Assignment 3: Q-Learning and Actor-Critic Algorithms

27th March 2025

1 Multistep Q-Learning

Consider the N-step variant of Q-learning described in lecture. We learn $Q_{\phi_{k+1}}$ with the following updates:

$$y_{j,t} \leftarrow \left(\sum_{t'=t}^{t+N-1} \gamma^{t'-t} r_{j,t'} \right) + \gamma^N \max_{\mathbf{a}_{j,t+N}} Q_{\phi_k} \left(\mathbf{s}_{j,t+N}, \mathbf{a}_{j,t+N} \right) \tag{1}$$

$$\phi_{k+1} \leftarrow \arg\min_{\phi \in \Phi} \sum_{j,t} \left(y_{j,t} - Q_\phi(\mathbf{s}_{j,t}, \mathbf{a}_{j,t}) \right)^2 \tag{2}$$

In these equations, j indicates an index in the replay buffer of trajectories $\mathcal{D}_k$. We first roll out a batch of B trajectories to update $\mathcal{D}_k$ and compute the target values in (1). We then fit $Q_{\phi_{k+1}}$ to these target values with (2). After estimating $Q_{\phi_{k+1}}$, we can then update the policy through an argmax:

$$\pi_{k+1} \left(\mathbf{a}_t \mid \mathbf{s}_t \right) \leftarrow \begin{cases} 1 \text{ if } \mathbf{a}_t = \arg\max_{\mathbf{a}_t} Q_{\phi_{k+1}} \left(\mathbf{s}_t, \mathbf{a}_t \right) \\ 0 \text{ otherwise.} \end{cases} \tag{3}$$

We repeat the steps in eqs. (1) to (3) K times to improve the policy. In this question, you will analyze some properties of this algorithm, which is summarized in **??** 1.

Algorithm 1: Multistep Q-Learning

Input: Iterations K, batch size B

Initialize random policy π_0, sample $\phi_0 \sim \Phi$;
for $k \leftarrow 0$ **to** $K-1$ **do**
  Update $\mathcal{D}_{k+1}$ with B new rollouts from π_k;
  Compute targets with (1);
  $Q_{\phi_{k+1}} \leftarrow$ update with (2);
  $\pi_{k+1} \leftarrow$ update with (3);
return π_K;

1.1 TD-Learning Bias (2 points)

We say an estimator $f_{\mathcal{D}}$ of f constructed using data $\mathcal{D}$ sampled from process P is *unbiased* when $\mathbb{E}_{\mathcal{D} \sim P}[f_{\mathcal{D}}(x) - f(x)] = 0$ at each x.

Assume $\hat{Q}$ is a noisy (but unbiased) estimate for Q. Is the Bellman backup $\mathcal{B}\hat{Q} = r(s,a) + \gamma \max_{a'} \hat{Q}(s',a')$ an unbiased estimate of $\mathcal{B}Q$?

☐ Yes

■ No

Answer

An unbiased estimator is expected value equals the true value. $\mathbb{E}[\mathcal{B}\hat{\mathcal{Q}}] = \mathcal{B}\mathcal{Q}$. so, $\mathbb{E}[\mathcal{B}\hat{\mathcal{Q}}(s',a')] = Q(s',a')$.

However, max operator introduces bias. The equality of $\mathbb{E}[f(X)] \geq f(\mathbb{E}[X])$ holds only if f is linear or X is deterministic. since $\hat{Q}$ is noisy, $\max_{a'} \hat{Q}(s',a')$ tends to overestimate $\max_{a'} Q(s',a')$. (random fluctuations favor higher values).

Thus, $\mathbb{E}[\mathcal{B}\hat{\mathcal{Q}}] \geq \mathcal{B}\mathcal{Q}$.

1.2 Tabular Learning (6 points total)

At each iteration of the algorithm above after the update from eq. (2), Q_{ϕ_k} can be viewed as an estimate of the true optimal Q^*. Consider the following statements:

I. $Q_{\phi_{k+1}}$ is an unbiased estimate of the Q function of the last policy, Q^{π_k}.

II. As $k \to \infty$ for some fixed B, Q_{ϕ_k} is an unbiased estimate of Q^*, i.e., $\lim_{k\to\infty} \mathbb{E}\big[Q_{\phi_k}(s,a) - Q^*(s,a)\big] = 0$.

III. In the limit of infinite iterations and data we recover the optimal Q^*, i.e., $\lim_{k,B\to\infty} \mathbb{E}\left[\|Q_{\phi_k} - Q^*\|_\infty\right] = 0$.

We make the additional assumptions:

- The state and action spaces are finite.
- Every batch contains at least one experience for each action taken in each state.
- In the tabular setting, Q_{ϕ_k} can express any function, i.e., $\{Q_{\phi_k} : \phi \in \Phi\} = \mathbb{R}^{S\times A}$.

When updating the buffer $\mathcal{D}_k$ with B new trajectories in **??** of **??** 1, we say:

- When learning *on-policy*, $\mathcal{D}_k$ is set to contain only the set of B new rollouts of π (so $|\mathcal{D}_k| = B$). Thus, we only train on rollouts from the current policy.
- When learning *off-policy*, we use a fixed dataset $\mathcal{D}_k = \mathcal{D}$ of B trajectories from another policy π'.

Indicate which of the statements **I-III** always hold in the following cases. No justification is required.

	I.	**II.**	**III.**
1. $N = 1$ and ...			
(a) on-policy in tabular setting	☐	☐	■
(b) off-policy in tabular setting	☐	■	■
2. $N > 1$ and ...			
(a) on-policy in tabular setting	☐	☐	■
(b) off-policy in tabular setting	☐	☐	■
3. In the limit as $N \to \infty$ (no bootstrapping) ...			
(a) on-policy in tabular setting	■	☐	■
(b) off-policy in tabular setting	☐	☐	☐

Answer

1. $N = 1$
   (a) on-policy in tabular setting
      I. No

      $y_{j,t} \leftarrow r_{j,t'} + \gamma \max_{a'_{j,t+1}} Q_{\phi_k}(s_{j,t+1}, a_{j,t+1})$. At N=1, when updating $Q_{\phi_{k+1}}$, we use Q_{ϕ_k}. Therefore, for the last policy π_k, $Q^{\pi_k} \neq Q_{\phi_k}$, so it is biased.

      II. No

      Since this is an on-policy $\mathcal{B}$, even as $k \to \infty$, Q_{ϕ_k} cannot unbiasedly estimate Q^*.

      III. Yes

      Unlike II, with infinite $\mathcal{B}$, we can observe all transitions. Therefore, Q_{ϕ_k} converges to Q^*.

   (b) off-policy in tabular setting
      i. No

      A fixed dataset $\mathcal{D} = \mathcal{D}_{\pi'}$ and buffer B contains trajectories from another policy π'. When $N = 1$, the TD-target estimates over Q_{current}, which causes bias.

      ii. Yes

      Considering longer horizons, as $N \to \infty$, the cumulative return is accurately estimated, so Q converges to Q^*. Tabular off-policy Q-learning on a fixed dataset with full return converges to Q^* as $t \to \infty$.

      iii. Yes, Better than II.

2. $N > 1$
   (a) on-policy in tabular setting
      I. No. Even though N is large, using bootstrapped Q leads to biased N-step return.

      II. No

      The fixed buffer B (generated under the current policy) leads to bias in computing N-step targets. $\Rightarrow$ Unbiased estimation is not possible.

      III. Yes. Infinite data corrects the bias over time, eventually converging to Q^*.

   (b) off-policy in tabular setting
      I. No. Target reflect π', not π_k. (same as $N = 1$).

      II. No. Even if the policy improves, the fixed buffer B prevents unbiased estimation of Q^*.

      III. Yes

3. $N = \infty$
   (a) on-policy in tabular setting
      I. Yes. Without bootstrapping, $y_{j,t}$ is the discounted return of π_k, making $Q_{\phi_{k+1}}$ unbiased for Q^{π_k}.

      II. No. Policy improves. But the fixed buffer B prevents unbiased estimation of Q^*.

      III. Yes

   (b) off-policy in tabular setting
      I. No. Returns are computed under π', not π_k.

      II. No. Fixed π' and buffer B limit convergence to Q^*.

      III. No. Without importance sampling, infinite N on $\pi' \neq \pi^*$ yields $Q^{\pi'}$, not Q^*.

1.3 Variance of Q Estimate (2 points)

Which of the three cases ($N = 1$, $N > 1$, $N \to \infty$) would you expect to have the highest-variance estimate of Q for fixed dataset size B in the limit of infinite iterations k? Lowest-variance?

Highest variance:

- ☐ $N = 1$
- ☐ $N > 1$
- ■ $N \to \infty$

Lowest variance:

- ■ $N = 1$
- ☐ $N > 1$
- ☐ $N \to \infty$

Answer

Variance increases with N.

As N increases, the target value $y_{j,t}$ includes more reward terms, each contributing additional noise. With a fixed dataset size B, we have fewer samples per state-action pair, which further amplifies this variance effect.

1.4 Function Approximation (2 points)

Now say we want to represent Q via function approximation rather than with a tabular representation. Assume that for any deterministic policy π (including the optimal policy π^*), function approximation can represent the true Q^π exactly. Which of the following statements are true?

- ☐ When $N = 1$, $Q_{\phi_{k+1}}$ is an unbiased estimate of the Q-function of the last policy Q^{π_k}.
- ■ When $N = 1$ and in the limit as $B \to \infty$, $k \to \infty$, Q_{ϕ_k} converges to Q^*.
- ■ When $N > 1$ (but finite) and in the limit as $B \to \infty$, $k \to \infty$, Q_{ϕ_k} converges to Q^*.
- ■ When $N \to \infty$ and in the limit as $B \to \infty$, $k \to \infty$, Q_{ϕ_k} converges to Q^*.

Answer

With infinite data and iterations, all cases ($N = 1$, $N > 1$, and $N \to \infty$) converge to Q^*. This convergence occurs because the policy becomes optimal through the $\arg\max$ updates, which is guaranteed when function approximation can perfectly represent the true Q-function.

1.5 Multistep Importance Sampling (5 points)

We can use importance sampling to make the N-step update work off-policy with trajectories drawn from an arbitrary policy. Rewrite (2) to correctly approximate a Q_{ϕ_k} that improves upon π when it is trained on data $\mathcal{D}$ consisting of B rollouts of some other policy $\pi'(\mathbf{a}_t \mid \mathbf{s}_t)$.

Do we need to change (2) when $N = 1$? What about as $N \to \infty$?

You may assume that π' always assigns positive mass to each action. [Hint: re-weight each term in the sum using a ratio of likelihoods from the policies π and π'.]

Answer

Recap N-step variant of Q-learning:

$$y_{j,t} \leftarrow \left(\sum_{t'=t}^{t+N-1} \gamma^{t'-t} r_{j,t'} \right) + \gamma^N \max_{\mathbf{a}_{j,t+N}} Q_{\phi_k} \left(\mathbf{s}_{j,t+N}, \mathbf{a}_{j,t+N} \right) \tag{4}$$

$$\phi_{k+1} \leftarrow \arg\min_{\phi \in \Phi} \sum_{j,t} \left(y_{j,t} - Q_\phi(\mathbf{s}_{j,t}, \mathbf{a}_{j,t}) \right)^2 \tag{5}$$

$$\pi_{k+1} \leftarrow \arg\min_{\phi \in \Phi} \sum_{j,t} \prod_{t'=t}^{t+N-1} \frac{\pi(\mathbf{a}_{j,t'}|\mathbf{s}_{j,t'})}{\pi'(\mathbf{a}_{j,t'}|\mathbf{s}_{j,t'})} \left(y_{j,t} - Q_\phi(\mathbf{s}_{j,t}, \mathbf{a}_{j,t}) \right)^2 \tag{6}$$

Both $N = 1$ and $N \to \infty$ need to change equation (5) to (6). But a single-step ration still necessary unless $\pi = \pi'$.
Also, $\prod_{t'=t}^{\infty} \frac{\pi(\mathbf{a}_{j,t'}|\mathbf{s}_{j,t'})}{\pi'(\mathbf{a}_{j,t'}|\mathbf{s}_{j,t'})}$ weights the entire return for $Q_{\phi_k} \to Q^*$ unless $Q_{\phi_k} \to Q^{\pi'}$.

2 Deep Q-Learning

2.1 Implementation

The first phase of the assignment is to implement a working version of Q-learning, with some extra bells and whistles like double DQN. Our code will work with both state-based environments, where our input is a low-dimensional list of numbers (like Cartpole), but we'll also support learning directly from pixels!

In addition to the double Q-learning trick (which you'll implement later), we have a few other tricks implemented to stabilize performance. You don't have to do anything to enable these, but you should look at the implementations and think about why they work.

- **Exploration scheduling for ϵ-greedy actor.** This starts ϵ at a high value, close to random sampling, and decays it to a small value during training.
- **Learning rate scheduling.** Decay the learning rate from a high initial value to a lower value at the end of training.
- **Gradient clipping.** If the gradient norm is larger than a threshold, scale the gradients down so that the norm is equal to the threshold.
- **Atari wrappers.**
  - **Frame-skip.** Keep the same constant action for 4 steps.
  - **Frame-stack.** Stack the last 4 frames to use as the input.
  - **Grayscale.** Use grayscale images.

2.2 Basic Q-Learning

Implement the basic DQN algorithm. You'll implement an update for the Q-network, a target network, and

What you'll need to do:

- Implement a DQN critic update in `update_critic` by filling in the unimplemented sections (marked with TODO(student)).
- Implement ϵ-greedy sampling in `get_action`
- Implement the TODOs in `run_hw3_dqn.py`.

  Hint: A trajectory can end (`done=True`) in two ways: the actual end of the trajectory (usually triggered by catastrophic failure, like crashing), or *truncation*, where the trajectory doesn't actually end but we

stop simulation for some reason (commonly, we truncate trajectories at some maximum episode length). In this latter case, you should still reset the environment, but the `done` flag for TD-updates (stored in the replay buffer) should be false.

- Call all of the required updates, and update the target critic if necessary, in `update`.

Testing this section:

- Debug your DQN implementation on `CartPole-v1` with `experiments/dqn/cartpole.yaml`. It should reach reward of nearly 500 within a few thousand steps.

Deliverables:

- Submit your logs of `CartPole-v1`, and a plot with environment steps on the x-axis and eval return on the y-axis.
- Run DQN with three different seeds on `LunarLander-v2`:

  Your code may not reach high return (200) on Lunar Lander yet; this is okay! Your returns may go up for a while and then collapse in some or all of the seeds.

- Run DQN on `CartPole-v1`, but change the `learning rate` to 0.05 (you can change this in the YAML config file). What happens to (a) the predicted Q-values, and (b) the critic error? Can you relate this to any topics from class or the analysis section of this homework?

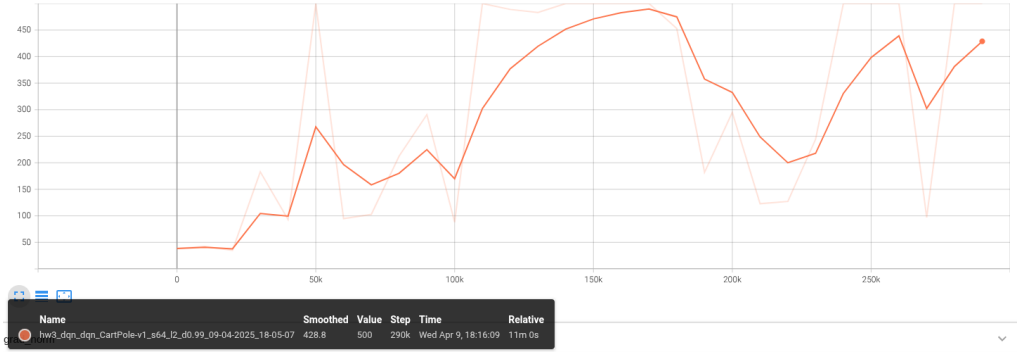

Figure 1: Evaluation return on `CartPole-v1` with DQN.

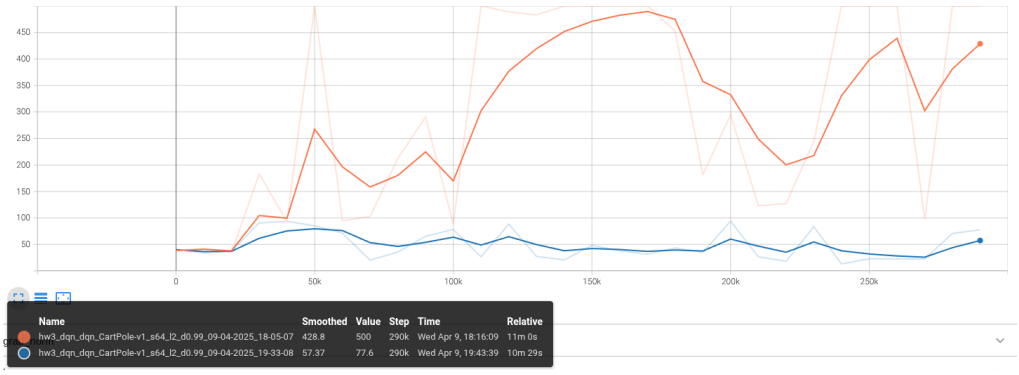

Figure 2: Evaluation return on `CartPole-v1` with DQN using lr 0.05.

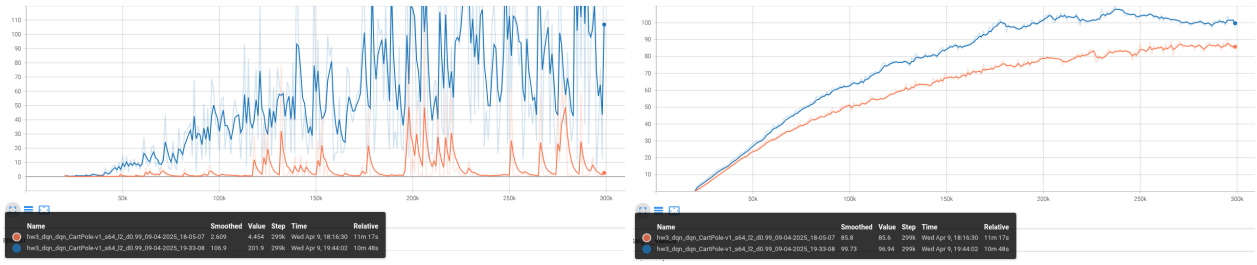

Figure 3: Left: Critic loss, Right: Predicted Q-values on `CartPole-v1`. **Blue** corresponds to a high learning rate (`lr = 0.05`), while **orange** shows the default learning rate (`lr = 0.001`).

Answer

In Figure 3, we can see that the critic loss is much noisy and the predicted Q-values increase rapidly. This is because the learning rate is too high, causing the Q-values to oscillate and diverge. The critic loss also becomes unstable, leading to poor performance in the DQN.

In DQN, the target for training is:

$$y_t = r_t + \gamma \max_{a'} Q_{\text{target}}(s_{t+1}, a')$$

The loss is:

$$\mathcal{L} = (Q(s_t, a_t) - y_t)^2$$

With a high learning rate α, gradient steps $\theta \leftarrow \theta - \alpha \nabla_\theta \mathcal{L}$ are very large, causing the Q-values to update too quickly. This can lead to divergence or oscillation.

2.3 Double Q-Learning

Let's try to stabilize learning. The double-Q trick avoids overestimation bias in the critic update by using two different networks to *select* the next action a' and to *estimate* its value:

$$a' = \arg\max_{a'} Q_\phi(s', a')$$

$$Q_{\text{target}} = r + \gamma(1 - d_t) Q_{\phi'}(s', a').$$

In our case, we'll keep using the target network $Q_{\phi'}$ to estimate the action's value, but we'll select the action using Q_ϕ (the online Q network).

Implement this functionality in `dqn_agent.py`.

Deliverables:

- Run three more seeds of the lunar lander problem:

  You should expect a return of **200** by the end of training, and it should be fairly stable compared to your policy gradient methods from HW2.

  Plot returns from these three seeds in red, and the "vanilla" DQN results in blue, on the same set of axes. Compare the two, and describe in your own words what might cause this difference.

- Run your DQN implementation on the `MsPacman-v0` problem. Our default configuration will use double-Q learning by default. You are welcome to tune hyperparameters to get it to work better, but the default parameters should work (so if they don't, you likely have a bug in your implementation). Your implementation should receive a score of around **1500** by the end of training (1 million steps. **This problem will take about 3 hours with a GPU, or 6 hours without, so start early!**

- Plot the average training return (`train_return`) and eval return (`eval_return`) on the same axes. You may notice that they look very different early in training! Explain the difference.

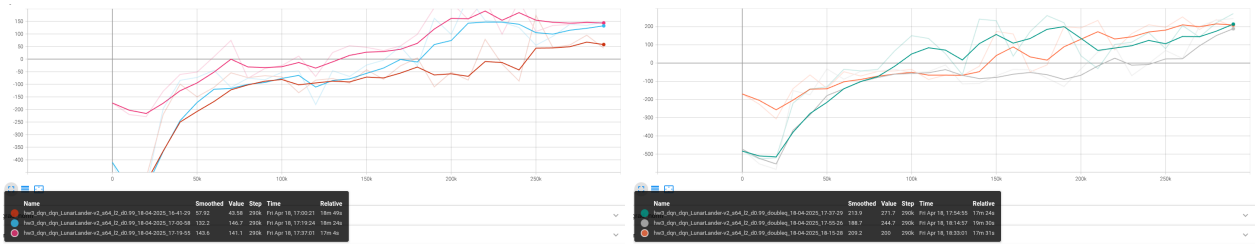

Figure 4: Left: DQN, Right: Double DQN on `LunarLander-v2`. Double DQN achieves around 200 return by the end of training.

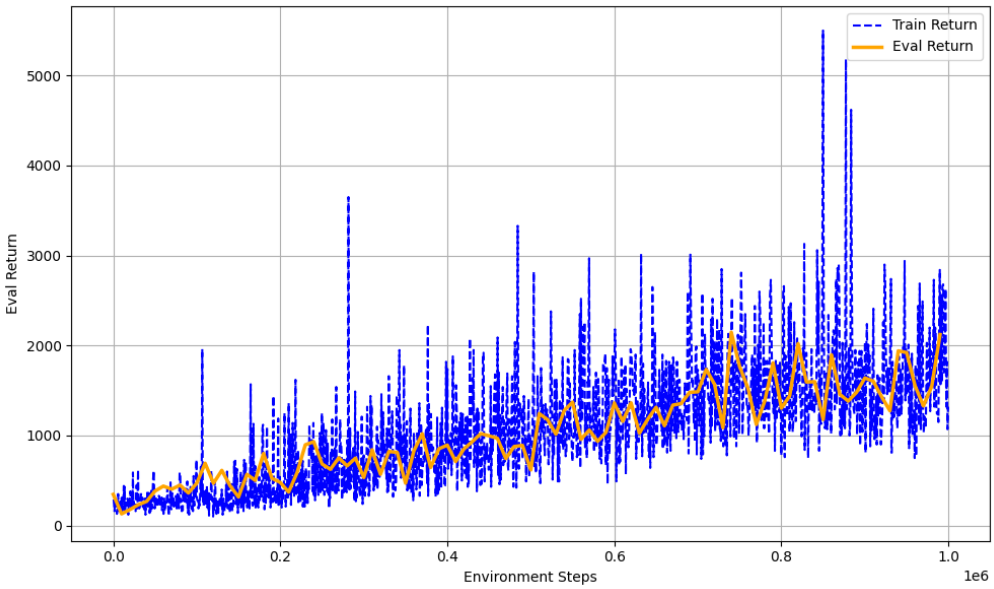

Figure 5: The training return (blue) and evaluation return (orange) for the `MsPacman-v0` environment, both shown over 1 million steps. The evaluation return reaches above 1500 by the end of training.

2.4 Experimenting with Hyperparameters

Now let's analyze the sensitivity of Q-learning to hyperparameters. Choose one hyperparameter of your choice and run at least three other settings of this hyperparameter, in addition to the one used in Question 1, and plot all four values on the same graph. Your choice what you experiment with, but you should explain why you chose this hyperparameter in the caption. Create four config files in `experiments/dqn/hyperparameters`, and look in `cs285/env_configs/basic_dqn_config.py` to see which hyperparameters you're able to change. You can use any of the base YAML files as a reference.

Hyperparameter options could include:

- Learning rate
- Network architecture
- Exploration schedule (or, if you'd like, you can implement an alternative to ϵ-greedy)

3 On-Policy Actor-Critic

3.1 Introduction

DQN works well in discrete action spaces. In continuous action spaces, though, calculating target values requires maximizing $Q_{\phi'}(s', a')$ over *all* values of a'!

Actor-critic methods are one way to deal with this limitation. In this framework, we have a *parametrized* policy π that gives us actions directly (rather than defining it implicitly by maximizing over Q-values). We can implement this in a very similar way to our policy gradient implementation from the previous homework:

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

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

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

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

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

(Note that this also coincides with GAE with $\lambda = 0$.)

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 (as we did in HW2), we can also *bootstrap* our estimate by fitting it 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.

3.2 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.

- TODO

3.3 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 Pendulum-v1 Now that you have implemented actor-critic, check that your solution works by running Cartpole-v0.
At the end, the best setting from above should match the policy gradient results from Cartpole in hw2 (200).

Question 5: Run actor-critic with more difficult tasks Use the best setting from the previous question to run InvertedPendulum and HalfCheetah: where `<ntu>_<ngsptu>` is replaced with the parameters you chose.

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.

SAC-related questions. We wanted to address some of the common questions that have been asked regarding Question 6 of the HW. The full algorithm for SAC is summarized below, the equations listed in this paper will be helpful for you: `https://arxiv.org/pdf/1812.05905`. Some definitions that will be useful:

1. What is alpha and how to update it: Alpha is the entropy regularization coefficient denoting how much exploration to add to the policy. You should update based on Eq. 18 in Section 5 in the above paper as follows:
$$J(\alpha) = \mathop{\mathbb{E}}_{a_t \sim \pi_t} [-\alpha \log_{\pi_t}(a_t|s_t) - \alpha\bar{\mathcal{H}}].$$
2. Target entropy is the negative of the action space dimension that is used to update the alpha term.
3. SquashedNorm: This is a function that takes in mean and std as in previous homeworks, and will give you a distribution that you can sample your action from.
4. To update the critic, refer to how to update Q-function parameters in Equation 6 of the paper above as follows:
$$J_Q(\theta) = Q_\theta(s_t, a_t) - (r(s_t, a_t) + \gamma(Q_{\bar{\theta}}(s_{t+1}, a_{t+1}) - \alpha \log(\pi_\phi(a_{t+1}, s_{t+1})))$$
5. To update the policy, follow Equation 18:
$$J(\alpha) = E_{a_t \sim \pi_t}[-\alpha \log \pi_t(a_t|s_t) - \alpha\bar{\mathcal{H}}]$$
6. You don't need to alter any parameters from the SAC run commands. The correct implementation should work with the provided default parameters.