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: uknqv@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 firstly provide some fundamental knowledges about meta reinforcement learning. Then we discuss a couple of algorithms in detail and analyse their experimental results.**

Disclaimer: Sections I, II, III, IV are written by Ruoheng Ma and Sections V, VI, VII are written by Meng Zhang.

I. Introduction

Reinforcement learning is an area of machine learning and concentrates on training an intelligent agent with an optimal strategy to interact with the environment so that a maximal future reward can be achieved. Like the other areas of machine learning, reinforcement learning algorithms needs a large number of samples to train the agent. But in real life, enough number of samples can not be easily obtained, while we also hope to train the model with better performance and less time. Therefore, meta learning, which is also a subfield of machine learning, is deployed into reinforcement learning to address such problem and fulfill the requirement. There are three common approaches in meta learning: model-based approach, metric-based approach and optimization-based approach. In this report, we focus on the optimization-based meta learning approach for reinforcement learning.

Usually, in a reinforcement learning task, the agent is trained with some fixed parameter during training process. But in optimization-based meta learning, these parameters can also be trained and adapted in the training, because such a model can be split into two stages and be viewed as learner, which is being trained to obtain optimal rewards for given samples, and meta-learner, which updates the learner model's parameters and accelerates the convergence, respectively. With optimization-based meta learning, fewer samples are needed to train the agent who is learning with gradient-based method due to the adaption of model meta-parameters, and the training time can also be reduced accordingly.

In this report, some fundamental knowledge about meta reinforcement learning is first presented in section II. Then, algorithms of meta learning hyperparameters and loss function will be presented in section III, IV. Afterwards, MAML and MAESN are introduced in section V, VI. Finally, conclusion is drawn in section VII.

II. Preliminaries

In this section, we provide some fundamental knowledge and notations 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). 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:

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	$V_\pi(s) = \mathbb{E}_\pi \left[G_t \mid S_t = s \right]$
Q_π	action-value function	$Q_\pi(s, a) = \mathbb{E}_\pi \left[G_t \mid S_t = s, A_t = a \right]$

In the framework of meta-reinforcement learning, a reinforcement learning model is extended to Fig. 1:

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.

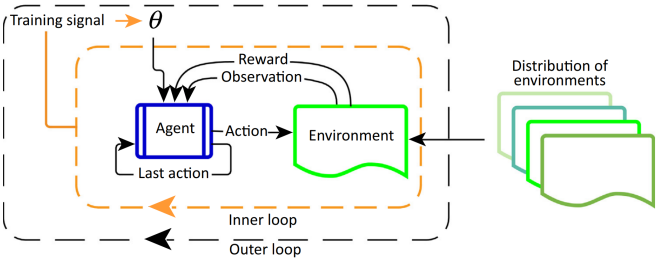

Fig. 1: Schematic of meta-reinforcement Learning. The outer loop trains the parameter weights θ, which determine the inner-loop learner ("Agent", instantiated by a recurrent neural network) that interacts with an environment for the duration of the episode. For every cycle of the outer loop, a new environment is sampled from a distribution of environments, which share some common structure[2].

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. 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 and are hand-selected and held fixed in normal training process. Following is 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 $\eta = \{\gamma\}$, t is the current time step, τ is the sample of experience being considered. 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 updated by the following formula, taking fixed η into account:

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

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 an 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:

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

The term $\frac{\mathrm{d}\theta'}{\mathrm{d}\eta}$ can be calculated as follows:

$$\begin{aligned}\frac{\mathrm{d}\theta'}{\mathrm{d}\eta} &= \frac{\mathrm{d}\theta}{\mathrm{d}\eta} + \frac{\partial f(\tau, \theta, \eta)}{\partial \eta} + \frac{\partial f(\tau, \theta, \eta)}{\partial \theta} \frac{\mathrm{d}\theta}{\mathrm{d}\eta} \\ &= \left(\mathrm{I} + \frac{\partial f(\tau, \theta, \eta)}{\partial \theta}\right) \frac{\mathrm{d}\theta}{\mathrm{d}\eta} + \frac{\partial f(\tau, \theta, \eta)}{\partial \eta}\end{aligned}$$

In practice, the gradient $\partial f(\tau, \theta, \eta)/\partial\theta$ is large and challenging to compute, because it is a N x N matrix, where N is the number of parameters in θ. Therefore, a simple approximation is to set $\partial f(\tau, \theta, \eta)/\partial\theta = 0$. Finally, $\Delta\eta$ is calculated as follows 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. As the gradient of the update with respect to η is figured out, the meta-parameters can now dynamically influence the parameter θ.

The validation of meta-gradient algorithm is executed on Arcade Learning Environment with agents built with the IMPALA framework. The agents are evaluated on 57 different Atari games and the scores are documented in the following table:

	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. 2: 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.

Four variants of IMPALA algorithm are evaluated. The first one is the baseline algorithm, while the other three are the variants with meta-gradient improvement upon different meta-parameter η. The entries of this table are the median of human-normalised scores. We can see from this table that the improved IMPALA algorithms perform better than the baseline one in most scenes. In addition, it is interesting that meta-gradient improvement upon more meta-parameters does not always yield a better performance. Furthermore, meta-gradient algorithm upon the discount factor γ can improved the performance much better in compare with the one upon the bootstrapping parameter λ.

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 policy 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, EPG adopts evolutionary strategies because there is no explicit way to write down a differentiable equation between the return and the loss function. 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 in Fig. 3. The algorithm works as follows: Because of adoption of evolutionary strategies, W workers will work in the inner loop. Then, at the beginning of each epoch in the outer loop, V multivariate normal vectors $\epsilon_v \in \mathcal{N}(0, \mathbf{I})$ of the same dimension as the loss function parameter ϕ are generated and assigned to V loss functions $L_v = L_{\phi+\delta\epsilon_v}$ respectively, with v being the v-th generated perturbed parameters. Then W workers are divided into V groups and each group obtain the v-th loss function.

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. 3: EPG algorithm.

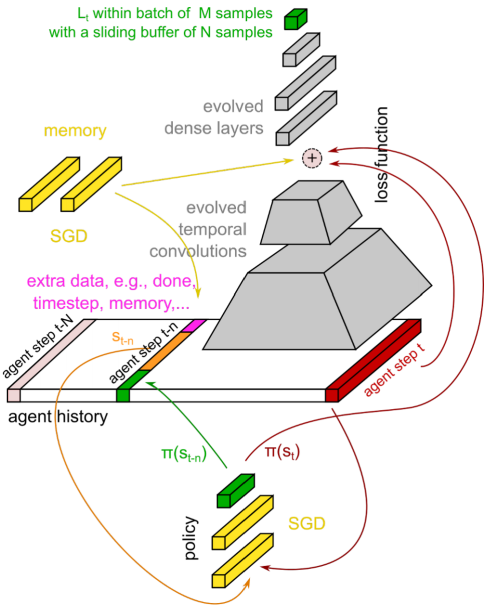

Fig. 4: Architecture of loss function L_ϕ.

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

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

At the end of the inner-loop training, the w-th return R_w 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 δ_{out} is the learning rate and $F(\phi + \sigma\epsilon_v) = \frac{R_{(v-1)*W/V+1}+\ldots+R_{v*W/V}}{W/V}$.

The architecture of the loss function can be seen in Fig. 4:

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. Fig. 5, 6, 7 show the evaluation results on RandomHopper, RandomWalker and RandomReacher, which require the agent to identify a randomly sampled environment at test time via exploratory behavior. These diagrams show the learning curves of EPG and the off-the-shelf policy gradient method, PPO, under different settings. We can see that EPG agents learn more quickly and obtain higher returns compared to PPO agents.

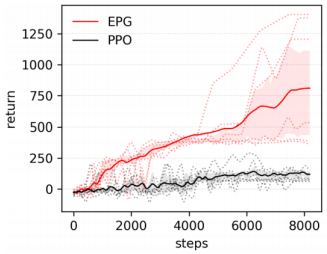

Fig. 5: RandomHopper testtime training over 128 (policy updates) x64 (update frequency) = 8196 timesteps: PPO vs no-reward EPG.

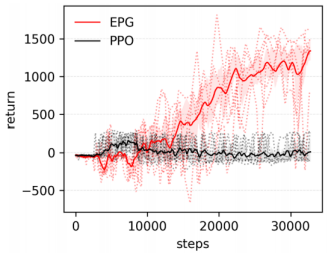

Fig. 6: RandomWalker testtime training over 256 (policy updates) x128 (update frequency) = 32768 timesteps: PPO vs no-reward EPG.

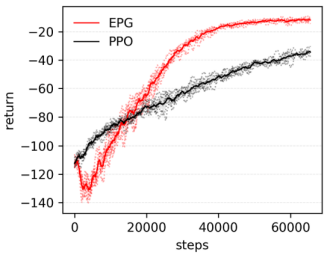

Fig. 7: RandomReacher testtime training over 512 (policy updates) x128 (update frequency) = 65536 timesteps: PG vs no-reward EPG.

Fig. 8, 9 and 10 show the ability of generalization of EPG. During metatraining, goals are randomly sampled on the positive x-axis (ant walking to the right) and at test time, goals are sampled from the negative x-axis (ant walking to the left). This task is illustrated in Fig. 8. Achieving generalization to the left side is not trivial, since it may be easy for a metalearner to overfit to the task metatraining distribution. As shown in Fig. 9, by training, RL^2 converges very fast, while EPG's performance catch up with RL^2 after 8192 steps. MAML achieves approximately the same final performance after taking a single SGD step (based on 8000 sampled steps).

But when it comes to the generalization phase, as seen in Fig. 10, the RL^2 and MAML algorithm are trapped by overfitting and yield negative result, while EPG trains the agent very quickly to the unseen task.

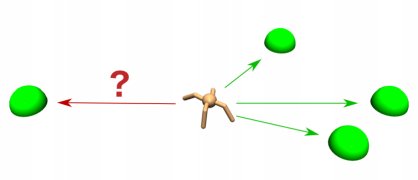

Fig. 8: GoalAnt task illustration.

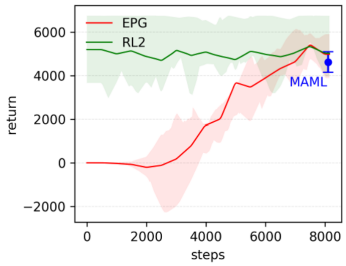

Fig. 9: Metatrained direction.

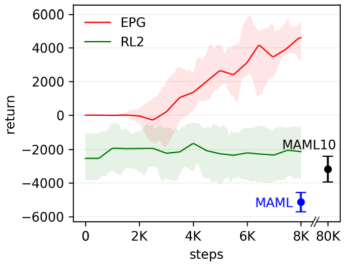

Fig. 10: Generalization direction.

V. MAML

As one of the essential meta-learning approaches, the algorithm of Model-Agnostic Meta-Learning (MAML), which was firstly introduced by Finn et al in [6], has been growing more and more popular in the field of meta-learning. As its name implies the approach offers a high compatibility to any model that learns through gradient descent.

The key idea of MAML is training a model for rapid adaptation on new tasks. More specifically, initial parameters of model are trained such that the model can achieve a significant performance on a new task after updating parameters with only a small number of gradient steps.

A. General Model-Agnostic Meta-Learning Algorithm

Firstly the algorithm is represented in a general form, which is suitable for the problems in various domains, including reinforcement learning. The objective of this approach is to learn an internal feature that is broadly applicable to all tasks in a task distribution $p(\mathcal{T})$, rather than a single task.

In the context of Model-Agnostic Meta-Learning, a model, that is expected to be able to adapt to a task distribution $p(\mathcal{T})$, is represented by a function f_θ with parameter θ. Generally, a task for meta-learning $\mathcal{T} = \{\mathcal{L}\left(\mathbf{x}_1, \mathbf{a}_1, \ldots, \mathbf{x}_H, \mathbf{a}_H\right) q\left(\mathbf{x}_1\right), q\left(\mathbf{x}_{t+1} \mid \mathbf{x}_t, \mathbf{a}_t\right), H\}$ is formed from a loss function $\mathcal{L}$, a distribution over initial observations $q\left(\mathbf{x}_1\right)$, a transition distribution $q\left(\mathbf{x}_{t+1} \mid \mathbf{x}_t, \mathbf{a}_t\right)$, and an episode length $\mathcal{H}$.

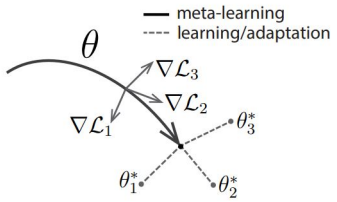

Fig. 11: Diagram of model-agnostic meta-learning algorithm (MAML), which optimizes for a representation θ that can quickly adapt to new tasks

The key point behind this approach is the transferability of internal representations learned from prior tasks over a distribution $p(\mathcal{T})$. The approach aims to find a set of weights θ, which are more sensitive to specific tasks, by changing the direction of gradient descent, so that the loss on new task can rapidly go down even with a very small number of data.

Algorithm 1 Model-Agnostic Meta-Learning

Require: $p(\mathcal{T})$: distribution over tasks
Require: α, β: step size hyperparameters
1: randomly initialize θ
2: **while** not done **do**
3: Sample batch of tasks $\mathcal{T}_i \sim p(\mathcal{T})$
4: **for all** $\mathcal{T}_i$ **do**
5: Evaluate $\nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$ with respect to K examples
6: Compute adapted parameters with gradient descent: $\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$
7: **end for**
8: Update $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})$
9: **end while**

Fig. 12: Model-Agnostic Meta-Learning algorithm

The principle of the algorithm refers to gradient by gradient, which means two gradient descents are conducted during the meta-training process. When applied to new tasks, the parameters of model θ become θ_i'. The updated parameter vector θ_i' is computed by doing the first gradient descent on each task $\mathcal{T}_i$ among batch of tasks, which is denoted as:

$$\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}\left(f_\theta\right)$$

where α represents step size fixed as hyperparameter and $\mathcal{L}_{\mathcal{T}_i}$ is loss of corresponding task. That is to say, we can get many different updated parameters θ_i' on different task $\mathcal{T}_i$. Actually the updated weights are not really for update of model, but temporarily used for the second gradient in the outer loop.

$$\min_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}\left(f_{\theta_i'}\right) = \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}\left(f_{\theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)}\right)$$

We can clearly see that the meta-optimization is performed over the model parameters θ, while the objective is computed using the updated model parameters θ_i'.

stochastic gradient descent (SGD) is performed in the approach to optimize the parameters of model. Note that here gradient is calculated on the sum of loss over all tasks in the batch of tasks, such that each task do a contribution to the whole model optimization.

$$\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}\left(f_{\theta_i'}\right)$$

where β is step size specially for the second gradient update.

B. Model-Agnostic Meta-Learning for Reinforcement Learning

For conventional reinforcement learning an agent is able to learn rules to solve specific problems, but one of the major limitations is that it is unable to generalize the learned policy to new task. On the contrary, a meta-learning model is expected to adapt to new tasks or new environments that have never been encountered during training. The combination of meta-learning and reinforcement learning, therefore, is of great significance to break the bottleneck of reinforcement learning. A representative example is Model-Agnostic Meta-Learning on reinforcement learning.

Wenn MAML is applied to reinforcement learning, the core idea and the algorithm do not change. The main differences lie in task and loss function. A task in meta-RL consists of an initial state distribution $q_i\left(\mathbf{x}_1\right)$, a transition distribution $q_i\left(\mathbf{x}_{t+1} \mid \mathbf{x}_t, \mathbf{a}_t\right)$, and the loss function $\mathcal{L}_{\mathcal{T}_i}$ corresponding to the reward function $\mathcal{R}$. Instead of images or datapoints used in supervised learning, the task of meta-RL is defined as a Markov decision process (MDP), that is composed of a set of states, a set of actions, transition probability function and reward function. We need to notice that it is the policy that is learned as a model which maps from states $\mathbf{x}_t$ to actions $\mathbf{a}_t$. The loss function for each task $\mathcal{T}_i$ is formulized as:

$$\mathcal{L}_{\mathcal{T}_i}\left(f_\phi\right) = -\mathbb{E}_{\mathbf{x}_t, \mathbf{a}_t \sim f_\phi, q_{\mathcal{T}_i}}\left[\sum_{t=1}^{H} R_i\left(\mathbf{x}_t, \mathbf{a}_t\right)\right]$$

where H is horizon, which indicates a limited number of sample trajectories in meta-RL.

Algorithm 3 MAML for Reinforcement Learning

Require: $p(\mathcal{T})$: distribution over tasks
Require: α, β: step size hyperparameters
1: randomly initialize θ
2: **while** not done **do**
3: Sample batch of tasks $\mathcal{T}_i \sim p(\mathcal{T})$
4: **for all** $\mathcal{T}_i$ **do**
5: Sample K trajectories $\mathcal{D} = \{(\mathbf{x}_1, \mathbf{a}_1, ...\mathbf{x}_H)\}$ using f_θ in $\mathcal{T}_i$
6: Evaluate $\nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$ using $\mathcal{D}$ and $\mathcal{L}_{\mathcal{T}_i}$ in Equation 4
7: Compute adapted parameters with gradient descent: $\theta_i' = \theta - \alpha \nabla_\theta \mathcal{L}_{\mathcal{T}_i}(f_\theta)$
8: Sample trajectories $\mathcal{D}_i' = \{(\mathbf{x}_1, \mathbf{a}_1, ...\mathbf{x}_H)\}$ using $f_{\theta_i'}$ in $\mathcal{T}_i$
9: **end for**
10: Update $\theta \leftarrow \theta - \beta \nabla_\theta \sum_{\mathcal{T}_i \sim p(\mathcal{T})} \mathcal{L}_{\mathcal{T}_i}(f_{\theta_i'})$ using each $\mathcal{D}_i'$ and $\mathcal{L}_{\mathcal{T}_i}$ in Equation 4
11: **end while**

Fig. 13: MAML for Reinforcement Learning

In terms of structure, algorithm 3 has no difference with algorithm 1. In K -shot reinforcement learning, K trajectories are sampled from the environment, so that the updated parameters θ_i' can be calculated by gradient descent. And the second gradient is performed on a new set of trajectories, that leads to real update of model parameters. Additionally, policy gradient methods are used here to estimate two gradients because the reward is not differentiable due to unknown dynamics.

Several experiments are conducted to evaluate the performance of MAML on reinforcement learning tasks. Among these experiments a neural network policy is trained as model with two hidden layers of size 100, and ReLU nonlinearities.

The first meta-RL evaluation is in the domain of $2D$ Navigation, where a goal position, that a point agent is expected to reach, is chosen randomly in a unit square. The state is the current $2D$position, and actions correspond to velocity control in the range [-0.1, 0.1]. What's more, the reward is the negative squared distance to the goal. The aim is to maximize the return by training neural network policy with the architecture of MAML.

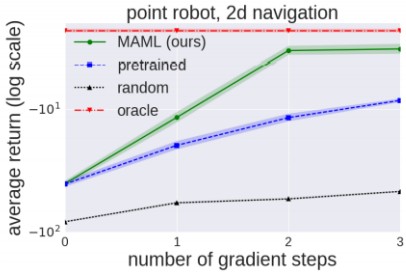

Fig. 14: quantitative results from $2D$ navigation task

The results of four approaches for adaptation on new tasks are showed in Figure 12. They are model that is initialized with MAML, conventional pretraining on the same set of tasks, random initialization, and an oracle policy that receives the goal position as input. We can easily find out that the model learned with MAML obviously outperforms the other three algorithms.

The other more complex evaluation is performed on high-dimensional locomotion tasks with the MuJoCo simulator [7], by which two different robots, half cheetah and ant, run in a particular direction or at a particular velocity. The reward in the goal velocity experiments is the negative absolute value between the current velocity of the agent and a random goal, while in the goal direction experiments we consider the magnitude of the velocity in either the forward or backward direction as the reward.

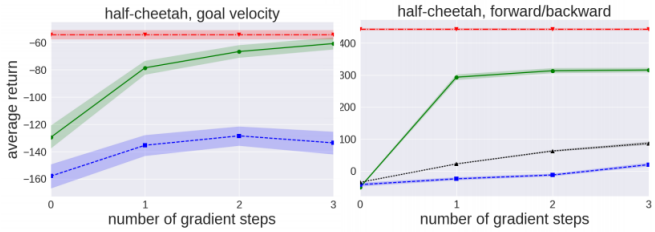

Fig. 15: results for the half-cheetah

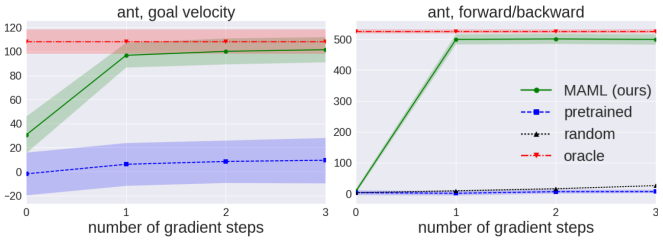

Fig. 16: results for the ant

From the results we know that the model with MAML performs a faster adaptation on its velocity and direction with a few gradient steps.

Through these two experiments we can draw a conclusion that algorithm MAML can enable faster adaptation on new problems in the field of reinforcement learning.

VI. MAESN

In this part, we introduce Model Agnostic Exploration with Structured Noise (MAESN) [8], by which an agent can be enabled to explore more effectively in new situations based on prior experience.

A. *overview*

The core idea of this gradient-based algorithm is to use prior experience both to initialize a policy and to learn a latent exploration space, by which the structured stochasticity injected policy can produce more effective exploration strategies on new tasks.

B. Meta-Learning Latent Variable Policies

The conventional action distribution in reinforcement learning is written as $\pi_\theta(a \mid s)$, which possesses no property of temporally coherent randomness throughout the trajectory because it is independent for each time step. To address this problem we introduce latent variables Z.

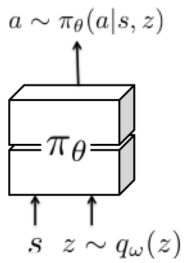

Fig. 17: latent variables

We condition the policy on random variables, which are randomly drawn from a learned latent distribution. As the latent variable is sampled once per episode, the structured stochasticity in the process of exploration can be certainly ensured. The latent variable conditioned policy is represented as $\pi_\theta(a \mid s, z)$, where $z \sim q_\omega(z)$ and $q_\omega(z)$ is latent variable distribution with parameter w.

Our aim is to meta-train the latent variable conditioned policy to generate coherent exploration strategies that perform faster adaptation on new tasks. Briefly, we jointly learn a set of policy parameters θ and the parameters of latent space distribution w, such that a policy gradient adaptation step can lead to maximal rewards.

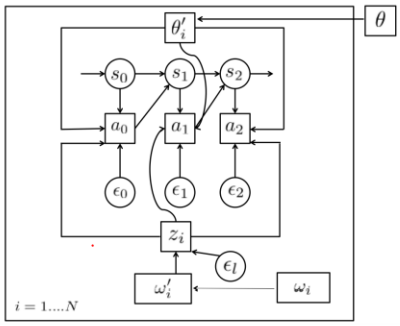

Fig. 18: Computation graph for MAESN

The objective of meta-training consists of not only sum of rewards under the post update parameters, but also the KL-divergence between the per-task pre-update distributions $q_{\omega_i}(z_i)$ and a prior $p(z)$, which ensures a effective structured exploration. The full meta-training problem can be stated mathematically as:

$$\max_{\theta,\omega_i} \sum_{i \in tasks} E_{\substack{a_t \sim \pi(a_t|s_t;\theta_i',z_i') \\ z_i' \sim q_{\omega_i'}(.)}} \left[\sum_t R_i(s_t) \right] - \sum_{i \in tasks} D_{KL}(q_{\omega_i}(.) \| p(z))$$

$$\omega_i' = \omega_i + \alpha_\omega \circ \nabla_{\omega_i} E_{\substack{a_t \sim \pi(a_t|s_t;\theta,z_i) \\ z_i \sim q_{\omega_i}(.)}} \left[\sum_t R_i(s_t) \right]$$

$$\theta_i' = \theta + \alpha_\theta \circ \nabla_\theta E_{\substack{a_t \sim \pi(a_t|s_t;\theta,z_i) \\ z_i \sim q_{\omega_i}(.)}} \left[\sum_t R_i(s_t) \right]$$

Fig. 19: meta-training procedure

The structure of training here is the same as the one in MAML, where two-level gradient is contained. The inner policy gradient is performed on the variational parameters for each task to get post-update parameters θ_i', ω_i', and then a meta-gradient is conducted to obtain θ, ω_0, ω_1, ..., ω_N.

Algorithm 1 MAESN meta-RL algorithm

```
Initialize variational parameters ω_i for each training task τ_i
for iteration k ∈ {1, ..., K} do
    Sample a batch of N training tasks from p(τ)
    for task τ_i ∈ {1, ..., N} do
        Gather data using the latent conditioned policy θ, (ω_i)
        Compute inner policy gradient on variational parameters via Equation (4) (optionally (5))
    end for
    Compute meta update on both latents and policy parameters by optimizing (3) with TRPO
end for
```

Fig. 20: MAESN algorithm

During the meta-train the sum of expected task rewards over all tasks is maximized and on the countrary the KL-divergence of pre-update distributions $q_{\omega_i}(z_i)$ against the prior$p(z_i)$ is minimized. According to training procedure we can understand MAESN as augmentation of MAML with a latent space to inject structured noise.

Wenn adapting to new task, the variational parameters ω_i used during meta-training are not suitable anymore. ω is adapted by using the policy gradient via backpropagation on the RL objective:

$$\max_\omega E_{a_t \sim \pi(a_t|s_t,\theta,z), z \sim q_\omega(.)} \left[\sum_t R(s_t) \right]$$

In the inner loop, backpropagate through the sampling operation is needed to compute the gradients with respect to ω, using either likelihood ratio or the reparameterization trick:

$$\nabla_\omega \eta = E_{a_t \sim \pi(a_t|s_t;\theta,z)} \left[\nabla \sim q_\omega(\cdot) \left[\nabla_\omega \log q_\omega(z) \sum_t R(s_t) \right] \right.$$

With the introduction of latent variables we achieve a structured stochascity, which leads to time-correlated explorations.

C. Experiments and Evaluations

In order to check the performance of meta-learned exploration strategies with MAESN, evaluations are conducted on three different task distributions with sparse rewards, which are Robotic Manipulation, Wheeled Locomotion and Legged Locomotion respectively.

In Robotic Manipulation, a robotic hand is expected to push blocks to randomly chosen target locations, which is considered as a typical multi-manipulation skill for robot.

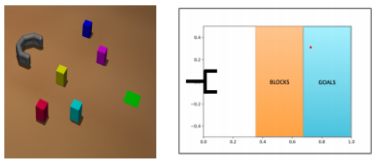

Fig. 21: Robotic Manipulation

In Wheeled Locomotion, a wheeled robot should move to different randomized goal locations with coherent exploration strategies.

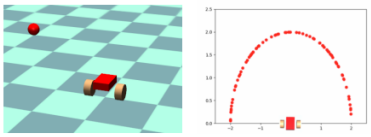

Fig. 22: Wheeled Locomotion

In Legged Locomotion, an ant walks to randomly placed goals, which requires carefully coordinated leg motions.

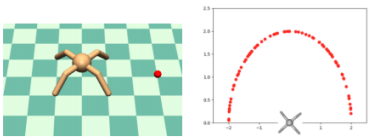

Fig. 23: Legged Locomotion

In the same time we compare MAESN with other 6 exploration approaches during meta-test on each task distribution, that are RL^2, MAML, simply learning latent spaces without fast adaptation(LatentSpace), TRPO, REINFORCE, and training from scratch with VIME.

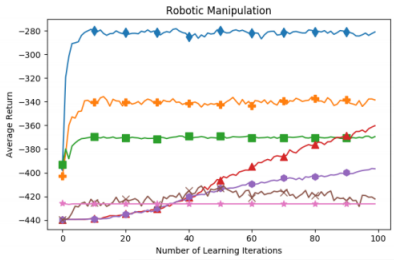

Fig. 24: results of Robotic Manipulation

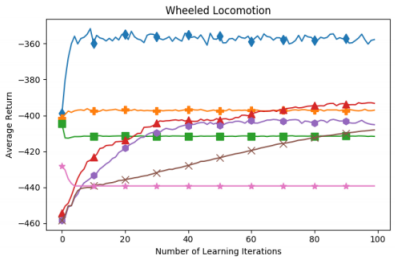

Fig. 25: results of Wheeled Locomotion

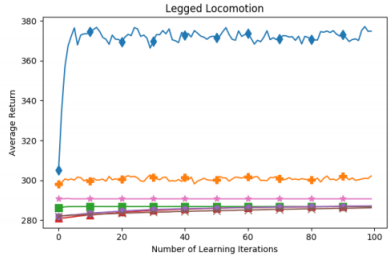

Fig. 26: results of Legged Locomotion

From the results of all exploration strategies we find that the latent variable based MAESN performs much better on new tasks compared to either prior meta-learning approaches or the methods learning from scratch, since MAESN is able to train latent space for fast adaptation.

VII. Conclusion

In this article we analysed four gradient-based algorithms of meta reinforcement learning, which are Meta-Gradient Reinforcement Learning, Evolved Policy Gradient, MAML and MAESN respectively.

The key point of Meta-Gradient Reinforcement Learning and Evolved Policy Gradient (EPG) is to achieve a better learning performance by updating the meta-parameters η of the learner model while the policy parameters are being learned. In EPG, evolutionary strategies are also deployed to speedup the convergence.

Model-Agnostic Meta-Learning (MAML) can enable an agent to quickly acquire a policy for a new test task using only a small amount of experience. Additionally, MAESN, which combines structured stochasticity with MAML, is able to provide a structured exploration behavior and to achieve fast adaptation on new task during meta-test time.

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

[6] Chelsea Finn, Pieter Abbee, Sergey Levine, *Model-Agnostic Meta-Learning for Fast Adaptation of Deep Networks*. https://arxiv.org/pdf/1703.03400.pdf

[7] Todorov, Emanuel, Erez, Tom, and Tassa, Yuval, *MuJoCo: A physics engine for model-based control*. https://ieeexplore.ieee.org/document/6386109

[8] Abhishek Gupta, Russell Mendonca, YuXuan Liu, Pieter Abbeel, Sergey Levine, *Meta-Reinforcement Learning of Structured Exploration Strategies*. https://arxiv.org/abs/1802.07245