CSE599G: Deep Reinforcement Learning

Homework 1: model-free policy gradient methods

Due: April 23, 2018

In this homework, the goal is to understand policy gradient methods and use them to solve some simple continuois control tasks. You should have the code infrastructure (mujoco, python wrapper, starter code) set-up before attempting this homework. If you have difficulties in this stage, contact the instructors during office hours. In the lectures, we discussed the REINFORCE algorithm (aka vanilla policy gradient or VPG) and some extensions like NPG, TRPO, PPO etc. We will compare some of these algorithms and also study some variance reduction techniques.

General instructions on experiments and submission

We have provided starter code for VPG and a linear baseline for variance reduction. We will be using the provided `Swimmer`, `HalfCheetah`, and `Ant` environments for this homework.

- We will be overall considering a budget of 500 trajectories (which is $500 \times 500 = 0.25$ million timesteps). How many policy updates and how to split the trajectory budget across the policy updates should be construed as a hyperparameter. One reasonable starting point is 10 trajectories per update and 50 updates.
- You should run all the problems below for 3 random seeds and report the average (for conferences, it is recommended to run close to 10 seeds).
- If you are trying multiple hyperparameter combinations, you should provide one plot per environment per question. For example, if you tried 5 hyperparameter combinations for the warmup problem below, you should have 3 plots (one each for `Swimmer`, `HalfCheetah`, and `Ant`), each containing 5 training curves. Label the plots with the hyperparameters.
- For the learning curves, you should plot the average return obtained with the `evaluation rollouts` performed with the mean policy (of the gaussian policy). This score is also logged with the starter code we provided and has the key name `eval_score` in the `log.pickle` file.

1 Warmup problem

1. *(*3 points) The first task is to run the provided VPG code on the three environments. You should play around with the following hyperparameter settings: (a) number of trajectories per policy update; (b) learning rate for the optimization step. Were you able to get reasonable results? Report your findings by providing the learning curves for the top 3 hyperparameter settings you found for each environment (one plot per environment, each plot containing 3 curves corresponding to 3 different hyperparameter settings).
(Note: It is fine if you are not able to get good results with this vanilla approach, just provide an honest summary of the findings.)

2 VPG with adaptive step size

1. *(*3 points) Now, we will perform a line-search to find a more reasonable step size. We will perform the linesearch on the KL-divergence. Recall that the policy gradient update has the following form:

$$\theta_{k+1} = \theta_k + \alpha g_{vpg} \tag{1}$$

i.e. we have the search direction to be the VPG direction, g_{vpg}, and a step-size (learning rate) of α. Let us pick a desired KL-divergence value, say $\bar{\delta}$. Now, we will consider initially a very large value of α, using which we can compute a "candidate" new parameters: $\hat{\theta}_{k+1} = \theta_k + \alpha g_{vpg}$. With this candidate new policy, we can compute the KL divergence between the successive policies:

$$\delta = KL\left(\pi_{\theta_k} || \pi_{\hat{\theta}_{k+1}}\right) \tag{2}$$

At each iteration, we shall refine our guess for α based on the following rule: if $\delta > \bar{\delta}$, then set $\alpha = 0.9\alpha$ and try again (i.e. recompute a new $\hat{\theta}_{k+1}$ using this new α), till we get a small enough step in the KL metric. Repeat the warm-up problem above with this algorithm and report the results. Now you should try different choices of $\bar{\delta}$ instead of α as in the warm-up problem.

3 Natural gradients

1. *(*5 points) A more principled way to ensure maximal performance improvement while moving along the KL metric is to use natural gradients. Recall that the natural policy

gradient takes the form:

$$\theta_{k+1} = \theta_k + \alpha_{npg}\ \hat{g}_{npg} = \theta_k + \sqrt{\frac{\bar{\delta}}{g_{vpg}^T\ g_{npg}}}\ g_{npg}, \tag{3}$$

where $g_{npg} = F^{-1} g_{vpg}$. We will implement this algorithm and repeat the empirical exercise as in the previous problems. You are expected to again try a few different settings for $\bar{\delta}$ and number of trajectories per iteration. In this problem, we will estimate the Fisher matrix as:

$$F = \frac{1}{|\mathcal{D}|} \sum_{i \in \mathcal{D}} \nabla \log \pi(a_i|s_i;\theta_k) \nabla \log \pi(a_i|s_i;\theta_k)^T \tag{4}$$

where $\mathcal{D}$ denotes (possibly subset of) samples collected using policy $\pi(.;\theta_k)$ and $|\mathcal{D}|$ the number of samples.

You can start this problem by copying `batch_reinforce.py` and writing the code for the computation of F as above. You will have to create a function that iterates over each data point (s_i, a_i) to compute the gradient of the `mean_LL` function of the policy with respect to the parameters of the policy. Once the Fisher matrix is computed, use the conjugate gradient algorithm (e.g. the implementation in `scipy`) to compute $F^{-1} g_{vpg}$. The `flat_vpg` function may provide some structure for how the gradient computation works.

(Note: in practice, for larger problems, we don't estimate the Fisher matrix this way. We use the other techniques discussed in the class – which are computationally much faster but at the same time harder to implement. We will provide you access to these more efficient methods for your course projects, if required.)

4 Variance Reduction

1. *(*4 points) In the class we discussed that variance reduction is critical to the success of policy gradient methods. We discussed how to use an approximate value function as baselines for the purpose of variance reduction. In this problem, you should implement a neural network baseline which predicts the empirical returns. In particular, for state and time (s, t), the targets are generated according to:

$$target(s,t) = \sum_{t'=t}^{T} \gamma^{t-t'} r_{t'}$$

Let w be the parameters of the MLP baseline. We train a network with weights w to solve the following optimization problem:

$$\underset{w}{\text{minimize}}\ \ ||MLP(s,t,w) - target(s,t)||_2^2$$

Use this baseline instead of the linear baseline, along with NPG, to solve the three environments. Fix the hyperparameters related to NPG to be the best you found in the previous question. Change only the baseline function, and report results for a few different choices of baseline architectures. For the inputs to the baseline MLP, use the same features function used in the linear baseline code. In other words, the linear baseline does a linear transformation of the features, whereas the MLP baseline will act on the same features and do a nonlinear transformation defined by the neural network.

5 Bonus points

Try other things that seem interesting to you. Talk to instructors for some ideas if needed. If you submit other things you tried, and give a concise summary of what worked, what didn't, along with plausible explanations, we will consider it for bonus points.