Assignment 2: Policy Gradients

Due September 25, 11:59 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.

Starter code
`https://github.com/berkeleydeeprlcourse/homework_fall2023/tree/master/hw2`

LaTeX assignment template
`https://github.com/berkeleydeeprlcourse/homework_fall2023/blob/main/hw2/hw2_template.tex`

Google has donated $50/student of cloud credit to students enrolled in the class. If you're enrolled in the class (or you are paying for your own GCP credits), you should follow the setup guide here:

`https://github.com/berkeleydeeprlcourse/homework_fall2023/blob/main/hw2/google_cloud/README.md`

Please don't use Colab for this assignment! It will probably crash or shutdown halfway through a long run. Running things on GCP or locally will be a much better experience.

If you use Google Cloud, you shouldn't need to use more than $15 of cloud credit in this assignment. HW3 will take a lot more compute power (and it needs a GPU), so you'll want to make sure you have enough to finish it on cloud without any issues.

2 Review

2.1 Policy gradient

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

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

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:

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

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

$$\nabla_\theta J(\theta) \approx \frac{1}{N}\sum_{i=1}^{N} \nabla_\theta \log \pi_\theta(\tau_i) r(\tau_i) \tag{6}$$

$$= \frac{1}{N}\sum_{i=1}^{N} \left(\sum_{t=0}^{T-1} \nabla_\theta \log \pi_\theta(a_{it}|s_{it})\right)\left(\sum_{t=0}^{T-1} r(s_{it}, a_{it})\right). \tag{7}$$

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:

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

2.2.4 Generalized Advantage Estimation

The quantity $\left(\sum_{t'=t}^{T-1} \gamma^{t'-t} r(s_{t'}, a_{t'})\right) - V_\phi^\pi(s_t)$ 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_ϕ^π. We can further reduce variance by also using V_ϕ^π in place of the Monte Carlo returns to estimate the advantage function as:

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

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

$$A_n^\pi(s_t, a_t) = \sum_{t'=t}^{t+n} \gamma^{t'-t} r(s_{t'}, a_{t'}) + \gamma^n V_\phi^\pi(s_{t+n+1}) - V_\phi^\pi(s_t). \tag{16}$$

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:

$$A_{GAE}^\pi(s_t, a_t) = \frac{1 - \lambda^{T-t-1}}{1 - \lambda} \sum_{n=1}^{T-t-1} \lambda^{n-1} A_n^\pi(s_t, a_t), \tag{17}$$

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:

$$A_{GAE}^\pi(s_t, a_t) = \frac{1}{1 - \lambda} \sum_{n=1}^{\infty} \lambda^{n-1} A_n^\pi(s_t, a_t) \tag{18}$$

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

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:

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

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

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

3 Policy Gradients

3.1 Implementation

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 4. (These sections are in `MLPPolicyPG:update` and `PGAgent:estimate_advantage`.)

3.2 Experiments

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

```
python cs285/scripts/run_hw2.py --env_name CartPole-v0 -n 100 -b 1000 \
    --exp_name cartpole

python cs285/scripts/run_hw2.py --env_name CartPole-v0 -n 100 -b 1000 \
    -rtg --exp_name cartpole_rtg

python cs285/scripts/run_hw2.py --env_name CartPole-v0 -n 100 -b 1000 \
    -na --exp_name cartpole_na

python cs285/scripts/run_hw2.py --env_name CartPole-v0 -n 100 -b 1000 \
    -rtg -na --exp_name cartpole_rtg_na

python cs285/scripts/run_hw2.py --env_name CartPole-v0 -n 100 -b 4000 \
    --exp_name cartpole_lb

python cs285/scripts/run_hw2.py --env_name CartPole-v0 -n 100 -b 4000 \
    -rtg --exp_name cartpole_lb_rtg

python cs285/scripts/run_hw2.py --env_name CartPole-v0 -n 100 -b 4000 \
    -na --exp_name cartpole_lb_na

python cs285/scripts/run_hw2.py --env_name CartPole-v0 -n 100 -b 4000 \
    -rtg -na --exp_name cartpole_lb_rtg_na
```

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).

- `-rtg` : Flag: if present, sets `reward_to_go=True`. Otherwise, `reward_to_go=False` by default.
- `-na` : Flag: if present, sets `normalize_advantages` to True, normalizing the advantages to have a mean of zero and standard deviation of one within a batch.
- `--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.

Deliverables for report:

- Create two graphs:
  - In the first graph, compare the learning curves (average return vs. number of environment steps) for the experiments prefixed with `cartpole`. (The small batch experiments.)
  - In the second graph, compare the learning curves for the experiments prefixed with `cartpole_lb`. (The large batch experiments.)

  For all plots in this assignment, the x-axis should be number of environment steps, logged as `Train_EnvstepsSoFar` (*not* number of policy gradient iterations).
- Answer the following questions briefly:
  - Which value estimator has better performance without advantage normalization: the trajectory-centric one, or the one using reward-to-go?
  - Did advantage normalization 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.

4 Using a Neural Network Baseline

4.1 Implementation

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 3. 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 $\left(\sum_{t'=t}^{T-1} \gamma^{t'-t} r(s_{it'}, a_{it'})\right) - V_\phi^\pi(s_{it})$.
- We will train the baseline network for *multiple* gradient steps for each policy update, determined by the parameter `baseline_gradient_steps`.

4.2 Experiments

Experiment 2 (HalfCheetah). Next, you will use your baselined policy gradient implementation to learn a controller for `HalfCheetah-v4`.

Run the following commands:

```
# No baseline
python cs285/scripts/run_hw2.py --env_name HalfCheetah-v4 \
    -n 100 -b 5000 -rtg --discount 0.95 -lr 0.01 \
    --exp_name cheetah
# Baseline
python cs285/scripts/run_hw2.py --env_name HalfCheetah-v4 \
    -n 100 -b 5000 -rtg --discount 0.95 -lr 0.01 \
    --use_baseline -blr 0.01 -bgs 5 --exp_name cheetah_baseline
```

You might notice that we omitted `-na` (normalize advantages). That's because in reality, advantage normalization is a very powerful trick, and eliminates the need for a baseline most of the simple environments that we test in.

Deliverables:

- Plot a learning curve for the baseline loss.
- Plot a learning curve for the eval return. You should expect to achieve an average return over 300 for the baselined version.
- Run another experiment with a decreased number of baseline gradient steps (`-bgs`) and/or baseline learning rate (`-blr`). How does this affect (a) the baseline learning curve and (b) the performance of the policy?
- **Optional:** Add `-na` back to see how much it improves things. Also, set `video_log_freq 10`, then open TensorBoard and go to the "Images" tab to see some videos of your HalfCheetah walking along!

5 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 3 (LunarLander-v2). You will now use your implementation of policy gradient with generalized advantage estimation to learn a controller for a version of `LunarLander-v2` with noisy actions. Search over $\lambda \in [0, 0.95, 0.98, 0.99, 1]$ to replace `<λ>` below. **Do not** change any of the other hyperparameters (e.g. batch size, learning rate).

```
python cs285/scripts/run_hw2.py \
    --env_name LunarLander-v2 --ep_len 1000 \
    --discount 0.99 -n 300 -l 3 -s 128 -b 2000 -lr 0.001 \
    --use_reward_to_go --use_baseline --gae_lambda <λ> \
    --exp_name lunar_lander_lambda<λ>
```

Deliverables:

- Provide a single plot with the learning curves for the `LunarLander-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 200 (180+).
- Consider the parameter λ. What does $\lambda = 0$ correspond to? What about $\lambda = 1$? Relate this to the task performance in `LunarLander-v2` in one or two sentences.

6 Hyperparameters and Sample Efficiency

One criticism of policy gradient methods is that they tend to be very *sample inefficient*. Examining your results for the previous problems, you'll notice that your algorithm takes hundreds of thousands or millions of steps in the environment before it is able to learn a good policy.

While improving sample inefficiency is in general an open challenge, we can do a lot better by just picking better hyperparameters! During training, we have to choose many hyperparameters and settings:

1. Discount factor
2. Network size
3. Batch size (super small batch sizes may have high variance, but very large batch sizes waste a lot of samples because you must recollect the entire batch for every gradient step)
4. Learning rate
5. Whether to use return-to-go
6. Whether to normalize advantages
7. Whether to use GAE, and if we do, what value of λ to use

Experiment 4 (InvertedPendulum). First, train a policy for the inverted pendulum problem without GAE and with otherwise default settings. Use five different seeds to get a good estimate of average performance:

```
for seed in $(seq 1 5); do
    python cs285/scripts/run_hw2.py --env_name InvertedPendulum-v4 -n 100 \
        --exp_name pendulum_default_s$seed \
        -rtg --use_baseline -na \
        --batch_size 5000 \
        --seed $seed
done
```

Your task is to tune hyperparameters so that your implementation reaches maximum performance (score of 1000) in fewer **environment steps** than these default settings (note: this is **not** the same as minimizing number of policy gradient iterations: one policy gradient iteration corresponds to `batch_size` environment steps). Try and find hyperparameters that reach a score of 1000 in as few steps as possible!

Deliverables:

1. Provide a set of hyperparameters that achieve high return on `InvertedPendulum-v4` in as few environment steps as possible.
2. Show learning curves for the average returns with your hyperparameters and with the default settings, with environment steps on the x-axis. Returns should be averaged over 5 seeds.

7 Extra Credit: Humanoid

Experiment 5 (Humanoid-v4). Let's put everything we've learned together to learn a policy for a more complicated environment! The `Humanoid-v4` environment in Gym trains a humanoid to walk, from scratch!

If you've implemented everything correctly, you shouldn't have to do anything new for this section. Just run the following command - we've filled in some hyperparameters that should work:

```
python cs285/scripts/run_hw2.py \
    --env_name Humanoid-v4 --ep_len 1000 \
    --discount 0.99 -n 1000 -l 3 -s 256 -b 50000 -lr 0.001 \
    --baseline_gradient_steps 50 \
    -na --use_reward_to_go --use_baseline --gae_lambda 0.97 \
    --exp_name humanoid --video_log_freq 5
```

We've enabled videos so you can enjoy the results! If everything works correctly, you should end up with a video of a 3D humanoid walking (or stumbling/skipping/running) along. It should achieve a final return of at least 600, but try to

WARNING! This will take a long time to run (as much as 20 hours). Do not attempt this experiment until you finish **ALL** of the other problems! It will not work if you have any bugs! **If it does not reach return 300 by 20 iterations, you probably have a bug!**

You are welcome to use any techniques you are familiar with to make training faster. We recommend:

- The vectorized Gym interface, to use multiple cores for simulating episodes
- Fast operations like `torch.cumsum` to speed up your GAE calculations

This experiment may take 10+ hours to run.

8 Analysis

Consider the following infinite-horizon MDP:

$$\begin{tikzcd} s_1 \arrow[loop left, "a_1"] \arrow[r, "a_2"] & s_F \end{tikzcd}$$

At each step, the agent stays in state s_1 and receives reward 1 if it takes action a_1, and receives reward 0 and terminates the episode otherwise. Parametrize the policy as stationary (not dependent on time) with a single parameter:

$$\pi_\theta(a_1|s_1) = \theta, \pi_\theta(a_2|s_1) = 1 - \theta$$

1. Applying policy gradients
   (a) Use policy gradients to compute the gradient of the expected return $J(\theta) = \mathbb{E}_{\pi_\theta} R(\tau)$ with respect to the parameter θ. **Do not use discounting.**

   Hint: to compute $\sum_{k=1}^{\infty} k\alpha^{k-1}$, you can write:

$$\sum_{k=1}^{\infty} k\alpha^{k-1} = \sum_{k=1}^{\infty} \frac{d}{d\alpha}\alpha^k = \frac{d}{d\alpha}\sum_{k=1}^{\infty}\alpha^k$$

   (b) Compute the expected return of the policy $\mathbb{E}_{\tau\sim\pi_\theta} R(\tau)$ directly. Compute the gradient of this expression with respect to θ and verify that this matches the policy gradient.

2. Compute the variance of the policy gradient in closed form and describe the properties of the variance with respect to θ. For what value(s) of θ is variance minimal? Maximal? (Once you have an exact expression for the variance you can eyeball the min/max).

   Hint: Once you have it expressed as a sum of terms $P(\theta)/Q(\theta)$ where P and Q are polynomials, you can use a symbolic computing program (Mathematica, SymPy, etc) to simplify to a single rational expression.

3. Apply return-to-go as an advantage estimator.
   (a) Write the modified policy gradient and confirm that it is unbiased.
   (b) Compute the variance of the return-to-go policy gradient and plot it on $[0, 1]$ alongside the variance of the original estimator.

4. Consider a finite-horizon H-step MDP with sparse reward:

$$\begin{tikzcd} s_1 \arrow[r, "a_1"] \arrow[rrd, "a_2"'] & s_2 \arrow[r, "a_1"] \arrow[rd, "a_2"] & s_3 \arrow[r, "a_1"] \arrow[d, dashed, "a_2"] & \cdots \arrow[r] \arrow[ld, dashed, "a_2"] & s_H \\ & & s_F & & \end{tikzcd}$$

   The agent receives reward $R_{\max}$ if it arrives at s_H and reward 0 if it arrives at s_F (a terminal state). In other words, the return for a trajectory τ is given by:

$$R(\tau) = \begin{cases} 1 & \tau \text{ ends at } s_H \\ 0 & \tau \text{ ends at } s_F \end{cases}$$

   Using the same policy parametrization as above, consider off-policy policy gradients via importance sampling. Assume we want to compute policy gradients for a policy π_θ with samples drawn from $\pi_{\theta'}$.
   (a) Write the policy gradient with importance sampling.
   (b) Compute its variance. How does it change when H becomes large?

9 Survey

Please estimate, in minutes, for each problem, how much time you spent (a) writing code and (b) waiting for the results. This will help us calibrate the difficulty for future homeworks.

- **Policy Gradients:**
- **Neural Network Baseline:**
- **Generalized Advantage Estimation:**
- **Hyperparameters and Sample Efficiency:**
- **Humanoid:**
- **Analysis:**
  1.1 **Applying policy gradients:**
  1.2 **Policy gradient variance:**
  1.3 **Return-to-go:**
  1.4 **Importance sampling:**

10 Submission

10.1 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).

(d) Your solutions from the analysis problems in section 8

10.2 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 `cs285/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 `cs285` folder with all the `.py` files, with the same names and directory structure as the original homework repository (excluding the `cs285/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-v4_12-09-2019_00-17-58
│  │  └─ events.out.tfevents.1567529456.e3a096ac8ff4
│  └─ ...
├─ cs285
│  ├─ agents
│  │  ├─ bc_agent.py
│  │  └─ ...
│  ├─ policies
│  │  └─ ...
│  └─ ...
├─ README.md
└─ ...
```

10.3 Turning it in

Turn in your assignment by the deadline on Gradescope. Uploade the zip file with your code to **HW2 Code**, and upload the PDF of your report to **HW2**.