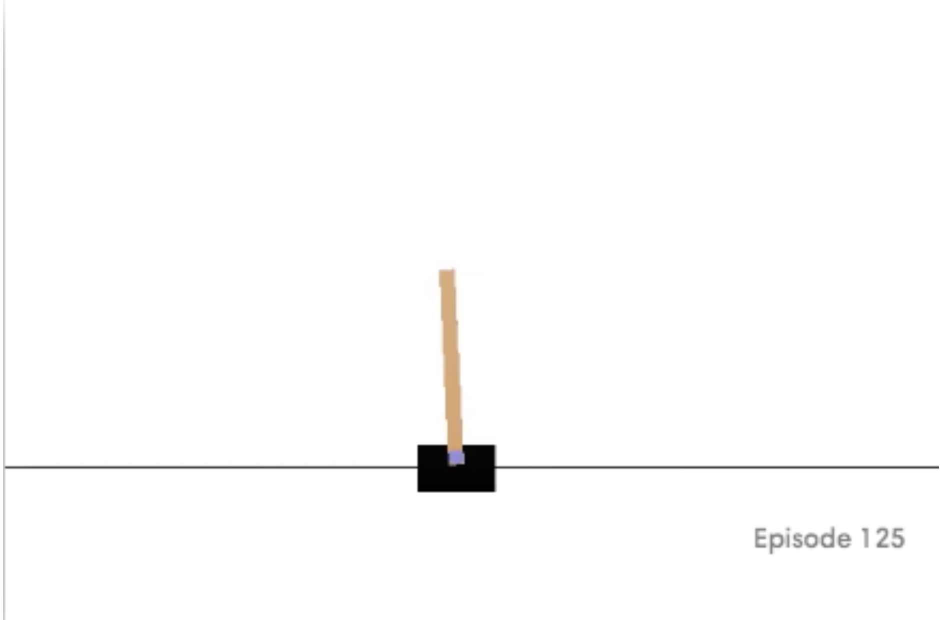
Learn to Play CartPole with PyTorch using PGPE

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

The CartPole enviroment:



"Keep the pole from falling for a short period of time", that is the task our algorithm will learn to perform in this post.

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

See the **sample code** on my Github page

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 do the parameter update.

ALGORITHM

Note:

- Typing LaTeX inline is not supported by my Markdown editor, so some of the equations may be a little difficult to read, refer to the original paper for details.

- 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.

Algorithm 1 The PGPE Algorithm without reward normalization: Left side shows the basic version, right side shows the version with symmetric sampling. $\boldsymbol{T}$ and $\boldsymbol{S}$ are $P \times N$ matrices with P the number of parameters and N the number of histories. The baseline is updated accordingly after each step. α is the learning rate or step size.

Initialize $\boldsymbol{\mu}$ to $\boldsymbol{\mu}_{\text{init}}$	Initialize $\boldsymbol{\mu}$ to $\boldsymbol{\mu}_{\text{init}}$
Initialize $\boldsymbol{\sigma}$ to $\boldsymbol{\sigma}_{\text{init}}$	Initialize $\boldsymbol{\sigma}$ to $\boldsymbol{\sigma}_{\text{init}}$
while TRUE **do**	**while** TRUE **do**
for $n = 1$ to N **do**	**for** $n = 1$ to N **do**
draw $\boldsymbol{\theta}^n \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{I}\boldsymbol{\sigma}^2)$	draw perturbation $\boldsymbol{\epsilon}^n \sim \mathcal{N}(\boldsymbol{0}, \boldsymbol{I}\boldsymbol{\sigma}^2)$
	$\boldsymbol{\theta}^{+,n} = \boldsymbol{\mu} + \boldsymbol{\epsilon}^n$
	$\boldsymbol{\theta}^{-,n} = \boldsymbol{\mu} - \boldsymbol{\epsilon}^n$
evaluate $r^n = r(h(\boldsymbol{\theta}^n))$	evaluate $r^{+,n} = r(h(\boldsymbol{\theta}^{+,n}))$
	evaluate $r^{-,n} = r(h(\boldsymbol{\theta}^{-,n}))$
end for	**end for**
$\boldsymbol{T} = [t_{ij}]_{ij}$ with $t_{ij} := (\theta_i^j - \mu_i)$	$\boldsymbol{T} = [t_{ij}]_{ij}$ with $t_{ij} := \epsilon_i^j$
$\boldsymbol{S} = [s_{ij}]_{ij}$ with $s_{ij} := \frac{t_{ij}^2 - \sigma_i^2}{\sigma_i}$	$\boldsymbol{S} = [s_{ij}]_{ij}$ with $s_{ij} := \frac{(\epsilon_i^j)^2 - \sigma_i^2}{\sigma_i}$
$\boldsymbol{r} = [(r^1 - b), \ldots, (r^N - b)]^{\mathrm{T}}$	$\boldsymbol{r}_T = [(r^{+,1} - r^{-,1}), \ldots, (r^{+,N} - r^{-,N})]^{\mathrm{T}}$
	$\boldsymbol{r}_S = [(\frac{r^{+,1} + r^{-,1}}{2} - b), \ldots, (\frac{r^{+,N} + r^{-,N}}{2} - b)]^{\mathrm{T}}$
update $\boldsymbol{\mu} = \boldsymbol{\mu} + \alpha \boldsymbol{T}\boldsymbol{r}$	update $\boldsymbol{\mu} = \boldsymbol{\mu} + \alpha \boldsymbol{T}\boldsymbol{r}_T$
update $\boldsymbol{\sigma} = \boldsymbol{\sigma} + \alpha \boldsymbol{S}\boldsymbol{r}$	update $\boldsymbol{\sigma} = \boldsymbol{\sigma} + \alpha \boldsymbol{S}\boldsymbol{r}_S$
update baseline b accordingly	update baseline b accordingly
end while	**end while**

The algorithm taken from the original paper is shown above. In this post we will implement the one on the left-hand side (the vanilla version). But as we will see later, this simple algorithm turned out to be very effective against benchmark control problems compared with to standard PG methods such as REINFORCE.

CODE

cartpole.py

```python
import os.path
import sys
import gym
from gym import wrappers
from agent import PGPE

env = gym.make('CartPole-v0')

# Gym's built-in monitor functionality
# We can later upload our result to OpenAI Gym
mod_path = os.path.dirname(os.path.abspath(sys.argv[0]))
save_path = os.path.join(mod_path, 'cartpole_experiment_1')

env = wrappers.Monitor(env, save_path, force=True)

# Create an agent instance
RL = PGPE(
    n_features=env.observation_space.shape[0],
    n_actions=env.action_space.n
)


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

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

        RL.store_reward(reward)
        r = RL.get_return()

        if done:
            print("Episode:", ep, "  Reward:", int(r))
            vt = RL.learn_and_sample()
            break

        observation = observation_

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

agent.py

```python
import numpy as np

```

```
import torch
import torch.nn as nn
from torch.autograd import Variable


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

        # Return tracker for episode
        self.ret = 0.

        self.model = nn.Sequential(
            nn.Linear(self.n_features, self.n_actions, bias=False),
            nn.Softmax()  # deterministic policy, pick action with greater value
        )

        # Learning rate for hyper-params mu and sigma
        self.Mu_lr = 0.2
        self.Sigma_lr = 0.1

        # Prepare hyper-params, store mean/var separately in lists
        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])

        # cast np array to torch variable
        s = Variable(torch.from_numpy(obs.astype(np.float32))).unsqueeze(0)

```

```
        a = self.model.forward(s).data.numpy()
        action = a[0].argmax()  # pick action with greater value
        return action

    def get_return(self):
        return self.ret

    def store_reward(self, r):
        self.ret += r  # Compute (un-discounted) return

    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])
```

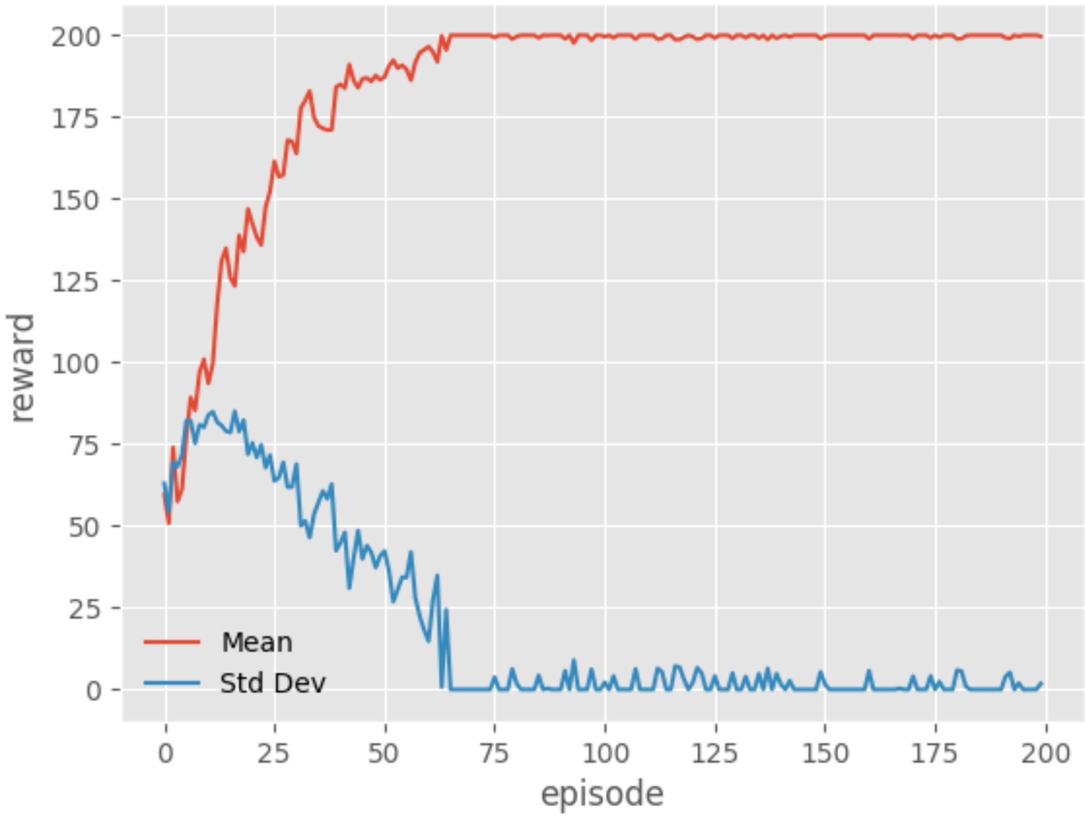
RESULT

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 of 187.5 and a standard deviation of reward of 37.5. The plot demonstrates the desired result: mean reward grows steadily toward 200 and the standard deviation decreases to 0 as the algorithm learns to prevent the pole from falling down 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.