Assignment 3: Policy Gradients

Due March 11, 5:59 pm

1 Introduction

The goal of this assignment is to experiment with policy gradient and its variants, including variance reduction tricks such as implementing reward-to-go and neural network baselines. The startercode can be found at

`https://github.com/milarobotlearningcourse/ift6163_homeworks/tree/master/hw3`

2 Review

2.1 Policy gradient

Recall that the reinforcement learning objective is to learn a θ^* that maximizes the objective function:

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_\theta(\tau)} [r(\tau)] \tag{1}$$

where each rollout τ is of length T, as follows:

$$\pi_\theta(\tau) = p(s_0, a_0, ..., s_{T-1}, a_{T-1}) = p(s_0)\pi_\theta(a_0|s_0) \prod_{t=1}^{T-1} p(s_t|s_{t-1}, a_{t-1})\pi_\theta(a_t|s_t)$$

and

$$r(\tau) = r(s_0, a_0, ..., s_{T-1}, a_{T-1}) = \sum_{t=0}^{T-1} r(s_t, a_t).$$

The policy gradient approach is to directly take the gradient of this objective:

$$\begin{aligned} \nabla_\theta J(\theta) &= \nabla_\theta \int \pi_\theta(\tau) r(\tau) d\tau && (2)\\ &= \int \pi_\theta(\tau) \nabla_\theta \log \pi_\theta(\tau) r(\tau) d\tau. && (3)\\ &= \mathbb{E}_{\tau \sim \pi_\theta(\tau)} [\nabla_\theta \log \pi_\theta(\tau) r(\tau)] && (4)\\ & && (5) \end{aligned}$$

In practice, the expectation over trajectories τ can be approximated from a batch of N sampled trajectories:

$$\begin{aligned} \nabla_\theta J(\theta) &\approx \frac{1}{N} \sum_{i=1}^{N} \nabla_\theta \log \pi_\theta(\tau_i) r(\tau_i) && (6)\\ &= \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_{it}|s_{it}) \right) \left(\sum_{t=0}^{T-1} r(s_{it}, a_{it}) \right). && (7) \end{aligned}$$

Here we see that the policy π_θ is a probability distribution over the action space, conditioned on the state. In the agent-environment loop, the agent samples an action a_t from $\pi_\theta(\cdot|s_t)$ and the environment responds with a reward $r(s_t, a_t)$.

2.2 Variance Reduction

2.2.1 Reward-to-go

One way to reduce the variance of the policy gradient is to exploit causality: the notion that the policy cannot affect rewards in the past. This yields the following modified objective, where the sum of rewards here does

not include the rewards achieved prior to the time step at which the policy is being queried. This sum of rewards is a sample estimate of the Q function, and is referred to as the "reward-to-go."

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_{it}|s_{it}) \left(\sum_{t'=t}^{T-1} r(s_{it'}, a_{it'}) \right). \tag{8}$$

2.2.2 Discounting

Multiplying a discount factor γ to the rewards can be interpreted as encouraging the agent to focus more on the rewards that are closer in time, and less on the rewards that are further in the future. This can also be thought of as a means for reducing variance (because there is more variance possible when considering futures that are further into the future). We saw in lecture that the discount factor can be incorporated in two ways, as shown below.

The first way applies the discount on the rewards from full trajectory:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_{it}|s_{it}) \right) \left(\sum_{t'=0}^{T-1} \gamma^{t'-1} r(s_{it'}, a_{it'}) \right) \tag{9}$$

and the second way applies the discount on the "reward-to-go:"

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_{it}|s_{it}) \left(\sum_{t'=t}^{T-1} \gamma^{t'-t} r(s_{it'}, a_{it'}) \right). \tag{10}$$

.

2.2.3 Baseline

Another variance reduction method is to subtract a baseline (that is a constant with respect to τ) from the sum of rewards:

$$\nabla_\theta J(\theta) = \nabla_\theta \mathbb{E}_{\tau \sim \pi_\theta(\tau)} \left[r(\tau) - b \right]. \tag{11}$$

This leaves the policy gradient unbiased because

$$\nabla_\theta \mathbb{E}_{\tau \sim \pi_\theta(\tau)} \left[b \right] = \mathbb{E}_{\tau \sim \pi_\theta(\tau)} \left[\nabla_\theta \log \pi_\theta(\tau) \cdot b \right] = 0.$$

In this assignment, we will implement a value function V_ϕ^π which acts as a *state-dependent* baseline. This value function will be trained to approximate the sum of future rewards starting from a particular state:

$$V_\phi^\pi(s_t) \approx \sum_{t'=t}^{T-1} \mathbb{E}_{\pi_\theta} \left[r(s_{t'}, a_{t'}) | s_t \right], \tag{12}$$

so the approximate policy gradient now looks like this:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_{it}|s_{it}) \left(\left(\sum_{t'=t}^{T-1} \gamma^{t'-t} r(s_{it'}, a_{it'}) \right) - V_\phi^\pi \left(s_{it} \right) \right). \tag{13}$$

2.2.4 Generalized Advantage Estimation

The quantity $\left(\sum_{t'=t}^{T-1} \gamma^{t'-t} r(s_{t'}, a_{t'}) \right) - V_\phi^\pi \left(s_t \right)$ from the previous policy gradient expression (removing the i index for clarity) can be interpreted as an estimate of the advantage function:

$$A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t), \tag{14}$$

where $Q^\pi(s_t, a_t)$ is estimated using Monte Carlo returns and $V^\pi(s_t)$ is estimated using the learned value function V^π_ϕ. We can further reduce variance by also using V^π_ϕ in place of the Monte Carlo returns to estimate the advantage function as:

$$A^\pi(s_t, a_t) \approx \delta_t = r(s_t, a_t) + \gamma V^\pi_\phi(s_{t+1}) - V^\pi_\phi(s_t), \tag{15}$$

with the edge case $\delta_{T-1} = r(s_{T-1}, a_{T-1}) - V^\pi_\phi(s_{T-1})$. However, this comes at the cost of introducing bias to our policy gradient estimate, due to modeling errors in V^π_ϕ. We can instead use a combination of n-step Monte Carlo returns and V^π_ϕ to estimate the advantage function as:

$$A^\pi_n(s_t, a_t) = \sum_{t'=t}^{t+n} \gamma^{t'-t} r(s_{t'}, a_{t'}) + \gamma^n V^\pi_\phi(s_{t+n+1}) - V^\pi_\phi(s_t). \tag{16}$$

Increasing n incorporates the Monte Carlo returns more heavily in the advantage estimate, which lowers bias and increases variance, while decreasing n does the opposite. Note that $n = T - t - 1$ recovers the unbiased but higher variance Monte Carlo advantage estimate used in (13), while $n = 0$ recovers the lower variance but higher bias advantage estimate δ_t.

We can combine multiple n-step advantage estimates as an exponentially weighted sum, which is known as the generalized advantage estimator (GAE). Let $\lambda \in [0, 1]$. Then we define:

$$A^\pi_{GAE}(s_t, a_t) = \frac{1 - \lambda^{T-t-1}}{1 - \lambda} \sum_{n=1}^{T-t-1} \lambda^{n-1} A^\pi_n(s_t, a_t), \tag{17}$$

where $\frac{1-\lambda^{T-t-1}}{1-\lambda}$ is a normalizing constant. Note that a higher λ emphasizes advantage estimates with higher values of n, and a lower λ does the opposite. Thus, λ serves as a control for the bias-variance tradeoff, where increasing λ decreases bias and increases variance. In the infinite horizon case ($T = \infty$), we can show:

$$A^\pi_{GAE}(s_t, a_t) = \frac{1}{1 - \lambda} \sum_{n=1}^{\infty} \lambda^{n-1} A^\pi_n(s_t, a_t) \tag{18}$$

$$= \sum_{t'=t}^{\infty} (\gamma\lambda)^{t'-t} \delta_{t'}, \tag{19}$$

where we have omitted the derivation for brevity (see the GAE paper https://arxiv.org/pdf/1506.02438.pdf for details). In the finite horizon case, we can write:

$$A^\pi_{GAE}(s_t, a_t) = \sum_{t'=t}^{T-1} (\gamma\lambda)^{t'-t} \delta_{t'}, \tag{20}$$

which serves as a way we can efficiently implement the generalized advantage estimator, since we can recursively compute:

$$A^\pi_{GAE}(s_t, a_t) = \delta_t + \gamma\lambda A^\pi_{GAE}(s_{t+1}, a_{t+1}) \tag{21}$$

3 Overview of Implementation

3.1 Files

To implement policy gradients, we will be building up the code that we started in homework 1. All files needed to run your code are in the `hw2` folder, but there will be some blanks you will fill with your solutions from homework 1. These locations are marked with `# TODO: get this from hw1` and are found in the following files:

- `infrastructure/rl_trainer.py`
- `infrastructure/utils.py`
- `policies/MLP_policy.py`

After bringing in the required components from the previous homework, you can begin work on the new policy gradient code. These placeholders are marked with `TODO`, located in the following files:

- `agents/pg_agent.py`
- `policies/MLP_policy.py`

The script to run the experiments is found in `scripts/run_hw2.py` (for the local option) or `scripts/run_hw2.ipynb` (for the Colab option).

3.2 Overview

As in the previous homework, the main training loop is implemented in `infrastructure/rl_trainer.py`.

The policy gradient algorithm uses the following 3 steps:

1. **Sample trajectories** by generating rollouts under your current policy.
2. **Estimate returns and compute advantages**. This is executed in the `train` function of **pg_agent.py**
3. **Train/Update parameters**. The computational graph for the policy and the baseline, as well as the update functions, are implemented in `policies/MLP_policy.py`.

4 Implementing Policy Gradients

You will be implementing two different return estimators within `pg_agent.py`. The first ("Case 1" within `calculate_q_vals`) uses the discounted cumulative return of the full trajectory and corresponds to the "vanilla" form of the policy gradient (Equation 9):

$$r(\tau_i) = \sum_{t'=0}^{T-1} \gamma^{t'} r(s_{it'}, a_{it'}). \tag{22}$$

The second ("Case 2") uses the "reward-to-go" formulation from Equation 10:

$$r(\tau_i) = \sum_{t'=t}^{T-1} \gamma^{t'-t} r(s_{it'}, a_{it'}). \tag{23}$$

Note that these differ only by the starting point of the summation.

Implement these return estimators as well as the remaining sections marked `TODO` in the code. For the small-scale experiments, you may skip those sections that are run only if `nn_baseline` is `True`; we will return to baselines in Section 6. (These sections are in `MLPPolicyPG:update` and `PGAgent:estimate_advantage`.)

5 Small-Scale Experiments

After you have implemented all non-baseline code from Section 4, you will run two small-scale experiments to get a feel for how different settings impact the performance of policy gradient methods.

Experiment 1 (CartPole). Run multiple experiments with the PG algorithm on the discrete `CartPole-v0` environment, using the following commands:

```
python run_hw3.py env_name=CartPole-v0 n=100 b=1000 dsa=true exp_name=q1_sb_no_rtg_dsa
    rl_alg=reinforce

python run_hw3.py env_name=CartPole-v0 n=100 b=1000 rtg=true dsa=true exp_name=
    q1_sb_rtg_dsa rl_alg=reinforce

python run_hw3.py env_name=CartPole-v0 n=100 b=1000 rtg=true exp_name=q1_sb_rtg_na rl_alg=
    reinforce

python run_hw3.py env_name=CartPole-v0 n=100 b=5000 dsa=true exp_name=q1_lb_no_rtg_dsa
    rl_alg=reinforce

python run_hw3.py env_name=CartPole-v0 n=100 b=5000 rtg=true dsa=true exp_name=
    q1_lb_rtg_dsa rl_alg=reinforce

python run_hw3.py env_name=CartPole-v0 n=100 b=5000 rtg=true exp_name=q1_lb_rtg_na rl_alg=
    reinforce
```

What's happening here:

- `n` : Number of iterations.
- `b` : Batch size (number of state-action pairs sampled while acting according to the current policy at each iteration).
- `dsa` : Flag: if present, sets `standardize_advantages` to False. Otherwise, by default, standardizes advantages to have a mean of zero and standard deviation of one.
- `rtg=true` : Flag: if present, sets `reward_to_go=True`. Otherwise, `reward_to_go=False` by default.
- `exp_name` : Name for experiment, which goes into the name for the data logging directory.

Various other command line arguments will allow you to set batch size, learning rate, network architecture, and more. You can change these as well, but keep them fixed between the 6 experiments mentioned above.

Deliverables for report:

- Create two graphs:
  - In the first graph, compare the learning curves (average return at each iteration) for the experiments prefixed with `q1_sb_`. (The small batch experiments.)
  - In the second graph, compare the learning curves for the experiments prefixed with `q1_lb_`. (The large batch experiments.)
- Answer the following questions briefly:
  - Which value estimator has better performance without advantage-standardization: the trajectory-centric one, or the one using reward-to-go?
  - Did advantage standardization help?
  - Did the batch size make an impact?

- Provide the exact command line configurations (or `#@params` settings in Colab) you used to run your experiments, including any parameters changed from their defaults.

What to Expect:

- The best configuration of CartPole in both the large and small batch cases should converge to a maximum score of 200.

Experiment 2 (InvertedPendulum). Run experiments on the `InvertedPendulum-v2` continuous control environment as follows:

```
python run_hw3.py env_name= InvertedPendulum-v2
    ep_len=1000 discount=0.9 n=100 l=2 s=64 b=<b*> lr=<r*> rtg=true
    exp_name=q2_b<b*>_r<r*> rl_alg=reinforce
```

where your task is to find the smallest batch size `b*` and largest learning rate `r*` that gets to optimum (maximum score of 1000) in less than 100 iterations. The policy performance may fluctuate around 1000; this is fine. The precision of `b*` and `r*` need only be one significant digit.

Deliverables:

- Given the `b*` and `r*` you found, provide a learning curve where the policy gets to optimum (maximum score of 1000) in less than 100 iterations. (This may be for a single random seed, or averaged over multiple.)
- Provide the exact command line configurations you used to run your experiments.

6 Implementing Neural Network Baselines

You will now implement a value function as a state-dependent neural network baseline. This will require filling in some `TODO` sections skipped in Section 4. In particular:

- This neural network will be trained in the `update` method of `MLPPolicyPG` along with the policy gradient update.
- In `pg_agent.py:estimate_advantage`, the predictions of this network will be subtracted from the reward-to-go to yield an estimate of the advantage. This implements $\left(\sum_{t'=t}^{T-1} \gamma^{t'-t} r(s_{it'}, a_{it'})\right) - V_\phi^\pi(s_{it})$.

7 More Complex Experiments

Note: The following tasks take quite a bit of time to train. Please start early! For all remaining experiments, use the reward-to-go estimator.

Experiment 3 (LunarLander). You will now use your policy gradient implementation to learn a controller for `LunarLanderContinuous-v2`. The purpose of this problem is to test and help you debug your baseline implementation from Section 6.

Run the following command:

```
python run_hw3.py
    env_name=LunarLanderContinuous-v2 ep_len=1000
    discount=0.99 n=100 l=2 s=64 b=40000 lr=0.005
    reward_to_go=true nn_baseline=true rl_alg=reinforce exp_name=q3_b40000_r0.005
```

Deliverables:

- Plot a learning curve for the above command. You should expect to achieve an average return of around 180 by the end of training.

Experiment 4 (HalfCheetah). You will be using your policy gradient implementation to learn a controller for the `HalfCheetah-v2` benchmark environment with an episode length of 150. This is shorter than the default episode length (1000), which speeds up training significantly. Search over batch sizes `b` $\in [10000, 30000, 50000]$ and learning rates `r` $\in [0.005, 0.01, 0.02]$ to replace `<b>` and `<r>` below.

```
python run_hw3.py env_name=HalfCheetah-v2 ep_len=150
    discount=0.95 n=100 l=2 s=32 b=<b> lr=<r> rtg=true nn_baseline=true rl_alg=reinforce
    exp_name=q4_search_b<b>_lr<r>_rtg_nnbaseline
```

Deliverables:

- Provide a single plot with the learning curves for the HalfCheetah experiments that you tried. Describe in words how the batch size and learning rate affected task performance.

Once you've found optimal values `b*` and `r*`, use them to run the following commands (replace the terms in angle brackets):

```
python run_hw3.py env_name=HalfCheetah-v2 ep_len=150
    discount=0.95 n=100 l=2 s=32 b=<b*> lr=<r*>
    exp_name=q4_b<b*>_r<r*>

python run_hw3.py env_name=HalfCheetah-v2 ep_len=150
    discount=0.95 n=100 l=2 s=32 b=<b*> lr=<r*> rtg=true
    exp_name=q4_b<b*>_r<r*>_rtg

python run_hw3.py env_name=HalfCheetah-v2 ep_len=150
    discount=0.95 n=100 l=2 s=32 b=<b*> lr=<r*> nn_baseline=true
    exp_name=q4_b<b*>_r<r*>_nnbaseline

python run_hw3.py env_name=HalfCheetah-v2 ep_len=150
    discount=0.95 n=100 l=2 s=32 b=<b*> lr=<r*> rtg=true nn_baseline=true
    exp_name=q4_b<b*>_r<r*>_rtg_nnbaseline
```

Deliverables: Provide a single plot with the learning curves for these four runs. The run with both reward-to-go and the baseline should achieve an average score close to 200.

8 Implementing Generalized Advantage Estimation

You will now use the value function you previously implemented to implement a simplified version of GAE-λ. This will require filling in the remaining `TODO` section in `pg_agent.py:estimate_advantage`.

Experiment 5 (HopperV2). You will now use your implementation of policy gradient with generalized advantage estimation to learn a controller for a version of `Hopper-v2` with noisy actions. Search over $\lambda \in [0, 0.95, 0.99, 1]$ to replace `<λ>` below. Note that with a correct implementation, $\lambda = 1$ is equivalent to the vanilla neural network baseline estimator. **Do not** change any of the other hyperparameters (e.g. batch size, learning rate).

```
python ift6131/scripts/run_hw3.py
    env_name=Hopper-v2 ep_len=1000 rl_alg=reinforce
    discount=0.99 n=300 l=2 s=32 b=2000 lr=0.001
    reward_to_go=true nn_baseline=true action_noise_std=0.5 gae_lambda=<$\lambda$>
    exp_name=q5_b2000_r0.001_lambda<$\lambda$>
```

Deliverables:

- Provide a single plot with the learning curves for the `Hopper-v2` experiments that you tried. Describe in words how λ affected task performance. The run with the best performance should achieve an average score close to 400.

9 Actor-Critic

9.1 Introduction

THe next part of this assignment requires you to modify policy gradients to an actor-critic formulation. Note that evaluation may take longer for actor-critic than policy gradient (on half-cheetah) due to the significantly larger number of training steps for the value function.

Recall the policy gradient from hw2:

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N}\sum_{t=1}^{T}\nabla_\theta \log \pi_\theta(a_{it}|s_{it})\left(\left(\sum_{t'=t}^{T}\gamma^{t'-t}r(s_{it'},a_{it'})\right) - V_\phi^\pi\left(s_{it}\right)\right).$$

In this formulation, we estimate the Q function by taking the sum of rewards to go over each trajectory, and we subtract the value function baseline to obtain the advantage

$$A^\pi(s_t,a_t) \approx \left(\sum_{t'=t}^{T}\gamma^{t'-t}r(s_{t'},a_{t'})\right) - V_\phi^\pi\left(s_t\right)$$

In practice, the estimated advantage value suffers from high variance. Actor-critic addresses this issue by using a *critic network* to estimate the sum of rewards to go. The most common type of critic network used is a value function, in which case our estimated advantage becomes

$$A^\pi(s_t,a_t) \approx r(s_t,a_t) + \gamma V_\phi^\pi\left(s_{t+1}\right) - V_\phi^\pi\left(s_t\right)$$

In this assignment we will use the same value function network from hw2 as the basis for our critic network. One additional consideration in actor-critic is updating the critic network itself. While we can use Monte Carlo rollouts to estimate the sum of rewards to go for updating the value function network, in practice we fit our value function to the following *target values*:

$$y_t = r(s_t,a_t) + \gamma V^\pi(s_{t+1})$$

we then regress onto these target values via the following regression objective which we can optimize with gradient descent:

$$\min_\phi \sum_{i,t}(V_\phi^\pi(s_{it}) - y_{it})^2$$

In theory, we need to perform this minimization every time we update our policy, so that our value function matches the behavior of the new policy. In practice however, this operation can be costly, so we may instead just take a few gradient steps at each iteration. Also note that since our target values are based on the old value function, we may need to recompute the targets with the updated value function, in the following fashion:

1. Update targets with current value function
2. Regress onto targets to update value function by taking a few gradient steps
3. Redo steps 1 and 2 several times

In all, the process of fitting the value function critic is an iterative process in which we go back and forth between computing target values and updating the value function to match the target values. Through experimentation, you will see that this iterative process is crucial for training the critic network.

9.2 Implementation

Your code will build off your solutions from homework 2. You will need to fill in the TODOS for the following parts of the code.

- In `policies/MLP_policy.py`, implement the `update` function for the class `MLPPolicyAC`. You should note that the AC policy class is in fact the same as the policy class you implemented in the policy gradient homework (except we no longer have a nn_baseline).
- In `agents/ac_agent.py`, finish the `train` function. This function should implement the necessary critic updates, estimate the advantage, and then update the policy. Log the final losses at the end so you can monitor it during training.
- In `agents/ac_agent.py`, finish the `estimate_advantage` function: this function uses the critic network to estimate the advantage values. The advantage values are computed according to

  $$A^\pi(s_t, a_t) \approx r(s_t, a_t) + \gamma V^\pi_\phi(s_{t+1}) - V^\pi_\phi(s_t)$$

  Note: for terminal timesteps, you must make sure to cut off the reward to go (i.e., set it to zero), in which case we have

  $$A^\pi(s_t, a_t) \approx r(s_t, a_t) - V^\pi_\phi(s_t)$$

- `critics/bootstrapped_continuous_critic.py` complete the TODOS in `update`. In `update`, perform the critic update according to process outlined in the introduction. You must perform

  `self.num_grad_steps_per_target_update * self.num_target_updates`

  number of updates, and recompute the target values every `self.num_grad_steps_per_target_update` number of steps.

9.3 Evaluation

Once you have a working implementation of actor-critic, you should prepare a report. The report should consist of figures for the question below. You should turn in the report as one PDF (same PDF as part 1) and a zip file with your code (same zip file as part 1). If your code requires special instructions or dependencies to run, please include these in a file called `README` inside the zip file.

Question 4: Sanity check with Cartpole Now that you have implemented actor-critic, check that your solution works by running Cartpole-v0.

```
python run_hw3.py env_name=CartPole-v0 n=100 b=1000 exp_name=q4_ac_1_1 ntu=1 ngsptu=1
    rl_alg=ac
```

In the example above, we alternate between performing one target update and one gradient update step for the critic. As you will see, this probably doesn't work, and you need to increase both the number of target updates and number of gradient updates. Compare the results for the following settings and report which worked best. Do this by plotting all the runs on a single plot and writing your takeaway in the caption.

```
python run_hw3.py env_name=CartPole-v0 n=100 rl_alg=ac b=1000 exp_name=q4_100_1 ntu=100
    ngsptu=1
```

```
python run_hw3.py env_name=CartPole-v0 n=100 rl_alg=ac b=1000 exp_name=q4_1_100 ntu=1
    ngsptu=100
```

```
python run_hw3.py env_name=CartPole-v0 n=100 rl_alg=ac b=1000 exp_name=q4_10_10 ntu=10
    ngsptu=10
```

At the end, the best setting from above should match the policy gradient results from Cartpole (200).

Deliverables:

- Plot a learning curve for the above commands and include that in the written submission.

Question 5: Run actor-critic with more difficult tasks Use the best setting from the previous question to run InvertedPendulum and HalfCheetah:

```
python run_hw3.py env_name=InvertedPendulum-v2 rl_alg=ac ep_len=1000 discount=0.95 n=100 l
    =2 s=64 b=5000 lr=0.01 exp_name=q5_<ntu>_<ngsptu> ntu=<> ngsptu=<>
```

where `<ntu>_<ngsptu>` is replaced with the parameters you chose.

```
python run_hw3.py env_name=HalfCheetah-v2 rl_alg=ac ep_len=150 discount=0.90
    scalar_log_freq=1 n=150 l=2 s=32 b=30000 eb=1500 lr=0.02 exp_name=q5_<ntu>_<ngsptu> ntu
    =<> ngsptu=<>
```

Your results should roughly match those of policy gradient. After 150 iterations, your HalfCheetah return should be around 150. After 100 iterations, your InvertedPendulum return should be around 1000. Your deliverables for this section are plots with the eval returns for both enviornments.

As a debugging tip, the returns should start going up immediately. For example, after 20 iterations, your HalfCheetah return should be above -40 and your InvertedPendulum return should near or above 100. However, there is some variance between runs, so the 150-iteration (for HalfCheetah) and 100-iteration (for InvertedPendulum) results are the numbers we use to grade.

Deliverables:

- Plot a learning curve for the above commands and include that in the written submission.

10 Dyna

Using your code so far you are going to create a version of Dyna. This version of Dyna is going to use the actor critic and policy gradient code you have written so far and combine it with your model based code from the last assignment. What you are supposed to implement will be similar to a simpler on-policy version of MBPO. Bascially, you will use your forward dynamics model from the last assignment to generate more on-policy data to train the policy.

1. In `agents/dyna_agent.py`, implement the `train` function for the class `MBAgent`. You will need to copy code from the policy gradient and actor critic code to use with your mb code.

What commands to run:

```
python run_hw3.py exp_name=q10_cheetah_n500_arch1x32 env_name=cheetah-ift6163-v0 discount
    =0.95 n_layers=2 size=32 learning_rate=0.01 scalar_log_freq=1 n=100 batch_size=5000
    training_batch_size=1024 rl_alg=dyna

python run_hw3.py exp_name=q10_cheetah_n500_arch1x32 env_name=cheetah-ift6163-v0 discount
    =0.95 n_layers=2 size=32 learning_rate=0.01 scalar_log_freq=1 n=100 batch_size=2000
    training_batch_size=1024 rl_alg=dyna
```

Deliverables:

- Plot a learning curve for the above commands and include that in the written submission.

11 Bonus!

Choose any (or all) of the following:

- A serious bottleneck in the learning, for more complex environments, is the sample collection time. In `infrastructure/rl_trainer.py`, we only collect trajectories in a single thread, but this process can be

fully parallelized across threads to get a useful speedup. Implement the parallelization and report on the difference in training time.

- In PG, we collect a batch of data, estimate a single gradient, and then discard the data and move on. Can we potentially accelerate PG by taking multiple gradient descent steps with the same batch of data? Explore this option and report on your results. Set up a fair comparison between single-step PG and multi-step PG on at least one MuJoCo gym environment.
- The actor critic algorithm can also be improved by adding constraints on how much the distribution changes. As a bonus you can implement the PPO clipping term ontop of the actor-critic algorithm to see how it improves performance.

12 Submission

12.1 Submitting the PDF

Your report should be a document containing

(a) All graphs and answers to short explanation questions requested for Experiments 1-4.

(b) All command-line expressions you used to run your experiments.

(c) (Optionally) Your bonus results (command-line expressions, graphs, and a few sentences that comment on your findings).

12.2 Submitting the code and experiment runs

In order to turn in your code and experiment logs, create a folder that contains the following:

- A folder named `run_logs` with all the experiment runs from this assignment. These folders can be copied directly from the `ift6131/data` folder. **Do not change the names originally assigned to the folders, as specified by `exp_name` in the instructions.** Video logging is disabled by default in the code, but if you turned it on for debugging, you need to run those again with `--video_log_freq -1`, or else the file size will be too large for submission.
- The `ift6131` folder with all the `.py` files, with the same names and directory structure as the original homework repository (excluding the `ift6131/data` folder). Also include any special instructions we need to run in order to produce each of your figures or tables in the form of a README file.

As an example, the unzipped version of your submission should result in the following file structure. **Make sure that the submit.zip file is below 15MB.**

If you are a Mac user, **do not use the default "Compress" option to create the zip**. It creates artifacts that the autograder does not like. You may use `zip -vr submit.zip submit -x "*.DS_Store"` from your terminal.

```
submit.zip
├── run_logs
│   ├── q1_lb_rtg_na_CartPole-v0_12-09-2019_17-53-4
│   │   └── events.out.tfevents.1567529456.e3a096ac8ff4
│   ├── q3_b40000_r0.005_LunarLanderContinuous-v2_12-09-2019_00-17-58
│   │   └── events.out.tfevents.1567529456.e3a096ac8ff4
│   └── ...
├── ift6131
│   ├── agents
│   │   ├── bc_agent.py
│   │   └── ...
│   ├── policies
│   │   └── ...
│   └── ...
├── README.md
└── ...
```

12.3 Turning it in

Turn in your assignment by the deadline on Gradescope. Uploade the zip file with your code to **HW3 Code**, and upload the PDF of your report to **HW3**.