CS 287, Fall 2019 Homework #5
Reinforcement Learning, Model-based reinforcement learning

Deliverable: Report submitted through Gradescope by Monday Dec 2nd, 23:59.

Final grades are given based on plots. Your report should be a PDF. Format specified in PS5_template.tex.

---

1. Proximal Policy Optimization (PPO)[regular: 80 pts; extra credits: 30 pts] For the ease of implementing algorithms, we will briefly go over some basic theory. We refer the students to CS287 slides for more rigorous derivation.

We refer the students to https://www.tensorflow.org/tutorials if you are not familiar with tensorflow.

try to install the requirements:

```
pip install -r requirements.txt
```

Visualization tools are introduced in the beginning of Problem 1. You will use this for both P1 and P2.

For each question, you should run three times on all three environments (Swimmer, HalfCheetah, Hopper) and plot the curves.

Your full PPO performance at 1E or 1F should be better than the vanilla policy gradient performance 1A.

We provide a template PS5_template.tex and some sample results.

We provide a bash file to run the very first experiment. But you need to change the environment names to run more experiments. You should also change exp_name every run to log your data in different folders. You can try:

```
export PYTHONPATH="/your/dir/to/ppo:$PYTHONPATH"
cd run_scripts
bash ppo_example.sh
```

In this section, you will implement a model-free algorithm PPO. We'll start from the vanilla policy gradient. Then we 'll add different components, explained in each subsection.

Visualization Tools: viskit For problem 1 and 2, you will use viskit to visualize your plot. you should run the following code:

```
cd viskit
python frontend.py data_path
```

Then log in with your chrome at localhost:5000. For Problem 1, please choose Y-Axis Attribute to be Train-AverageReturn. For problem 2, please choose Y-Axis Attribute to be Policy-AverageReturn. Then save the image with your corresponding curve. Please click unrelated curves' legend to disable them. Please select series **split by name** in the interface.

1A. Policy Gradient [20 pts] Policy gradient methods work by computing an estimator of the policy gradient and plugging it into a stochastic gradient ascent algorithm. The most commonly used gradient estimator has the form

$$\hat{g} = \hat{E}_t[\nabla_\theta \log \pi(a_t|s_t) r(\tau)]$$

where π_θ is a stochastic policy and $r(\tau)$ is an estimator or computation of the advantage function at timestep t. Here, the expectation $\hat{E}_t$ indicates the empirical average over a finite batch of samples, in an algorithm that alternates between sampling and optimization.

Implementation: 0) In practice, we are minimizing an objective. Keep that in mind and be careful about the sign of your objective.

1) You will implement this in the hw5/algos/ppo.py and hw5/policies/distributions/diagonal_gaussian.py files. In diagonal gaussian, the log_std is a vector that contains the diagonal elements with the log of the stds. You will learn how to write the policy gradient objective. Search YOUR CODE HERE FOR PROBLEM 1A for the place you need to code.

2) Change the input argument **discount** to 0.99, and in hw5/utils/utils.py you will implement the discount_cumsum function that will be used in sampler/base.py. You will learn how to compute the discounted sum of the rewards which will re-weight the future rewards to the current state. Search YOUR CODE HERE FOR PROBLEM 1A for the place you need to code.

You only need to plot 2) the discounted reward result.

1B. Baselines[20 pts] To further reduce the variance, we introduce the baseline, which is b in:

$$\hat{g} = \hat{E}_t[\nabla_\theta \log \pi(a_t|s_t)(r(\tau) - b)].$$

A standard baseline is a fitted function that predicts the expected return from the current state or observation.

Implementation:

1) Currently, we are using zero baseline. Please change the input argument **use_baseline** to 1. In hw5/baselines/linear_baseline.py, you will implement how to fit a linear baseline as well as the predict function. You will learn how we can use a linear function to predict the returns, which can be used as baseline to reduce variance. Search YOUR CODE HERE FOR PROBLEM 1B for the place you need to code.

1C. Likelihood Ratio[10 pts] Now we try to use a different surrogate objective:

$$\max_\theta \quad \hat{E}_t[\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)} \hat{A}_t] \tag{1}$$

where $\hat{A}_t$ is the advantage estimation, previously denoted as $r(\tau) - b$. In PPO we use a clipped surrogate object as an alternative. In this section, we will implement the likelihood ratio $\frac{\pi_\theta(a|s)}{\pi_{\theta_{old}}(a|s)}$ in this part.

Implementation:

1) Please change the input argument **use_ppo_obj** to 1. Your code will be in hw5/algos/ppo.py and hw5/policies/distributions/diagonal_gaussian.py. Search for YOUR CODE HERE FOR PROBLEM 1C for the place you need to code.

1D. Clipping[10 pts] Without a constraint, maximization of the objective would lead to an excessively large policy update; hence, we now consider how to modify the objective, to penalize changes to the policy that move the ratio away from 1. The main objective we will use is the following:

$$\hat{E}_t[min(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t)]$$

where $r_t(\theta)$ is the likelihood ratio computed in 1C.

Implementation

1) Please change the variable **use_clipper** to 1. Your code will be in hw5/algos/ppo.py. Search for YOUR CODE HERE FOR PROBLEM 1D for the place you need to code.

1E. Entropy bonus[20 pts] In many works, an entropy (H) of the policy is calculated. This corresponds to the spread of action probabilities. Intuitively, if the policy outputs actions with relatively similar probabilities, then entropy will be high; but if the policy suggests a single action with a large probability then entropy will be low. We use the entropy as a means of improving exploration, by encouraging the model to be conservative regarding its sureness of the correct action.

There are two ways to add the entropy bonus in PPO: V1) We add a weighted entropy bonus in the final surrogate loss. V2) We have a full max-entropy formulation that treats the entropy as rewards.

In this problem, we will implement the entropy bonus of an action from a diagonal gaussian policy in the format of V1. We will explore format V2 in the extra credit problem 3 on SAC. (You are of course more than welcome to also implement format V2 into PPO in your own time.)

Implementation

1) Please change the variable **use_entropy** to 1. Your code will be in hw5/algos/ppo.py and hw5/policies/distributions/diagonal_gaussian.py. Search for YOUR CODE HERE FOR PROBLEM 1E for the place you need to code.

1F. (Extra credits) Generalized Advantage Estimator (GAE) [20 pts] We know that to compute the return we can compute the advantage $\hat{A}_t^1 = -V(s_t) + r_{t+1} + \gamma V(s_{t+1})$. This is also called TD residual. If we have more real samples after t, we can have $\hat{A}_t^n = -V(s_t) + r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n V(s_{t+n})$. These equations result from a telescoping sum, and we see that $\hat{A}^(_t$ involves a n-step estimate of the returns, minus a baseline term $V(s_t)$.The tradeoff

here is that the estimators $A^(n)_t$ with small n have low variance but high bias, whereas those with large n have low bias but high variance. Hence, GAE is defined as:

$$A_t^{GAE(\gamma,\lambda)} = (1-\lambda)(A_t^{(1)} + \lambda A_t^{(2)} + \lambda^2 A_t^{(3)} + \cdots) \tag{2}$$
$$= (1-\lambda)(\delta_t^V + \lambda(\delta_t^V + \gamma\delta_{t+1}^V) + \lambda^2(\delta_t^V + \gamma\delta_{t+1}^V + \gamma^2\delta_{t+2}^V) + \cdots) \tag{3}$$
$$= (1-\lambda)(\delta_t^V(1+\lambda+\lambda^2+\cdots) + \gamma\delta_{t+1}^V(\lambda+\lambda^2+\cdots) + \cdots) \tag{4}$$
$$= (1-\lambda)(\delta_t^V\frac{1}{1-\lambda} + \gamma\delta_{t+1}^V\frac{\lambda}{1-\lambda} + \cdots) \tag{5}$$
$$= \sum_{l=0}^{\infty}(\gamma\lambda)^l\delta_{t+l}^V \tag{6}$$

where $\delta_t^V = r_t + \gamma V(s_{t+1})V(s_t)$. In this problem, we will implement this GAE.
We refer the readers to https://arxiv.org/pdf/1506.02438.pdf for better understanding.

Implementation

1) Please change the input argument **use_gae** to 1. Your code will be in hw5/samplers/base.py. Search for YOUR CODE HERE FOR PROBLEM 1F for the place you need to code.

1G. (Extra credits) Hyperparameter Tuning[10 pts] Play with the hyperparameters including learning rate, batchsize, discount factor, gae λ, etc. Provide your best results here. Credit will be given if this is better than your best curve from 1A to 1E (or 1F if you choose to do 1F).

2. Model-based PPO [regular: 20 pts; extra credits: 40 pts]

Now that we have PPO in our hands, we move on to model-based reinforcement learning. Generally, there are three major steps for a model-based algorithm: 1. Collect data under current policy, 2. Learn dynamics model from past data, 3. Improve policy by using dynamics model. Usually a model-based algorithm performs these three steps iteratively. In this problem, we will implement a model-based algorithm based on the PPO we just implemented.

Model-based algorithms are usually considered more sample-efficient than model-free algorithms. Then learned model can also be stored for other tasks.

2A Model-based PPO with a single model [20 pts]

2A.1 Learning the dynamics After running the (initally random) policy for a while, you need to use the collected data to train your dynamics model with supervised learning. We specifically choose ℓ_2 loss for this problem. Note that here your dynamics model is another neural network.

Implementation

1) Your code will be in hw5/dynamics/mlp_dynamics.py. Search for YOUR CODE HERE FOR PROBLEM 2A.1 for the place you need to code.

2A.2 Predict the delta To make your model-based RL algorithm work, you need to pre-process your data to make it easier for the model to learn. In this problem, we pre-process the data so that a model learns to predict $s_{t+1} - s_t$ instead of s_t.

Implementation

1) Your code will be in hw5/dynamics/mlp_dynamics.py. Search for YOUR CODE HERE FOR PROBLEM 2A.2 for the place you need to code.

2A.3 data normalization To make your model-based RL algorithm work, you need to pre-process your data to make it easier for the model to learn. In this problem, we pre-process the data so that a model learns to predict normalized delta based on normalized states and actions.

Implementation

1) Your code will be in hw5/dynamics/mlp_dynamics.py and hw5/dynamics/utils.py. Search for YOUR CODE HERE FOR PROBLEM 2A.3 for the place you need to code.

Now you should have a full model that can be trained. You can run:

```
cd run_scripts
bash mb_ppo_example.sh
```

2B (Extra credit) Investigation of dynamics model loss [20 pts]

In this problem, you will investigate how different loss function affect the performance of model-based algorithms. Please find your loss implementation for 2A.1 and try ℓ_1 loss and ℓ_2 loss without the square.

2C (Extra credit) Model-ensemble PPO[20 pts]

In this problem, you will implement model-ensemble PPO. Instead of training one single model, you can now train 5 dynamics models with the same dataset. Then using an ensemble of them to improve the policy. More details can be found in the course slides.

Implementation

1) Your code will be in hw5/dynamics/mlp_dynamics_ensemble.py and you need to modify the input argument ensemble to 1. Search for YOUR CODE HERE FOR PROBLEM 2C for the place you need to code. As this is the (hardest) extra credit problem, we only provide minimal starter code for this part. You can also write from scratch for this part.

3 (Extra credit) Soft Actor Critic [extra credits: 30 pts] We refer the students to learn the theory of SAC in the lecture slides and the original paper. Soft actor-critic is a value-based algorithm that maximizes the maximum entropy objective. The algorithm learns both the soft Q-function Q^π, and a soft policy, π, and we sometimes refer to them simply as the Q-function

and policy. The soft Q-function is defined as

$$Q_t^\pi(s_t, a_t) \triangleq r(s_t, a_t) + E_{(s_{t+1}, a_{t+1}, \cdots) \sim p_\pi}(\sum_{l=1}^{T} r(s_{t+l}, a_{t+l}) - \alpha \log \pi_t(a_{t+l}|s_{t+l}))$$

and it satisfies the bellman equation

$$Q_t^\pi(s_t, a_t) = r(s_t, a_t) + E_{s_{t+1} \sim p(s_{t+1}|s_t, a_t)}[V_{t+1}^\pi(s_{t+1})]$$

where

$$V_t^\pi(s_t) \triangleq E_{a_t \sim \pi(a_t|s_t)}[Q_t^\pi(s_t, a_t) - \alpha \log \pi_t(a_t|s_t)]$$

is the soft value function. We can evaluate the soft Q value of a fixed policy π by applying Equation 3 to each time step. This procedure is called soft policy evaluation. We can now write the objective in terms of the soft Q-function as

$$J(\pi) = E_{s_0 \sim p_{s_0}}[D_{\mathrm{KL}}(\pi_0(\cdot|s_0)||\exp(\frac{1}{\alpha} Q_0^\pi(s_0, \cdot)))] + const$$

Optimization of the above objective is called policy improvement step. **all the experiments should run 3 random seeds on HalfCheetah-v2, Hopper-v2 and Ant-v2.**

3A Loss functions [20 pts]

3A.1 Loss function: Q loss

In sac.py, look for the method _q_function_loss_for. The method takes in the Q-function Q_θ and the target value function V_ψ, and outputs a value function loss over a minibatch. The two input arguments are Keras Network instances (defined in nn.py), and they provide an easy interface to construct neural networks for arbitrary inputs. For example, to obtain a N 1 tensor of Q values for a minibatch of size N, you can write

```
q_values = q_function((self._observations_ph, self._actions_ph))
```

where self._observations_ph is a $N \times |S|$ and self._actions_ph is a $N \times |A|$ placeholder tensor. All placeholders that you will need for this assignments are created in _create_placeholders method. Note that you can pass in any tf.Tensor objects as an input to a Keras Network object. Please implement the loss for Q:

$$J_Q(\theta) = E_{(s,a,s') \sim D}[(Q_\theta(s, a) - (r(s, a) + \gamma V_\psi(s'))^2]$$

Implementation

1) Your code will be in sac.py. Search for YOUR CODE HERE FOR PROBLEM 3A.1 for the place you need to code.

3A.2 Loss function: V loss

Your second task is to implement the value function loss in _value_function_loss_f. You will implement the loss for value function:

$$J_V(\psi) = E_{s \sim D}[(V_\psi(s) - E_{a \sim \pi_\phi(a|s)}[Q_\theta(s, a) - \alpha \log \pi_\phi(a|s)])]$$

To sample actions from the policy π_ϕ (and evaluate their log-likelihoods), you can call

```
actions, log_pis = policy(self._observations_ph)
```

which creates two tensors: an $N \times |A|$ dimensional tensor containing actions sampled from the policy, and an N dimensional tensor containing the corresponding log-likelihood. For now, ignore the input parameter q_function2, it will be relevant in a later part.

Implementation

1) Your code will be in sac.py. Search for YOUR CODE HERE FOR PROBLEM 3A.2 for the place you need to code.

3A.3 Double Q [10 pts]

Value-based methods suffer from positive bias in the Q value updates. The expectation of the sample estimator of the target Q-value is an upper bound of the true target Q-value. The bias accumulates when we keep applying the Bellman backup operator. To mitigate this effect, we can construct two Q functions and only select the minimum of them. To implement soft actor-critic with two Q-functions, you will need to:

1) Construct two independent Q-functions, with parameters θ_1 and θ_2, and learn them by minimizing the loss. Use the same target value function V as a target for both Q-functions.

2) Replace the original Q value with the minimum of Qs.

Implementation

1) Your code will be in sac.py. Search for YOUR CODE HERE FOR PROBLEM 3A.3 for the place you need to code.

After all the implementations are done. You can run:

```
python train_mujoco.py --env_name HalfCheetah-v2 --exp_name reinf -e 3
```

to perform SAC. Then please use plot.py to generate the plots. You can play with parameters such as reparameterize or two_qf to find the best performance and submit it with the correct exp_name.