CS 277: Control and Reinforcement Learning (Winter 2021)
Assignment 5

Due date: Friday, March 12, 2021 (Pacific Time)

Roy Fox
https://royf.org/crs/W21/CS277/

General instructions: In theory questions, a formal proof is not needed (unless specified otherwise); instead, briefly explain informally the reasoning behind your answers. In practice questions, include a printout of your code as a page in your PDF, and a screenshot of TensorBoard learning curves (`episode_reward_mean`, unless specified otherwise) as another page.

Part 1 Actor–Critic vs. Control-as-Inference (50 points + 10 bonus)

Recall that Actor–Critic algorithms represent an actor π_θ and a critic V_ϕ. Many such algorithms use a temporal-difference loss to update the critic and a policy-gradient loss to update the actor. The simplest such algorithm we saw gathers on-policy experience (s, a, r, s'), and then uses the critic's Bellman error $\delta = r + \gamma V_{\bar{\phi}}(s') - V_\phi(s)$ (with $V_{\bar{\phi}}$ a target network) to compute the critic loss $\mathcal{L}_\phi = \frac{1}{2}\delta^2$; and uses the critic's advantage estimate $\hat{A} = r + \gamma V_\phi(s') - V_\phi(s)$ to compute the actor loss $\mathcal{L}_\theta = \log \pi_\theta(a|s)\hat{A}$. The algorithm then employs any gradient-based optimizer on the total loss $\mathcal{L}_{\text{AC}} = \mathcal{L}_\phi + \eta\mathcal{L}_\theta$, with η a coefficient relating the two losses.

Also recall that, in the Control-as-Inference framework, the optimal policy is

$$\pi(a|s) = \frac{\pi_0(a|s) \exp \beta Q(s,a)}{Z(s)}, \tag{1}$$

with the normalizer ("partition function") $Z(s) = \mathbb{E}_{a|s\sim\pi_0}[\exp \beta Q(s,a)]$. If this policy is plugged into the bounded Bellman optimality equation, we get

$$V(s) = \frac{1}{\beta} \log Z(s) = \frac{1}{\beta} \log \mathbb{E}_{a|s\sim\pi_0}[\exp \beta Q(s,a)].$$

The SQL algorithm represents a Q-value function Q_θ, and given off-policy experience (s, a, r, s') uses the Bellman error $\tilde{\delta} = r + \gamma V_{\bar{\theta}}(s') - Q_\theta(s,a)$ to compute the loss $\mathcal{L}_{\text{SQL}} = \frac{1}{2}\tilde{\delta}^2$. Here $V_{\bar{\theta}}$ is computed by

$$V_{\bar{\theta}} = \frac{1}{\beta} \log \mathbb{E}_{a|s\sim\pi_0}[\exp \beta Q_{\bar{\theta}}(s,a)] \tag{2}$$

from a target network $Q_{\bar{\theta}}$.

1. Rearranging (1), we get

$$Q(s,a) = V(s) + \frac{1}{\beta} \log \frac{\pi(a|s)}{\pi_0(a|s)}. \tag{3}$$

   Consider implementing the SQL algorithm by parametrizing $Q_{\theta,\phi}$ as the function (3) of an actor network π_θ and a critic network V_ϕ. There is no dedicated network for Q (beyond the actor and critic networks), and no target network for Q. Instead, there is a target critic network $V_{\bar{\phi}}$, which is used in the Bellman error $\tilde{\delta}$ instead of $V_{\bar{\theta}}$ (i.e. (2) is not used now). Write down the SQL loss $\mathcal{L}_{\text{SQL}}$ in terms of this $Q_{\theta,\phi}$. (15 points)

2. Write an expression for a pseudo-reward[1] $\tilde{r}$, such that the AC critic loss $\mathcal{L}_\phi$, with $\tilde{r}$ substituted for r, is equivalent to the SQL loss $\mathcal{L}_{\text{SQL}}$. (15 points)

3. Show that the gradient $\nabla_\theta \mathcal{L}_\theta$ of the AC actor loss $\mathcal{L}_\theta$, with $\tilde{r}$ substituted for r, is almost[2] equivalent to the gradient $\nabla_\theta \mathcal{L}_{\text{SQL}}$ of the SQL loss with respect to the policy parameters θ. (10 points)

4. Put the previous two questions together by showing that, with $\tilde{r}$ substituted for r in AC, the gradient of $\mathcal{L}_{\text{AC}}$ is (almost) equivalent to the gradient of $\mathcal{L}_{\text{SQL}}$ with respect to both θ and ϕ. What is the equivalent of β in AC? (10 points)

5. (**Bonus**) (Warning: hard question) AC is an on-policy algorithm (can only work with on-policy data), while SQL is an off-policy algorithm. This means that the above equivalence only holds for on-policy data. Which part of this equivalence fails for off-policy data? (10 points)

Part 2 Option–Critic algorithm (50 points)

1. Download the following implementation of the Option–Critic algorithm: https://github.com/alversafa/option-critic-arch. Read `option_critic.ipynb`, and make the following changes:

   (a) In parts 3 and 4, add color bars (see `matplotlib.pyplot.colorbar`) to the heat maps.

   (b) In part 4, plot the following three histograms:

      i. For each option h (on the x-axis), the number of times option h was called in an episode.

      ii. For each option h (on the x-axis), the average number of actions option h took each time it was called before it terminated.

      iii. For each option h (on the x-axis), the total number of actions it took in an episode (summed over all times it was called).

   In each of these histograms, plot the average and standard-deviation error bars over 10 episodes.

   Run the code, and attach the resulting plots. (10 points)

2. Is the agent high-fitting (i.e. a single option solves much of the entire task)? Is it low-fitting (i.e. options terminate very quickly, such that the meta-policy solves much of the entire task)? Explain which results make you think so and why. (5 points)

3. One way to reduce high-fitting is to make the options simpler. In the next question, you'll implement options that try to move towards a single position μ_h in 2D space. Specifically, the action distribution is

$$\pi_{\mu_h}(a|s) = \frac{\exp(-d(f(s,a),\mu_h))}{\sum_{\bar{a}} \exp(-d(f(s,\bar{a}),\mu_h))},$$

   where $f(s,a)$ is the state that would follow s when action a is taken (if there weren't walls), and $d(s_1,s_2) = \frac{1}{2}\|s_1 - s_2\|_2^2$. Note that the option policy is now parametrized by μ_h. Recall that the option policy gradient in the Option–Critic algorithm is $\nabla_{\mu_h} \mathcal{L}_h(s,a) = -\nabla_{\mu_h} \log \pi_h(a|s) Q_h(s,a)$. Write the expression for the gradient in the case of the above policy. (10 points)

4. In this question, you'll implement the option class above. Read the implementation of the current option policy class `utils.SoftmaxPolicy`. It parametrizes the policy with parameters $\theta_{s,a}$ such that the softmax policy is

$$\pi_\theta(a|s) = \frac{\exp \tau^{-1}\theta_{s,a}}{\sum_{\bar{a}} \exp \tau^{-1}\theta_{s,\bar{a}}},$$

   where τ is a temperature hyperparameter. The class has the following methods:

[1] $\tilde{r}$ is called a pseudo-reward because it's not a fixed function of s and a, but may change during the run of the algorithm.

[2] It would be exactly equivalent if $\hat{A}$ in AC was also using the critic's target network for $V_{\bar{\phi}}(s')$.

- The method `Q_U` just returns the parameters, and is poorly named so don't get confused — it's not returning Q values at all.
- The method `pmf` takes the parameters and applies softmax to get $\pi_\theta(a|s)$ for all actions a in a given state s. Computing softmax can be numerically unstable if parameters become very large or very small, so notice how this function uses logsumexp to compute this in a numerically stable way.
- The method `sample` then samples an action for a given state.
- The method `update` takes an option policy gradient step over the parameters. It's argument `Q_h` is the same as what we called Q_h, and is provided by the critic. Note that, in the original parameterization, $\mathcal{L}_\theta(s,a)$ for a given state s and action a depends only on $\theta_{s,a}$ for that action and $\theta_{s,\tilde{a}}$ for other actions. The gradient therefore only touches those parameters, and for them:

  $$\nabla_{\theta_{s,\tilde{a}}} \mathcal{L}_\theta(s,a) = Q_h(s,a) \nabla_{\theta_{s,\tilde{a}}} \log \sum_{\bar{a}} \exp \tau^{-1} \theta_{s,\bar{a}} = \tau^{-1} \pi(\tilde{a}|s) Q_h(s,a),$$

  and similarly

  $$\nabla_{\theta_{s,a}} \mathcal{L}_\theta(s,a) = -\tau^{-1}(1-\pi(a|s)) Q_h(s,a).$$

  Note how the current code implements this update.

Based on this, implement the new option class, with the new parametrization μ_h. The code for the Option–Critic algorithm will only use methods `sample` and `update` of your class, but you can have any other methods that you find helpful. Some things to note:

- You can initialize the option policy parameters μ_h however you want.
- The `state` argument is an integer. To get the (x,y) position in the grid world, you can use `env.tocell` (see here: https://github.com/alversafa/option-critic-arch/blob/master/fourrooms.py#L42). It may help to pass the `env` to the policy object constructor.
- The `action` argument is also an integer. To get the direction in the grid world, you can use `env.directions`.
- The state that follows `env.tocell(state)` when taking action `env.directions(action)` is their sum (if it's not a wall, but for the purpose of the policy just take their sum).
- Remember to *descend* on the loss.

Replace the `option_policies` with your implementation. Compare the results of your code with different numbers of options, and compare to the original code. (25 points)