Homework 3

CMU 10–703: Deep Reinforcement Learning (Fall 2025)

OUT: September 23rd

DUE: October 1th by 11:59 pm ET

Instructions: START HERE

- **Collaboration policy:** You may work in groups of up to three people for this assignment. It is also OK to get clarification (but not solutions) from books or online resources after you have thought about the problems on your own. You are expected to comply with the University Policy on Academic Integrity and Plagiarism[1].
- **Late Submission Policy:** You are allowed a total of 10 grace days for your homeworks. However, no more than 2 grace days may be applied to a single assignment. Any assignment submitted after 2 days will not receive any credit. Grace days do not need to be requested or mentioned in emails; we will automatically apply them to students who submit late. We will not give any further extensions so make sure you only use them when you are absolutely sure you need them. See the Assignments and Grading Policy for more information about grace days and late submissions [2]
- **Submitting your work:**
  - **Gradescope:** Please write your answers and copy your plots into the provided LaTeX template, and upload a PDF to the GradeScope assignment titled "Homework 3." Additionally, export your code ([File → Export .py (if using Colab notebook)]) and upload it the GradeScope assignment titled "Homework 3: Code." Each team should only upload one copy of each part. Regrade requests can be made within one week of the assignment being graded.

[1] https://www.cmu.edu/policies/

[2] https://cmudeeprl.github.io/703website_f25/logistics/

Introduction

In Homework 1, you implemented foundational policy gradient algorithms such as REINFORCE and Advantage Actor-Critic (A2C). In this assignment, you will extend those ideas by implementing three widely used modern policy gradient algorithms: Proximal Policy Optimization (PPO), Twin Delayed Deep Deterministic Policy Gradient (TD3), and Soft Actor-Critic (SAC). Among these, PPO and SAC in particular have become essential algorithms in deep reinforcement learning research and applications.

Note - Make sure to take a look at the `ReadMe.md` to set up your conda environment for this homework assignment.

Problem 0: Collaborators

Please list your name and Andrew ID, as well as the names and Andrew IDs of your collaborators.

Problem 1: Proximal Policy Optimization (50 pts)

Begin by reading the PPO paper (https://arxiv.org/abs/1707.06347) to familiarize yourself with the method you will implement. For this problem, you will focus on implementing Equations (7), (8), and (9) from the paper.

PPO improves upon the standard policy gradient algorithms from Homework 1 (REINFORCE and A2C) by introducing a more stable and efficient update mechanism. To appreciate this contribution, recall the limitations of earlier approaches:

- **REINFORCE**: A pure Monte-Carlo method that estimates gradients directly from sampled returns. It suffers from high variance, making training slow and unstable.
- **A2C (Advantage Actor-Critic)**: Reduces variance by incorporating a learned value function baseline, but still permits large, destabilizing policy updates.

PPO addresses this instability by constraining policy updates. Rather than maximizing the raw policy gradient, PPO optimizes a clipped surrogate objective that limits how much the probability ratio between new and old policies can change. This constraint yields more conservative updates, preventing destructive shifts while maintaining the simplicity of first-order gradient optimization.
All of your code for this problem will be written in `ppo_agent.py`.

Problem 1.1: PPO Loss

Because PPO is an actor-critic method, its loss function contains both an actor (policy) component and a critic (value) component. PPO's innovation lies in how the actor loss is defined, ensuring stability during training.

Problem 1.1.1: Clipped Policy Objective

The standard policy gradient objective arises from the need to improve the policy π_θ while reusing data collected under an older policy $\pi_{\theta_{\text{old}}}$. Since trajectories are generated using $\pi_{\theta_{\text{old}}}$, we cannot directly evaluate expectations under the new policy. To correct this mismatch, we use importance sampling. Intuitively, if the new policy assigns higher probability to an action than the old policy did, that sample is upweighted; if it assigns lower probability, it is downweighted. This correction makes the gradient estimator unbiased with respect to the new policy, even though the data came from the old one.

Formally, the standard policy gradient objective is:

$$J(\theta) = \mathbb{E}_{\tau \sim \pi_{\theta_{\text{old}}}} \left[\sum_{t=0}^{T-1} \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \hat{A}_t \right], \tag{1}$$

where $\hat{A}_t$ is the estimated advantage at timestep t. The advantage weighting ensures that actions with positive advantage are reinforced while those with negative advantage are discouraged. While this formulation is unbiased, it can become unstable when the ratio takes extreme values:

1. Large or small ratios introduce high variance into the gradient estimate.
2. Large updates can push the policy into poorly performing parameter space regions.
3. The advantage estimates $\hat{A}_t$, computed under the old policy, may not remain reliable if the new policy is too different.

To address this, PPO defines the clipped surrogate objective (Equation 7 in the paper). Let

$$r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}, \tag{2}$$

then the clipped objective is:

$$L^{CLIP}(\theta) = \hat{\mathbb{E}}_t \left[\min\Big(r_t(\theta)\hat{A}_t,\ \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \Big) \right], \tag{3}$$

where ϵ is a hyperparameter. The `clip` keeps the probability ratio $r_t(\theta)$ bounded between $1-\epsilon$ and $1+\epsilon$. Sometimes the unclipped ratio $r_t(\theta)\hat{A}_t$ is more conservative than the clipped version, so the min operator selects whichever is smaller.

This design guarantees conservative updates in both directions:

- If $\hat{A}_t > 0$, L^{CLIP} prevents the policy from increasing the probability of an action too aggressively.
- If $\hat{A}_t < 0$, L^{CLIP} prevents the policy from reducing the probability of an action too aggressively.

You should convince yourself by examining the equations that this conservative behavior always holds, regardless of the sign of the advantage.

Implement this clipped objective in `section 1.1.1` of the `PPOAgent._ppo_loss(...)` function.

Hint: Use the old and new log probabilities (already defined for you) to compute the ratio. Additionally, in the code ϵ is saved as `self.clip_coef`.

Problem 1.1.2: Complete PPO Loss Function

The full PPO loss augments the clipped policy loss with a value function loss and an entropy bonus. From Equation 9 of the PPO paper, the combined objective is:

$$L_t^{CLIP+VF+S}(\theta) = \hat{\mathbb{E}}_t \left[L_t^{CLIP}(\theta) - c_1 L_t^{VF}(\theta) + c_2 S[\pi_\theta](s_t) \right], \tag{4}$$

where

$$L_t^{VF}(\theta) = \left(V_\theta(s_t) - V_t^{\text{targ}} \right)^2 \qquad \text{(value function loss)}, \tag{5}$$

$$S[\pi_\theta](s_t) = -\sum_a \pi_\theta(a|s_t) \log \pi_\theta(a|s_t) \qquad \text{(entropy bonus)}. \tag{6}$$

Here V_t^{targ} is the target value (commonly the empirical return G_t), and c_1, c_2 are scalar weights balancing the critic loss and entropy bonus.

Components of the PPO Loss:

- **Policy Loss**: Encourages improved actions while constraining updates to remain close to the old policy.
- **Value Function Loss**: Trains the critic to provide accurate value estimates for advantage computation.
- **Entropy Bonus**: Encourages exploration and prevents premature convergence to deterministic policies.

Implement this combined loss in `section 1.1.2` of the `PPOAgent._ppo_loss(...)` function.

`Hint:` Use `self.vf_coef` and `self.clip_coef`. Pay attention to the signs in your total loss function. The sign conventions presented in the paper (and copied above) may be a bit confusing. Remember that pytorch minimizes loss. For your value function targets, you can directly use the `returns` variable that we define at the top of the loss function (you will be calculating these returns in the next question).

Problem 1.2: Generalized Advantage Estimation

During our loss computation, we assume access to advantage estimates $\hat{A}_t$ and returns (new targets for the value function). In HW1, you implemented advantage estimates with N-step bootstrapping, for this homework you will be implementing a more commonly used method, Generalized Advantage Estimation (GAE). While N-step bootstrapping requires selecting a fixed horizon N, GAE instead uses a parameter λ to combine advantages across all horizons, offering a smoother bias–variance tradeoff.

GAE's parameter $\lambda \in [0, 1]$ allows us to balance between bias and variance in our advantage estimates. The GAE advantage estimator is defined as:

$$\hat{A}_t^{GAE(\gamma,\lambda)} = \sum_{l=0}^{\infty} (\gamma\lambda)^l \delta_{t+l} \tag{7}$$

where $\delta_t = r_t + \gamma V(s_{t+1}) - V(s_t)$ is the temporal difference (TD) error.

In practice, for a finite episode of length T, this becomes:

$$\hat{A}_t^{GAE(\gamma,\lambda)} = \sum_{l=0}^{T-t-1} (\gamma\lambda)^l \delta_{t+l} \tag{8}$$

This can be computed efficiently using the recursive relation:

$$\hat{A}_t = \delta_t + \gamma\lambda\hat{A}_{t+1} \tag{9}$$

$$\hat{A}_{T-1} = \delta_{T-1} \tag{10}$$

With the above formulation, it should be clear how a smaller λ brings the GAE closer to 1-step TD (lower variance, but higher bias). Similarly, a large λ brings the GAE close to the Monte Carlo estimate (higher variance but lower bias)

Implement GAE in `section 1.2` of your code in the function `PPOAgent._compute_gae(...)`.

Note: The above formulation assumes a continuous environment. However, we are working in an environment with terminal states. If the final state has a `done` flag - this means that we have reached a terminal state (success or failure). If not, this means that the environment was truncated due to time limits before reaching a terminal state. Think about what needs to happen at these final states - when should we be using a value function estimate for s_{t+1}?

`Hint:` After calculating the advantages, the computation of the returns is a simple formula.

Problem 1.3: PPO Update Loop

A central feature of Proximal Policy Optimization (PPO) is its training procedure, which alternates between data collection and policy updates. In each iteration, PPO collects trajectories (or rollouts) using the current policy, then repeatedly optimizes the actor and critic networks through mini-batch gradient descent. The canonical PPO algorithm, as presented in the original paper (Algorithm 1), is shown below:

Algorithm 1 PPO Algorithm (Canonical)

procedure PPO, Actor-Critic Style
  for *iteration* $= 1, 2, \ldots$ **do**
    for *actor* $= 1, 2, \ldots, N$ **do**
      Run policy $\pi_{\theta_{\text{old}}}$ *in environment for* T *timesteps*
      Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
    end for
    Optimize surrogate L *wrt* θ, *with* K *epochs and minibatch size* $M \leq NT$
    $\theta_{\text{old}} \leftarrow \theta$
  end for
end procedure

In this homework, the same update logic is adapted slightly to maintain compatibility across different experiments. The pseudocode below illustrates this implementation. While the structure looks different, you should convince yourself that it is functionally equivalent to the canonical PPO procedure.

Algorithm 2 PPO Training Loop (Your Implementation)

procedure PPO Training Loop
  Initialize environment and actor-critic network π_θ
  Initialize rollout buffer
  $N =$ *num steps per update*
  while *total_steps* $<$ *max_steps* **do**
    Collect rollout step by running policy $\pi_{\theta_{\text{old}}}$
    if *rollout finished* **then**
      Compute GAE advantages $\hat{A}_1, \ldots, \hat{A}_T$
      Add rollout with advantages to buffer
      if *steps collected with* $\pi_{\theta_{\text{old}}} \geq N$ **then**
        Optimize L *wrt* θ, *with* K *epochs and minibatch size* $M \leq N$
        $\theta_{\text{old}} \leftarrow \theta$
      end if
    end if
  end while
end procedure

Problem 1.3.1: Environment Step Function

Implement `PPOAgent.step(self, transition)`.

This function is executed at every environment step and is responsible for maintaining the rollout buffer, updating internal state, and triggering policy updates when necessary. Specifically:

- When an episode terminates, compute Generalized Advantage Estimates (GAE) for the rollout.
- Store the processed rollout in the buffer.
- Reset the temporary rollout storage.
- If the required number of steps has been reached, update the policy and relevant instance variables.

Useful helper functions include:

```
self._compute_gae(...)
self._prepare_batch(...)
self._rollout_buffer._add_batch(...)
self._perform_update(...)
```

Problem 1.3.2: Update Function

Implement `PPOAgent._perform_update(self)`.

This function is responsible for training the actor and critic networks using data sampled from the rollout buffer. Specifically:

- Sample minibatches corresponding to the current policy iteration, using the filter argument (e.g., `filter={"iteration": [self._policy_iteration]}`).
- Normalize advantages across the batch to improve stability
- For each minibatch, perform gradient updates on the actor and critic for K epochs using the PPO loss (`_ppo_loss(...)`).

Note: Filtering by iteration allows you to reuse the buffer without resetting it each time, enabling future experiments that incorporate more off-policy data.

Hint: You must use these lines of code in your implementation for proper logging:

```
loss, stats = self._ppo_loss(minibatch)
all_stats.append(stats)
```

You should use the following line to help with stability. Also, our naming convention here is a bit misleading, but for PPO, `self.actor` is a network that is used both for the actor and the critic (critic just has a separate head).

```
torch.nn.utils.clip_grad_norm_(self.actor.parameters(), self.max_grad_norm)
```

Problem 1.4: Experiment – Clipped Objective vs. KL Objective

A key design choice in PPO is how to constrain the new policy π_θ from moving too far away from the old policy $\pi_{\theta_{\text{old}}}$. In previous sections, we introduced the clipped surrogate objective as one way to enforce this constraint. In this section, you will compare that approach against an alternative: penalizing divergence between the two policies using the KL divergence.

Problem 1.4.1: Clipped Objective

This corresponds to the loss you implemented in Section 1.1.1. Run the provided training code with the default hyperparameters and plot the resulting learning curves.

If your implementation is correct, the moving-average return (MA(50)) should approach or exceed 200 by the end of training. Note that the raw returns will be noisy, so do not be concerned about short-term oscillations. If your best-performing policy never reaches a return of 200, that indicates a mistake in your code.

You may experiment with hyperparameters if you like, but the provided defaults (including seeds) have been tested and are sufficient to achieve the expected performance.

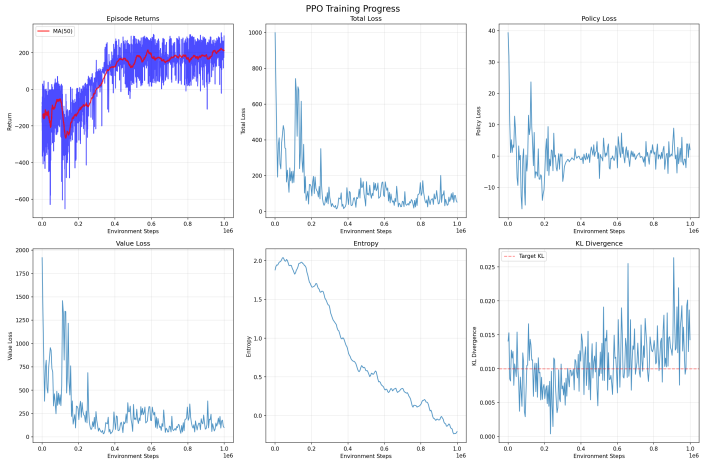
TODO: In your solution, include both the generated plots and the reported final performance.

Command (15 min):

```
python runner.py --agent ppo
```

Solution

Final performance: 226.3 ± 83.4



Problem 1.4.2: KL-Penalized Objective

Instead of clipping, another natural way to limit policy updates is to explicitly penalize divergence from the old policy. The KL divergence,

$$D_{KL}(\pi_{\theta_{\text{old}}}(\cdot|s) \,\|\, \pi_\theta(\cdot|s))\,,$$

measures how dissimilar the new policy is from the old one at each state. PPO incorporates this idea in an alternative objective (Equation 8 of the paper):

$$L^{KLPEN}(\theta) = \hat{\mathbb{E}}_t \left[\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)} \hat{A}_t - \beta D_{KL}(\pi_{\theta_{\text{old}}}(\cdot|s_t) \,\|\, \pi_\theta(\cdot|s_t)) \right], \tag{11}$$

Update β as follows:

$$\begin{cases} \beta \leftarrow \beta/2 & \text{if } d < d_{\text{targ}}/1.5, \\ \beta \leftarrow 2\beta & \text{if } d > d_{\text{targ}} \times 1.5. \end{cases}$$

Here, $\hat{A}_t$ is the advantage estimate and β is a hyperparameter that controls the strength of the KL penalty. Large deviations between old and new policies are explicitly discouraged.

Implement this KL-penalized loss in `PPOAgent._ppo_loss(...)` and implement the beta update in `PPOAgent._perform_update(...)`.

Hint: The KL divergence can be approximated using the log ratio of old to new probabilities: $\log(\pi_{\theta_{\text{old}}}/\pi_\theta)$.

TODO: Rerun training (commenting out the clipped loss), plot the generated curves below and include the final training performance.

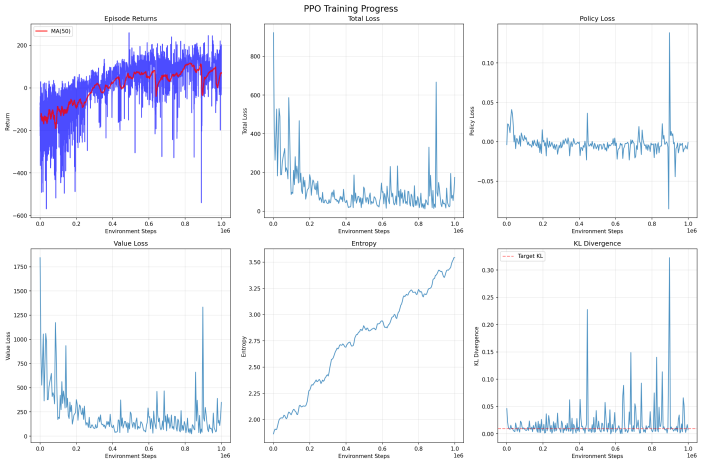
Reflection: Which approach seems to yield more stable training in your runs—clipping or KL penalty? What advantages and drawbacks do you notice in each case?

Command (15 min): You must comment out the Problem 1.1.1 code (PPO Clipped Surrogate Objective Loss) before running this command.

```
python runner.py --agent ppo
```

Solution

Final performance: 205.4 ± 101.8
In my runs, using a KL penalty tends to produce more stable training, with smoother learning curves and fewer large oscillations. In contrast, clipping allows the policy to achieve higher returns more quickly, but at the cost of increased variance and occasional instability.


PPO Training Progress
Episode Returns
MA(50)
Return
Environment Steps
1e6
Total Loss
Total Loss
Environment Steps
1e6
Policy Loss
Policy Loss
Environment Steps
1e6
Value Loss
Value Loss
Environment Steps
1e6
Entropy
Entropy
Environment Steps
1e6
KL Divergence
Target KL
KL Divergence
Environment Steps
1e6

Problem 1.5: Experiment – Clipping Threshold

The clipping threshold ϵ directly controls how conservative PPO's updates are. In this section, you will vary ϵ and observe how training behavior changes. For these experiments you can use the `--clip_coef` argument to `runner.py` to start trainings with different clipping thresholds.

For all of these experiments, comment out the KL divergence loss and go back to using the clipped objective.

Problem 1.5.1: High Clipping Threshold

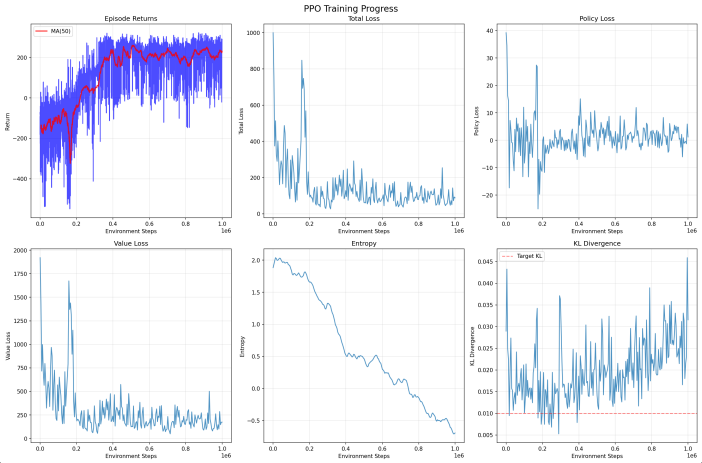
Rerun training with larger clipping threshold (0.3). Insert the plots here, and report the final environment returns. Explain how the learning dynamics differ from Section 1.4.1. Why might larger ϵ lead to this behavior? Does a higher threshold make the algorithm behave more or less like the vanilla policy gradient method? What trade-offs do you observe?

Command (15 min):

```
python runner.py --agent ppo --clip_coef 0.3
```

Solution

Final performance: 217.3 ± 95.4
With a larger clipping threshold ($\epsilon = 0.3$), the policy can deviate more from the old policy, leading to faster but less stable learning. This increased flexibility allows greater exploration and potentially higher rewards, though with higher variance and oscillations in returns. As ϵ increases, PPO behaves more like the vanilla policy gradient method, trading stability for exploration and larger policy updates.



Problem 1.5.3: Low Clipping

Rerun training with a smaller clipping threshold (0.05). Insert the plots here, and report the final environment returns. Discuss how the learning dynamics differ from Section 1.4.1. What does this tell you about the role of clipping in stabilizing updates? How does lowering

the clipping threshold affect stability and final performance? What behavior would you expect in higher-dimensional or more complex environments?

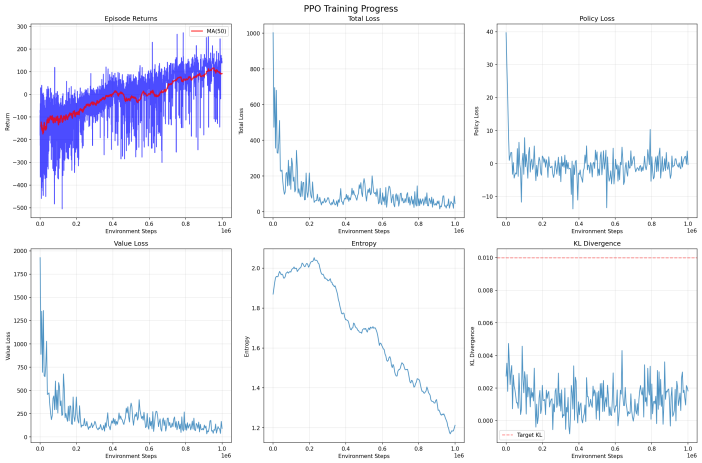
Command (15 min):

```
python runner.py --agent ppo --clip_coef 0.05
```

Solution

Final performance: 161.2 ± 137.9

With a smaller clipping threshold ($\epsilon = 0.05$), the final returns were lower because updates are very conservative and learning is slower. The strong clipping reduces variance and keeps training stable, but it also limits exploration and the policy's ability to improve quickly. In higher-dimensional or more complex environments, this conservative behavior would likely hurt performance as the agent might not explore enough to find good strategies.



Problem 1.6: Experiment – Off-Policy PPO

So far, PPO has been trained in an on-policy manner: the buffer only provides samples from the current policy iteration. Here, you will relax this constraint by allowing the buffer to sample from all stored trajectories, including older ones.

For all of these experiments, comment out the KL divergence loss and go back to using the clipped objective.

Problem 1.6.1 - Full Off-Policy PPO

Modify your sampling logic so that you pull a batch from the full dataset rather than only the current iteration (it is okay if this sampling includes some data from the current policy). Rerun training, insert the resulting plots, report final return, and compare against Section 1.4.1. How does using off-policy data affect training stability and overall performance? What might explain the differences you observe compared to strictly on-policy PPO? What changes could you make to get better performance while still using off-policy data?

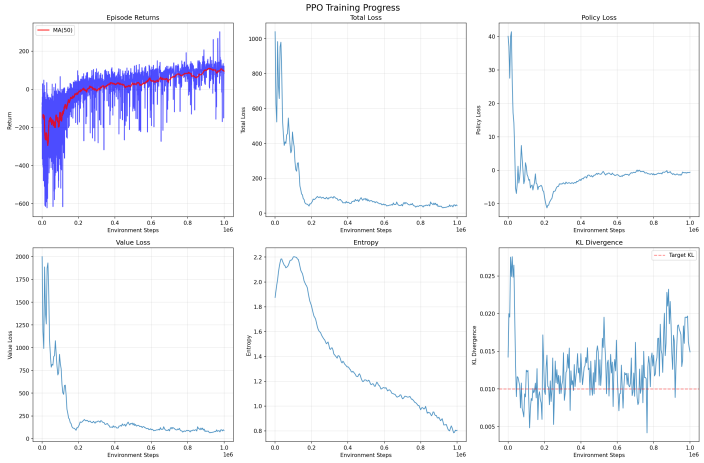
Command (15 min): Before running this command, change your logic for sampling batches. This is located in `PPOAgent._perform_update(...)`

```
python runner.py --agent ppo
```

Solution

Final performance: -2.8 ± 195.6

Using off-policy data makes training less stable since PPO assumes the samples come from the current policy old data throws that off and leads to noisy updates. That's likely why the performance fluctuates so much compared to normal PPO. To fix this while still using off-policy data, I could limit how old the samples are so they're too different from the current policy.



Problem 1.6.2 - Half Off-Policy PPO

Modify your sampling logic so that half of your batch is sampled from the full dataset and half from is sampled from the current iteration. Rerun training, insert the resulting plots, report final return, and compare against Section 1.4.1 and Section 1.6.1. How does the inclusion of more on-policy data affect the training stability and overall performance? What might explain any differences you notice between these graphs and section 1.6.1?

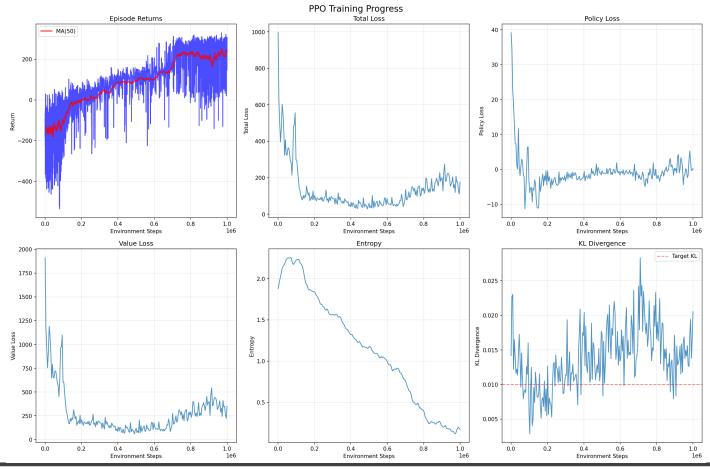
Command (15 min): Before running this command, change your logic for sampling batches. This is located in `PPOAgent._perform_update(...)`

```
python runner.py --agent ppo
```

Solution

Final performance: 184.7 ± 128.7

Including more on-policy data makes training more stable since the updates are based on samples from the current policy, reducing distribution mismatch and variance in gradients. This leads to smoother learning and better overall returns. The differences from section 1.6.1 likely come from using fresher, more consistent data instead of old off-policy samples.



Problem 2: Twin Delayed DDPG - TD3 (35 pts)

Begin by reading the TD3 paper (https://arxiv.org/abs/1802.09477) to familiarize yourself with the method you will implement.

In Homework 2, you implemented DQN and Double DQN for discrete action spaces, and saw how Double DQN reduced the overestimation bias of vanilla DQN. TD3 extends these Q-learning ideas to continuous action spaces by building on Deep Deterministic Policy Gradient (DDPG). Whereas DQN evaluates a finite set of discrete actions with a Q-function, DDPG uses an actor-critic framework: the actor outputs continuous actions, and the critic estimates their Q-values. However, plain DDPG is often unstable and still suffers from overestimation.

TD3 introduces three modifications that parallel lessons from DQN and Double DQN:

- **Twin critics:** Two Q-networks are trained in parallel, and the minimum of their predictions is used for updates. This "double" trick reduces overestimation bias.
- **Delayed policy updates:** The actor (policy) is updated less frequently than the critics, so policy improvements are based on more accurate Q-estimates.
- **Target policy smoothing:** Noise is added to target actions when computing Q-targets, preventing the critic from overfitting to narrow Q-function peaks.

Together, these changes yield more stable and reliable training in continuous control tasks. Conceptually, TD3 is the continuous-action successor to Double DQN, combining Q-learning stabilization techniques with an actor-critic architecture.
All of your code for this problem will be written in `td3_agent.py`.

TD3 Algorithm

Algorithm 3 TD3 Algorithm (Canonical)

Initialize deterministic actor μ_θ, critics Q_{ϕ_1}, Q_{ϕ_2}, targets, replay $\mathcal{D}$
for each environment step **do**
    Select action $a_t = \text{clip}(\mu_\theta(s_t) + \epsilon, a_{min}, a_{max}),\ \epsilon \sim \mathcal{N}(0, \sigma^2)$
    Step env; store $(s_t, a_t, r_t, s_{t+1}, d_t) \in \mathcal{D}$
    if ready to update **then**
        Sample batch; compute target action $\tilde{a}' = \text{clip}(\mu_{\theta'}(s') + \epsilon',\ a_{\min},\ a_{\max}),\ |\epsilon'| \le c$
        Target: $y = r + \gamma(1-d)\min\{Q'_1(s', \tilde{a}'), Q'_2(s', \tilde{a}')\}$
        Update critics: minimize $\sum_i \|Q_{\phi_i}(s,a) - y\|^2$
        if update iteration % `policy_delay` == 0 **then**
            Update actor: minimize $-\mathbb{E}[Q_{\phi_1}(s, \mu_\theta(s))]$
            Soft-update actor/critic targets with τ
        end if
    end if
end for

Problem 2.1.1: Network Initialization

Lets start by implementing the network initialization in `TD3Agent.__init__(...)`.

In TD3, you will maintain not only an actor and two critics, but also corresponding *target networks* for each. The role of these target networks is to stabilize training when bootstrapping. Recall that bootstrapping means constructing a training target using the network's own predictions for future states. If the same networks you are updating are also used to generate these training targets, the target values shift every time you update, leading to instability. By using separate, slowly updated target networks to generate the bootstrapped training targets, TD3 ensures that the reference values change more gradually, making learning more stable.

In this problem, you will set up the actor, two critics, and their corresponding target networks. Initialize the targets to match the online networks at the start of training.
Hint: Actors and Critics can be initialized as follows:

```
Actor(
    obs_dim=self.obs_dim,
    act_dim=self.act_dim,
    act_low=self.act_low,
    act_high=self.act_high,
    hidden=(64, 64),
).to(self.device)
```

```
Critic(
    self.obs_dim,
    self.act_dim,
    hidden=(64, 64)
).to(self.device)
```

Problem 2.1.2: TD3 Target with Policy Smoothing

Next, implement the update step in `TD3Agent._td3_update_step(...)`. In TD3, the target is computed using the target actor (with added noise for smoothing) and the element-wise minimum of the two target critics:

$$
\begin{aligned}
y &= r + \gamma(1-d)\,\min\Big(Q'_1(s',\tilde{a}'),\; Q'_2(s',\tilde{a}')\Big) \\
\tilde{a}' &= \mathrm{clip}\Big(\mu'(s') + \epsilon,\; a_{\min},\; a_{\max}\Big) \\
\epsilon &\sim \mathcal{N}(0,\sigma^2), \quad |\epsilon| \le c
\end{aligned}
\tag{12}
$$

Here μ' denotes the target actor, Q'_1, Q'_2 are the target critics, and ϵ is Gaussian noise (clipped to a maximum magnitude c). This noise helps prevent the policy from overfitting to sharp peaks in the Q-function.

Hints:

- To get a deterministic action from actor network A, you can use `A(obs).mean_action`
- Sample noise with `torch.randn_like`, scale it by `self.policy_noise`, and clamp to $[-$`self.noise_clip`, `self.noise_clip`$]$.

- Clamp resulting actions to `(self.act_low, self.act_high)`.

Problem 2.1.3: Critic Update

Minimize the sum of MSE losses of both critics against y with a single optimizer step. Make sure to zero grad the critic optimizer before calling backwards, then stepping.

$$\mathcal{L}_Q = \mathbb{E}\big[(Q_1(s,a) - y)^2 + (Q_2(s,a) - y)^2\big]. \tag{13}$$

This will be implemented in `TD3Agent._td3_update_step` (critic block).

Problem 2.1.4: Actor Update (Delayed)

Every `policy_delay` steps, update the actor to maximize the Q-value estimated by the first critic:

$$\mathcal{L}_\pi = -\mathbb{E}_s\left[Q_1(s, \mu_\theta(s))\right]. \tag{14}$$

After updating the actor, also apply soft updates to all target networks (actor and critics). You should be making calls to `self._soft_update`, but soft update rule itself will be implemented in the next problem.
Implement this in `TD3Agent._td3_update_step` (actor block).

Problem 2.1.5: Polyak Target Updates

After each action update, softly update target networks using Polyak averaging. For parameters θ (online network) and θ' (target network), the update is

$$\theta' \leftarrow (1-\tau)\,\theta' + \tau\,\theta, \tag{15}$$

where $\tau \in (0, 1]$ controls how fast the target tracks the online network.

Implement this in `TD3Agent._soft_update`.
Hint: Use `torch.no_grad()` and iterate with `zip(params, target_params)`.

Problem 2.2: Exploration Noise with Actions

The actor in TD3 is deterministic, so without added noise, the policy will fail to explore. To address this, TD3 injects Gaussian noise into the actor's output when interacting with the environment:

$$a = \mu_\theta(s) + \epsilon, \qquad \epsilon \sim \mathcal{N}(0, \sigma^2).$$

Implement this in `TD3Agent.act`:
Hints:

- To get a deterministic action from actor network A, you can use `A(obs).mean_action`

- Sample noise with `torch.randn_like`, scale it by `self.exploration_noise`.
- Clamp the final action to `(self.act_low, self.act_high)` before returning it.

Problem 2.3: Delayed Policy Updates

In TD3, the critics are updated at every training step, but the actor (and target networks) are updated less frequently. This is controlled by the parameter `self.policy_delay` (commonly set to 2). In practice, this means: update critics every time, and only update the actor and targets once every `self.policy_delay` critic updates.

You will implement this scheduling logic inside `TD3Agent._perform_update`.

Hints:

- Use `self.update_count` to keep track of updates
- Set a flag (`do_actor_update`)to indicate whether the actor should update on this step.
- You **must** set `stats = self._td3_update_step(batch, do_actor_update)` for appropriate logging

Problem 2.4: Environment Step Function (TD3Agent.step)

Implement `TD3Agent.step(self, transition)` using the TD3 rules:

- Add transitions to buffer and maintain `self.total_steps` (this is done for you).
- Don't update when `self.warmup_steps` or `self.batch_size` is larger than the buffer size
- Otherwise, update every `self.update_every` steps
- If not updating, return an empty dictionary

Problem 2.5: TD3 Evaluation

Run the TD3 agent code with the default hyperparameters and plot the resulting learning curves. In your solution, include both the generated plots and the reported final performance.

If your implementation is correct, the moving-average return (MA(50)) should approach or exceed 200 by the end of training. Note that the raw returns will be noisy, so do not be concerned about short-term oscillations. If your best-performing policy never reaches a return of 200, that indicates a mistake in your code. You may experiment with hyperparameters if you like, but the default values have been tested and are sufficient to achieve the expected performance.

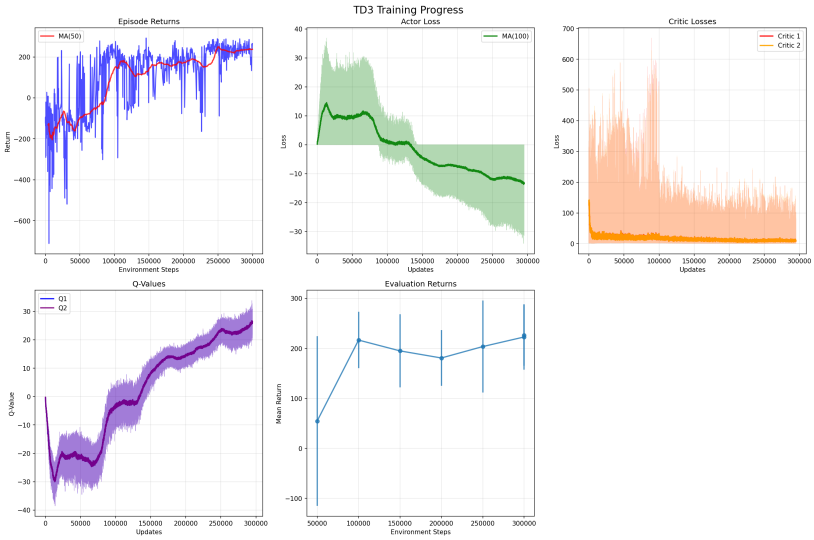
Note - For TD3, you may not see positive returns until around 250000 iterations.

Command (30 min):

```
python runner.py --agent td3 --total_steps 500000
```

Solution

NOTE: Was running out of memory on my machine, had to reduce total steps to 300000
Final performance: 226.1 ± 60.6



Problem 2.5: Target Policy Smoothing

Run the TD3 agent code with `self.policy_noise == 0`. Run this and compare with baseline, why do you see the results you do? What role does target policy smoothing play?

In your solution, include both the generated plots and the reported final performance.

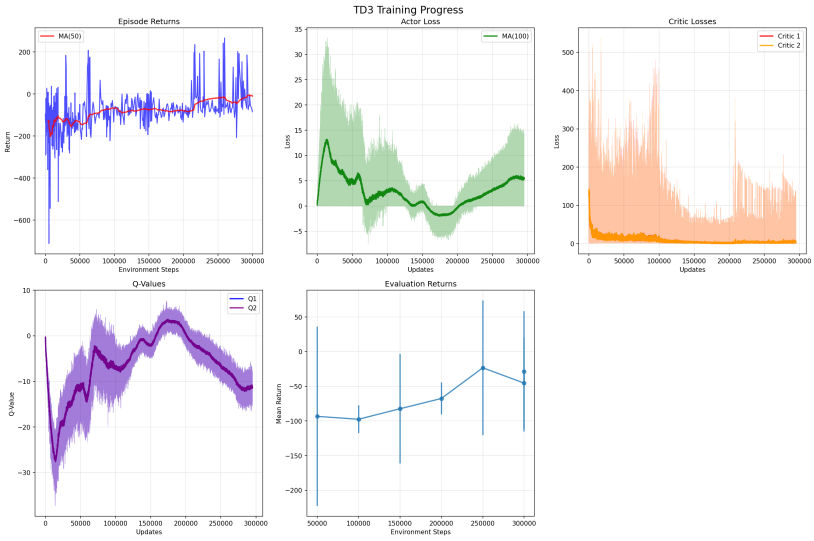
Command (30 min):

```
python runner.py --agent td3 --total_steps 500000 --policy_noise 0
```

Solution

NOTE: Was running out of memory on my machine, had to reduce total steps to 300000
Final performance: -45.3 ± 66.0

When policy_noise = 0, target policy smoothing is disabled, so target actions are deterministic and the critic can overfit to sharp Q-value peaks. This makes training less stable and can hurt performance. Adding noise normally smooths the targets, reducing overestimation and improving stability.



Problem 2.6: Policy Delay

Run the TD3 agent code with `self.policy_delay == 1, 4`. The baseline is 2. Run these versions and compare with baseline, why do you see the results you do? What role does setting the policy delay parameter play?

In your solution, include both the generated plots and the reported final performance.

Command (30 min each):

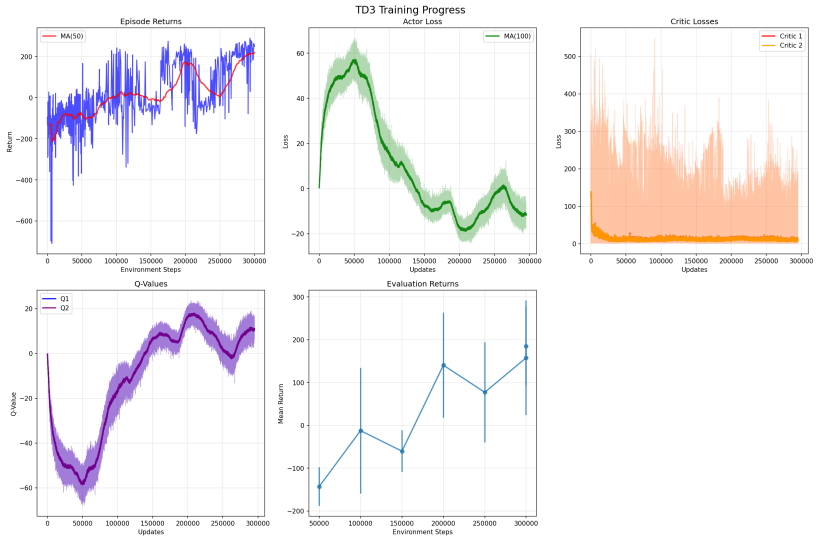
```
python runner.py --agent td3 --total_steps 500000 --delay 1
python runner.py --agent td3 --total_steps 500000 --delay 4
```

Solution

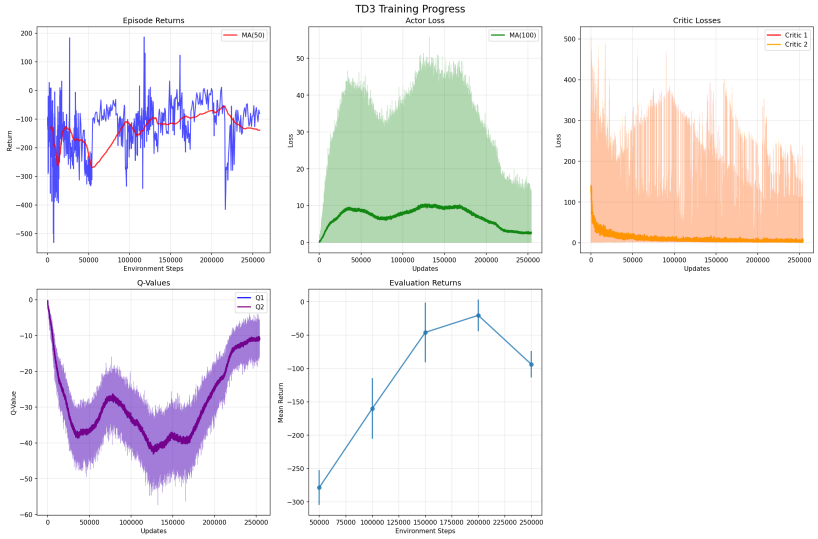
NOTE: Was running out of memory on my machine, had to reduce total steps to 300000
NOTE: for policy delay = 4, training process got killed at 250000 steps due to memory issues

With policy_delay = 1, the actor updates more often, speeding up learning but increasing variance. With policy_delay = 4, updates are less frequent, giving the critics time to converge, which makes learning slower but more stable. Policy delay balances learning speed and stability.

Final performance: 184.9 ± 91.7 (policy delay = 1)



Final performance: -93.8 ± 19.8 (policy delay = 4)



Problem 3: Soft Actor-Critic - SAC (15 pts)

Begin by reading the SAC paper (https://arxiv.org/abs/1801.01290) to familiarize yourself with the method you will implement.

SAC is an off-policy actor-critic algorithm that combines ideas from both the policy gradient family (HW1: REINFORCE, A2C, PPO) and the Q-learning family (HW2: DQN, Double DQN, TD3). Like TD3, it uses twin critics to reduce overestimation bias and target networks to stabilize bootstrapped updates. From the policy gradient side, SAC inherits the idea of training a stochastic policy with gradients, rather than a deterministic actor. This stochasticity is built directly into the objective through an **entropy bonus**, which encourages the policy to remain diverse rather than collapsing to a single action too quickly.

The SAC objective balances two terms:

- **Expected return:** As in other actor-critic methods, the policy is trained to maximize predicted Q-values.
- **Entropy maximization:** The policy is simultaneously trained to maximize entropy, i.e., to act as randomly as possible while still pursuing reward. This leads to more robust policies and improved exploration.

All of your code for this problem will be written in `sac_agent.py`.

You will be able to reuse much of the structure from TD3, as SAC borrows many of the same components (twin critics, target networks, off-policy updates) while extending them with a stochastic actor and entropy maximization.

SAC Algorithm

Algorithm 4 SAC Algorithm

Initialize actor π_θ, critics Q_{ϕ_1}, Q_{ϕ_2}, targets $\phi'_i \leftarrow \phi_i$, replay $\mathcal{D}$
for each environment step **do**
    Select action $a_t \sim \pi_\theta(\cdot|s_t)$; execute in env; store $(s_t, a_t, r_t, s_{t+1}, d_t) \in \mathcal{D}$
    if warmup complete and update is due **then**
        for $k = 1$ to UTD ratio **do**
            Sample batch from $\mathcal{D}$; compute target $y = r + \gamma(1-d)[\min Q' - \alpha \log \pi]$
            Update critics by minimizing $\sum_i \|Q_{\phi_i}(s,a) - y\|^2$
            Update actor by minimizing $\alpha \log \pi_\theta(a|s) - \min_i Q_{\phi_i}(s,a)$
            (Optional) Update temperature α
            Soft-update targets: $\phi'_i \leftarrow (1-\tau)\phi'_i + \tau\phi_i$
        end for
    end if
end for

Problem 3.1: SAC Implementation

Problem 3.1.1: Network Initialization

Let's begin by implementing the network initialization in `SACAgent.__init__(...)`.

SAC, like TD3, uses twin critics with corresponding *target networks* to stabilize training when bootstrapping. However, unlike TD3, SAC's actor is *stochastic*: it outputs both a mean and a standard deviation for the action distribution. This allows SAC to directly optimize entropy as part of its objective, encouraging diverse exploration. Because of this, we actually do not need to maintain a target actor network.

Hint: We can initialize actors and critics in the same way as TD3. For deterministic actions in TD3, we were just taking the mean action from the actor. But our actor implementation naturally allows for sampling stochastic actions (feel free to check the implementation in `policies.py`).

```
Actor(
    obs_dim=self.obs_dim,
    act_dim=self.act_dim,
    act_low=self.act_low,
    act_high=self.act_high,
    hidden=(64, 64),
).to(self.device)
```

```
Critic(
    self.obs_dim,
    self.act_dim,
    hidden=(64, 64)
).to(self.device)
```

Problem 3.1.2: Soft Bellman Target

You will be implementing the update step in this homework. SAC uses a soft bellman target to train the Q-function - this target incorporates a minimum between the twin value networks and subtracts a baseline term that incorporates temperature that increases/decreases exploration. Compute the soft target with entropy term:

$$y = r + \gamma(1-d)\Big[\min\big(Q_1'(s',a'), Q_2'(s',a')\big) - \alpha \log \pi_\theta(a'|s')\Big], \tag{16}$$

where $a' \sim \pi_\theta(\cdot|s')$ is drawn using `rsample()` (reparameterization) or set to the policy mean. Make sure to clamp the action log probability to the bounds [-20,20].
Implement this in the `SACAgent.sac_update_step` (soft bellman target block).

Problem 3.1.3: Critic Update

Continuing the update step implementation, you will code the critic update. The critic update in SAC minimizes the sum of MSE losses for both critics against y:

$$\mathcal{L}_Q = \mathbb{E}\big[(Q_1(s,a) - y)^2 + (Q_2(s,a) - y)^2\big]. \tag{17}$$

Backprop once over the concatenated critic parameters and step the optimizer. Make sure that you zero-grad the optimizer before calling backward(). You should also use `clip_-`

`grad_norm` on the critic1 and critic2 parameters, setting max norm to 1.0.
Implement this in `SACAgent.sac_update_step` (critic block).

Problem 3.1.4: Actor Update

Contininuing in the `SACAgent.sac_update_step` function, you will implement the update for the actor as well.

The formula looks this:

$$\mathcal{L}_\pi = \mathbb{E}_{s\sim\mathcal{D},\,\epsilon}\Big[\alpha\,\log\pi_\theta(a|s) - \min(Q_1(s,a), Q_2(s,a))\Big], \quad a = \pi_\theta(s;\epsilon). \tag{18}$$

Make sure you clamp the log probabilities using the bounds [-20, 20]. Also, make sure to take the mean of the expression inside of the expected value to get the correct value in the code. Please make sure to also use `clip_grad_norm` on the actor parameters to max norm value of 1.0.

Problem 3.1.5: Polyak Target Updates

After each update, softly update target critics. For parameters θ (online network) and θ' (target network), the update is:

$$\theta' \leftarrow (1-\tau)\,\theta' + \tau\,\theta, \tag{19}$$

where $\tau \in (0, 1]$ controls how fast the target tracks the online network. This is the same update that we see in TD3.
You should implement this within `SACAgent._soft_update_`.
Call this soft target update after the existing updates in `SACAgent._sac_update_step`.
Hint: Use `torch.no_grad()` and iterate with `zip(params, target_params)`.

Problem 3.2: Environment Step

Implement `SACAgent.step(self, transition)` using the SAC rules:

- Add transitions to buffer and maintain `self.total_steps` (this is done for you).
- Don't update when `self.warmup_steps` or `self.batch_size` is larger than the buffer size
- Otherwise, update every `self.update_every` steps
- If not updating, return an empty dictionary

Implement this in the `SACAgent.step` function.

Problem 3.3: SAC Evaluation

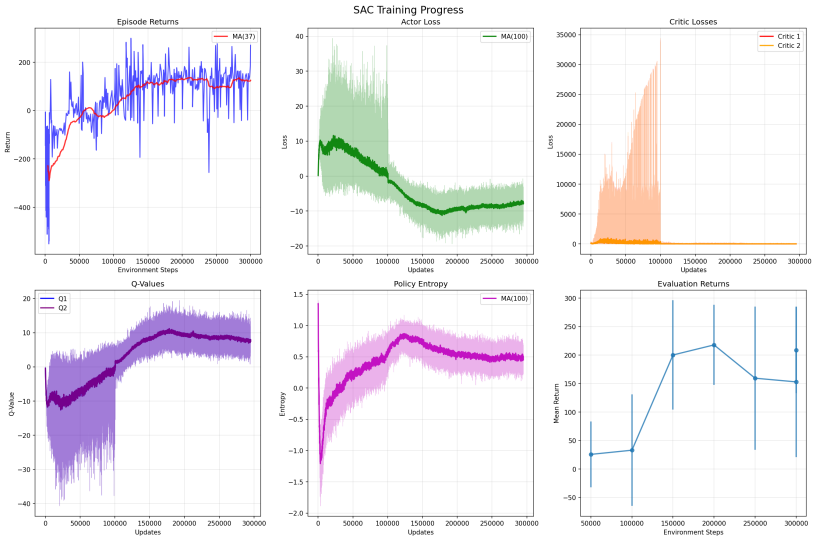
Run the SAC agent code with the default hyperparameters and plot the resulting learning curves. If your implementation is correct, the moving-average return (MA(50)) should

approach or exceed 200 by the end of training. Note that the raw returns will be noisy, so do not be concerned about short-term oscillations. If your best-performing policy never reaches a return of 200, that indicates a mistake in your code. You may experiment with hyperparameters if you like, but the provided defaults (includ- ing seeds) have been tested and are sufficient to achieve the expected performance.

Command (40 min):

```
python runner.py --agent sac --total_steps 500000
```

Solution



Problem 3.4 UTD ratio

Experiment with the number of gradient steps per environment step. This can be controlled with `--utd_ratio N`. Analyze sample efficiency vs. stability and wall-clock when comparing to the baseline in 3.3 when running with utd = 2 and utd = 4. In your solution include the generated plots, the final environment return, and the total run time. Note the lower number of total steps to help save some time, so keep this in mind when comparing to 3.3.

Command (40 min):

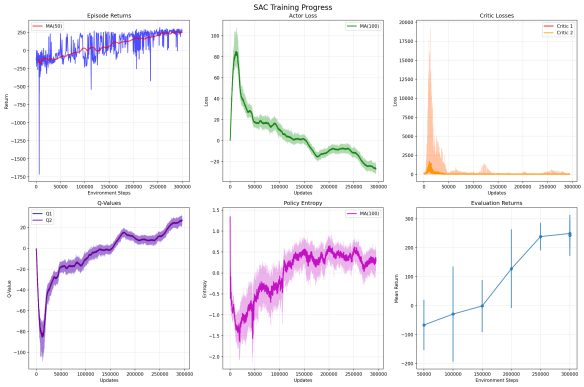
```
python runner.py --agent sac --total_steps 300000 --utd_ratio 2
python runner.py --agent sac --total_steps 300000 --utd_ratio 4
```

Solution

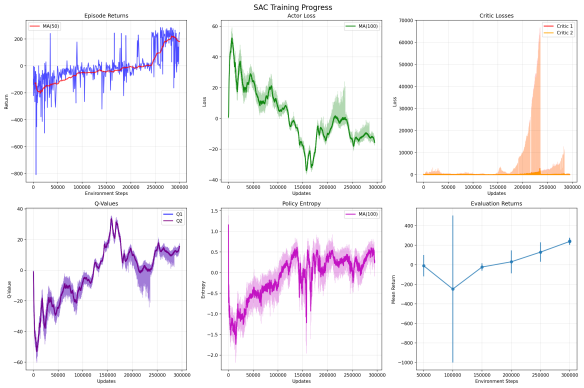
Increasing –utd_ratio improves sample efficiency but can reduce stability. With utd=2, learning is faster and still fairly stable, while utd=4 leads to instability.

Overall, higher UTD ratios trade stability for efficiency and speed

Final performance: 242.8 ± 70.6 (utd = 2) Training Run Took: 1:51:59.382648



Final performance: 242.9 ± 32.3 (utd = 4) Training Run Took: 1:42:03.280795



Problem 3.5: Backup Action

In `SACAgent.act()`, change the action generation to deterministic action generation (using `self.actor(obs_t).mean_action`). How does taking the mean action affect performance? Plot and write one to two sentences on the differences and explain why, when comparing to 3.3 baseline.

Command (40 min): Before running this command, make sure to change the action generation in `SACAgent.act()`.

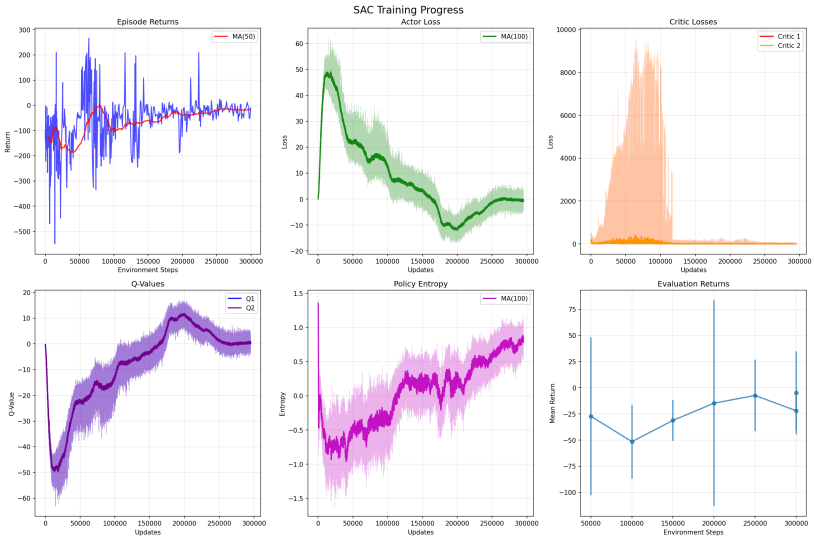
```
python runner.py --agent sac --total_steps 500000
```

Solution

NOTE: Was running out of memory on my machine, had to reduce total steps to 300000

Final performance: -4.9 ± 39.6

When using deterministic actions (mean action), the policy loses its stochasticity and exploration capability. This leads to poorer performance compared to the baseline where actions are sampled from the policy distribution.



Problem 4: Feedback

Feedback: You can help the course staff improve the course by providing feedback. What was the most confusing part of this homework, and what would have made it less confusing?

This homework was well-structured, however I found the PPO implementation to be difficult to follow at times as there were subtle important implementation details not mentioned in the pseudocode. I spent a considerable amount of time debugging this part. I also ran into memory issues when running the TD3 and SAC experiments. My machine has 16GB of RAM, but I would find the process constantly getting killed due to high memory usage. I had to reduce the total steps to 300000 to get it to run.

Time Spent: How many hours did you spend working on this assignment? Your answer will not affect your grade.

Alone	20
With teammates	
With other classmates	
At office hours	