Quan Vuong (quan.hovuong@gmail.com)
PPO implementation details of the official implementation by OpenAI (for mujoco).

PPO Algorithm

Algorithm 1 Parameter Update for PPO

Require: A neural net π_θ that parameterizes the policy.
Require: A neural net V_ϕ that approximates V^{π_θ}.
Require: Hyperparameter: N (number of trajectories), T (length of each trajectory).

1: **for** k = 1, 2, ... **do**
2: under policy π_{θ_k}, sample N trajectories, each of size T $(s_{it}, a_{it}, r_{it}, b_{it}, s_{i(t+1)}), \quad i = 1, \dots, N, t = 1, \dots, T$
#Calculate Advantage Value and Target for V_ϕ
3: $\delta_{it} \leftarrow r_{it} + \gamma b_{it} V_{\phi_k}(s_{i(t+1)}) - V_{\phi_k}(s_{it}), \quad i = 1, \dots, N, t = 1, \dots, T$
4: $A_{it} \leftarrow \delta_{it} + \sum_{l=1}^{T-t} (\gamma\lambda)^l \delta_{i(t+l)} \prod_{j=t}^{t+l-1} b_j, \quad i = 1, \dots, N, t = 1, \dots, T$
5: $\mu_{A_i} \leftarrow \frac{1}{T} \sum_t A_{it}, \quad i = 1, \dots, N$
6: $std_{A_i} \leftarrow \sqrt{\frac{1}{T} \sum_t (A_{it} - \mu_{A_i})^2}, \quad i = 1, \dots, N$
7: $A_{it} \leftarrow \frac{A_{it} - \mu_{A_i}}{std_{A_i}}, \quad i = 1, \dots, N, t = 1, \dots, T$
8: **for** i = 1, ..., N **do**
9: $V^{targ}(s_{iT}) \leftarrow r_{iT} + b_{iT} \gamma V_{\phi_k}(s_{i(T+1)})$
10: **for** t = T-1, ..., 1 **do**
11: $V^{targ}(s_{it}) \leftarrow r_{it} + b_{it} \left[\gamma(1-\lambda) V_{\phi_k}(s_{i(t+1)}) + \gamma\lambda V^{targ}(s_{i(t+1)})\right]$
12: **end for**
13: **end for**
#Finish Calculating Advantage Value and State Value
#Update parameters
14: $L_i(\phi_k) = \frac{1}{T} \sum_t (V^{targ}(s_{it}) - V_{\phi_k}(s_{it}))^2, \quad i = 1, \dots, N$
15: $\phi_{k+1} \leftarrow \phi_k - \frac{1}{N} \sum_i \nabla_\phi L_i(\phi_k)$
16: $L_i(\theta_k) = \frac{1}{T} \sum_t \min(\frac{\pi_{\theta_k}(a_{it}|s_{it})}{\pi_{\theta_k}(a_{it}|s_{it})} A_{it}, clip(\frac{\pi_{\theta_k}(a_{it}|s_{it})}{\pi_{\theta_k}(a_{it}|s_{it})}, 1-\epsilon, 1+\epsilon) A_{it}), \quad i = 1, \dots, N$
17: $\theta_{k+1} \leftarrow \theta_k + \frac{1}{N} \sum_i \nabla_\theta L_i(\theta_k)$
#Finish update parameters
18: **end for**

$b_{it} = 0$ **if** s_{it} **is a terminal state.** $b_{it} = 1$ **otherwise.**

For step 9 to 12, the code uses minibatch SGD. I left that out here for simplicity purposes. Note that for states $s_{i(T+1)}, i = 1, \dots, N$, the only quantity wrt these states that we need to calculate is $V_\phi(s_{i(T+1)})$. This can be obtained by plugging the states into V_ϕ. Note that Algorithm 1 assumes that the timesteps in each trajectory belong to the same episode. If not, the algorithmic description will be more involved. In practice, the timesteps within each trajectory can come from different episodes.

Summary:

0) After running π_{θ_k} for N trajectories, the algorithm has two main parts. In the first part, it obtains advantage estimates A_{it} and critic targets for π_{θ_k}. In the second part, it uses the critic targets to update the critic, and uses the advantage estimates to update the actor.

1) More specifically, after running π_{θ_k} for N trajectories, the algorithm calculates advantage estimates A_{it} using the data from the N trajectories and the critic network ϕ_k, which critiques θ_k. This is a little different in what is in Levine's actor-critic lecture on GAE. There Levine seems to imply we would base the estimates on a critic network ϕ_k that is obtained by solving a regression problem with the data generated from π_{θ_k}.

2) After obtaining the advantage estimates A_{it}, these estimates are normalized for each trajectory separately.

3) The algorithm then obtains targets for the critic network ϕ_k. These targets are obtained by using the accumulated returns

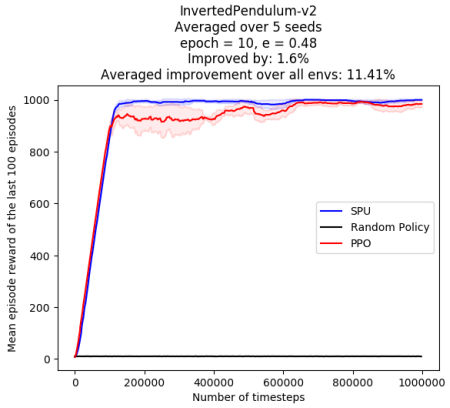
obtained from policy π_{θ_k}, and also the critic network ϕ_k. The two approaches are weighted by λ and $1-\lambda$.

4) We are now in position to obtain ϕ_{k+1} and θ_{k+1}. For ϕ_{k+1}, this done by solving a regression problem using the targets. For θ_{k+1}, we do the PPO update (generalized PG) using the advantage estimates A_{it} for π_{θ_k}.

Performance Calculation

If we look at the performance graph below, the red line indicates at each timestep (x-axis) the performance (y-axis) of PPO averaged over 5 seeds. For each seed, the performance at a specific timestep is the average episode reward over the last 100 episodes. An episode reward is the sum of the undiscounted timestep rewards, i.e. Monte Carlo returns. The purpose of this section is to explain the ideas in the last 3 sentences.

We can also express in math notation how to obtain the y values in the graph below. Let the performance of PPO with seed c at timestep x be P_{cx}. Thus, at timestep x, the corresponding y-values is $\frac{\sum_c P_{cx}}{5}$. Note that in the graph below, the red line is continuous. But in reality, we obtain discrete (x, y) data points. This is because policy evaluation is only performed after a fixed number of timesteps. For example, if we evaluate the policy after every trajectory, each trajectory of length T, then we have the y-values for $x = T, 2T, \ldots$. Assuming x divides T, Algorithm 2 shows how P_{cx} is calculated. For values on the x-axis that do not divide T, we draw a straight line between the two neighboring P values. This is done to obtain a continuous line graph from discrete data points.



Algorithm 2 Performance Calculation

Require: A neural net π_θ that parameterizes the policy.
Require: Hyperparameter T (length of each trajectory).
Require: A seed c.
Require: A First-In First-Out queue B whose maximum size is 100.
Require: Variable R to keep track of episode reward. Initialized to 0.

for k=1, 2, ... **do**
  #Determine how to obtain the starting state of the trajectory
  if $k = 1$ or $s_{(k-1)(T+1)}$ is a terminal state **then**
    $s_{k1} \leftarrow$ new randomized starting state ▷ explained under **Randomized Initial State Selection**
  else
    $s_{k1} \leftarrow s_{(k-1)(T+1)}$
  end if
  #Finished obtaining the starting state of the trajectory
  #Generate the trajectory
  for i=1, ..., T **do**
    $a_{ki} \sim \pi_{\theta_k}(.|s_{ki})$
    Obtain $r_{ki}, s_{k(i+1)}$ by taking action a_{ki}
    $R \leftarrow R + r_{ki}$
    if $s_{k(i+1)}$ is a terminal state **then**
      Add R to B
      $R \leftarrow 0$
      $s_{k(i+1)} \leftarrow$ new randomized starting state
    end if
  end for
  #Finished generating the trajectory
  $P_{c(k*T)} = mean(B)$
  Obtain θ_{k+1} from θ_k by running your favorite algorithm with trajectory $\{s_{k1}, a_{k1}, r_{k1}, \ldots, s_{kT}, a_{kT}, r_{kT}, s_{k(T+1)}\}$
end for
Outputs: the values $P_{c(k*T)}$ corresponding to the values on the x-axis $k * T, k = 1, 2, \ldots$ for seed c =0

standard deviation) before we plug them into backprop. This way we're always encouraging and discouraging roughly half of the performed actions. Mathematically you can also interpret these tricks as a way of controlling the variance of the policy gradient estimator."

This paragraph is under the section **More general advantage functions**. He provides a link that he says provide more in-depth explanation, but that link doesn't actually discuss this. Actually, we don't know how important advantage value normalization is. We have never tried running PPO without it. I can try it at some point after we're done with our experiments.

Why not fit ϕ_k to get ϕ_{k+1} and then use ϕ_{k+1} to obtain advantage value estimates for π_k

In the GAE paper where Schulman introduces this kind of advantage estimation, the authors explain their reasoning (copied and pasted below):

"Note that the policy update θ_i to θ_{i+1} is performed using the value function V_ϕ for advantage estimation, not $V_{\phi_{i+1}}$. Additional bias would have been introduced if we updated the value function first. To see this, consider the extreme case where we overfit the value function, and the Bellman residual $r_t + \gamma V(s_{t+1}) - V(s_t)$ becomes zero at all timesteps -the policy gradient estimate would be zero."

This paragraph is at the end of section 6.1

Action Sampling

At each timestep, the action is a vector with continuous entries. Let $a_t = [a_{t_1}, \ldots, a_{t_D}]$, then the output of the last layer of the policy network is $[\mu_1, \ldots, \mu_D]$. Then $a_{t_k} \sim \mathcal{N}(\mu_k, std_k)$. std_k is a trainable parameter of the policy network whose value does not change when the input to the network changes.

Calculation of probability of action

In PPO for mujoco, the action dimensions are assumed to be independent with each sampled from a normal distribution. Thus action has a multivariate normal distribution with diagonal covariance matrix.

OpenAI implementation gets the probability of an action by first computing its negative log probability. Below, D is the size of the action dimension.

$$neglogp = 0.5 \sum_{i=1}^{D} \frac{a_i - \mu_i}{\sigma_i} + 0.5D \log(2\pi) + \sum_{i=1}^{D} log(\sigma_i)$$

$$\Rightarrow logp = -0.5 \sum_{i=1}^{D} \frac{a_i - \mu_i}{\sigma_i} - 0.5D \log(2\pi) - \sum_{i=1}^{D} log(\sigma_i)$$

$$\Rightarrow p = exp\{-0.5 \sum_{i=1}^{D} \frac{a_i - \mu_i}{\sigma_i} - 0.5D \log(2\pi) - \sum_{i=1}^{D} log(\sigma_i)\}$$

We have:

$$e^{-0.5D \log(2\pi)} = e^{log(2\pi)^{\frac{-D}{2}}} = \frac{1}{(2\pi)^{\frac{D}{2}}}$$

$$e^{-\sum_{i=1}^{D} log(\sigma_i)} = e^{-\log(\prod_{i=1}^{D} \sigma_i)} = (\prod_{i=1}^{D} \sigma_i)^{-1} = \frac{1}{(\prod_{i=1}^{D} \sigma_i)} = \frac{1}{|\Sigma|^{\frac{1}{2}}}$$

$$\Rightarrow p = \frac{1}{(2\pi)^{\frac{D}{2}} |\Sigma|^{\frac{1}{2}}} exp\{-\frac{1}{2} \sum_{i=1}^{D} \frac{a_i - \mu_i}{\sigma_i}\}$$

which is the pdf of the distribution.

Randomized Initial State Selection

Every mujoco environment has a hard-coded initial state. At the beginning of each episode, the randomized initial state = the hard-coded state + noise. How the noise is generated differs between the different mujoco environments. To make this concrete, let us look at the pendulum environment.

In the pendulum environment, the state is a continuous vectors of size 4 [position of first joint, position of second joint, velocity of
The hard-coded initial state is $[0, 0, 0, 0]$. Each new randomized initial state is thus: $[0, 0, 0, 0] + [noise_1, noise_2, noise_3, noise_4]$ where:

$noise_i \sim Uniform(-0.01, 0.01)$

$+$ denotes element-wise addition.

Mini-batches

In algorithm 1 in the paper, before the policy is updated, we collect NT data points (from NT timesteps). The algorithm then "Optimize surrogate L wrt θ, with K epochs and minibatch size M $\leq$ NT". What the sentence in quote does is below:

Algorithm 3 PPO Minibatch

Require: NT data points $\{s_i, a_i, \pi_{\theta_k}(a_i|s_i), V^{targ}(s_i), A_i\}, \quad i = 1, \ldots, NT$

1: **for** epoch $= 1, 2, ...,$ K **do**
2: shuffle NT data points
3: #After shuffling, there are still NT data points but their orders are randomized
4: **for** $m = 1, 2, ..., NT/M$ **do**
5: From NT data points, sample without replacement a mini-batch $B \leftarrow \{s_j, a_j, \pi_{\theta_k}(a_j|s_j), V^{targ}(s_j), A_j\}, \quad j = 1, \ldots, M$
6: $L(\phi) \leftarrow \frac{1}{M}\sum_j (V^{targ}(s_j) - V_\phi(s_j))^2$
7: $\phi \leftarrow \phi - \alpha \nabla_\phi L(\phi)$
8: $L(\theta) \leftarrow \frac{1}{M}\sum_j \min(\frac{\pi_\theta(a_j|s_j)}{\pi_{\theta_k}(a_j|s_j)} A_j, clip(\frac{\pi_\theta(a_j|s_j)}{\pi_{\theta_k}(a_j|s_j)}, 1-\epsilon, 1+\epsilon) A_j)$
9: $\theta \leftarrow \theta + \alpha \nabla_\theta L(\theta)$
10: **end for**
11: **end for**

The sampling on line 5 is sequentially across mini-batch, i.e. if $m = 1$, we take the first M data points out of NT, if $m = 2$, we take the next M data points, etc. This is equivalent to if we were to:

- Not shuffle the NT data points on line 2.

- Sample uniformly without replacement on line 5.

But the way it's done in Algorithm 3 is easier to implement in practice.

States Normalization

Before being fed to the policy network, states are normalized by subtracting the running mean and dividing by the running std.

The rest of the document explains how I arrived at my conclusion by looking at the code execution path. You can stop reading here if you are only interested in the implementation details and not in how I arrived at my conclusions.

Action Sampling

During training, rollouts are generated by the function *traj_segment_generator* invoked here, which invoke the policy network here, which invoke the tf entry point created here. The entry point receives a state and a boolean indicating whether the policy should be stochastic. As indicated on this line, if stochastic is true, then the action is sampled from the action distribution, otherwise the action is the mode. The action distribution is created on this line with 2 inputs. The first is *pdtype*, which is created here and specified the type of distribution. For mujoco, the action distribution is of type *DiagGaussian*. The second is the params of the distribution, which are created here. As we can see, the means of the actions are the output of the last layer of the policy network and the *logstd* is initialized to 0. Finally, this line indicates the fact that $a_{t_k} \sim \mathcal{N}(\mu_k, 1)$.

Mini-batches

The update loop for the policy parameter starts here. Access to the NT data points is abstracted away with a *Dataset* class. The class's *iterate_once* function is a Python generator, which upon being called the first time, shuffles the NT data points and returns M data points. Subsequent calls to the generator return the next M data points without reshuffling the NT data points.

Advantage Value Calculation

here is where the advantage values are used to compute the surrogate objective L. This is the function that triggers the tf computation graph to get the gradient of L. This function receives the advantage values as the argument *atarg* and is invoked here. The advantage values *atarg* are computed here by *add_vtarg_and_adv*. The function *add_vtarg_and_adv* receives T as an argument. So it's also important to understand how s, a, r are obtained here.