Learning Environmental Calibration Actions for Policy Self-Evolution

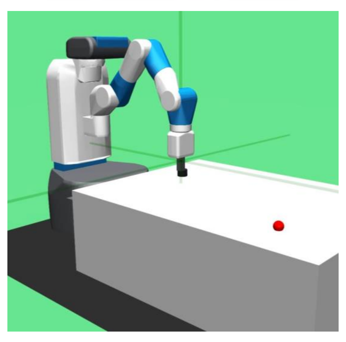
Chao Zhang, Yang Yu, Zhi-Hua Zhou emails: {zhangc, yuy, zhouzh}@lamda.nju.edu.cn

National Key Laboratory for Novel Software Technology, Nanjing University

LAMDA Learning And Mining from DatA http://lamda.nju.edu.cn

1.Motivation

Physical world reinforcement learning is highly costly



Simulators are usually very helpful

Simulation error is inevitable

- measure inaccuracy
- physical world changes

Previuosly:
simulated policy
\+ manual adjustment

Can the policy be self-evolvable to adapt to its environment?

2.Idea

Learn a meta-policy over environments

1. collect a set of previous environments
2. associate policies with environments
3. learn a mapping from environment features to policy parameters

assume:
same state and action spaces,
but different transitions

such policy is generalizable to new environments

In a new environment: map the environmental features to the policy

But, how to obtain the environmental features?

[Peng et al. 2018]: Implicitly learned in the LSTM policy model

Our idea: explicitly extract features by probing the environment

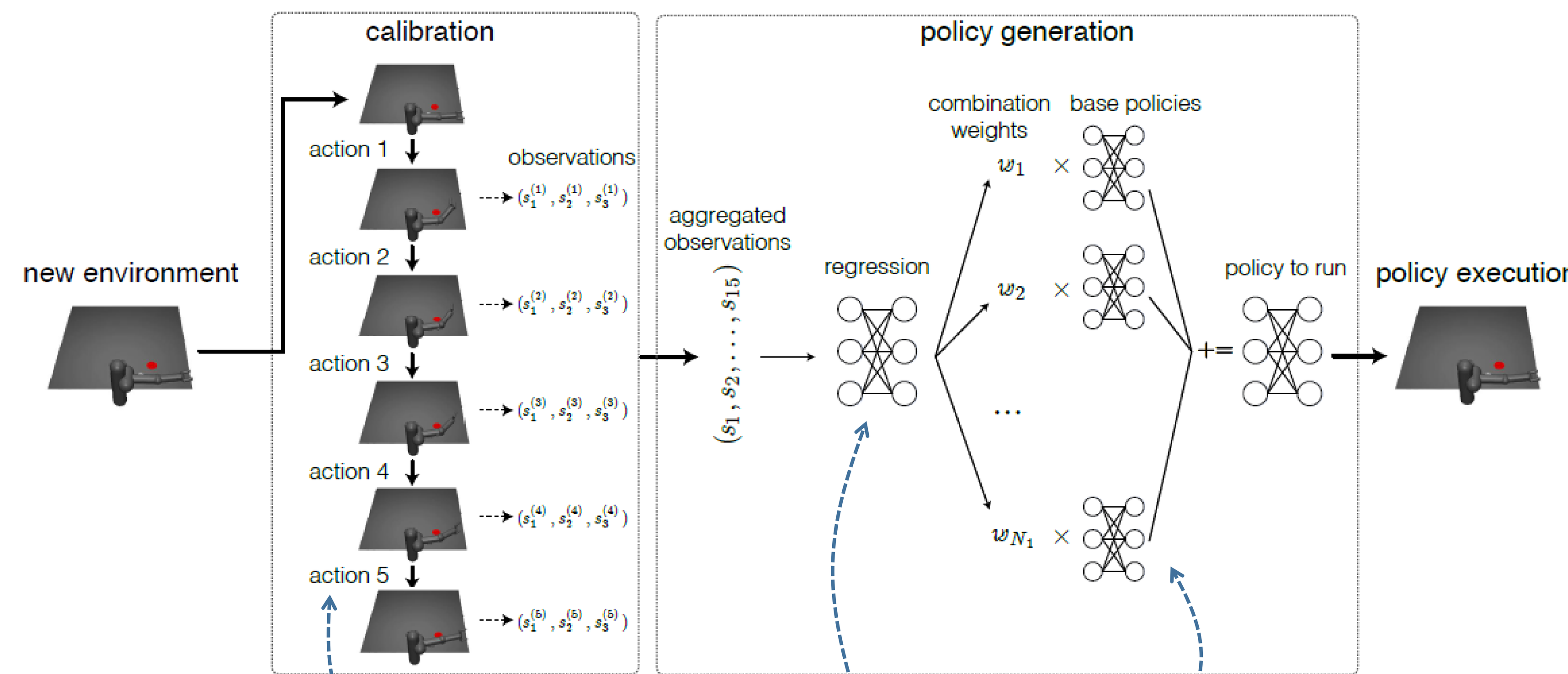
3.Proposed Method: POSEC

Framework

Extraction of environmental features:

1. run some calibration actions
2. observe the environment states after each action
3. the observations are used as the features

With the environment features, aggregate previous policies for the new environment



need to solve

calibration actions **regression model** **base policies**

Implementation

Assume a configurable simulator is available to generate environments

base policies

Sample M_1 different configurations of the environment

In every environment: train a policy heavily, as a base policy

base policies set: $\{\pi_1, \pi_2, \ldots, \pi_{M_1}\}$

combination weights

linear combination policy: $\pi_w(a|s) = \sum_{t=1}^{M_1} \frac{w_t}{\sum_{t=1}^{M_1} w_t} \pi_t(a|s)$

Draw another set of M_2 different configurations of the environment

In each environment: the reward objective function about $\pi_w(a|s)$

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

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

solved by a derivative-free optimization method [Yu et al., IJCAI'16; Hu et al., AAAI'17]

solved combination weights ($W_1, W_2, W_3, \ldots, W_{M1}$) of base policies

regression model

In each of the M_2 environment:

run the calibration actions A (assume given)

the environmental features → instance

combination weights → label

instance → regression model → label

calibration actions

Draw the final set of M_3 configurations **{MDP$_1$, …, MDP$_{M3}$}**

For every environment, and calibration actions A:

environment features: $F(\text{MDP}, A)$

predicted combination weights: $\theta^{*\top} F(\text{MDP}, A)$

combined policy: $\pi_{\theta^{*\top} F(\text{MDP}, A)}$

Optimal actions:

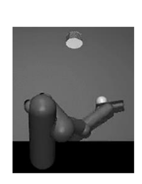
$$A^* = \arg\max_A \sum_{i=1}^{M_3} \int P_{\pi_{\theta^{*\top} F(\text{MDP}''_i, A)}}(\tau) R(\tau) d\tau$$

solved by the derivative-free optimization again

4.Experiments

Experimental task (State 23 dim, Action 7 dim)

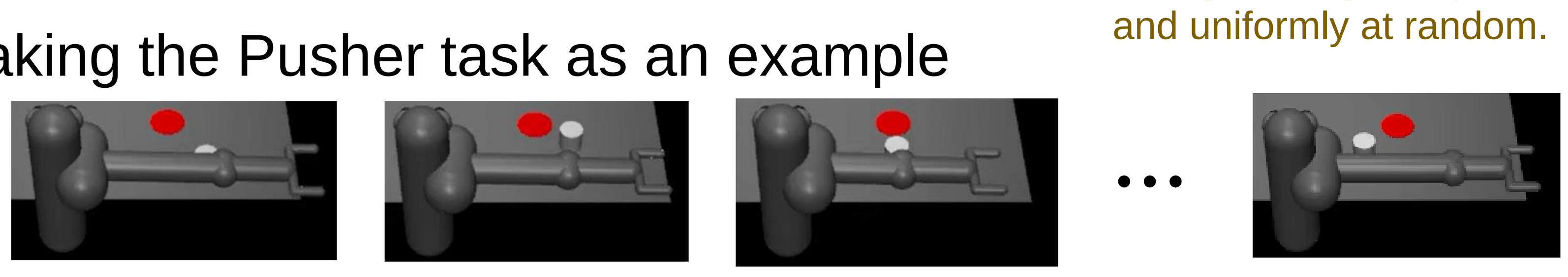
Robotic arm controlling to accomplish tasks:

- Pusher: pushes a cylinder onto a coaster 
- Striker: hits a ball to a target 
- Thrower: throws a ball into a box 

Task generation:

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.

Taking the Pusher task as an example

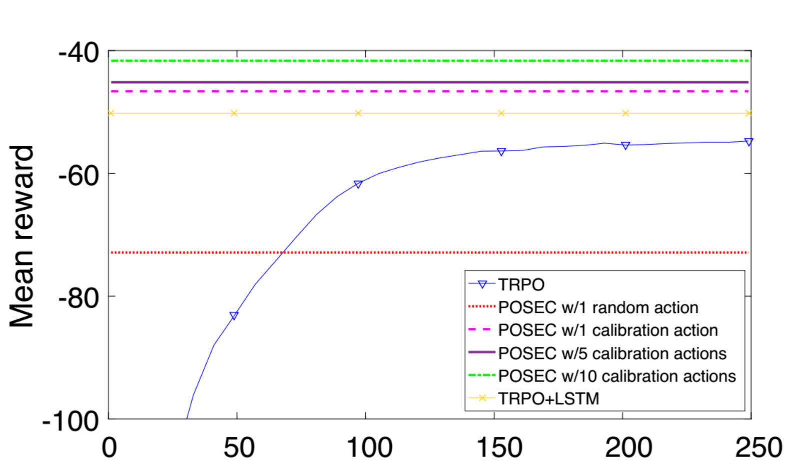


Three policy learning methods in the new environment

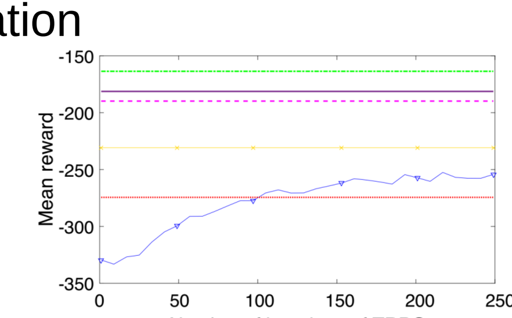
Experimented algorithms

1. The policy trained directly from scratch
2. The policy trained using LSTM for environment adaptation
3. The policy evolved by POSEC

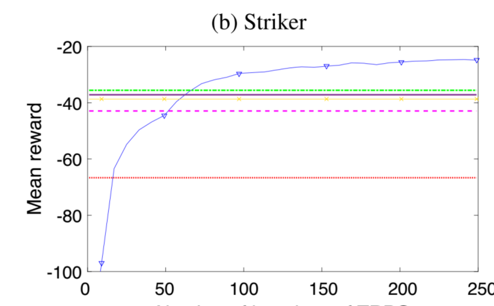
Comparisons of performance in new environments (TRPO+LSTM need to be trained online)



(a) Pusher

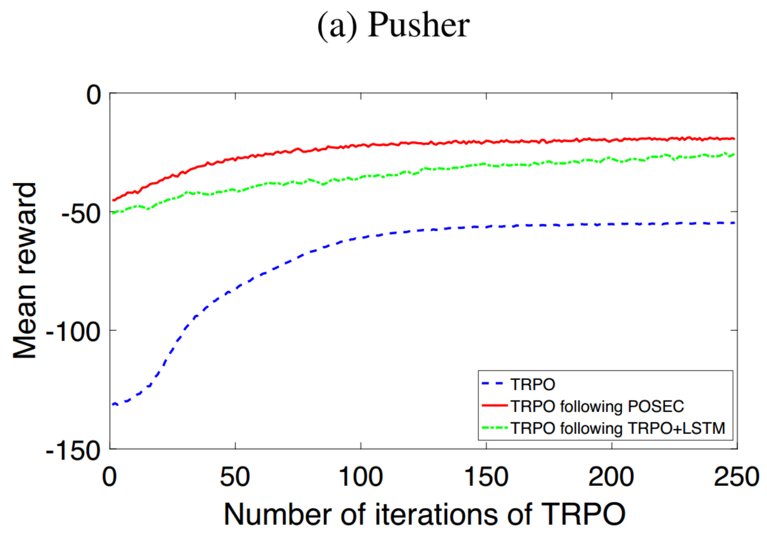


(b) Striker

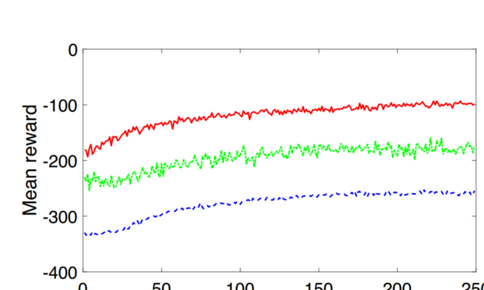


(c) Thrower

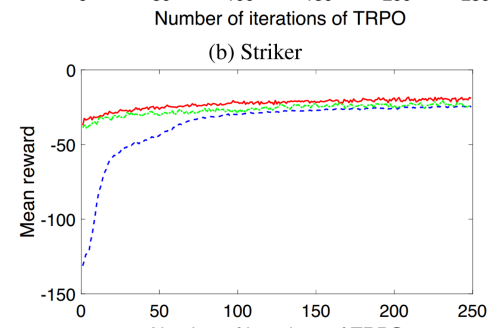
Comparison of refinement training from different initial policies.



(a) Pusher



(b) Striker



(c) Thrower

source codes can be found at: https://github.com/eyounx/POSEC

demo video: