CS224R Spring 2023 Homework 2
Online Reinforcement Learning

Due 5/3/2023

SUNet ID: 06661759
Name: Paul-Emile Giacomelli
Collaborators: Yoni Gozlan, Josselin Sommerville

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 ℓ^2 norm of the difference between the agent's torso position and the goal position, which 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. 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	10.0
w_1	1.0
w_2	0.15
w_3	-0.25

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 ℓ^2 norm of the difference between the cube's position and the hand palm position,
- $r_{t,\text{cube ori}}$ is the ℓ^2 norm of the difference between the cube's orientation and the goal orientation,
- $r_{t,\text{cube vel}}$ is the ℓ^2 norm of the cube's linear velocity, and
- $r_{t,\text{actuator}}$ is the ℓ^2 norm of the control inputs.

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 hand moves a bit erratically, but the cube is always on the palm and rotates only once to the desired position.
The weight w_3 multiplying the l^2 norm of the control inputs $r_{t,actuator}$ is very low (0.1) which explains why the movement of the finger is fast and frequent.
Also, the fact that w_0 and w_2 are "big" explain why the cube is always on the palm and does not have much velocity, as these coefficients multiply $r_{t,cubepos}$ and $r_{t,cubevel}$.
Finally, w_1 is low (3.0) which which is coherent with the cube only rotating once.

(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 hand is not moving.
Here, we have the same w_0, w_1, and w_2 as before. However, w_3 is now equal to 1.0, which explains why the hand is not moving. Indeed, the agent aims at

minimizing the cost function, and as moving a finger would increase the cost function (it would move the cube which woud have an impact on the cost function given the high values of w_0 and w_2, and a big impact with the norm of the control inputs given the value of w_3), the hand chooses not to move which is indeed the best behaviour to minimize the cost function in this case.
However, in opposition to (2.c.), the best behaviour is to keep the initial position of the hand as the fairly "big" values of w_0, w_1, and w_2 would induce an increase of the cost function.

(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
```

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

The hand is totally "relaxed" and does not move.
As w_0, w_1, and w_2 are equal to 0, they have no influence on the reward function so the hand has no reason to control the position, rotation and velocity of the cube. Therefore, only the norm of the control inputs have an impact on the cost function. As moving a finger would increase the cost function, the hand does not move and gets "relaxed" because it has no reason to hold its initial position as in (2.b.) (because of the values of w_0, w_1, and w_2).

Problem 2:

1. Optimizing behaviour in environments with sparse reward functions is difficult due to limited reward supervision. To alleviate that, we have provide the agent with 5 successful demonstrations, which we will use to pre-train with behaviour cloning. In **ac.py** complete the **pretrain** function of the **PixelACAgent** class to train **both** the policy and encoder using the supervised behaviour cloning loss. That is, given state-action pairs o_t, a_t optimize the loss

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

with respect to both π_θ and f_θ.

2. In the second part, we will try to improve the performance of the policy with additional fine-tuning with reinforcement learning. Your implementation will be in the **update** method of the **PixelACAgent** class.

   - We begin by implementing the critic update using the standard Bellman objective. Consider transitions $(o_t, a_t, r_t, o_{t+1}, \gamma)$ and implement the following steps:
     (a) Process the observations o_t, o_{t+1} through the augmentation and encoder networks to obtain features $f_\theta(\text{aug}(o_t))$ and $f_\theta(\text{aug}(o_{t+1}))$.
     (b) Sample next state actions from the policy $a'_{t+1} \sim \pi_\theta(f_\theta(\text{aug}(o_{t+1})))$.
     (c) Compute the Bellman targets

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

     where $\bar{Q}_{\theta^i}$ and $\bar{Q}_{\theta^j}$ are two randomly sampled critics.
     (d) Compute the loss:

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

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

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

   - In the final part, we will improve the policy. First sample an action from the actor $a'_t \sim \pi_\theta(\textbf{sg}(f_\theta(\text{aug}(o_t))))$ and compute the objective:

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

     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 evaluation results from Tensorboard.

See page 7-8.

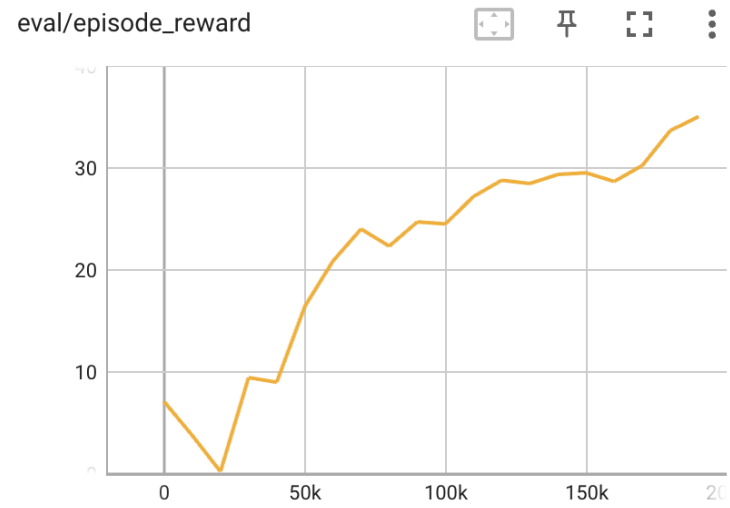
- In the final part, we will explore some optimization parameter choices. The update-to-data (UTD) ratio stands for the number of gradient steps we do, with respect to each environment step. So far we have used only 2 critics and UTD of 1. Repeat the previous part with:

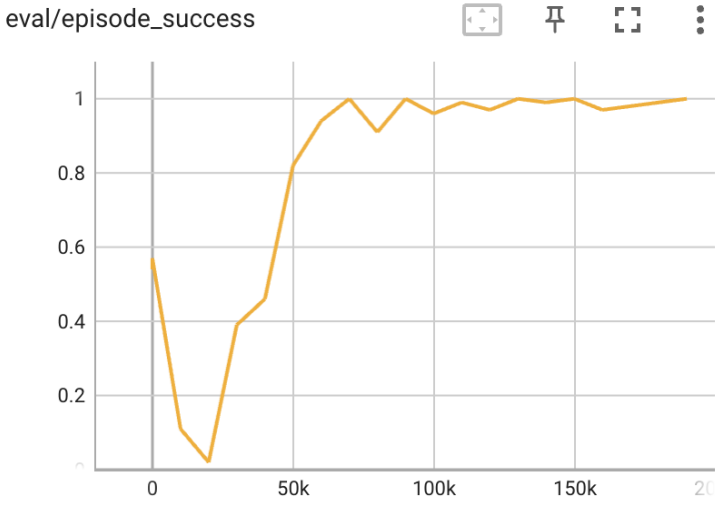
```
python train.py agent.num_critics=10 utd=5
```

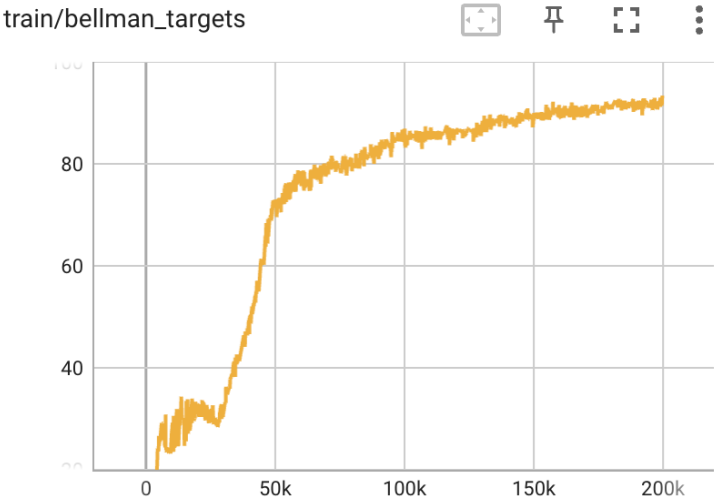
Attach your results and compare them to the previous question. Provide an explanation of why we observe these effects.

See page 9-10 for figures.

We can see that by increasing the number of critics, the agent can receive more accurate value estimates, leading to better performance.
We can also see that with a higher UTD ratio, the target values used to update the critics are more out-of-date but also more stable. This can help prevent over-fitting and improve the stability of the training.
However, increasing the number of critics can increase the computational time and potentially cause over-fitting. In our case, the training times were ≈2h for case 1 and ≈5h for case 2.
Therefore, the choice of the number of critics and the value of the UTD ratio is a trade-off between the stability of the training, the risk of over-fitting, and the computational time and resources.


eval/episode_reward
30
20
10
0
50k
100k
150k


eval/episode_success
1
0.8
0.6
0.4
0.2
0
50k
100k
150k


train/bellman_targets
80
60
40
0
50k
100k
150k
200k

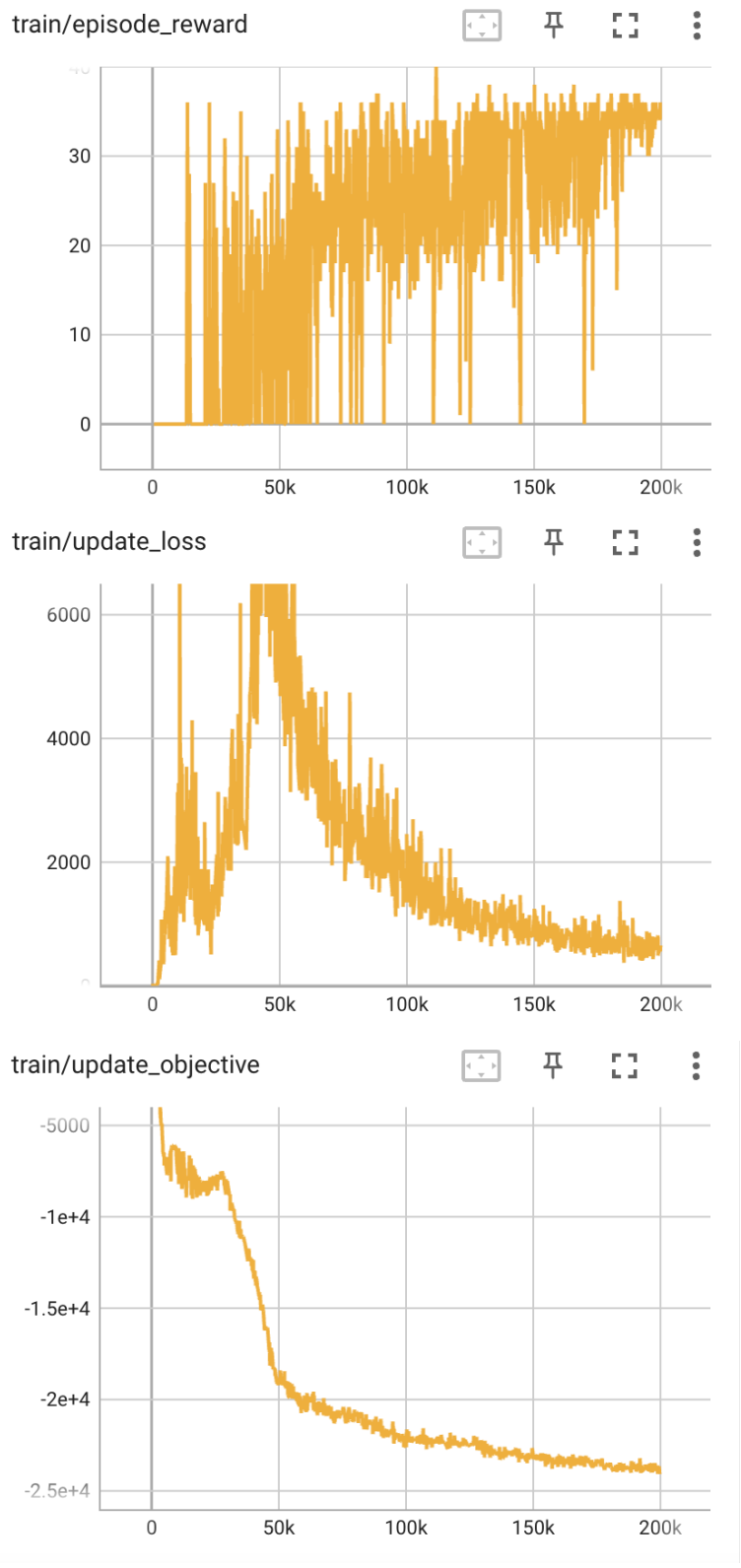
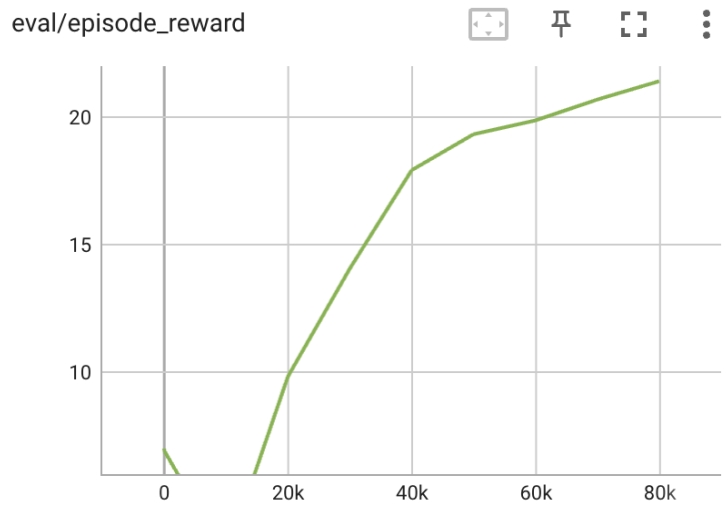
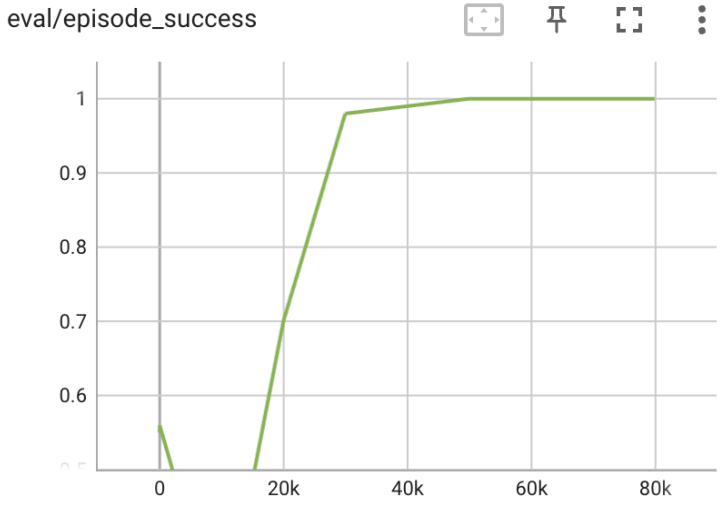
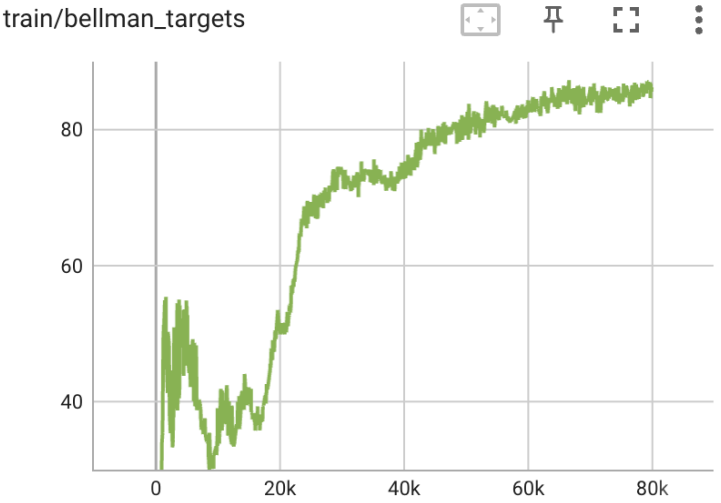


Figure 1: Training with num_critics = 2 and utd = 1


eval/episode_reward
20
15
10
0
20k
40k
60k
80k


eval/episode_success
1
0.9
0.8
0.7
0.6
0
20k
40k
60k
80k


train/bellman_targets
80
60
40
0
20k
40k
60k
80k

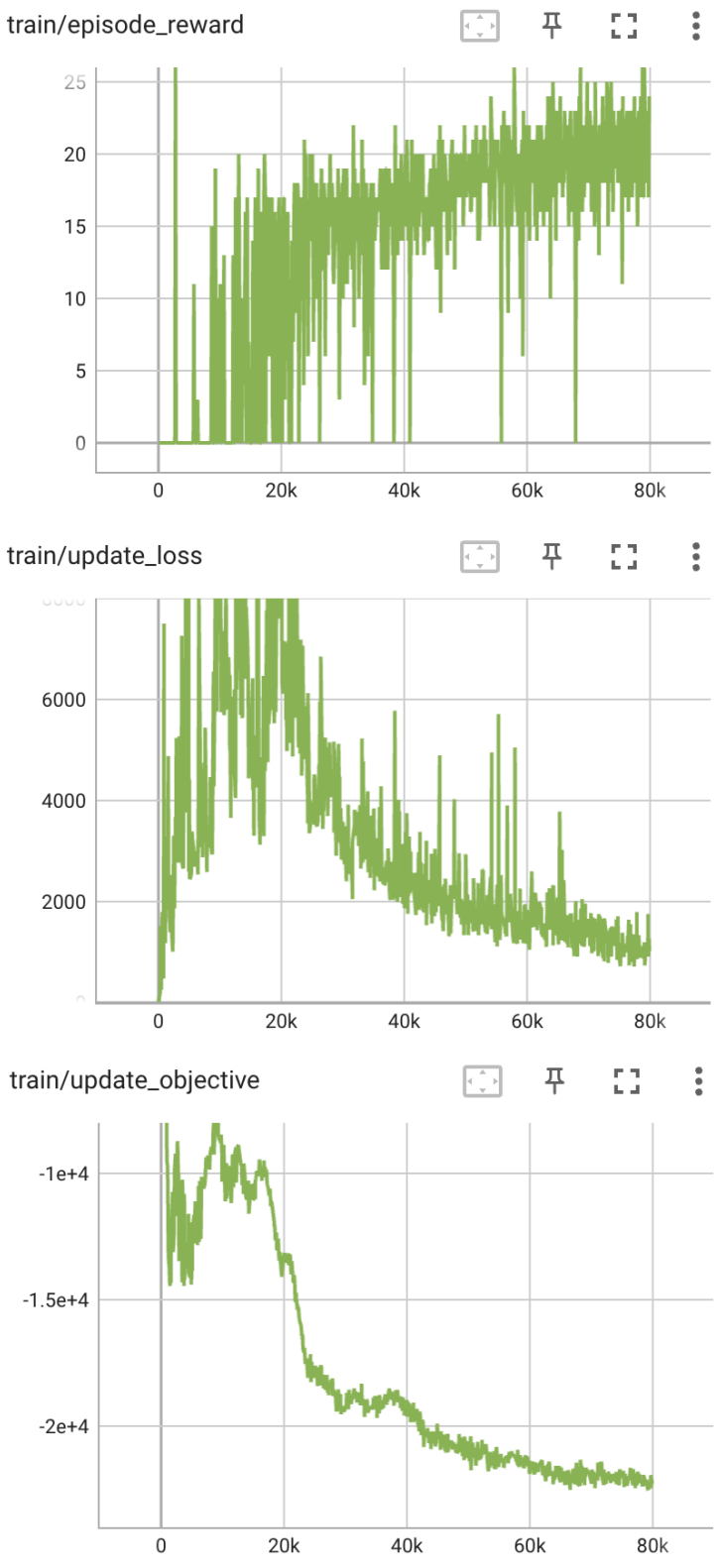


Figure 2: Training with num_critics = 10 and utd = 5