Meta Reinforcement Learning

Ruoheng Ma
Karlsruhe Institute of Technology
Karlsruhe, Germany
Email: ukeyt@student.kit.edu

Meng Zhang
Karlsruhe Institute of Technology
Karlsruhe, Germany
Email: @student.kit.edu

***Abstract*—Meta reinforcement learning is a research area that combines reinforcement learning and meta learning. It applies meta learning approaches on reinforcement learning tasks to generalize the training result of reinforcement learning and adapts it to unseen tasks. In this report, we provide some fundamental knowledge about meta reinforcement learning. Then we discuss a couple of algorithms in detail and show their experimental results.**

Disclaimer: Section II, III, IV by Ruoheng Ma and Section...by Meng Zhang.

I. Introduction

This demo file is intended to serve as a "starter file" for IEEE conference papers produced under LaTeX using IEEEtran.cls version 1.8b and later. I wish you the best of success.

II. Preliminaries

In this section, we provide some information about the framework of meta reinforcement learning.

In the framework of reinforcement learning, the agent is continuously interacting with the environment to learn an optimal strategy in order to get a maximal future reward. Such a model can be framed as a Markov Decision Process (MDP) as depicted in the following Fig. 1.

A Markov decision process consists of five elements $\mathcal{M} = \langle \mathcal{S}, \mathcal{A}, P, R, \gamma \rangle$, whose definition is shown in the following table:

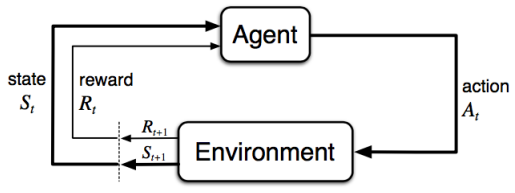

Fig. 1. The agent-environment interaction in a Markov decision process[1].

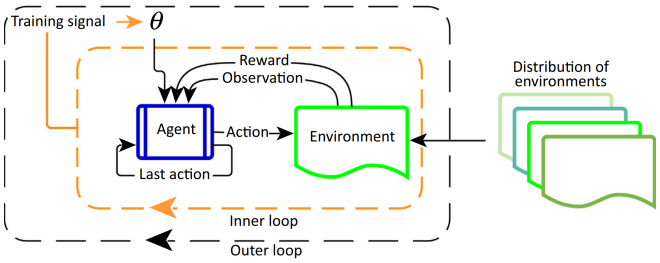

Fig. 2. Schematic of meta-reinforcement Learning. The outer loop trains the parameter weights θ, which determine the inner-loop learner ("Agent", instantiated by a recurrent neural network) that interacts with an environment for the duration of the episode. For every cycle of the outer loop, a new environment is sampled from a distribution of environments, which share some common structure.[2]

symbol	definition	function
$\mathcal{S}$	states of the agent	
$\mathcal{A}$	actions that the agent can choose	
P	transition probability function	transition function: $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \times \mathcal{R} \to \mathbb{R}_{\geq 0}$ state-transition function: $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to \mathbb{R}_{\geq 0}$
R	reward function	$\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$
γ	discounting factor for future rewards	

Except for the notation of MDP, the reinforcement learning framework also consists of the following components:

symbol	definition	function
π	policy function	deterministic: $\pi : \mathcal{S} \to \mathcal{A}$ stochastic: $\pi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}_{\geq 0}$
G	return function	$G_t = \sum_{i=0}^{T} \gamma^i R_{i+t+1}$
V_π	state value function	$V_\pi(s) = \mathbb{E}_\pi [G_t \mid S_t = s]$
Q_π	action-value function	$Q_\pi(s, a) = \mathbb{E}_\pi [G_t \mid S_t = s, A_t = a]$

In the framework of meta-reinforcement learning, a reinforcement learning model is extended to the following Fig. 2:

The agent is trained over specific learning tasks and optimized for the best performance in the inner loop, while in outer loop, the parameter of the learner is adjusted accordingly to yield the best performance on a distribution of tasks, including potentially unseen tasks.

III. Meta-Gradient Reinforcement Learning

One of the algorithm in meta reinforcement learning is the meta-gradient reinforcement learning introduced in [3] by Google DeepMind. The main idea of this algorithm is to update the meta-parameters η to achieve a better learning performance. The details of this algorithm is presented in this section.

In deep reinforcement learning, the environment model is often unknown to the agent. Therefore, the true value function has to be approximated by a function known as *return* with parameter θ. This return function plays an important role in determining the characteristics of the reinforcement learning algorithm. Usually, the return function is parameterised by a discount factor γ and the bootstrapping parameter λ, which are denoted as meta-parameters η in this paper. Throughout normal training process, these two parameters γ and λ are hand-selected and held fixed. Following are the *n*-step return[1] depending on fixed γ:

$$g_\eta(\tau_t) = R_{t+1} + \gamma R_{t+2} + \ldots, +\gamma^{n-1} R_{t+n} + \gamma^n v_\theta(S_{t+n})$$

where $\eta = \{\gamma\}$ and the λ return[4] depending on fixed γ and λ:

$$g_\eta(\tau_t) = R_{t+1} + \gamma(1-\lambda) v_\theta(S_{t+1}) + \gamma\lambda g_\eta(\tau_{t+1})$$

where $\eta = \{\gamma, \lambda\}$. In addition, the parameter θ is also updated by the following formula, taking fixed η into account:

$$\theta' = \theta + f(\tau, \theta, \eta)$$

where τ is the sample of experience being considered.

In order to achieve better performance, the algorithm introduced in [3] updates η during the training process. Besides an objective function $J(\tau, \theta, \eta)$, whose value the parameter θ is being updated to reduce, a meta-objective defined as $\bar{J}(\tau', \theta', \bar{\eta})$ is also included into this algorithm and the purpose of updating η is to increase its value.

Here is how the algorithm works: First, the algorithm starts with parameter θ and applies the update function on the first sample, resulting in a updated parameter θ'. After that, the performance is measured by $\bar{J}(\tau', \theta', \bar{\eta})$, where τ' is the next sample following τ and $\bar{\eta}$ is a fixed meta-parameter. Then, the gradient of the meta-objective with respect to the meta-parameters η is obtained by applying the chain rule:

$$\frac{\partial \bar{J}(\tau', \theta', \bar{\eta})}{\partial \eta} = \frac{\partial \bar{J}(\tau', \theta', \bar{\eta})}{\partial \theta'} \frac{\mathrm{d}\theta'}{\mathrm{d}\eta}$$

Finally, $\Delta\eta$ is calculated to maximize the meta-objective:

$$\Delta\eta = -\beta \frac{\partial \bar{J}(\tau', \theta', \bar{\eta})}{\partial \theta'} \frac{\partial f(\tau, \theta, \eta)}{d\eta}$$

where β is the learning rate for updating η.

The validation of meta-gradient algorithm is executed on Arcade Learning Environment with an agent built with the IMPALA framework. The agents are evaluated on 57 different Atari games. The result is shown below:

	η	Human starts		No-op starts	
		$\gamma = 0.99$	$\gamma = 0.995$	$\gamma = 0.99$	$\gamma = 0.995$
IMPALA	$\{\}$	144.4%	211.9%	191.8%	257.1%
Meta-gradient	$\{\lambda\}$	156.6%	214.2%	185.5%	246.5%
		$\bar{\gamma} = 0.99$	$\bar{\gamma} = 0.995$	$\bar{\gamma} = 0.99$	$\bar{\gamma} = 0.995$
Meta-gradient	$\{\gamma\}$	233.2%	267.9%	280.9%	275.5%
Meta-gradient	$\{\gamma, \lambda\}$	221.6%	292.9%	242.6%	287.6%

Fig. 3. Experiment result of meta-gradient algorithm. "Human starts" means episodes are initialized to a state that is randomly sampled from human play, while "No-op starts" means each episode is initialized with a random sequence of no-op actions.

IV. Evolved Policy Gradient

Another algorithm for meta reinforcement learning is evolved policy gradient (EPG)[5]. Unlike meta gradient algorithm which focuses on learning the return function, EPG aims to learn a surrogate loss function $L_\phi = L_{\phi+\sigma\epsilon_i}$, which is parameterized by parameter ϕ along with a small Gaussian noise $\epsilon_i \sim \mathcal{N}(0, \mathbf{I})$ as perturbed terms.

EPG contains an inner loop and an outer loop during its execution. In inner loop, the algorithm functions as normal policy gradient algorithm that tries to update parameter θ according to the following formula:

$$\theta^* = \arg\min_\theta \mathbb{E}_{\tau \sim \mathcal{M}, \pi_\theta}[L_\phi(\pi_\theta, \tau)]$$

where θ^* is the updated parameter θ, $\mathcal{M}$ is a sampled Markov decision process, τ is an episode of $\mathcal{M}$ with horizon H. The objective of the inner loop is to minimize the loss function L_ϕ. In outer loop, the loss function parameter ϕ is updated as shown below:

$$\phi^* = \arg\max_\phi \mathbb{E}_{\mathcal{M} \sim p(\mathcal{M})} \mathbb{E}_{\tau \sim \mathcal{M}, \pi_{\theta^*}}[R_\tau]$$

where $p(\mathcal{M})$ is a distribution over MDPs, π_{θ^*} is the agent's policy trained with the loss function L_ϕ and $R_\tau = \sum_{t=0}^{H} \gamma^t r_t$ is the discounted episodic return of τ. The goal of the outer loop is to achieve high expected returns in the MDP distribution.

Pseudocode of EPG algorithm is shown below:

Algorithm 1: Evolved Policy Gradients (EPG)

[Outer Loop] for epoch $e = 1, \ldots, E$ **do**
  Sample $\boldsymbol{\epsilon}_v \sim \mathcal{N}(\mathbf{0}, \mathrm{I})$ and calculate the loss parameter $\boldsymbol{\phi} + \sigma\boldsymbol{\epsilon}_v$ for $v = 1, \ldots, V$
  Each worker $w = 1, \ldots, W$ gets assigned noise vector $\lceil wV/W \rceil$ as $\boldsymbol{\epsilon}_w$
  for each worker $w = 1, \ldots, W$ **do**
    Sample MDP $\mathcal{M}_w \sim p(\mathcal{M})$
    Initialize buffer with N zero tuples
    Initialize policy parameter $\boldsymbol{\theta}$ randomly
    [Inner Loop] for step $t = 1, \ldots, U$ **do**
      Sample initial state $s_t \sim p_0$ if $\mathcal{M}_w$ needs to be reset
      Sample action $a_t \sim \pi_{\boldsymbol{\theta}}(\cdot|s_t)$
      Take action a_t in $\mathcal{M}_w$ and receive r_t, s_{t+1}, and termination flag d_t
      Add tuple (s_t, a_t, r_t, d_t) to buffer
      if $t \bmod M = 0$ **then**
        With loss parameter $\boldsymbol{\phi} + \sigma\boldsymbol{\epsilon}_w$, calculate losses L_i for steps $i = t - M, \ldots, t$ using buffer tuples $i - N, \ldots, i$
        Sample minibatches mb from last M steps shuffled, compute $L_{\mathrm{mb}} = \sum_{j \in \mathrm{mb}} L_j$, and update the policy parameter $\boldsymbol{\theta}$ and memory parameter (Eq. (5))
    In $\mathcal{M}_w$, using trained policy $\pi_{\boldsymbol{\theta}}$, sample several trajectories and compute mean return R_w
  Update the loss parameter $\boldsymbol{\phi}$ (Eq. (6))
Output: Loss $L_{\boldsymbol{\phi}}$ that trains π from scratch according to inner loop scheme, on MDPs $\sim p(\mathcal{M})$

Fig. 4. EPG algorithm.

The algorithm works as follows: Assumed that W workers are working in the inner loop. Then, at the beginning of each epoch in the outer loop, V multivariate normal vectors $\epsilon_v \in \mathcal{N}(0, I)$ of the same dimension as the loss function parameter ϕ are generated and assigned to V loss functions $L_\omega = L_{\phi+\delta\epsilon_v}$ respectively, with v being the v-th generated perturbed parameters.

Afterwards, the inner loop launches, each worker samples a random MDP from the task distribution $\mathcal{M}_\omega \sim p(\mathcal{M})$ and trains the policy π_θ along with the loss function L_ω given from the outer loop and updates the parameter θ as follows:

$$\theta \leftarrow \theta - \delta_{in} \cdot \nabla_\theta L_\omega(\pi_\theta, \tau_{t-M,\ldots,t})$$

At the end of the inner-loop training, the return R_ω is returned by each worker and is aggregated in the outer loop. Then, the parameter ϕ of the loss function is updated according to the rule shown below:

$$\phi \leftarrow \phi + \delta_{out} \cdot \frac{1}{V_\sigma} \sum_{v=1}^{V} F(\phi + \sigma\epsilon_v)\epsilon_v$$

where $F(\phi + \sigma\epsilon_v) = \frac{R_{(v-1)*W/V+1} + \ldots + R_{v*W/V}}{W/V}$.

The architecture of the loss function can be seen in the following figure:

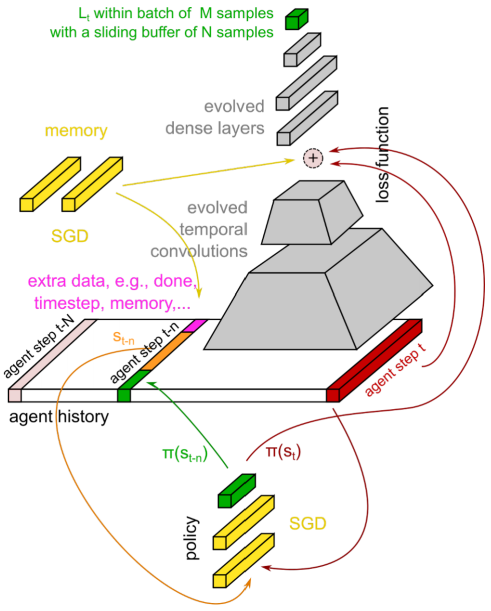

Fig. 5. Architecture of loss function L_ϕ.

As depicted in the diagram above, the architecture contains a memory unit, which is actually a single-layer neural network accepting constant input vector and is used to store the loss function value, as well as an experience buffer with limited storage, which stores the agent's N most recent experience steps, in the form of a list of tuples (s_t, a_t, r_t, d_t) at time step t, where d_t is the trajectory termination flag.

In addition, the architecture consists of temporal convolutional layers which generate a context vector $f_{context}$, and dense layers, which output the loss. The dense layer outputs the loss function L_t at time step t by taking a batch of M sequential samples $\{s_i, a_i, d_i, mem, f_{context}, \pi_\theta(\cdot|s_i)\}_{i=t-M}^{t}$. To generate the context vector, first, the data samples $\{s_i, a_i, d_i, mem, \pi_\theta(\cdot|s_i)\}_{i=t-N}^{t}$ are stack together to form a matrix, second, this matrix is processed by the temporal convolutional layers which outputs the context vector $f_context$.

This algorithm is evaluated under several randomized continuous control MuJoCo environments. For example, the following diagrams show results on RandomHopper, RandomWalker and RandomReacher.

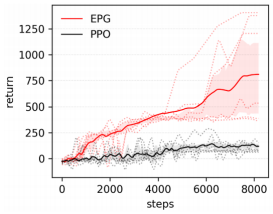

Fig. 6. RandomHopper testtime training over 128 (policy updates) x64 (update frequency) = 8196 timesteps: PPO vs no-reward EPG.

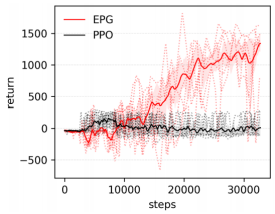

Fig. 7. RandomWalker testtime training over 256 (policy updates) x128 (update frequency) = 32768 timesteps: PPO vs no-reward EPG.

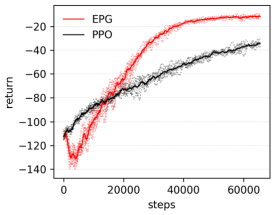

Fig. 8. RandomReacher testtime training over 512 (policy updates) x128 (update frequency) = 65536 timesteps: PG vs no-reward EPG.

V. MAML

VI. Conclusion

The conclusion goes here.

Acknowledgment

The authors would like to thank...

References

[1] R. S. Sutton and A. G. Barto., *Reinforcement learning: An introduction*.

[2] Matthew Botvinick, Sam Ritter, Jane X. Wang, Zeb Kurth-Nelson, Charles Blundell, and Demis Hassabis, *Reinforcement Learning, Fast and Slow*. URL:https://www.cell.com/action/showPdf?pii=S1364-6613%2819%2930061-0

[3] Zhongwen Xu, Hado van Hasselt and David Silver, *Meta-Gradient Reinforcement Learning*. URL:https://proceedings.neurips.cc/paper/2018/file/2715518c875999308842e3455eda2fe3-Paper.pdf

[4] R. S. Sutton, *Learning to predict by the methods of temporal differences*.

[5] Rein Houthooft, Richard Y. Chen, Phillip Isola, Bradly C. Stadie, Filip Wolski, Jonathan Ho, Pieter Abbeel, *Evolved Policy Gradients*. URL:https://papers.nips.cc/paper/2018/file/7876acb66640bad41f1e1371ef30c180-Paper.pdf