CS294-112 Deep Reinforcement Learning HW5: Meta-Reinforcement Learning

Due November 13th, 11:59 pm

1 Introduction

Deep reinforcement learning algorithms usually require a large number of trials. So far with the tools we have learned in this course, learning a new task entails re-collecting this large dataset and training from scratch. Intuitively, knowledge gained in learning one task should help to learn new, related tasks more quickly. Humans and animals are able to learn new tasks in just a few trials. In this assignment, we design a reinforcement learning algorithm that leverages prior experience to figure out how to solve new tasks quickly. In recent literature, such methods are referred to as *meta-reinforcement learning* Mishra et al. [2018], Finn et al. [2017], Wang et al. [2016], Duan et al. [2016].

2 Background

2.1 Notation

Formally, we will define a task as a finite-horizon discounted Markov Decision Process (MDP) M[1], drawn from a distribution $p_M : M \to R_+$. Each MDP M is defined as $M = (S, A, P, r, \rho_0, \gamma, T)$ in which S is the set of states, A the set of actions, $P : S \times A \to R_+$ the dynamics distribution, $r : S \times A \to [-R_{max}, R_{max}]$ a bounded reward function, ρ_0 the initial state distribution, $\gamma \in [0, 1]$ the discount factor, and T the horizon. Given a state s, the agent takes action a, receives reward r and a boolean indicating if the episode has ended d. While these ideas can be applied to value-based methods as well, we will focus on policy gradient approaches, in which we optimize a stochastic policy π_θ, parameterized by θ. The objective is to maximize its expected discounted return, $J(\pi_\theta) = E[\sum_{t=0}^{t} \gamma^t r(s_t, a_t)]$.

[1] Infinite-horizon, undiscounted, and first-exit formulations are also possible, and are straightforward generalizations of this definition.

2.2 Contextual Policies

We define a *trial* to be a set of episodes collected from a single task. This is analogous to a batch of trajectories that might be used to update the policy in a policy gradient method. Our goal is to learn a policy that solves new tasks drawn from p_M within a few trials. To do this, the policy must leverage knowledge from previously learned tasks. A simple solution is to fine-tune a pre-trained policy on the collected trajectories. This approach is not very robust and often fails in practice without additional technical improvements, because it may be hard to jump out of the optimum that the policy converged to from the training tasks.

Instead, we condition a single policy on a *context*, which encodes the task. This context is a function of the agent's prior experience. Given $M \sim p_M$, the policy initially acts conditioned on an empty context. As experience is accumulated, the policy is able to infer information about the task. To enable this behavior, the policy is optimized end-to-end to solve M given the context.

2.3 Recurrent policy

One way to incorporate context is by concatenating (s, a, r, d) tuples across time and feeding them as input into the policy (rather than having the input only be the state s). The action taken, the reward received, and whether the episode ended provide important context for the agent to infer what task it is solving. If we were to implement this with a fully connected policy architecture, the number of policy parameters grows linearly with the length of the history, quickly becoming intractable. Instead, we can model the policy as a recurrent network. We'll take a short detour into how recurrent networks work.

2.3.1 Recurrent Neural Networks

Ah, recurrent neural networks (RNN), the deepest networks of all[2]. Unlike a memoryless feedforward policy that takes in a state s_t and outputs a_t as

$$a_t = \pi(s_t),$$

[2] https://www.youtube.com/watch?v=0YLpp$_T VhLY$

an RNN based policy also contains a hidden state h that is carried through time:

$$\begin{aligned} a_0, h_1 &= \pi(s_0, h_0) \\ a_1, h_2 &= \pi(s_1, h_1) \\ a_2, h_3 &= \pi(s_2, h_2) \\ a_3, h_4 &= \pi(s_3, h_3) \\ &\vdots \end{aligned}$$

The next hidden state h_{t+1} is a function of both the current state s_t and the current hidden state h_t. Therefore, at a high level, an RNN policy may be implemented like so:

$$\begin{aligned} h_{t+1} &= f(s_t, h_t) \\ a_t &= g(s_t, h_{t+1}) \end{aligned}$$

such that

$$a_t, h_{t+1} = \pi(s_t, h_t).$$

Unlike a feedfoward network, the hidden state allows the RNN to remember information that has happened many timesteps in the past. In particular, given a task drawn from p_M, if the observations contain information about the reward structure of the environment, then the RNN policy may be able to infer the task by aggregating reward information over multiple timesteps in the hidden state, such that it can quickly learn to solve that task. At each timestep the policy receives (s, a, r, d) as input, which is used to update the hidden state. The next action is a function of this hidden state, and is thus a function of the policy's experience up to that timestep. The same hidden state is updated throughout the trial. During training, we sample $M \sim p_M$, and roll out the policy to collect data for the update. The policy is optimized to maximize the total discounted reward accumulated across the whole trial (not a single episode). We optimize the usual policy gradient objective, using backpropagation through time (BPTT) [Werbos, 1990] to compute gradients dependent on the agent's prior experience. This approach is based on the recent papers [Duan et al., 2016, Wang et al., 2016]. By training on a distribution of tasks (rather than a single task as earlier in this course), the policy's hidden state can learn to encode a representation of the task, which the policy can then leverage to solve it.

2.4 Learning to Learn

So far we have viewed the recurrent policy as a simple modeling choice to allow the agent to condition actions on a history of trajectories. However, we can

also view this approach as learning to learn. In this view, the recurrent policy is itself a learning algorithm, which learns over time as it accumulates more history. The "learning" that the policy performs for a new task simply consists of ingesting the context and outputting the action. The "learned" learning algorithm is therefore implemented in the weights of the network. Thus we can think of this formulation as consisting of an inner and outer loop. In the inner loop, the agent learns to complete a single task (via updating the hidden state as more history is accumulated). In the outer loop, the agent is optimized to learn the task faster (via policy gradient optimization). While perhaps somewhat contrived in the context of solely this model, this viewpoint is powerful as a way to look at the problem of transfer learning. We can consider different kinds of inner loop learners, and as long as the learning process is differentiable, it can be optimized by the outer loop. For example, instead of a hidden state update, the inner learner could be gradient descent on model parameters [Finn et al., 2017].

3 Installation

Obtain the starter code from https://github.com/berkeleydeeprlcourse/homework/tree/master/hw5/multi. It is similar to the code you used for HW2, and requires the same dependencies. For this assignment, you will be working with a simple 2-D point mass environment. To train the policy, the basic command with all defaults is

```
python train_policy.py --env_name <environment_name> --exp_name
↪  <experiment_name>
```

4 Implementation

While the code is based on the HW2 code, for better performance the vanilla policy gradient has been replaced with an off-policy actor-critic algorithm proximal policy optimization (PPO) [Schulman et al., 2017]. Take a look at these modifications if you like, but you won't have to know anything about PPO to complete this assignment.

4.1 Problem 1: Context as Task ID

We'll start off by giving the policy a task identifier rather than learning it from experience, in effect telling the policy which task it should perform, rather than forcing it to figure it out from reward signals. Note that we won't be able to generalize to new tasks not seen during training, but this experiment will

validate that we can learn a contextual policy. For this we'll use a simple 2-D point mass environment with four goal locations. At each time step, the agent receives a penalty equal to the negative Euclidean distance from the goal. Fill in the parts of the code labeled "Problem 1" in `point_mass_observed.py`.

4.2 Problem 2: Meta-Learned Context

Now we will learn to infer the task from prior experience and rewards. We will continue to use the point mass environment for this part, but goals will be sampled randomly from a square. You will implement the data collection, which requires you to construct "meta-states" of concatenated (s, a, r, d) tuples, as well as the recurrent "meta-learner." Fill in the parts of the code labeled "Problem 2" in `train_policy.py`.

4.3 Problem 3: Generalization

In the previous problem, testing goals could be arbitrarily close to training goals. Here we evaluate generalization to tasks strictly outside of the training set. Divide the state space into checkerboard pattern, where the alternating colors correspond to alternating train/test goals. Fill in the parts of the code labeled "Problem 3" in `point_mass.py`.

5 Deliverables

Use `plot.py` to plot learning curves. The hyper-parameter defaults should work, but they may not be the best - feel free to modify them.

5.1 Problem 1

Run the command

```
python train_policy.py 'pm-obs' --exp_name <experiment_name>
 ↪  --history 1 -lr 5e-5 -n 100 --num_tasks 4
```

You should get an average return of around -50. Include a plot of the average return in your report.

5.2 Problem 2

Compare the performance of the feed-forward and recurrent architectures for different lengths of history. (For a history length of 60, the recurrent network

should achieve an average return of about -100.) You can modify the history length with the option `--history` and you can switch architectures with the `--recurrent` flag. Be sure to roughly control for the number of model parameters when comparing the two architectures. Include a plot of average return for both architectures for at least three different history lengths. Discuss your results. What minimum history length is needed? Which architecture works better? If you change any hyper-parameters, discuss the result.

5.3 Problem 3

Compare the performance of the policy on training goals and testing goals. Vary the granularity of the checkerboard and comment on the generalization performance as the training and testing distributions become more different. Include plots comparing training and testing average returns for at least two different settings.

5.4 Submission

Turn in both parts of the assignment on Gradescope as one submission. Upload the zip file with your code to **HW5 Code Multi-Task**, and upload the PDF of your report to **HW5 Multi-Task**.

References

Yan Duan, John Schulman, Xi Chen, Peter L Bartlett, Ilya Sutskever, and Pieter Abbeel. Rl ̃2: Fast reinforcement learning via slow reinforcement learning. *arXiv preprint arXiv:1611.02779*, 2016.

Chelsea Finn, Pieter Abbeel, and Sergey Levine. Model-agnostic meta-learning for fast adaptation of deep networks. *arXiv preprint arXiv:1703.03400*, 2017.

Nikhil Mishra, Mostafa Rohaninejad, Xi Chen, and Pieter Abbeel. A simple neural attentive meta-learner. In *International Conference on Learning Representations*, 2018. URL https://openreview.net/forum?id=B1DmUzWAW.

John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

Jane X Wang, Zeb Kurth-Nelson, Dhruva Tirumala, Hubert Soyer, Joel Z Leibo, Remi Munos, Charles Blundell, Dharshan Kumaran, and Matt Botvinick. Learning to reinforcement learn. *arXiv preprint arXiv:1611.05763*, 2016.

Paul J Werbos. Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560, 1990.