Learn to Play CartPole with PyTorch using PGPE

Code: *agent.py*, *cartpole.py*

See the **final result** on my OpenAI Gym profile

See the **sample code** on my Github

INTRODUCTION

In this post, we will apply another policy-based algorithm to solve the CartPole environment. Unlike REINFORCE, searching the optimal policy in the action space, policy gradient with parameter-based explorations(PGPE) searches the optimal policy in the parameter space. The idea is to equip each model parameter with a *prior distribution* to sample from. At the beginning of each roll-out, we sample a set of model parameters, and follow a *deterministic* controller(policy) for the episode thereafter. Just like REINFORCE, the original PGPE adopts an episodic setting. After an episode finishes, we compute the gradient of the objective function w.r.t. *hyper-parameters of the prior distribution* and update, again, the hyper-parameters.

ALGORITHM

Note: Some notations have been modified to better fit in the context, hence you may find some inconsistency in notations with the original paper.

1. Each action follows the *deterministic policy* (the δ term) with model parameters δ drawn from the *prior distribution*, parametrized by ρ

   $$p(a|s,\rho) = \int_{\Theta} p(\theta|\rho)\delta_{F_\theta(s)=a}d\theta$$

   which means the prbobability of taking action a under state s is exactly the probability of drawing parameter θ given hyper-parameter ρ, since the delta function(our deterministic controller) contributes no stochasticity in the equation.

2. The objective $J(\rho)$ is defined to be

   $$J(\rho) = \int_{\Theta}\int_{H} p(h,\theta|\rho)r(h)dhd\theta$$

   which is the *expected return* taken over all possible parameters we can draw from the prior and the history generated by following policy $p(h,\theta)$.

3. Apply the classic log-likelihood trick, we obtain the gradient of the objective w.r.t. hyper-parameter ρ

   $$\nabla_\rho J(\rho) = \int_{\Theta}\int_{H} p(h,\theta|\rho)\nabla_\rho \log p(h,\theta|\rho)r(h)dhd\theta$$

4. Observe that the history is conditionally independent of ρ given θ, that is, $p(h, \theta|\rho) = p(h|\rho)p(\theta|\rho)$ and the partial derivative reduces, $\nabla_\rho \log p(h, \theta|\rho) = \nabla_\rho \log p(\theta|\rho)$. Combining all of the above, the gradient estimator reveals:

$$\nabla_\rho J(\rho) \approx \frac{1}{N} \sum_{n=1}^{N} \nabla_\rho \log p(\theta^n|\rho) r(h^n)$$

   Remark the above gradient estimator is completely determined by the agent's model paremters *without* the knowledge of environment dynamics. Information concerning environment dynamics encapsulates in the term $p(h|\theta)$, as we get rid of it when taking the log value of the term and differentiate w.r.t. ρ. Hence PGPE is a *model-free* algorithm, just as REINFORCE.

5. Here we assume each model parameter θ_i follows a normal distribution independent with one another. Hence the hyper-parameters ρ consists of a sequence of means and standard deviations $\rho = ((\mu_i, \sigma_i)_i)$. Direct computation gives

$$\nabla_\mu \log p(\theta|\rho) = \frac{\theta - \mu}{\sigma^2}, \quad \nabla_\sigma \log p(\theta|\rho) = \frac{(\theta - \mu)^2 - \sigma^2}{\sigma^3}$$

   Finally, we choose a stepsize proportional to the variance, $\alpha_\rho = \alpha\sigma^2$. The stepsize may be different for different parameters as we will see below.

$$\Delta\mu = \alpha_\mu (r - b)(\theta - \mu), \quad \Delta\sigma = \alpha_\sigma (r - b) \frac{(\theta - \mu)^2 - \sigma^2}{\sigma}$$

   where b is the reward baseline for variance reduction purpose.

CODE

For brevity, I only explain the code snippet I think may raise difficulty to the reader and omit most part of it. Full code can be found in my GitHub.

cartpole.py

```python
#...

# Train for 200 episodes
for ep in range(200):
    observation = env.reset()

    while True:
        # The typical RL env-agent paradigm
        action = RL.choose_action(observation)
        observation_, reward, done, info = env.step(action)

        RL.store_reward(reward)

        if done:
            # Learn in an episodic basis
            vt = RL.learn_and_sample()
            break

        observation = observation_

# Close env.
env.close()
```

agent.py

```python
#...

class PGPE:
    def __init__(self, n_actions, n_features):
        #...

        # Prepare hyper-params and initialize model params
        self.Param = list(self.model.parameters())
        self.Mu = []
        self.Sigma = []
        for p in self.Param:
            # initialize hyper-params
            self.Mu.append(torch.normal(torch.zeros(p.size()), torch.ones(p.size())))
            self.Sigma.append(2 * torch.ones(p.size()))

            # Sample initial model params
            p.data = torch.normal(self.Mu[-1], self.Sigma[-1])

    def choose_action(self, obs):
        # Scale input to (-1, 1):
        #   1. if range is finite -> divide by range
        #   2. if range is infinite -> take tanh
        obs[0] /= 2.4
```

```
        obs[1] = np.tanh(obs[1])
        obs[2] /= 41.8
        obs[3] = np.tanh(obs[3])

        #...

        # Deterministic policy, pick action with greater value
        action = a[0].argmax()
        return action

    #...

    def learn_and_sample(self):
        # Scale return to [0, 1]
        _r = self.ret / 200

        # reset return tracker
        self.ret = 0.

        for i in range(len(self.Param)):
            # Learning
            # These are the T and S matrices in the original paper
            _T = self.Param[i].data - self.Mu[i]
            _S = (_T ** 2 - self.Sigma[i] ** 2) / self.Sigma[i]

            # Update means
            _delta_Mu = self.Mu_lr * _r * _T
            self.Mu[i] += _delta_Mu

            # Update standard deviations
            _delta_Sigma = self.Sigma_lr * _r * _S
            self.Sigma[i] += _delta_Sigma

            # Freeze params if hit target reward, else re-sample
            if _r < 1.:
                self.Param[i].data = torch.normal(self.Mu[i], self.Sigma[i])
```

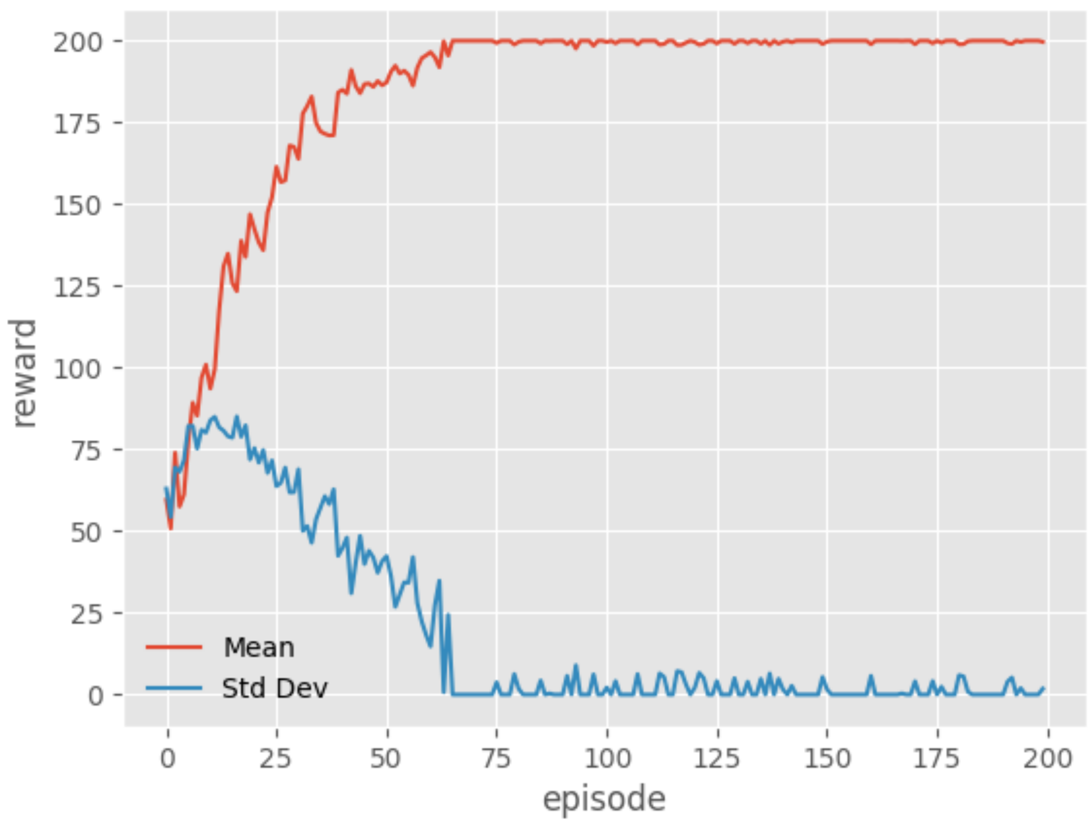
RESULT

The result of this algorithm can be viewed on my OpenAI Gym page. I also provide a gist link there so you can easiliy reproduce the result. As you can see, the algorithm converges quickly, taking only 5 episodes to solve the CartPole environment. The model consists of only 8 parameters and without performing any sophiscated initialization trick (in fact, random initialization from normal(0, 2) is used to draw initial model parameters). Other results provided on the OpenAI Gym cartpole environment page utilizing the REINFORCE algo often take up dozens more episodes to converge to a stable policy.

By the way, in case anyone is interested in what the input states are in the cartpole env. Here is the official OpenAI gym github page providing the background information. Note the env setting is a bit different from the one on the wiki page. But it doesn't really matter since we scale the input state when it comes in.

ROBUSTNESS

To further evaluate our algorithm, we roll out 30 trials, each with 200 episodes. Then compute the mean and standard deviation of reward **at each episode** over these 30 trials to get a sense of how much reward in average we can get for each episode. The plot demonstrates the result:



To interpret, say, at the 50th episdoe, the algorithm gets an average reward of 187.5 and reward standard deviation being 37.5. The plot demonstrates the desired outcome: **mean reward grows steadily toward 200 and the standard deviation decreases to 0 as the algorithm learns to prevent the pole from falling for 200 steps.** Also, we can see the algorithm *solves* the envir. at around 63th step, where it achieves an average reward of 195.0 over 100 consecutive episodes thereafter.