# CS224R Spring 2025 Homework 2
# Online Reinforcement Learning

Due 5/2/2025

SUNet ID: agliang
Name: Agnes Liang
Collaborators:

By turning in this assignment, I agree by the Stanford honor code and declare that all of this is my own work.

## Problem 1: Impact of Reward Functions

1. Design a reward function for the Quadruped task such that the agent walks in a clockwise circle (watch `mujoco_mpc/videos/part1.avi` for an example of the desired behavior). The structure of the reward function for the Quadruped task is:

$$r_t = -w_0 \cdot r_{t,\text{height}} - w_1 \cdot r_{t,\text{pos}}(w_2, w_3) + c$$

where

- $r_{t,\text{height}}$ is the absolute difference between the agent's torso height over its feet and the target height of 1,
- $r_{t,\text{pos}}(w_2, w_3)$ is the $\ell^2$ norm of the difference between the agent's torso position and the goal position. The goal moves at each time-step according to the desired *walk speed* $w_2$ and *walk direction* $w_3$, and
- $c$ consists of other reward terms for balance, effort, and posture.

You will design the reward function by choosing values for $w_0, w_1, w_2$, and $w_3$, which can be any real number. Here is how you can run the Quadruped task with $w_0 = w_1 = w_2 = w_3 = 0$:

```
./build/bin/mjpc --task="Quadruped Flat" --steps=100 \
    --horizon=0.35 --w0=0.0 --w1=0.0 --w2=0.0 --w3=0.0
```

The program will run the simulator for the specified number of time-steps and save a video in the `mujoco_mpc/videos/` directory.

Fill in Table 1 with the parameters of your reward function.

| Parameter | Value |
|---|---|
| $w_0$ | **1.0** |
| $w_1$ | **10** |
| $w_2$ | **0.35** |
| $w_3$ | **-1.0** |

Table 1: Parameters of your reward function.

2. In this next part, you will see how different reward functions impact the agent's behavior in the In-Hand Manipulation task. The structure of the reward function for the Hand task is:

$$r_t = -w_0 \cdot r_{t,\text{cube pos}} - w_1 \cdot r_{t,\text{cube ori}} - w_2 \cdot r_{t,\text{cube vel}} - w_3 \cdot r_{t,\text{actuator}}$$

where

- $r_{t,\text{cube pos}}$ is the $\ell^2$ norm of the difference between the cube's position and the hand palm position,
- $r_{t,\text{cube ori}}$ is the $\ell^2$ norm of the difference between the cube's orientation and the goal orientation,
- $r_{t,\text{cube vel}}$ is the $\ell^2$ norm of the cube's linear velocity, and
- $r_{t,\text{actuator}}$ is the $\ell^2$ norm of the control inputs (i.e., the forces applied that cause the robot to move).

For each part below, you will watch the videos located in `mujoco_mpc/videos`. Then, in 1-2 sentences, describe the agent's behavior and why the reward function leads to this behavior.

(a) Watch `mujoco_mpc/videos/part2a.avi`, which was generated with the parameters $w_0 = 20$, $w_1 = 3$, $w_2 = 10$, and $w_3 = 0.1$:

```
./build/bin/mjpc --task="Hand" --steps=100 \
    --horizon=2.5 --w0=20.0 --w1=3.0 --w2=10.0 --w3=0.1
```

**Describe the agent's behavior and why the reward function leads to this behavior in 1-2 sentences.**

The agent grasps the cube and keep it on the ahnd while also allowing some rotation to happen as the hand moves. This behavior results from the high weight on maintaining the cube in hand as $w0 = 20$ while only moderately penalizing orientation changes $w1 = 3$, allowing for some rotational motion during manipulation.

(b) Watch `mujoco_mpc/videos/part2b.avi`, which was generated with the parameters $w_0 = 20$, $w_1 = 3$, $w_2 = 10$, and $w_3 = 1$:

```
./build/bin/mjpc --task="Hand" --steps=100 \
    --horizon=0.25 --w0=20.0 --w1=3.0 --w2=10.0 --w3=1.0
```

**Describe the agent's behavior and why the reward function leads to this behavior in 1-2 sentences.**

The agent stays still and does not move the cube because the reward function strongly penalizes effort with a high actuator weight $w3 = 1.0$. This combined with the very short planning horizon of 0.25, the agent chooses not to move since it cannot pan far ahead and moving would cost too much of control effort.

(c) Watch `mujoco_mpc/videos/part2c.avi`, which was generated with the parameters $w_0 = 0$, $w_1 = 0$, $w_2 = 0$, and $w_3 = 1$:

```
./build/bin/mjpc --task="Hand" --steps=100 \
    --horizon=2.5 --w0=0.0 --w1=0.0 --w2=0.0 --w3=1.0
```

**Hint:** An episode terminates when the cube falls out of the hand.

**Describe the agent's behavior and why the reward function leads to this behavior in 1-2 sentences.**

The agent is not able to grasp or stabilize the cube which falls quickly out of the hand. This is because all-task related rewards (w0,w1, and w2) are set to 0 which makes it so that the agent has no incentive to hold, orient or stabilize the cube and only the minimizing actuator effort $w3 = 1.0$ is rewarded.

## Problem 2:

1. Optimizing behavior in environments with sparse rewards is difficult due to limited reward supervision. To alleviate this, we provide the agent with 20 successful demonstrations, which we will use to pre-train with behavior cloning. In **ac.py** complete the **bc** function of the **ACAgent** class to train the policy using supervised behavior cloning. That is, given state-action pairs $o_t, a_t$ optimize the loss

$$\mathcal{L}_{\pi_\theta, f_\theta}(o_t, a_t) = -\log \pi_\theta(a_t|o_t)$$

   We will also use this to update the actor during later RL training, to improve stability.

2. In the second part, we will try to improve the performance of the policy with additional fine-tuning with reinforcement learning. Reinforcement learning allows the robot to improve using its own experience, without needing additional demonstrations. Your implementation will be in the **ACAgent** class.

   - We begin by implementing the **update_critic** method using the Bellman objective. Consider transitions $(o_t, a_t, r_t, o_{t+1}, \gamma)$ and implement the following steps:
     (a) Sample next state actions from the policy $a'_{t+1} \sim \pi_\theta(o_{t+1})$.
     (b) Compute the Bellman targets

$$y = r_t + \gamma \min\{\bar{Q}_{\theta^i}(o_{t+1}, a'_{t+1}), \bar{Q}_{\theta^j}(o_{t+1}, a'_{t+1})\}$$

     where $\bar{Q}_{\theta^i}$ and $\bar{Q}_{\theta^j}$ are two randomly sampled target critics. We take the minimum over two Q-values to mitigate critic overestimation errors.
     (c) Compute the loss:

$$\mathcal{L}_{Q_\theta, f_\theta} = \sum_{i=1}^{N} (Q_{\theta^i}(o_t, a_t) - \mathbf{sg}(y))^2$$

     where sg stands for the stop gradient operator.
     (d) Take a gradient step with respect to the critic parameters.
     (e) Update the target critic parameters using exponential moving average.

$$\bar{Q}_{\theta^i} = (1-\tau)\bar{Q}_{\theta^i} + \tau Q_{\theta^i}$$

     By using an exponential moving average, our target critic parameters are updated more slowly than those of the main critic.

   - Next, we will improve the policy in the **update_actor** method. Sample an action from the actor $a'_t \sim \pi_\theta(o_t)$ and compute the objective that optimizes the actor to maximize the Q-value estimates from the critics:

$$\mathcal{L}_{\pi_\theta} = -\frac{1}{N} \sum_{i=1}^{N} Q_{\theta^i}(o_t, a'_t)$$

     Take a gradient step on this objective with respect to the policy only.

- Once you are done, run the RL fine-tuning with

```
python train.py agent.num_critics=2 utd=1
```

**Attach the plot "eval/episode_success" for this run from Tensorboard. You should achieve a success rate of at least 90% before 100K environment steps.**

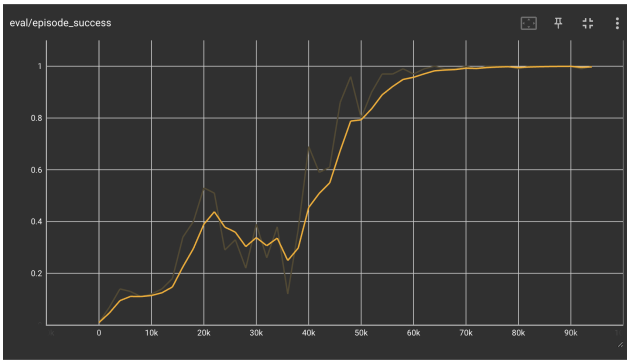


Figure 1: Plot for 2 critics, UTD=1

- In the final part, we will explore some optimization parameter choices. The update-to-data (UTD) ratio stands for the number of critic gradient steps we do, with respect to each environment step. So far we have used only 2 critics and UTD of 1. Repeat the previous part with:

```
python train.py agent.num_critics=10 utd=5
```

Here, we perform critic gradient steps 5x more often. However, too many gradient steps can make learning unstable by increasing error in the critic Q-values.

**Attach the plot "eval/episode_success" for this run and compare it to the previous question. Provide an explanation of why we observe these effects.**

Figure 2 shows the evaluation success rate for the configuration with 10 critics and an update-to-data (UTD) ratio of 5. Compared to the previous setting with 2 critics and a UTD of 1, this setup leads to faster learning, with the agent reaching near-perfect success rates before 30k environment steps. This improvement can be attributed to the increased number of critic gradient updates per environment step, which provides more learning signal and improves sample efficiency. However, the plot also exhibits mild fluctuations during the early stages of training, reflecting some instability in learning. This instability arises because performing too many updates to the critic can cause the Q-value estimates to become inaccurate, which in turn negatively affects the actor's learning. Overall, increasing the number of critics and UTD ratio boosts performance but can introduce learning variance if not tuned carefully.

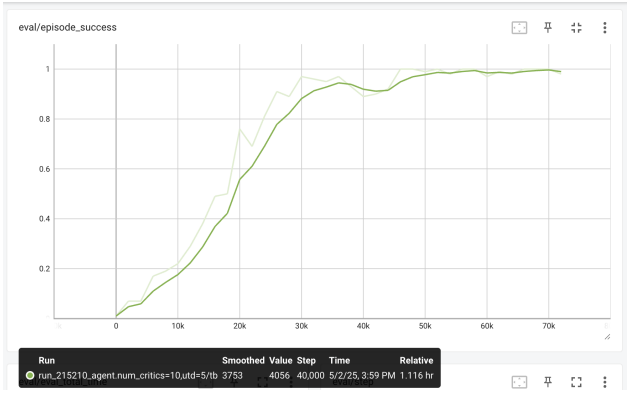


Figure 2: Plot for 10 critics, UTD = 5