Solving RL problems with PPLs Summary

PENGYUAN SHI, WITH GUIDANCE FROM PROFESSOR YIZHOU ZHANG

1 INTRODUCTION

This is the documentation of the URA project in 2020 Fall term. The topic is to solve RL(Reinforcement Learning) using PPLs(Probabilistic Programming Languages). In particular, we worked on the Cart-pole and Mountain-Car problem from OpenAI Gym library, by using Pyro.

2 DOCUMENTATION FOR THE PROJECT

The main structure of our program to solve RL problems involves three parts, the model, guide and inference. The model contains simulation of the trajectory, represents the probability of a trajectory (Levine 2018):

$$p(\tau) = \left[p(s_1) \prod_{t=1}^{T} p(s_{t+1}|s_t, a_t) \right] exp\left(\sum_{t=1}^{T} r(s_t, a_t) \right)$$

The aim of RL is to approximate the optimal policy function $\pi(a_t|s_t)$. We use a neural network $q(a_t|s|t)$ to represent the policy function and use approximate posterior distributions (called guide in Pyro) $q(\tau)$ to approximate the distribution $p(\tau)$ by Structured Variational Inference (SVI). The guide function in our program has form (Levine 2018):

$$q(\tau) = q(s_1) \prod_{t=1}^{T} q(s_{t+1}|s_t, a_t) q(a_t|s_t)$$

It also contains simulation of the trajectory and we need to fix $q(s_1) = p(s_1)$ and $q(s_{t+1}|s_t, a_t) = p(s_{t+1}|s_t, a_t)$ by setting the same initial state and using the same transition function.

3 CARTPOLE

The final version of cartpole PPL model can be found at https://github.com/frankmao666/pyro-nn/blob/master/cartpole_svi_revised.py. In our program, we use a three-layer network to represents the policy function:

```
class Policy(nn.Module):
    def __init__(self):
        super().__init__()
        self.neural_net = nn.Sequential(
            nn.Linear(4, 128),
            nn.ReLU(),
            nn.Linear(128, 1),
            nn.Sigmoid())

    def forward(self, observation):
        prob = self.neural_net(observation)
        return prob
```

Author's address: Pengyuan Shi, with guidance from Professor Yizhou Zhang.

The output for the neural network is a floating point between 0 and 1 that represents the input to Bernoulli distribution.

In the definition of model, it samples a trajectory from the environment and uses **pyro.factor** statement to factor the execution. The trajectory with higher total reward will have a higher factor, i.e.

```
pyro.factor("Episode_{}".format(episode), total_reward * alpha)
```

The guide function is almost identical to the model except it invokes the Policy object (neural network) to output the probability of drawing an action while in model, the probability(input) to Bernoulli distribution that drawing actions is 0.5.

Note that in each inference step, we should make sure the run of guide and model in the same **svi.step** will have the same initial state so they will have the same number of actions drawn from **pyro.sample** primitive. Plus we are using the same transition function, **env.step**, so the implementations of the model and guide are probabilistic equivalent to the two equations shown in section 2.

When doing inference, we will use

```
learning_rate = 8e-4 #1e-5
optimizer = optim.Adam({"lr":learning_rate})
svi = SVI(model, guide, optimizer, loss=Trace_ELBO())
```

Note that each **svi.step** will run guide and model once, and the samples drawn in model are copied from the samples in guide. We will also need to ensure the initial state setting for a guide and model run is random so we can learn a general policy function. That is exactly what **reset_init_state()** is doing.

4 MOUNTAIN-CAR

For Mountain-Car problem, we use a slightly different neural network to represent to policy function and redefine the total reward that the agent will receive from a simulation, which is used in **pyro.factor** statement.

```
class Policy(nn.Module):
    def __init__(self):
        super().__init__()
        self.neural_net = nn.Sequential(
            nn.Linear(2, 24),
            nn.ReLU(),
            nn.Linear(24, 24),
            nn.ReLU(),
            nn.Linear(24, 3),
            nn.Softmax())

    def forward(self, observation):
        prob = self.neural_net(observation)
        return prob
```

Since it is hard for the agent to solve the problem by taking naive actions at the beginning, we need to define an alternative reward system if the problem has not been solved. First, if the problem is solved successfully, the total reward is defined by $[MaxReward - timeTakeToSolve]$, so if the agent

solves the problem with more time, it will receive less reward. However, if the problem is not solved within a specific time duration, the total reward is defined by $[|FinalPosition + TargetPosition|]$, so if the agent is closer to the target position, it will receive more rewards, but the amount of rewards is much lower than it gets when solving the problem.

The full implementation can be accessed from https://github.com/frankmao666/pyro-nn/blob/master/mountain-car-svi.py.

5 RESULTS

For Cartpole, with setting $num\_steps = 3000$ which indicates the inference will do 3000 gradient steps, and each one will draw a sample trajectory, the program solves the Cartpole problem with an average score 925 in 100 testing episodes (i.e. survive for 925 time steps, higher is better). The processing time is around 11 minutes without using a GPU. Note that we have set $MAXTIME = 1000$, setting it to a lower value (eg. 500) can decrease the processing time.

For Mountain-Car, with setting $num\_steps = 30000$, the program can solve the Mountain-Car problem with an average testing score 114 in 100 testing episodes (i.e. using 114 time steps to solve the problem, lower is better). The processing time is around 53 minutes without using a GPU.

6 IMPROVEMENT

Improvements can still be addressed by tuning hyper-parameters such as the learning rate and neural network to accelerate inference and improve performance. Additionally, Pyro baselines can possibly be applied to reduce the variance.

7 REFERENCE

Levine, S. (2018). Reinforcement Learning and Control as Probabilistic Inference: Tutorial and Review.