Reimplementation and Extension of Meta-Gradient Reinforcement Learning with an Objective Discovered Online

Yiqiao Qiu

January 28th 2021

Abstract

This project is a reimplementation and extension of the paper Meta-Gradient Reinforcement Learning with an Objective Discovered Online. This paper and it's original version Meta-Gradient Reinforcement Learning discuss a method to adaptively adjust meta-parameters such as bootstrapping factor λ and discount factor γ. In the reinforcement learning environment, not only the neural network parameters are optimized, but also the meta-parameters of the reinforcement learning are optimized through the gradient descent of the meta-objective function, to better adapt to the environment and make the best decision.

1 Introduction

Among them, reinforcement learning is mainly to interact with the environment. Each agent in the reinforcement learning process will be in a certain state S. The state space can be continuous or discrete. The agent can have a corresponding action A in the state S, action space can also be continuous or discrete.

Then, according to the rules of the environment, in the single-agent reinforcement learning environment, the agent will get a reward, denoted as R_s^a. The goal of the agent is to maximize the cumulative reward obtained during the entire MDP interaction process and make the best action a_i in each state S_i. At the same time, we should have a long-term perspective to consider future returns. At the same time, the future reward should obviously have a certain discount ratio compared to the current reward, which is the first meta parameter γ mentioned.In this case, we set the corresponding step size to n, which means that in the k step we should generate $\{S_k, A_k, R_{k+1}, S_{k+1}, A_{k+1}, \ldots, R_{k+n}, S_{k+n}\}$ such an interaction sequence, we believe that such an interaction sequence brings back to the Agent as $G_k = \sum_{i=0}^{n} \gamma^i R_{k+i}$, where this n is also a meta parameter, but it is discrete.

Generally speaking, we establish a state value function $v^\pi(s)$ and an action-state value function $Q^{\pi}(s, a)$ to measure the value of each state and each The value of the action performed in the state. In the process of fitting real rewards, the MDP process and the Model-free method are separated according to the knowability and unknowability of the environment. For the case where the environment is known, according to the Bellman equation.

If the state space and action space of the environment are infinite dimensional or continuous space, the computational cost of this method will be very expensive, which requires the environment to be Markovian, and the state transition matrix must be known. This is unrealistic in the real world.

Therefore, the corresponding Model-free method generates MDP sequence by interacting with the environment to fit the value function. Among them, it is divided into Monte Carlo method and Time series Differential method.

The Monte Carlo method is to allow the current Agent to directly interact with the environment to the end state, and generate a finite MDP sequence. The corresponding reward is defined as $G_n = \sum_{i=0}^{n} \gamma^i R_i$. In order to estimate $v(s)$. For each state s, if it is visited, then the count $N(s)$ plus 1, the corresponding total value $S(s)$ plus G_n, and finally estimated $v(s) = \frac{S(s)}{N(s)}$.

The timing difference learning algorithm has a step size n as a parameter, and when selecting the next action, there are two methods, one is on-policy and the other is off-policy. Generally speaking, two classic algorithms are derived from specifying $n = 1$:SARSA algorithm and Q-Learning algorithm.Both of the two algorithms only need to be executed under the current state of $s$through the current strategy of $\pi(a|s)$and the $\epsilon great$strategy, that is, to select the action that maximizes the state action-value function with the probability of $1 - \epsilon$, and to act randomly with the probability of ϵ. Use this action a to interact with the environment to get the following state s'. Then, if it is SARSA algorithm, it will repeat the $\epsilon - greedy$ strategy to get the next action a'.

Due to the complexity of various optimization problems in the reinforcement learning environment, it is necessary to introduce a deep network as a function approximator. For example, the convolution neural network is better for image processing. Therefore, for those reinforcement learning problems that use image array to represent the state, it is a better method to use deep convolution network for image feature extraction, and then use the full connection layer for strategy and value fitting, which is the classic DQN method.

For complex deep networks, optimization itself is not a simple matter, because we need to adopt a small batch stochastic gradient descent method, and various optimizers need some set hyperparameters. Therefore, if you want to train a deep network on different problems, you need to constantly debug the hyperparameters. In the reinforcement learning task, the objective function is also the meta-parameter specified in the reinforcement learning method, such as discount factor γ, bootstrapping factor λ, step size n, etc. So many changing hyperparameters will make our adjustment process more difficult.

2 Related Work

Because some relatively new methods are used in this article, the meta-gradient reinforcement learning method is improved based on these methods as a baseline. So first describe these related methods here.

2.1 Policy Gradient Optimization and Actor-Critic Algorithm

In many reinforcement learning environments, it is not very accurate to directly optimize the state-action value function of the neural network output, and use the $\epsilon - greedy$ strategy to obtain the agent action strategy distribution $\pi(a|s)$ Methods. Because the ϵ here also needs to be adjusted gradually, or an appropriate value needs to be set. Therefore, the output state-value function vector $\{Q(s, a_i)\}_{i=1}^{n}$ is modified to the policy distribution vector $\{\pi^{\theta}(a_i|s)\}_{i=1}^{n}$, also output a static state value $v^{\theta}(s)$.
The goal of the agent is to reduce the error between the value function $v^{\theta}(s)$ and the real value G_n. Therefore, the policy gradient of the loss function is

$$\nabla_{\theta} J_{policy}(\theta) = \sum_{j=1}^{m} \gamma^{j} (G_j - v^{\theta}(S_j)) \nabla_{\theta} log \pi_{\theta}(A_j|S_j)$$

This is the loss gradient generated by Actor from MDP interaction trace:$\{S_1, A_1, R_2, S_2, A_2, \ldots, S_m\}$.
Critic use the output state value from Agent to approximate the current return G_n, so it's goal is to reduce the mean square loss between $v^{\theta}(s)$ and G_n, here is the value gradient:

$$\nabla_{\theta} J_{value}(\theta) = \sum_{j=1}^{m} \gamma^{j} (G_j - v^{\theta}(S_j)) \nabla_{\theta} v^{\theta}(S_j)$$

Where $G_j = (\sum_{i=j}^{j+n} \gamma^{i-j} R_{i+1}) + \gamma^{n+1} v^{\theta}(S_{t+j})$ is the n-step TD-target we want to learn.

2.2 V-trace and IMPALA

2.2.1 IMPALA

IMPALA: Scalable Distributed Deep-RL with Importance Weighted Actor-Learner Architectures.This is the method introduced by deepmind in ICML2018, which developed a new distributed agent IMPALA (Importance Weighted Actor-Learner Architecture) thatnot only uses resources more efficiently in single-machine training but also scales to thousands of machines without sacrificing data efficiency orresource utilisation. A stable learning at high throughput by combining decoupled acting and learning with a novel off-policy correctionmethod called V-trace was also developed.
The IMPALA is described as an architecture consists of a set of actors,repeatedly generating trajectories of experience, and one ormore learners that use the experiences sent from actors

to learn $\pi(a|s)$ off-policy.
Every actor updates itsown local policy μ to the latest learner policy π and runs it for n steps in its environment, which will generate the trajectory of $\{S_i, A_i, R_i\}_{i=1}^{n}$ with the corresponding policy distribution $\mu(a|s)$. However, the learner policy π is potentially several updates ahead of the actor's policy μ at the time of update, therefore there is a policy-lag between the actors and learner(s).

2.2.2 V-trace

With the IMPALA distributed Actor-Critic method mention above, we need to set the off-policy target learned by the agent. Here is definition of V-trace target. Consider a trajectory $\{x_t, A_t, R_t\}_{i=s}^{s+n}$ generated by the actor following some policy μ. Here define the n-steps V-trace target for value function $V^{\theta}(x_s)$:

$$v_s = V(x_s) + \sum_{t=s}^{s+n-1} \gamma^{t-s}(\prod_{i=s}^{t-1} c_i)\delta_t V, \delta_t V = \rho(r_t + \gamma V(x_{t+1} - V(x_t)))$$

$\delta_t V$ is a temporal difference for value function $V^{\theta}(x_s)$. Define $\rho_t = \min(\bar{\rho}, \frac{\pi(a_t|x_t)}{\mu(a_t|x_t)})$ and $c_i = \min(\bar{c}, \frac{\pi(a_t|x_t)}{\mu(a_t|x_t)})$ as the truncated importance sampling weights(make use of the notation $\prod_{i=s}^{t-1} c_i = 1$ for $s = t$).
Specially, in the on-policy case $\mu = \pi$, and assuming $\bar{c} \geq 1$, then all $c_i = 1$ and $\rho_t = 1$, we rewrite V-trace target as $v_s = V(x_s) + \sum_{t=s}^{s+n-1} \gamma^{t-s}(r_t + \gamma V(x_{t+1}) - V(x_t)) = (\sum_{t=s}^{s+n-1} \gamma^{t-s} r_t) + \gamma^n V(x_{s+n})$, which is the on-policy n-steps Bellman target. So this property allows one to use the same algorithm for off-policy and on-policy data.
The importance sampling weights c_i and ρ_t play different roles. The weight ρ_t appears in the temporal difference $\delta_t V$ and defines the fixed point of this update rule.

2.2.3 V-trace Actor-Critic Algorithm

Finally, combine IMPALA, V-trace target and actor-critic algorithms we get the format of V-trace Actor-Critic Algorithm. Assume current state value function is $v^{\theta}(s)$, current policy is $\pi_{\theta}(a|s)$, the value function needs to reduce the L2 loss between V-trace target, the value gradient is

$$\nabla_{\theta} J_{value}(\theta) = (v_s - v^{\theta}(x_s))\nabla_{\theta} v^{\theta}(x_s)$$

policy gradient is

$$\nabla_{\theta} J_{policy}(\theta) = \rho_s \nabla_{\theta} log \pi_{\theta}(a_s|x_s)(r_s + \gamma v_{s+1} - v^{\theta}(x + s))$$

An entropy regularization like A3C could also used to prevent premature convergence to local optimal:

$$\nabla_{\theta} J_{entropy}(\theta) = \nabla_{\theta}(-\sum_a \pi_{\theta}(a|x_s) log \pi_{\theta}(a|x_s))$$

3 Method

In this paper, the author proposes to optimize these meta-parameters accordingly. Set a meta objective function to optimize the value of the meta parameter the gradient descent of meta objective function. At the same time, we can deduce the mathematical expressions of the relationship between optimization problems and meta-parameters known from reinforcement learning. Therefore, with the progress of the reinforcement learning process, the meta-parameters can also be optimized at the same time, which avoids the problem that the value and strategy learned by the agent cannot be well-fed back to the RL environment due to inappropriate and fixed meta-parameters.

3.1 Meta-Gradient Reinforcement Learning

Meta-gradient RL is a two-level optimisation process. A meta-learned inner loss L_η^{inner} is parameterised by meta-parameters η and the agent tries to optimise L_η^{inner} to update θ. For a trajectories sequence $\tau = \{\tau_i, \tau_{i+1}, \tau_{i+2}, \ldots, \tau_{i+M}, \tau_{i+M+1}\}$. The inner loss is obtained to update the Agent parameters θ with meta-parameter η fixed. Every inner loss update parameters with the gradient:$\nabla\theta_i L_\eta^{inner}(\tau_i, \theta_i)$. So the θ is updated M times as

$$\theta_i \xrightarrow{\eta} \theta_{i+1} \xrightarrow{\eta} \ldots \xrightarrow{\eta} \theta_{i+M-1} \xrightarrow{\eta} \theta_{i+M}$$

Now we obtain an outer loss to update the meta-parameter η with the updated Agent parameters θ_{i+M} and the last interaction sample τ_{i+M+1} in the trajectories. The outer loss gradient of η is $\nabla_\eta L^{outer}(\tau_{i+M+1}, \theta_{i+M})$, which is the meta-gradient mentioned above. We can use chain rule to compute the it by $\frac{\partial L^{outer}}{\partial \eta} = \frac{\partial L^{outer}}{\partial \theta`} \frac{\partial \theta`}{\partial \eta}$.
The meta-gradient algorithm above can be applied to any differentiable component of the update rule,for example to learn the discount factor γ and bootstrapping factor λ.

In this paper, we apply meta-gradients to learn the meta-parameters ofthe update target g_η online, where η are the parameters of a neural network. We call this algorithm FRODO (Flexible Reinforcement Objective Discovered Online). The following sections instantiate the FRODO algorithm for value prediction, value-based control and actor-critic control, respectively.

3.2 Prediction and Value-based Control

With the meta-parameter η based target $G_\eta(\tau)$, the value gradient of the Agent parameters θ is:

$$\nabla_\theta L_{value}^{inner}(\theta) = (G_\eta(\tau) - v^\theta(s))\nabla_\theta v^\theta(S)$$

After M updates, we compute the outer loss L^{outer} from a validation trajectory $\tau`$ as the squared difference between the predicted value and a canonical multi-step bootstrapped return $G(\tau`)$, so in classic Reinforcement Learning, we use the gradient of Agent parameters $\theta`$:

$$\nabla_{\theta`} L^{outer} = (G(\tau`) - v^{\theta`}(S`))\nabla_{\theta`} v^{\theta`}(S`)$$

With chain rule mentioned above, we could obtain the meta gradient of η, and θ_t is interpreted and treated as a function of η, which was held fixed during several updates to θ.Any standard RL update can be used in the outer loss, such as Q-learning, Q(λ), or Sarsa.

3.3 Actor-Critic in Control

In actor-critic algorithms, the update target is used both to compute policy gradient update to the policy, as well as to update the critic. As form an A2C update with $g_\eta(\tau)$, the gradient of Agent parameters is:

$$\nabla_\theta L^{inner}(\theta) = (g_\eta(\tau) - V(S))\nabla_\theta log\pi(a|s) + c_1(g_\eta(\tau) - V(S))\nabla_\theta V(S) + c_2\nabla_\theta H(\pi(*|s))$$

First part is policy gradient, second part is value gradient, third part is entropy regularization, which was weighted by c_1 and c_2.
The meta-gradient can be computed on the validation trajectory τ‘ using the classic actor-critic update:

$$\nabla_{\theta‘} L^{outer} = (G(\tau‘) - V(S‘))\nabla_{\theta‘} log\pi(A‘|S‘) + c_1(g_\eta(\tau‘) - V(S‘))\nabla_{\theta‘} V(S‘) + c_2\nabla_{\theta‘} H(\pi(*|S‘))$$

Where θ‘ is the updated Agent parameters, S‘ is the state in τ‘, in a word, the same loss function gradient which is formed by policy gradient, value gradient and entropy regulization, combined with $\frac{\partial \theta‘}{\partial \eta}$ used to compute the meta-gradient.

4 Experiments

Here I'll show the performance of meta-gradient algorithm in some Reinforcement Learning Environments. I'll use PyTorch Deeplearning Framework to complete my experiment. Every experiment is in such form:

1. agent interact with environment, generate out full trajectory τ
2. use Actor-Critic loss to update the agent's policy and value function from $\tau_{1:T-1}$, where the target of the value function is $G_\eta(\tau)$ output by meta-network. repeat (2) for several times.
3. use the last item of the trajectory as the evaluate item and the n-step Monte-Carlo target to update the target encoding parameters from meta-network.

4.1 Motivating Examples

The motivating examples is two simple MDP environment. They are all in Actor-Critic Architecture, specially here the meta-network of the agent will use the observation state, reward, discount factor and next_state value output from agent-network to feed into a Long Short-Term Memory network with 32 hidden union, with a 32-1 Linear layer to encode a $G_\eta(\tau)$ as the value function approximate target, where τ is the interaction trajectory.

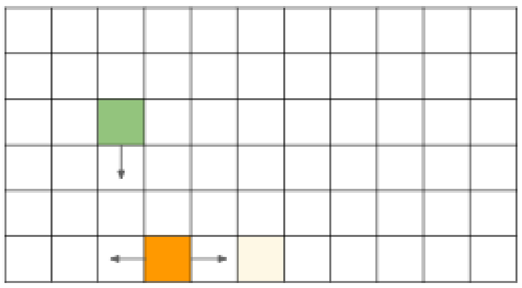

FIGURE 1: “Catch” for learning bootstrapping

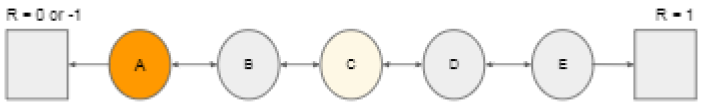

FIGURE 2: “5-state random walk” for non-stationarity

4.1.1 bootstrapping experiments

We use a simple 6×11 environment called “Catch”. The agent controls apaddle located on the bottom row of the grid. The agent starts in the centre and, on each step, it canmove on cell to the left, one cell to the right or stand still. At the beginning of each episode a pellet appears in a random start location on the top row. On each step, the pellet move down on cell. Whenthe pellet reaches the top row the episode terminates. The agent receives a reward of 1 if it caught the pellet, -1 otherwise. All other rewards are zero. Here is my reimplementation result of the bootstrapping experiment, which is the average score of 10 different random seed.

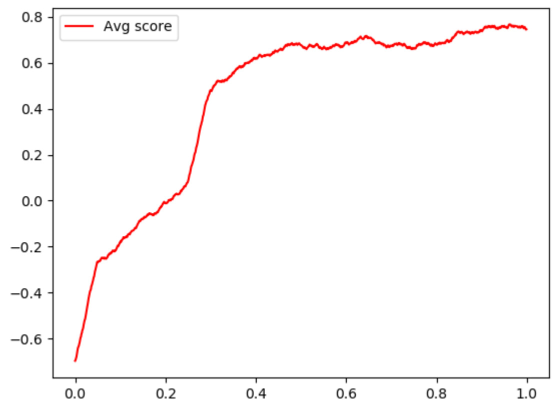

FIGURE 3: My “Catch” experiment result

4.1.2 non-stationarity experiments

We use a non-stationary variant of the “5-state Random Walk” environment. The agent starts in the centre of a chain, and moves randomly left or righton each step. Episodes terminate when the agent steps out of either boundary, on termination areward is provided to the agent; on the left side, the reward is either 0 or -1(switching every 960 time-steps, which corresponds to 10 iterations of FRODO), on the right side the reward is always 1.Each trajectory has 16 time steps.

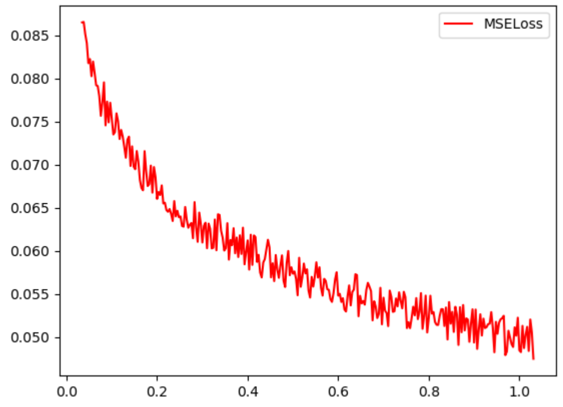

FIGURE 4: My "Random-Walk" experiment result

5 Extension of Algorithm

In the original paper, the inner loss use the encoding result $G_\eta(\tau)$ output from meta-network to be the target of value function, while the outer loss (meta loss-function) still use tradictional n-step Monte-Carlo loss.
While the network hasn't been trained, the random seed determine how the parameters of the network to be inited. And the target output from meta-network $G_\eta(\tau)$ is not good at begin, which will lead the value function and policy distribution from agent-network θ to a local minimum of this complex optimization space of these Nerual Network parameters θ. Although with the training of meta-network and agent-network continues, it can finally constringency to some local minimum with the Stochastic Gradient Descent. But different set of the random seed affect the final performance of the Algorithm.
Hence, I have an idea to make an extension of this meta-gradient Reinforcement Learning Algorithm. This idea is inspired by a paper which producted recently:Meta Pseudo Labels. And it reach the state of art of the key problem: Image 1000 classification on ImageNet dataset. It's top-1 accuracy reach 90%. In Meta Pseudo Labels, after the student model generate the pseudo label from teacher model, the quality of the pseudo label is also used as a reward signal for the teacher model. The teacher is also updated with the principle in Reinforcement Learning, which means it want to maximize the performance of the pseudo labels generated by its student model.
In order to make an extension with our algorithm, we could take the agent output $v^\theta(s)$ and $\pi^\theta(a|s)$ as a pseudo label generated by the agent-network, its performance measured by actor-critic inner loss function in trajectory (*), which may also be an negetive reward of the meta-network which had generated an object $G_\eta(\tau)$.
(*) means that in this actor-critic loss function, we didn't compute the loss for value function

$v^{\theta}(s)$ and encoded object $G_{\eta}(\tau)$, instead, only with the evaluate object setting in outer loss function. In conclusion, the original algorithm could be extend as above:

$$\nabla_{\eta} L^{outer}_{extend} = b_1 \nabla_{\theta} L^{inner}_{revise}(\theta) \frac{\partial \theta}{\partial \eta} + \nabla_{\theta`} L^{outer} \frac{\partial \theta`}{\partial \eta}$$

b_1 is a weight hyper-parameter. Where as:

$$\nabla_{\theta} L^{inner}_{revise}(\theta) = (G(\tau) - V(S)) \nabla_{\theta} log\pi(a|s) + c_1(G(\tau) - V(S)) \nabla_{\theta} V(S) + c_2 \nabla_{\theta} H(\pi(*|s))$$

$$\nabla_{\theta`} L^{outer} = (G(\tau`) - V(S`)) \nabla_{\theta`} log\pi(A`|S`) + c_1(g_{\eta}(\tau`) - V(S`)) \nabla_{\theta`} V(S`) + c_2 \nabla_{\theta`} H(\pi(*|S`))$$

With the meta-network being trained better, we could reduce b_1 to let the agent mainly approximate the value function output by meta-network, which means to set b_1 as $\frac{b_1}{L^{outer}}$.

5.1 Experiment of Extension Algorithm

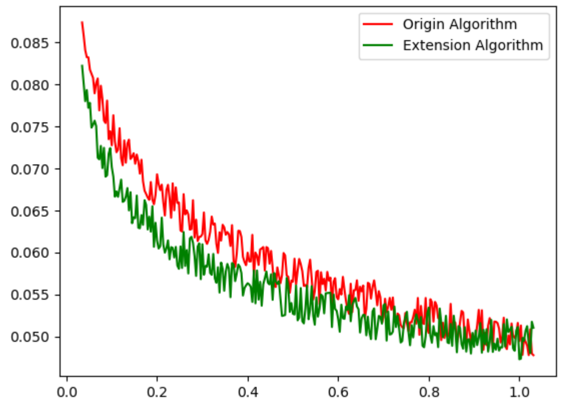

FIGURE 5: Extension “Catch” experiment result

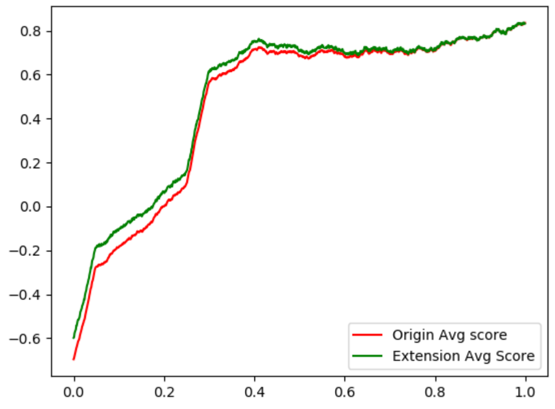

FIGURE 6: Extension “Random-Walk” experiment result

With this kind of extension, the meta-network could also consider the quality of objective function it created for agent-network. Since we know that most of the objective function like n-step TD-learning, SARSA, Q-learning are still widely used in tradictional Reinforcement

Learning, so with the approximating procedure of the agent-network parameters, which trys to optimize both of fixed objective function and $G_\eta(\tau)$ output from meta-network, could the agent-network avoid to be mislead by not well trained meta-network at the beginning. We could still mainly use $G_\eta(\tau)$ to be the objective function after the meta-network as achieve a better performance in objective discovering online.
This extension algorithm has a difference from the original one is that it reduces the begin loss from ground truth value function. The beginning loss from the original algorithm is corrected by the changing of learned objective function, which adjusts from only approximate the bias object $G_\eta(\tau)$ produced by insufficent training meta-network to both learning correct n-step TD loss and $G_\eta(\tau)$.
I still do experiment of this extension algorithm with the same two environment as above: "Catching" environment and "random-walk" environment.
The results are showed in figure 5 and 6.
It's obvious that the extension algorithm performs better than the original one at the beginning of training procedure. With the meta-network getting better, the performance of extension algorithm converge to the original one. Because as the definition of my extension:

$$\nabla_\eta L_{extend}^{outer} = \frac{b_1}{L^{outer}} \nabla_\theta L_{revise}^{inner}(\theta) \frac{\partial \theta}{\partial \eta} + \nabla_{\theta'} L^{outer} \frac{\partial \theta'}{\partial \eta}$$

The lower outer loss is, the fewer we need to learn from original revise TD-target function like n-step TD-learning. Finally, the meta-network only needs to use mainly meta-gradient $\nabla_{\theta'} L^{outer} \frac{\partial \theta'}{\partial \eta}$ to update.

5.2 Implementation Details of some Python code

Here I provide some of my implementation details like the original author, which is finished in PyTorch.

```python
import torch
import torch.nn as nn
import torch.optim as optim

class Agent_net(nn.Module):
    def __init__(self):
        self.hidden_layer = nn.Linear(7, 32)
        self.value_out = nn.Linear(32, 1)
        self.policy_out = nn.Linear(32, 2)

    def forward(self, x):
        x = self.hidden_layer(x)
        value = self.value_out(x)
        policy = self.policy_out(x)
        return torch.log_softmax(policy, dim=1), value

```

```python
class Agent(object):
    def __init__(self, seed):
        torch.manual_seed(seed)
        self.meta_network = nn.Sequential(
            nn.LSTM(input_size=4, hidden_size=32,
                num_layers=1,bias=False,
                batch_first=True, dropout=0,
                bidirectional=False),
                nn.Linear(32, 1)
            )

        self.agent_network = Agent_net()
        self.meta_optim = optim.RMSprop(
            self.meta_network.parameters(), lr=1e-2)
        self.agent_optim = optim.RMSprop(
            self.agent_network.parameters(), lr=1e-1)


    def agent_pred(self, obs):
        return self.agent_network(obs)

    def meta_pred(self, eta_inputs):
        input_vec = []
        traj_len = eta_inputs['obs'].shape[0]
        for i in range(traj_len):
            obs = eta_inputs['obs'][i]
            reward = eta_inputs['reward'][i]
            discount = eta_inputs['discount'][i]
            value = eta_inputs['value'][i]
            input_i = torch.Tensor(
                [obs, reward, discount, value])
            input_vec.append(input_i)
        input_vec = torch.stack(input_vec, dim=0)
        print(input_vec.shape)
        g_ret = self.meta_network(input_vec)
        return g_ret

    def discounted_return(self,
        rewards,discounts,next_n_value):
        traj_len = rewards.shape[0]
        ret = next_n_value*(discounts**traj_len)
        for i in range(traj_len):
            ret += rewards[i]*(discounts[i]**i)
        return ret

    def _inner_loss(self, trajectory):
        obs = trajectory.obs
        rewards = trajectory.reward[1:]
        discounts = trajectory.discount[1:]
        policy, next_values = self.agent_pred(obs)
        eta_inputs = {
                'obs':obs[:-1],
                'reward':rewards,
                'discount':discounts,
                'value':next_values[1:],
        }

        G_target = self.meta_pred(eta_inputs)
        td_error = G_target - next_values[:-1]
        policy_loss = (td_error*policy).mean()
        value_loss = (td_error*next_values).mean()
        return policy_loss+value_loss

    def _outer_loss(self, trajectory):
        obs = trajectory.obs
        rewards = trajectory.reward[1:]
        discounts = trajectory.discount[1:]
        policy, next_values = self.agent_pred(obs)
        G = self.discounted_return(
            rewards, discounts, next_values[-1])
        td_error = G.detach() - next_values[:-1]
        policy_loss = (td_error*policy).mean()
        value_loss = (td_error*next_values).mean()
        return policy_loss+value_loss

    def agent_update(self, trajectory):
        inner_loss = self._inner_loss(trajectory)
        inner_loss.backward()
        self.agent_optim.step()
        return inner_loss.item()

    def agent_update_M(self, trajectory_arr):
        #M=5
        for trajectory in trajectory_arr[:-1]:
            self.agent_update(trajectory)
            val_traj = trajectory_arr[-1]
            outer_loss = self._outer_loss(val_traj)
            return outer_loss

    def two_level_update(self, traj_arr):
        outer_loss = self.agent_update_M(traj_arr)
        outer_loss.backward()
        self.meta_optim.step()
        return outer_loss.item()
```