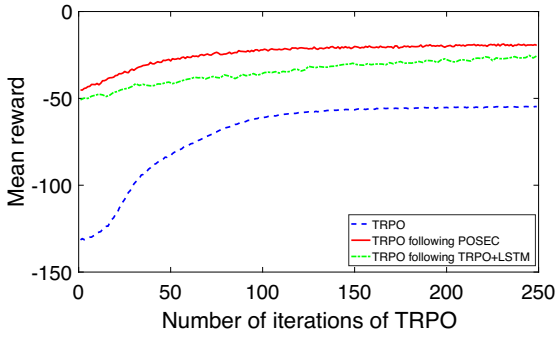


(b) Striker

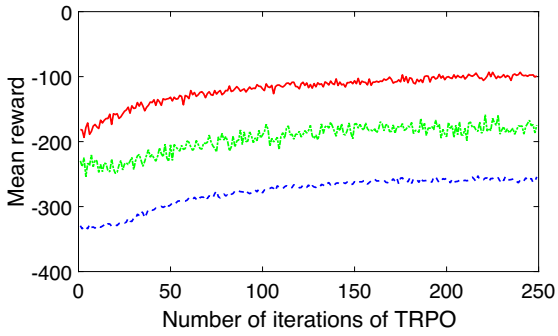


(c) Thrower

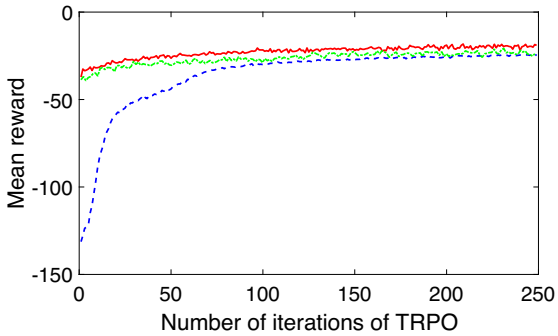
Figure 3: The performance comparisons of POSEC with different actions, LSTM approach, and TRPO trained from scratch.



(a) Pusher



(b) Striker



(c) Thrower

Figure 4: Comparison of refinement training from different initial policies.

scratch. This enables the online training of the policy in applications in order to further improve the performance.

5 Conclusion

In this paper, we propose the POSEC (POlicy Self-Evolution by Calibration) approach for fast policy self-evolution to fit new environments. POSEC runs calibration actions in the new environment, observes the outcome of the actions as the features of the environments, and reuses the base policies to form the self-evolved policy. Experiments on three robotic arm controlling tasks show that POSEC can effectively make the agent understand the environment through the calibration actions, resulting in self-evolved policy with good performance. Note that in this process, the robotic arm in the new environment needs only execute a few (e.g. 5) actions, comparing with millions of samples required by reinforcement learning approaches from scratch. Meanwhile, the performance of the policies evolved by POSEC are strongly competitive with baselines, including the policy learned from scratch and the policy trained using LSTM for environment adaptation. We hope this technique, as an effort towards the *learnware*, would be helpful to learn policies in real-world expensive-to-train tasks. Meanwhile, the policy self-evolution under more general situations, such as the environments with different state and/or action spaces, is the focus of our next research.

References

[Brafman and Tennenholtz, 2002] Ronen I. Brafman and Moshe Tennenholtz. R-MAX – A general polynomial time algorithm for near-optimal reinforcement learning. *Journal of Machine Learning Research*, 3:213–231, 2002.

[Brockman *et al.*, 2016] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. OpenAI Gym. *arXiv:1606.01540*, 2016.

[Deisenroth and Rasmussen, 2011] Marc Deisenroth and Carl E. Rasmussen. PILCO: A model-based and data-efficient approach to policy search. In *Proceedings of the 28th International Conference on Machine Learning*, pages 465–472, Bellevue, WA, 2011.

[Duan *et al.*, 2016] Yan Duan, Xi Chen, Rein Houthooft, John Schulman, and Pieter Abbeel. Benchmarking deep reinforcement learning for continuous control. In *Proceedings of the 33rd International Conference on Machine Learning*, pages 1329–1338, New York, NY, 2016.

[Finn *et al.*, 2017] Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. In *Proceedings of the 34th International Conference on Machine Learning*, pages 1126–1135, Sydney, Australia, 2017.

[Hansen and Ostermeier, 2001] Nikolaus Hansen and Andreas Ostermeier. Completely derandomized self-adaptation in evolution strategies. *Evolutionary computation*, 9(2):159–195, 2001.

[Hu *et al.*, 2017] Yi-Qi Hu, Hong Qian, and Yang Yu. Sequential classification-based optimization for direct policy search. In *Proceedings of the 31st Association for the Advancement of Artificial Intelligence*, pages 2029–2035, San Francisco, CA, 2017.

[Jong and Stone, 2007] Nicholas K. Jong and Peter Stone. Model-based exploration in continuous state spaces. In *Proceedings of the 7th International Symposium on Abstraction, Reformulation, and Approximation*, pages 258–272, Berlin,Germany, 2007.

[Kirkpatrick *et al.*, 1983] Scott Kirkpatrick, Daniel Gelatt, and Mario P. Vecchi. Optimization by simulated annealing. *Science*, 220(4598):671–680, 1983.

[Lazaric *et al.*, 2008] Alessandro Lazaric, Marcello Restelli, and Andrea Bonarini. Transfer of samples in batch reinforcement learning. In *Proceedings of the 25th International Conference on Machine Learning*, pages 544–551, Helsinki, Finland, 2008.

[Mehta *et al.*, 2008] Neville Mehta, Sriraam Natarajan, Prasad Tadepalli, and Alan Fern. Transfer in variable-reward hierarchical reinforcement learning. *Machine Learning*, 73(3):289–312, 2008.

[Mnih *et al.*, 2015] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin Riedmiller, Andreas K. Fidjeland, and Georg Ostrovski. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, 2015.

[Munos, 2014] Rémi Munos. From bandits to Monte-Carlo tree search: The optimistic principle applied to optimization and planning. *Foundations and Trends in Machine Learning*, 7(1):1–130, 2014.

[Pan *et al.*, 2011] Sinno Jialin Pan, Ivor W. Tsang, James T. Kwok, and Qiang Yang. Domain adaptation via transfer component analysis. *IEEE Transactions on Neural Networks*, 22(2):199–210, 2011.

[Peng *et al.*, 2017] Xue-Bin Peng, Marcin Andrychowicz, Wojciech Zaremba, and Pieter Abbeel. Sim-to-Real transfer of robotic control with dynamics randomization. *arXiv:1710.06537*, 2017.

[Ratitch and Precup, 2004] Bohdana Ratitch and Doina Precup. Sparse distributed memories for on-line value-based reinforcement learning. In *Proceedings of the 15th European Conference on Machine Learning*, pages 347–358, Pisa, Italy, 2004.

[Schulman *et al.*, 2015] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. Trust region policy optimization. In *Proceedings of the 33nd International conference on Machine Learning*, pages 1889–1897, Lille, France, 2015.

[Shahriari *et al.*, 2016] Bobak Shahriari, Kevin Swersky, Zi-Yu Wang, Ryan P. Adams, and Nando de Freitas. Taking the human out of the loop: A review of Bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2016.

[Silver *et al.*, 2016] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George A. Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, and Marc Lanctot. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529(7587):484–489, 2016.

[Strehl *et al.*, 2006] Alexander L. Strehl, Li-Hong Li, Eric Wiewiora, John Langford, and Michael L. Littman. PAC model-free reinforcement learning. In *Proceedings of the 23rd International Conference on Machine Learning*, pages 881–888, Pittsburgh, PA, 2006.

[Sutton and Barto, 1998] Richard S. Sutton and Andrew G. Barto. *Introduction to reinforcement learning*. MIT Press Cambridge, 1998.

[Sutton *et al.*, 2000] Richard S. Sutton, David A. McAllester, Satinder P. Singh, and Yishay Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Proceedings of the 14th Advances in Neural Information Processing Systems*, pages 1057–1063, Denver, CO, 2000.

[Watkins and Dayan, 1992] Chris Watkins and Peter Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.

[Yu *et al.*, 2016] Yang Yu, Hong Qian, and Yi-Qi Hu. Derivative-free optimization via classification. In *Proceedings of the 13th Association for the Advancement of Artificial Intelligence*, pages 2286–2292, Phoenix, AZ, 2016.

[Zhou, 2016] Zhi-Hua Zhou. Learnware: On the future of machine learning. *Frontiers of Computer Science*, 10(4):589–590, 2016.