Proximal Policy Optimization Demo

Rishikesh Vaishnav

July 12, 2018

Code

- The code for this project is available at: `https://github.com/rish987/Reinforcement-Learning/blob/master/demos/ppo/code/ppo.py`.

Implementation Details

Pseudocode:

- Initialize policy parameter θ.
- Iterate until convergence:
  - Initialize/clear list S of $\{G, \pi_\theta(s,a), s, a\}$.
  - Generate N_τ trajectories $\{\tau\}$, saving $\pi_\theta(s,a)$ for each s, a encountered.
  - For each trajectory $\tau \in \{\tau\}$:
    - For each $\{s,a\} \in \tau$:
      - Calculate discounted return $G_\theta(s,a)$ from this time to end of episode.
      - Retrieve $\pi_\theta(a|s)$ at this (s,a).
      - Store $(G, \pi_\theta(a|s), s, a)$ in S.
  - Use automatic differentiation library to calculate the gradient $\nabla_\theta L^{CLIP}(\theta)$ of:

$$L^{CLIP}(\theta) = \frac{1}{|S|} \sum_{(G,\pi_{\theta_{old}}(a|s),s,a)\in S} min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}G, clip\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}, 1-\epsilon, 1+\epsilon\right)G\right)$$

  - $\theta = \theta + \alpha \nabla_\theta L^{CLIP}(\theta)$.

Parameter Settings:

- $\epsilon = 0.2$ (as per Schulman et. al.)
- $\gamma = 1$ (adjusted to maximize empirical performance)

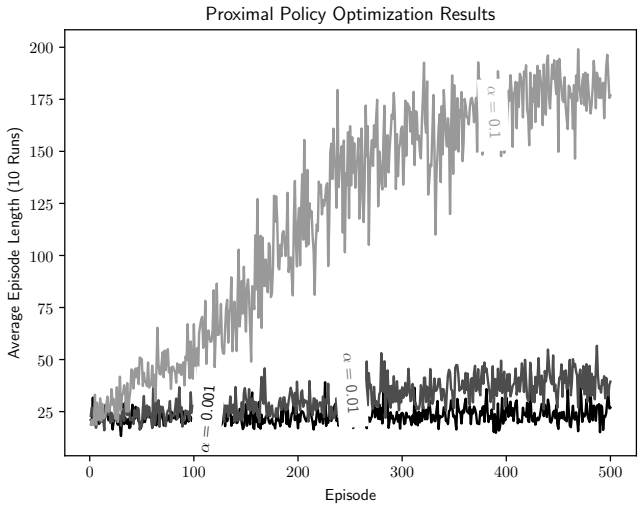
Policy Function Encoding:

- Each state-action pair is converted to the feature vector $x(s, a)$. Letting S_{obs} and S_{act} be the size of the observation and action spaces, respectively, the size of the vector is $S_{obs} \times S_{act}$, where all features are 0 except for the S_{obs} features starting at index $S_{obs} \times a$, which are set to the environment's parameterization of s.
  - In this case, $S_{obs} = 4$ and $S_{act} = 2$.
- The policy function $\pi(a|s, \theta)$ performs the softmax on a parameterized linear mapping of feature vectors:

$$\pi(a|s, \theta) = \frac{e^{\theta^T x(s,a)}}{\sum_b e^{\theta^T x(s,b)}}$$

- Gradient calculation was performed on the objective using the autograd library. Various constant learning rates were tested.

Results



- Clearly, the learning rate had a significant effect on performance. Smaller learning rates were prohibitive, but after the learning rate passed a certain threshold, the problem became feasable.
- It is important to note that the sample complexity is far inferior to TRPO. This may reflect a poor choice in hyperparameters more than an inherent weakness in the algorithm. However, because the update at each iteration was very simple, the code ran significantly faster.
- Possible areas of improvement include a dynamic learning rate, more effective trajectory per iteration count, and a different epsilon.