The Implementation of MAML on HalfCheetah

Zirong Chen
Department of Computer Science
Vanderbilt University
Nashville, USA
zirong.chen@vanderbilt.edu

***Abstract*—The data-driven machine learning approaches involved in deep learning and deep reinforcement learning are data-hungry and require too many training examples or episodes. Meta learning offers an elegant solution to this problem. The goal of Meta-Learning is to let the agent learn how to deal with the new tasks faster with even fewer learning samples by learning the learnt experiences from other different tasks. In this work, Model-agnostic meta-learning (MAML) [1] is implemented on the HalfCheetah [2] PyBullet environment [3]. Further experiments on different learning setups are done to illustrate its learning efficiency.**

***Index Terms*—Deep Reinforcement Learning, Meta Learning, Few-shot Learning**

I. Motivation and Problem Description

It is normal for human beings to learn quickly from previously learnt skills or knowledge. For example, from walking forwards to walking backwards. However, with most Reinforcement Learning agents being environment-/task-specific, training from scratch is time-consuming and a lot of approaches do not guarantee convergence. There exist urgent calls for a more efficient learning approach. MAML [1] proposes a solution to this problem by (1)learning the best parameter settings from a group of tasks; (2)using the best parameter settings as initial parameters for a new task; (3)hopefully accelerating the learning process of the new task. In meta RL, leveraging the ideas behind Few-shot Learning [4], [5], the agent is expected to learn from its past experiences from many other tasks to aid the learning process on the new task. The generalization could be achieved in different ways, such as learning a generalizable representation of the data or learning a good set of policy parameters that can be used as initial parameters for new tasks.

The environment used for this work is HalfCheetah from PyBullet environments. In this environment, the goal is to train an agent which is able to make the half-cheetah-like robot run as far as it can.

Briefly speaking, Meta RL usually consists of two layers of optimization, which appear to be two loops. In the inner loop, the agent interacts with the environment and maximizes the cumulative rewards (the return). In the outer loop, a meta-learner samples an environment at each iteration and optimizes the policy to improve the agent's performance over environment distribution.During the evaluation process, different numbers of the inner loops will be conducted and compared with each other.

II. Background

A. Machine Learning v.s. Meta Learning

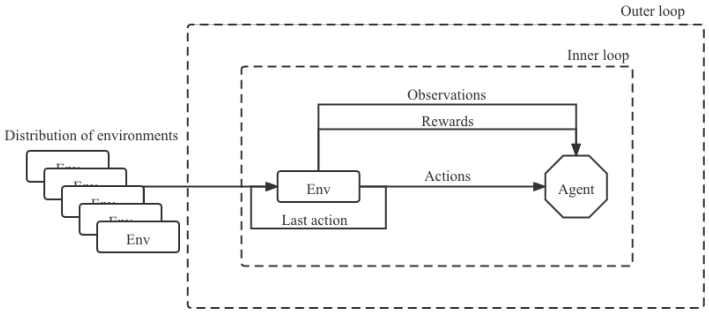
Supervised learning gives models like classifier or a regressor, which is a mapping f from input x to its corresponding label y. During inference, the goodness of the mapping f is evaluated by loss function. Meta Learning somehow shares some part of this idea with Machine Learning, however, the mapping f_θ now is a policy with parameters denoted as θ. The goal of meta learning is to learn the best θ which gives the best policy f_θ. More specifically, Meta Learning agents

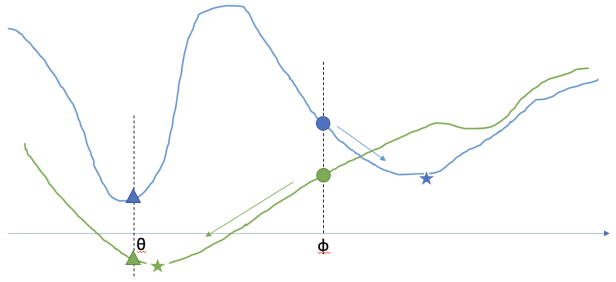
- Learn from a group of tasks T_i, $i \in I$
  - every T_i is sampled from the same task distribution $p(T)$
  - every T_i has its loss function L_{T_i}
- Validate on new samples from T_i
- The model (θ) gives the least validation error for all i will be the optimal
- We then use θ for a new task T_j, such that T_j $p(T)$ but $j \notin I$. And hopefully T_j's learning can be quick

B. Model Pretraining v.s. Meta Learning

In this implementation of Meta RL, the aim is to seek a good set of policy parameters that can be used as initial parameters for new tasks. And by assigning these initial parameters, the agent is supposed to get better performance eventually. However, in Model Pretraining, the model will be assigned with a set of parameters which aims to give better performance instantly.

In Fig. 2., θ is the parameter set given by Model Pretrainig, which is able to give a lower sum of two loss function, marked as triangles, and will not take the gradient descent





process into account. However, Meta Learning gives a set of initial parameters ϕ to start with and eventually gets a better result, marked as stars, after optimization, see gradient descent directions following both arrows.

III. Approach

A. Methodology

The idea of MAML is to learn the initial parameters for a policy such that the policy can quickly achieve its maximal performance on a new task after only a few updating steps with a small number of training trajectories.

1) Overview: The relationship between inner and outer updates are shown in Fig.3. The initial set of parameters begins with ϕ_0 and samples some task τ_1 from $p(\tau)$. During the first step, the paramters for inner loop are updated from θ_1 to θ_1' and the direction is marked as $\overrightarrow{i_1^1}$. Similar as θ_1'' and $\overrightarrow{i_2^1}$. In step-2 adaptation, the initial parameter set ϕ_0 will be updated to ϕ_1 with the same direction $\overrightarrow{o_1}$ used for updating θ_1'' from θ_1'. Similar when updating ϕ_2 from ϕ_1. Those dash lines can be understood as inner loop procedures and the solid lines can be understood as outer loop procedures.

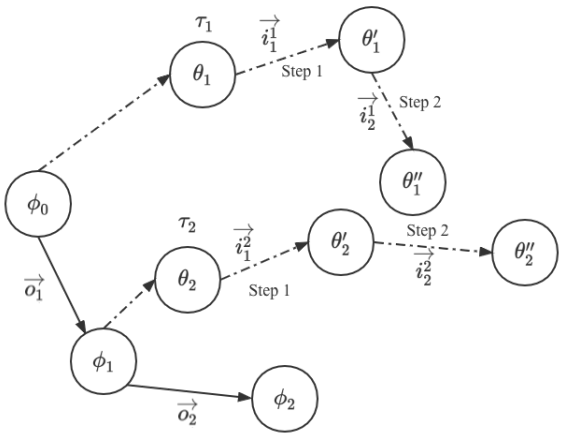
2) Algorithm Details: The objective function of the MAML is defined as Eq.1.

$$\min_{\theta} \Sigma_{\tau_i \sim p(\tau)} L_{\tau_i}(f_{\theta_i'}) = \Sigma_{\tau_i \sim p(\tau)} L_{\tau_i}(f_{\theta_i - \alpha \nabla_\theta L_{\tau_i}(f_\theta)}) \quad (1)$$

Where τ_i is an environment or task sampled from environment distribution $p(\tau)$, θ is the policy parameter, θ_i' is the adapted policy parameter on environment τ_i. The loss function on a single environment is defined as the negative accumulative reward, i.e Eq.2.

$$L_{\tau_i}(f_\phi) = -\mathbb{E}_{x_t, a_t \sim f_\phi, q_{\tau_i}} \left[\Sigma_{t=1}^{H} R_i(x_t, a_t)\right] \quad (2)$$

The detailed Meta RL algorithm is shown in Alg.1:



Algorithm 1 MAML for Reinforcement Learnin

Require: $p(\tau)$: distribution over tasks
Require: α, β: step size hyper-parameters
1: Randomly initialize θ
2: **while** not done **do**
3: Sample batch of tasks $\tau \sim p(\tau)$
4: **for** all τ_i **do**
5: Sample K trajectories $D = \{(x_1, a_1, ..., x_H)\}$ using f_θ in τ_i
6: Evaluate $\nabla_\theta L_{\tau_i}(f_\theta)$ using D and L_{τ_i} in Eq.2.
7: Compute adapted parameters with gradient descent $\theta_i' = \theta - \alpha \nabla_\theta L_{\tau_i}(f_\theta)$
8: Sample trajectories $D_i' = \{(x_1, a_1, ..., x_H)\}$ using f_θ^i in τ_i
9: **end for**
10: Update $\theta \leftarrow \theta - \beta \nabla_\theta \Sigma_{\tau_i \sim p(\tau)} L_{\tau_i}(f_\theta')$ using each D_i' and L_{τ_i} in Eq.2.
11: **end while**

Inner Loop (from line 4 to line 8 in Alg.1.): The inner loop adapts the parameter from θ to θ_i' with Policy Gradient, aiming to maximize the performance on a sampled task τ_i.

Outer Loop (from line 2 to line 10 in Alg.1): The outer loop updates θ with another policy gradient such that the agent achieves good performance on all tasks

B. Implementation Details

[1]

1) Environment Divergence: In the initial setups for MAML, multiple environments for similar tasks are needed. However, to make this task easier to train, the HalfCheetah environment is modified into two different task: walking forwards and walking backwards. By doing so, the assumption can be made that both tasks are different however close enough for meta learning. Implications about this adjustments can be found as:

- Render mode is no longer supported;
- Hyper-parameters reported on the HalfCheetah leaderboard will no longer be usable;
- Total rewards might be discounted.

2) Optimize Non-leaf Tensors: Most pre-written optimizers provided by open-sourced packages like PyTorch or TensorFlow only support leaf tensor optimization. However, in MAML, the adapted parameters θ_i' is a non-leaf tensor that depends on parameter θ. One naive solution to tackle this is to store θ and all the θ_i' somewhere and load them back to the neural networks afterwards. However, this will not work, since any load, save, copy and set operations will detach the tensor

[1]Some insights are from OpenAI implemetaion [6], RLLab [7], my online GitHub [8] and UCB CS285 online courses.

from the computation graph. It will stop the back-propagation process. One solution is to override the forward function using lower-level APIs.

3) Override Forward Functions: Default forward pass in neural networks does not satisfy MAML's implementation requirements. Instead, `torch.nn.Module`[2] or `tf.keras.Model`[3] is needed for 'manually' calculating forward pass. To further deal with the particular parameter handling network-wisely, the neural network should process the input with its own copy of the parameter set if no alternative parameter set is provided, i.e. Line 5 in Alg. 1. Those parameters stored in networks can be accessed by using `named_parameters()`[4] or `trainable_variables()`. To better retrieve those named parameters in sequences, in this work, they are all stored using `OrderedrDict`[5] with trackable names assigned.

4) Compose Own Optimizers: Due to similar reasons as stated above, the pre-written optimzers provided by PyTorch or TensorFlow cannot be used. This issue comes from the in-place fashion of modifying parameter tensors. In-place modifications will also block the computation graph too. Although writing an new optimizer might sound knotty, the good news is that working from scratch is not needed. The `torch.autograd.grad`[6] or `tf.GradientTape.gradient`[7] can stil be used under this situation. Thus, only the learning_rate needs to be taken care of in the inner-loop gradient.

C. Environment Setup

The detailed environmental setups can be found in the table below.

Package	Version
Python	3.9.7
PyTorch	1.10.0
Gym	0.21.0
PyBullet	3.2.0

IV. Experimental Studies and Results

Parallel experiments are done on following two different devices. Those experiments done on Macbook will be marked with hashtag # later while comparing performances from different settings. Running experiments on different devices might not be the best way for comparison, but due to the limited time, it saves time.

Hardware	Macbook	PC
Processor	M1	i7-11700F
System	Monterey 12.0.1	Windows 11 Home
RAM	16GB	32GB

Default hyper-parameter settings are as following:

- Inner Learning Rate: 0.1
- Outer Learning Rate: 1e-4
- Discount Factor γ: 0.99
- Number of Trajectories: 20
- Horizon (max len of trajs): 200
- Clip Ratio: 0.2
- Lambda λ (PPO): 0.9
- C1(PPO): 0.5

Next, implementation tricks and hyper-parameters that will greatly affect final performances or training efficiency will be discussed separately.

A. Implementation Tricks[8]

Tricks	Training time reduced
No trick applied	N/A
Shared models for Pi and V net	$\sim$5mins #
Backward estimation of rewards	$\sim$3mins #

B. Hyper-Parameters

1) Learning Rate: [9] After tuning the whole framework, the choice of learning rates has significant effects on final results.

Inner Learning Rate	Observations
1e-5	$\sim$40 AvgRet after 60 episodes$*$
1e-3	$\sim$50 AvgRet eventually #
0.1	$\sim$98 AvgRet eventually #

$*$The training was terminated once the returns were dropping in five consecutive epochs

Outer Learning Rate	Observations
0.1	$\sim$30 AvgRet after 60 epochs$*$
0.01	$\sim$42 AvgRet eventually #
1e-4	$\sim$98 AvgRet eventually #

$*$The AvgRet increases sharply and reaches $\sim$60 after first 10 epochs, however drops dramatically after 60 epochs

Although the pick of learning rates might be unreasonable, like in this implementation, 0.1 as inner learning rate and 1e-4 as outer learning rate, it still plays a key role in learning process. And from the experimental observations, MAML is extremely sensitive to different learning rates.

2) Random Seeds: The experiments are done with a controlled random seed for `numpy`, `PyTorch` and `random`. It is interesting to find that different random seeds can also make impacts on final performances, take STEP-1 Adaptation for example.

Random Seed	Observations
157	$\sim$90 AvgRet eventually
2020	$\sim$99 AvgRet eventually
2021	$\sim$92 AvgRet eventually

[2]https://pytorch.org/docs/stable/generated/torch.nn.Module.html

[3]https://www.tensorflow.org/api_docs/python/tf/keras/Model

[4]https://pytorch.org/docs/stable/_modules/torch/nn/modules/module.html#Module.named_parameters

[5]https://docs.python.org/3/library/collections.html#collections.OrderedDict

[6]https://pytorch.org/docs/stable/generated/torch.autograd.grad.html

[7]https://www.tensorflow.org/api_docs/python/tf/GradientTape

[8]Time Reduced Per Epoch; Comparison was only done in STEP-1 Adaptation on Macbook.

[9]Comparison was only done in STEP-0 Adaptation on PC.

3) Adaptation Steps: The number of adaptation steps determines how many steps that the agents for sub-tasks are allowed to use for learning. In other words, the adaptation steps are the number of shots in its few-shot learning scenario.

The number of adaptation steps will also have influences on training time, since the number represents how many steps the agent will spend on learning.

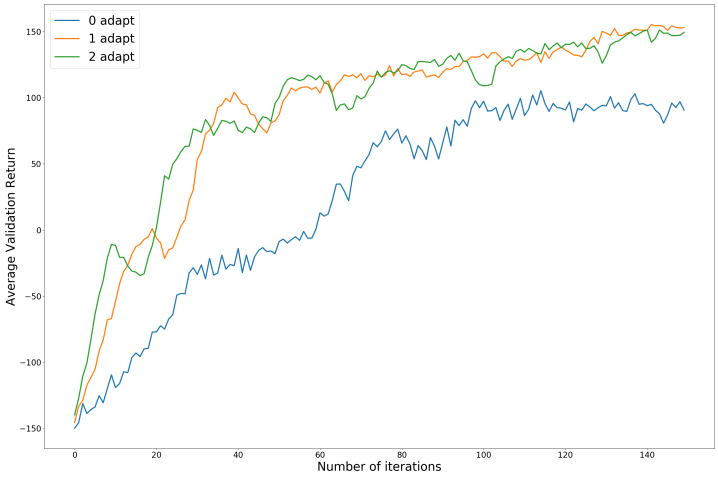
Adaptation Step	Training Time
STEP-0	∼4mins per epoch
STEP-1	∼6mins per epoch
STEP-2	∼12mins per epoch

The number of adaptation steps is also critical to final training results. Below is the table of performances in terms of final average returns with respect to the number of adaptation steps.

AdaptStep	Forward	Backward	All
STEP-0	∼99.2 #	∼97.3 #	∼98.3 #
STEP-1	∼159.9	∼148.4	∼154.8
STEP-2	∼181.5	∼144.8	∼164.1

V. Discussion and Conclusion

In this work, different implementation tricks and hyper-parameters are tried out for Meta RL. The Overall performance of MAML with the default hyper-parameter settings as number of iterations increases as shown as below.



Here are some findings from the final results:

- As the number of adaptation steps increases, the model tends to give better performances, however less time efficiency.
- The overall performances can be found as STEP-0 ¡¡ STEP-1 ≈¡ STEP-2, however the performance is the best compared with STEP-0 and STEP-1.
- The running time taken for different numbers of steps can be found as STEP-0 ¡ STEP-1 ¡ STEP-2, which obeys the no-free-lunch law.
- STEP-1 and STEP0-2 reach faster convergences compared with STEP-0.
- Steady improvements are observed in all three algorithms, if the time is allowed, better and better performance will be achieved over time.

A. When to use Meta RL

After reading several supplementary materials, here I conclude situations to implement MAML:

- When the model has nice second-order derivatives [9];
- When the distribution of tasks $p(\tau)$ is fairly static and share similar structures [5], [10];
- When tasks have similar input distributions [9];
- Tasks can be learned at a similar rate [11]

MAML offers freedom in underlying model design and it is also a unified algorithm for dealing with RL, classification and regression. If a Meta RL algorithm is well-trained, it will find us good initial parameters to share between tasks and minimize the number of updates required for each task.

B. Implications about Meta RL

However, MAML also comes up with some implications from what I have read:

- It is unsuitable for Imitation Learning [12];
- It might bring gradient instability [9];
- It sometimes prefer static learning rates [11];
- It provides little insight for how to build a model.

Acknowledgment

Here I address my gratitude towards Professor Gautam Biswas, Dr. Marcos Quinones-Grueiro and Avisek Naug for their insightful thoughts on this final project.

I would also like to thank Haotian Xue from Google and Dr. Zhiwen Tang from Meta(Facebook) for their advice and guidance while I was implementing Meta RL.

References

[1] C. Finn, P. Abbeel, and S. Levine, "Model-agnostic meta-learning for fast adaptation of deep networks," in *International Conference on Machine Learning*. PMLR, 2017, pp. 1126–1135.
[2] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym," 2016.
[3] E. Coumans and Y. Bai, "Pybullet, a python module for physics simulation for games, robotics and machine learning," http://pybullet.org, 2016–2021.
[4] J. Snell, K. Swersky, and R. S. Zemel, "Prototypical networks for few-shot learning," *arXiv preprint arXiv:1703.05175*, 2017.
[5] Y. Wang, Q. Yao, J. T. Kwok, and L. M. Ni, "Generalizing from a few examples: A survey on few-shot learning," *ACM Computing Surveys (CSUR)*, vol. 53, no. 3, pp. 1–34, 2020.
[6] J. Achiam, "Spinning Up in Deep Reinforcement Learning," 2018.
[7] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel, "Benchmarking deep reinforcement learning for continuous control," in *International conference on machine learning*. PMLR, 2016, pp. 1329–1338.
[8] R. Z. Chen, "Drl," https://github.com/RexZChen/DRL, commit = 675589983374be4adf3e21fa5eda811325df57f8, 2020.
[9] A. Antoniou, H. Edwards, and A. Storkey, "How to train your maml," *arXiv preprint arXiv:1810.09502*, 2018.
[10] E. Parisotto, J. L. Ba, and R. Salakhutdinov, "Actor-mimic: Deep multitask and transfer reinforcement learning," *arXiv preprint arXiv:1511.06342*, 2015.

[11] H. S. Behl, A. G. Baydin, and P. H. Torr, "Alpha maml: Adaptive model-agnostic meta-learning," *arXiv preprint arXiv:1905.07435*, 2019.
[12] Y. Duan, J. Schulman, X. Chen, P. L. Bartlett, I. Sutskever, and P. Abbeel, "Rl ′2: Fast reinforcement learning via slow reinforcement learning," *arXiv preprint arXiv:1611.02779*, 2016.

APPENDIX

Here is the instruction about how to run the code:

- STEP-0: `python main.py --meta_iteration 150 --num_adapt_steps 0 -- meta_batch_size 40 -K 10 -- meta_lr 1e-4 --alpha 0.1 > 0_adapt_output`
- STEP-1: `python main.py --meta_iteration 150 --num_adapt_steps 1 -- meta_batch_size 40 -K 10 -- meta_lr 1e-4 --alpha 0.1 > 1_adapt_output`
- STEP-2: `python main.py --meta_iteration 150 --num_adapt_steps 2 -- meta_batch_size 40 -K 10 -- meta_lr 1e-4 --alpha 0.1 > 2_adapt_output`