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 of reinforcement learning. It applies meta learning approaches on reinforcement learning tasks to generalize the training result of reinforcement learning and adapt it to unseen tasks. Currently, multiple meta reinforcement learning methodologies are being researched, such as optimization-based approach, hyperparameter-based approach and so on. In this report, we first provide some fundamental knowledge about meta reinforcement learning. Then we give an overview of state-of-the-art meta reinforcement learning algorithms. Finally, a couple of algorithms are discussed in detail and their experimental results are also presented.**

Disclaimer: Section...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. Fundamental Knowledge

In this section, we provide some fundamental knowledge for ease of understanding the algorithms discussed in the following sections.

A. Formulation of Reinforcement Learning and Meta Learning

In reinforcement learning, a value function

$$G_t =$$

no predefined value function is provided to tell the score of an action. In practice, a proxy of the true value function, known as a *return*, is used for estimation and optimization for the value function.

B. Hyperparameter

C. empty

III. Meta-Gradient Reinforcement Learning

One of the algorithm in meta reinforcement learning is the meta-gradient reinforcement learning introduced in [1] by Google DeepMind. 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. In [1], it is defined as shown below:

$$G_\eta^{(n)}(\tau_t) = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n v_\theta(s_{t+n})$$

$$G_\eta^{\lambda}(\tau_t) = (1-\lambda)\sum_{n=1}^{\infty} \lambda^{n-1} G_\eta^{(n)}$$

The two parameters are hand-selected and held fixed throughout training. In [1], these parameters are denoted by $\eta = \{\gamma, \lambda\}$ and θ is updated by the following formula, taking η into account:

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

where τ is the sample of experience being considered.

In order to achieve better performance, η is also updated in the training process. A meta-objective defined as follows is used to define the rule to update η:

$$\bar{J}(\tau', \theta', \bar{\eta}) = (g_{\bar{\eta}}(\tau') - v'_\theta(S'))^2$$

where τ' is the next sample and θ' is the updated parameter θ.

$\Delta\eta$ is defined below:

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

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. 1. 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)[2]. 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 $L_{\text{mb}} = \sum_{j \in \text{mb}} L_j$, 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. 2. 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:

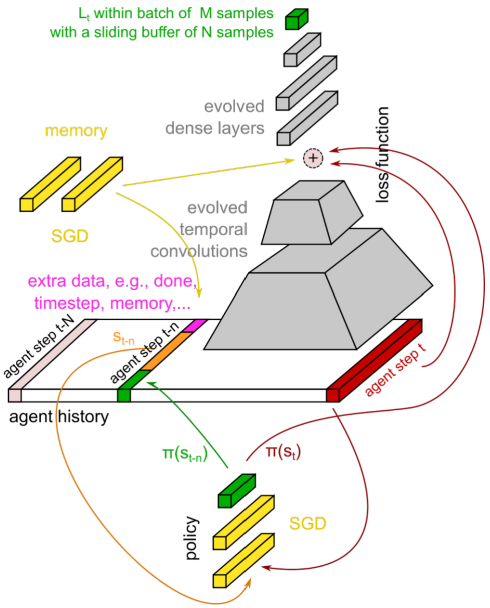

Fig. 3. Architecture of loss function L_ϕ.

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.

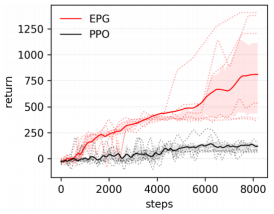

Fig. 4. RandomHopper testtime training over 128 (policy updates) x64 (update frequency) = 8196 timesteps: PPO vs no-reward EPG.

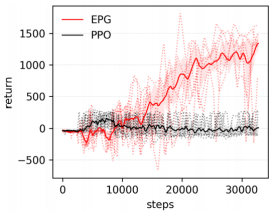

Fig. 5. RandomWalker testtime training over 256 (policy updates) x128 (update frequency) = 32768 timesteps: PPO vs no-reward EPG.

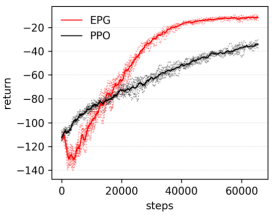

Fig. 6. RandomReacher testtime training over 512 (policy updates) x128 (update frequency) = 65536 timesteps: PG vs no-reward EPG.

V. MAML

VI. Conclusion

The conclusion goes here.

Acknowledgment

The authors would like to thank...

References

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

[2] 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