CSE 579-AU24: Intelligent Control through Learning and Optimization
Homework 2 - Policy Gradient and Actor Critic

Due November 10 @ 11:59pm PST

The goal of this homework is to get a basic understanding of on-policy and off-policy reinforcement learning. In particular, you will be implementing policy gradient and actor critic algorithms. Please refer to the git link for the assignment `https://github.com/WEIRDLabUW/CSE579-hw2/tree/main`

Related Lectures: This homework largely involves knowledge from Lecture 4 to Lecture 7 on policy gradient and off-policy RL.

Collaboration: Students can discuss questions, but each student MUST write up their own solution, and code their own solution. We will be checking code/PDFs for plagiarism.

Late Policy: This assignment may be handed in up to 3 days late. If you have used up your 10 late days this quarter, there will be a penalty of 10% of the maximum grade per day.

1 Code Overview

The starter code is written in Python and depends on NumPy and Matplotlib as well as Pytorch. If you are new to Pytorch, please refer to the following tutorial [Link]. The README describes how to install packages in a conda environment to solve this assignment. We also provide a link to a Google Colab `https://colab.research.google.com/drive/1PmagnEr-OOMuIzNr25csAumyJ3Z6UpLQ?authuser=4` notebook below that can be used for cloud execution. We strongly recommend using either a Linux machine or Colab rather than Windows or Mac.

- `main.py` - overall launcher with environment creation and policy alternation [DO NOT MODIFY except for hyper parameter tuning]
- `environment.yml` - Conda env file to install dependencies [DO NOT MODIFY]
- `rollouts.py` - Evaluating learned policy reward and success rate [DO NOT MODIFY]
- `policy_gradient.py` - Code for implementing policy gradient [TODOs]
- `actor_critic.py` - Code for implementing actor critic [TODOs]
- `sac.py` - Code for implementing Soft Actor Critic [TODOs]
- `agents.py, networks.py` - Modules and networks for the SAC and AC algorithm [DO NOT MODIFY]

Runtime on Colab Using a t4 GPU and the default hyper parameters with pendulum, it should take around 20 minutes for policy gradient and 12-15 minutes for actor critic.

Environment Details This is the same conda environment as in HW1, so feel free to reuse that environment. Otherwise, to get started, you must install the correct dependencies for the code to run. There are several

ways to do this - if you have a Linux machine, install a conda environment (install Anaconda from here `https://docs.anaconda.com/free/anaconda/install/index.html`) with the correct dependencies from the provided environment.yml file using the following command

```
conda env create -f environment.yml
conda activate cse542a1
```

This should give you an environment with all the necessary dependencies, and you can run the code by running `main.py` as described in each section below.

If you do not have a Linux machine, it is advisable for you to use Google Colab instead. This will provide you with a single GPU instance that should be sufficient to run this code. The Colab instance can be found `https://colab.research.google.com/drive/1PmagnEr-OOMuIzNr25csAumyJ3Z6UpLQ#scrollTo=OlrpNThnUqJg`, and all the installs are completed by just running the initial cells inline. Please talk to the course staff if you have environment setup issues!

The Inverted Pendulum environment is a simulation of a classic control problem called the inverted pendulum. The objective is to control the movement of an inverted pendulum by applying appropriate forces to keep it balanced. You may refer to this following link for further details: [Link]. You may find this [link] useful as a reference for policy gradient.

We also use the Ant Environment, which simulates an ant walking forward. The objective is to apply the appropriate forces to stay upright and move towards the right

2 Policy Gradient [30 points]

```
$ python main.py --task policy_gradient
```

In this part, you will implement Policy Gradient, specifically REINFORCE (likelihood ratio), with a baseline for variance reduction. Recall from the lecture that policy gradient methods are high variance and can be reduced by using a learned baseline. Our goal here is to construct an objective that maximizes the expected sum of rewards. This can be computed as the sum of discounted rewards over a trajectory (look in the code for more hints). The optimization can be done by forming a surrogate objective at each step, which is the negative log-likelihood of actions given states, multiplied by the discounted return to go. This surrogate objective can then be optimized using standard PyTorch optimizers, like Adam, to update parameters. This process can be repeated to learn an optimal policy. We have added a detailed pseudo code at the bottom of this document if you need it, but try to do the homework without looking!

2.1 Expected Results

We provide an example below which serves as a reference for the expected output; please note that the success rate and reward may fluctuate:

```
$ python main.py --task policy_gradient --env pendulum
using device cuda
Episode: 0, reward: 6.984375, max path length: 20
Episode: 10, reward: 11.328125, max path length: 48
Episode: 20, reward: 19.140625, max path length: 58
Episode: 30, reward: 35.828125, max path length: 97
Episode: 40, reward: 74.21875, max path length: 140
```

```
Episode: 50, reward: 136.484375, max path length: 200
Episode: 60, reward: 188.828125, max path length: 200
Episode: 70, reward: 197.0, max path length: 200
...
Success rate:  0.97
Average reward (success only):  200.0
Average reward (all):  199.77
```

2.2 Execution

1. Fill in the blanks in the code marked with TODO in the `policy_gradient` function in `policy_gradient.py`.
2. Plot the reward during the training with default hyperparameters. Report the final success rate and average reward using the evaluate function that is provided to you.
3. Try training policy gradient without subtracting the baseline when computing returns and/or without normalizing the returns. Plot the reward during the training. Report success rate and average reward using the evaluate function that is provided to you. How does performance differ?

3 Actor Critic [30 points]

```
$ python main.py --task actor_critic --env pendulum
```

In this part, you will be implementing actor-critic algorithms. Recall from the class that on-policy RL methods such as policy gradient you've implemented in Part 1, are sample inefficient both because we cannot effectively use past data, as well as suffering from high variance. In contrast, off-policy actor-critic methods can update the value function and policy using data from a replay buffer, which stores all the past experiences - both being off-policy and lowering variance.

An actor critic algorithm has two main components: an actor (policy) and a critic (Q function). The actor learns a policy that selects actions based on the current state, while the critic estimates the Q function to evaluate the quality of the actions taken by the actor. The Q function will be learned by optimizing the Bellman error, while the actor will be learned by trying to find actions that maximize expected Q values. In this homework, you will fill in two main things - (1) the Bellman error for the critic loss that enables learning of the Q function. This is the off-policy Q Bellman error we discussed in lecture. (2) the actor loss that computes the expected Q values for actions sampled according to the policy, and tries to maximize these. Note that we will do this via reparameterization, so you will need to fill in ACPolicy (in utils.py) to sample actions via reparameterization, and then use this to compute the reparameterized actor loss. More hints in the code! Remember you are using a "target" network on the RHS of your Bellman equation. We have added detailed pseudocode at the bottom of this document if you need it, but try to do the homework without looking!

3.1 Expected Results

When you execute the code, you may get outputs similar to the ones below. Please note that the success rate and reward may fluctuate. The actor entropy are set to zero as they are not part of the actor-critic algorithm

```
$ python main.py --task actor_critic
using device cuda
step 5000, batch_r 0.8398, critic_l 0.8139, actor_l -0.001, actor_ent 0, alpha_l 0
step 10000, batch_r 0.9375, critic_l 0.3434, actor_l -20.4693, actor_ent 0, alpha_l 0
```

```
step 15000, batch_r 0.9727, critic_l 0.0119, actor_l -26.8561, actor_ent 0, alpha_l 0
eval step 15076, average episode reward 88.6, average episode length 89.6
step 20000, batch_r 1.0, critic_l 0.0094, actor_l -44.599, actor_ent 0, alpha_l 0
...
eval step final, average episode reward 200.0, average episode length 200.0
```

3.2 Execution

1. Fill in the blanks in the code marked with TODO in `actor_critic.py`.
2. Plot the return during the training with default hyperparameters. Report the final average reward using the evaluate function that is provided to you.
3. Try training actor critic where target Q-values are calculated not through a target Q-network but the current Q-network. Plot the reward during the training. Report success rate and average reward using the evaluate function that is provided to you. How does performance differ?

4 Soft Actor Critic [30 points]

In this part, you will implement the Soft Actor-Critic (SAC) algorithm, a popular off-policy actor-critic method (paper). You might find this blog post very helpful. SAC builds upon the traditional actor-critic framework but introduces two key differences that enhance its performance:

Entropy Regularization: Unlike standard actor-critic methods, SAC incorporates an entropy augmented objective.

$$J(\pi) = \mathbb{E}_{\pi}\left[\sum_{t} r(s_t, a_t) - \alpha \log(\pi(a_t \mid s_t))\right] \tag{1}$$

This makes the optimal policy not only the expected return but also the expected entropy of its action distributions. This encourages exploration by incentivizing the policy to choose more diverse actions, leading to a more robust and exploratory learning process. This α can be fixed but is a learned parameter in our implementation, which keeps our entropy close to a specified target level.

Double Q Networks: An additional improvement is that SAC uses multiple Q functions to help prevent optimistic Q value estimations. This essentially occurs when out-of-distribution states are passed into the Q function; you can get an overly optimistic approximation. To fix this, SAC uses two Q networks and uses the minimum Q value predicted to improve performance.

5 Execution

1. Fill in the blanks in the code marked with the TODO in sac.py. We have provided Psudeo code in the appendix similar to the other problems.
2. Plot the batch reward during training on the pendulum env with the default hyperparameters. Report the average reward on your final trained policy using the evaluate function provided to you.
3. We have also included the Ant environment, a much harder control problem than the pendulum. Increase the number of train steps for AC and for SAC to 100,000. Run both of these algorithms on the ant environment and compare the results of the two methods. Explain why you think the performance differs and why this is the case.

5.1 expected results

```
$python main.py --task sac --env pendulum
using device cuda
step 5000, batch_r 0.832, critic_l 1.478, actor_l 0.0122, actor_ent -0.2915, alpha_l 0.0709
step 10000, batch_r 0.9258, critic_l 1.8954, actor_l -17.7841, actor_ent -0.6569, alpha_l 0.013
step 15000, batch_r 0.9648, critic_l 0.0236, actor_l -27.4914, actor_ent -0.9386, alpha_l 0.0009
eval step 15110, average episode reward 200.0, average episode length 200.0
...
eval step final, average episode reward 200.0, average episode length 200.0
```

6 Discussion [10 points]

1. Describe the role of the value function in actor-critic methods and how it differs from policy gradient methods.
2. Discuss the advantages and disadvantages of policy gradient, actor-critic, and soft-actor-critic methods in terms of sample efficiency, stability, and convergence properties.

7 Submission

We will be using Canvas to submit the assignments. Please submit the written assignment answers as a PDF. For the code, submit a zip file of the entire working directory.

8 Appendix

8.1 Policy Gradient Pseudocode

The following pseudocode may be helpful: You are expected to get a success rate high than 0.8 and average reward (all) higher than 180 for the policy gradient method on the pendulum task. The following pseudocode may be helpful:

```
def policy_gradient():
    initialize policy neural network
    instantiate baseline neural network
    for i in range(max_num_iters):
        trajectories = rollout(policy) // roll out current learned policy
        returns = compute_returns(policy) // compute return to go from observations
        returns = normalize(returns) // normalize returns by subtracting mean, dividing by std
        for i in range(baseline_training_iters): // train baseline via regression
            baseline_prediction = baseline(trajectories['observations'])
            loss_baseline = MSE(baseline_prediction, returns)
            loss_baseline.update() // update baseline only

        baseline_prediction = baseline(trajectories['observations']) // compute final baseline
            prediction

        mean_predicted, std_predicted = policy(trajectories['observations'])
```

```
        log_probs = log_density(trajectories['actions'], mean_predicted, std_predicted)

        loss_policy = -log_probs*(returns - baseline_prediction)
        loss_policy.update() // update policy only
    return
```

Note the compute returns function is a little tricky - at every step t, compute the return to go as $\sum_{t'=t}^{T} \gamma^{t'-t} r(s_{t'})$. This essentially starts the return to go from $\gamma^0 = 1$ and keeps discounting by γ as time goes on. More hints on the code on how to do this using numpy cumulative sums.

8.2 Actor Critic Pseudo-code

You are expected to get a success rate higher than 0.8 and an average reward (all) higher than 180 for the actor-critic method on the pendulum task. The following pseudocode may be helpful:

```
def update_actor(observation):
    dist = actor(observation)
    action = sample from dist
    actor_Q = critic(observation, action)
    // The loss is just the negative Q values.
    actor_loss = -actor_Q

def update_critic(obs, action, reward, next_obs, not_done)
    q_pred = self.critic(obs, action)
    // We do not want gradient flow through the target Q. Use torch.no grad to disable gradient when
        calculating it.
    next_dist = actor(next_obs)
    next_action = sample next_dist
    next_Q = target_critic(next_obs, next_action)
    target_Q = reward + not_done_no_max * discount * next_Q

    critic_loss = MSE(q_pred, target_Q)
```

8.3 Soft Actor-Critic Psudeocode

This is the pseudo-code for soft actor-critic that includes calculating the loss for both the actor, the critic, and the α parameter. If you do not want the gradient to flow through something, use the .detach() method in the torch.

```
def update_actor(observations):
    // Sample actions from the actor-network and calculate log likelihood
    actor_distritution = actor_network(observations)
    actions = sample from actor distribution
    log_prob = actor_distribution.log_prob(actions)

    // get the Q values:
    q_vals = min(double_Q_Network(observations, actions))

    // calculate actor loss. Note we do not want the gradient to flow through α here.
    // The first term is the entropy term, and the second is the actor loss from the standard
        actor-critic.
    actor_loss = α * log_prob - q_vals
```

```
    // Optimize the alpha parameter to be closer to our target entropy. Note we do not want the
        gradient to flow through the log probs here.
    α_loss = α * (-log_prob - target_entropy)

def update_critic(obs, action, reward, next_obs, not_done)
    // We do not want the gradient to flow through the target_Q. You can use torch.no_grad to
        calculate it
    next_action_dist = actor_network(next_obs)
    next_action = sample from next_action_dist
    next_log_prob = next_action_dist.log_prob(next_action)
    target_Q1, target_Q2 = self.target_critic(next_obs, next_action)
    // We do not want the gradient to flow through the alpha parameter
    target_V = min(target_Q1, target_Q2) - α * log_prob
    // use not_done to account for terminal state
    target_Q = reward + (not_done * discout_factor * target_V)

    // get current estimates:
    current_Q1, current_Q2 = self.critic(obs, action)
    critic_loss = MSE(current_Q1, target_Q) +MSE(current_Q2, target_Q)
```