TTIC 31170: Robot Learning and Estimation (Spring 2025)

Problem Set #4

Due Date: May 22, 2025

1 Markov Decision Process: Grid World [20 points]

Consider an agent that navigates in the grid world depicted in Figure 1. The agent can move in each of the cardinal directions, $\mathcal{A} = \{\text{N}, \text{E}, \text{S}, \text{W}\}$, but the state transition is stochastic: if the agent tries to move North, it will go East or West, each with probability $\frac{p_{\text{noise}}}{2}$; if the agent tries to move East, it will go North or South, each with probability $\frac{p_{\text{noise}}}{2}$; if the agent tries to move South, it will go East or West, each with probability $\frac{p_{\text{noise}}}{2}$; and if the agent tries to move West, it will go North or South, each with probability $\frac{p_{\text{noise}}}{2}$. Black cells denote obstacles and if the agent moves towards an obstacle or the environment boundary, it will stay in its current cell, i.e., $P(s_{t+1} = 11 \,|\, s_t = 6, a = \text{North}) = 1 - p_{\text{noise}}$. Once the agent moves, it is able to observe its current state.

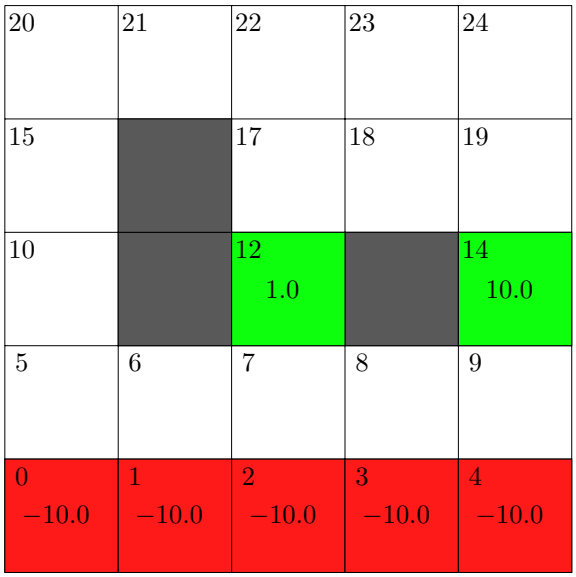


Figure 1: A 5×5 grid world.

The states rendered as green and red are absorbing states. Executing any action in these states will conclude the episode. Taking an action in any of the states in the bottom row (a cliff) will result in a reward of $R(s_t, s_{t+1} = \text{end}) = -10.0$ for $s_t \in \{0, 1, 2, 3, 4\}$. Any action in states $s_t = 12$ or $s_t = 14$ results in a reward of $R = 1.0$ and $R = 10.0$, respectively.

The goal is to find the optimal policy $\pi : \mathcal{S} \to \mathcal{A}$ that maximizes the expected discounted reward for an infinite time horizon. We can model this problem as an MDP $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{R}, \gamma, T \rangle$,

where we are assuming that $T = \infty$.

(a) **[12pts]** One way to solve for the optimal policy is to use value iteration, whereby we initially assign each state a value of 0.0 and perform a series of Bellman updates (backups) until the updated value function is within ϵ of the previous value function:

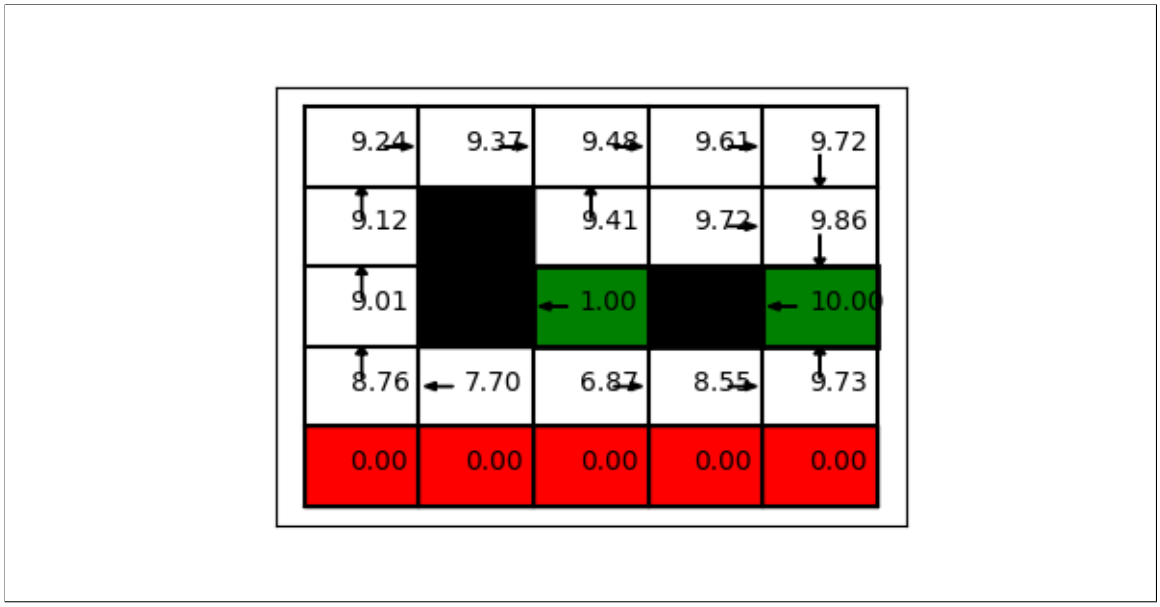
$$\|V_{i+1}(s) - V_i(s)\| < \epsilon \quad \text{where } \|U\| = \max_s |U(s)|$$

Included with this problem is a file `GridWorld.py` that provides a Python class for the grid world problem. Implement value iteration within the `ValueIteration(epsilon)` function, which returns the optimal value function, the optimal policy, and the number of iterations necessary for convergence. You can validate your implementation by using the examples from class, where we considered different settings for p_{noise} and γ.

What is included: There are two files included with this problem set, `GridWorld.py` that defines the MDP class, which is the one that you should edit, and `RunMDP.py`, which calls the value iteration function and draws the resulting value function and optimal policy. You can run value iteration as follows:

```
$ python RunMDP.py --noise 0.2 --gamma 0.99 --epsilon 0.001
```

What to hand in: Your `GridWorld.py` and `RunMDP.py` files, along with the figure that depicts the optimal value function and policy for the above parameter settings.



(b) **[2pts]** Run value iteration for the following two parameter settings, $(p_{\text{noise}} = 0.2, \gamma = 0.1, \epsilon = 0.001)$ and $(p_{\text{noise}} = 0.2, \gamma = 0.99, \epsilon = 0.001)$. Provide the resulting figure for each of these settings and explain the differences in the resulting policies.

We observe that varying the gamma from 0.1 in the first figure to 0.99 in the second dramatically increases the values of the resulting value function. This makes sense since gamma, the discount factor, weights the value of future rewards that are one or more steps in the future and explains the higher valeus obtainable from grid locations that are more than one grid away from the green locations. It also explains the difference in the optimal policies for the squares directly next to the green square with reward 1. In the low gamma setting, the optimal policy dictates moving towards it, while in the high gamma setting, the optimal policy dictates moving towards the green square with reward 10 instead.

(c) **[4pts]** Compare the number of iterations necessary for different settings of the discount factor, $(p_{\text{noise}} = 0.2, \gamma = 0.1, \epsilon = 0.001)$ and $(p_{\text{noise}} = 0.2, \gamma = 0.99, \epsilon = 0.001)$. Why are more iterations required for one setting for γ over the other?

Now, consider the effect of an increased noise likelihood, and compare the number of required iterations for $(p_{\text{noise}} = 0.2, \gamma = 0.99, \epsilon = 0.001)$ and $(p_{\text{noise}} = 0.5, \gamma = 0.99, \epsilon =$

0.001). Explain the difference.

When varying the discount factor, we see that a higher discount factor requires more iterations to converge (20 vs 5 in this case). This is because a higher discount factor weights potential future reward higher, meaning more exploration is necessary to observe the optimal policy over the potential longer sequences of actions. In contrast, a low discount factor leads to only considering the rewards from extremely short sequences of moves, which can be explored much faster.

The increased noise likelihood also means more iterations to converge (43 vs 20 in this case) since there is noise in each potential action taken that must be resolved to determine the value of a particular policy. Specifically, transitions are more likely to lead to intermediary values, which makes changes less pronounced (which is also why the values are smaller across the board when noise is higher) and thus each iteration improves the value function less, meaning more iterations are necessary to reach the convergence threshold.

(d) **[2pts]** If sequential backups of the value function are within ϵ according to the above norm, one can show that the error in the value function estimate relative to the optimal value function is bounded as

$$\|V_{i+1}(s) - V^*(s)\| < \frac{2\epsilon\gamma}{1-\gamma}$$

Explain the dependence on γ.

We see that as gamma goes to 0, only the immediate reward matters and future reward is ignored, therefore the bound goes to 0. As gamma goes to 1, the denominator goes to 0 which results in the bound blowing up and so we have a very long effective horizon, with errors decaying slowly and even a tiny ϵ having potential to cause a large final error. Between 0 and 1, we see that the bound grows roughly linearly for small gamma and then accelerates.

2 Reinforcement Learning: Monte Carlo vs TD(0) [5 points]

(a) **[5pts]** Consider an agent acting in an environment consisting of two states $\mathcal{S} = \{x, y\}$. We would like to estimate the value function (i.e., $V^\pi(x)$ and $V^\pi(y)$) for a given policy in a model-free way based upon observations of state-reward sequences. In particular, suppose that we observe the following episodes (where absence of an entry for $(s_2, \mathcal{R}_2)$ indicates that the episode terminated)

What estimate of the value function would you get if you used the TD(0) algorithm? What would you get if you used Monte Carlo prediction? Compare the results in terms of how correct they are for the given episodes and how they may generalize.

Using TD(0), we see that x was followed once by reward 0 and next state y, so $\bar{R}(x) = 0, \hat{P}(y|x) = 1$ and y always terminated with its next-state value of 0, and $\bar{R}(y) = 0.75$. With these statistics, we see that

$$V_{TD}(y) = \bar{R}(y) = 0.75$$

Episode	$(s_1, \mathcal{R}_1)$, $(s_2, \mathcal{R}_2)$
1	$(x, 0)$, $(y, 0)$
2	$(y, 1)$
3	$(y, 1)$
4	$(y, 1)$
5	$(y, 1)$
6	$(y, 0)$
7	$(y, 1)$
8	$(y, 1)$

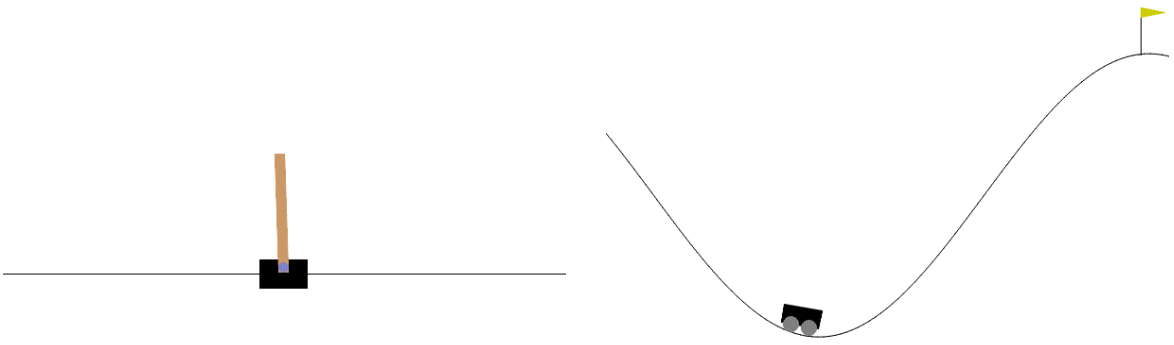
$$V_{TD}(x) = \bar{R}(x) + \gamma V_{TD}(y) = \gamma \cdot 0.75$$

If we use MC first-visit prediction, we observe the following over the corresponding episodes:

1. $(x, 0) \to (y, 0)$, so we have a return of 0+0=0 from x and a return of 0 from y
2. $(y, 1)$, so we have a return of 1 from y
3. $(y, 1)$ so we have a return of 1 from y
4. $(y, 1)$ so we have a return of 1 from y
5. $(y, 1)$ so we have a return of 1 from y
6. $(y, 0)$ so we have a return of 0 from y
7. $(y, 1)$ so we have a return of 1 from y
8. $(y, 1)$ so we have a return of 1 from y

Altogether, we seen that x is seen once with return 0, so $\hat{V}_{MC}(x) = 0$ and y is seen 8 times with 6 returns of 1 and two of 0, so $\hat{V}_{MC}(y) = \frac{6}{8} = 0.75$.

Comparing the two, we see that TD(0) makes an "error" on the single x sample, but is consistent with the dynamics of $x \to y$. The MC method exactly matches the only observed return from x and the average from y. We also see that MC has high variance for rarely visited states. In particular, since we only saw x once, its estimates will fluctuate until many more episodes are gathered. In contrast, TD shares information through bootstrapping, so it is faster with reduced variance, but is exposed to possible bias.



(a) `CartPole-v1` (b) `CartPole-v1`

Figure 2: Continuous control domains.

3 Model-Free Reinforcement Learning [35 points]

In this problem, we will implement and experiment with two model-free reinforcement-learning algorithms—one value-based (Q-learning) and one policy-based (REINFORCE policy gradient)—on two classic control environments.

- **CartPole-v1** (Figure 2a): The cart-pole problem is a classic control task in which a pole is attached to a cart that can move along a horizontal track. Without the ability to apply torque to the pole, the goal is to apply forces to the cart to balance the pole upright, starting from an initial state where the pole may be leaning (i.e., it may be necessary to first swing the pole).

  We can formulate this as a reinforcement learning (RL) problem. We can consider the state $s = \begin{bmatrix} x & \dot{x} & \theta & \dot{\theta} \end{bmatrix}^\top \in \mathbb{R}^4$ to be a four-dimensional vector that includes the cart position x and velocity $\dot{x}$, the angle of the pole θ (measured in radians from vertical, with 0 corresponding to the upward vertical direction and positive angles increasing counter-clockwise), and the pole's angular velocity $\dot{\theta}$. There are two available actions $\mathcal{A} = \{\text{push left}, \ \text{push right}\}$ that correspond to applying a fixed force to the cart either in the left or right direction. In the default setting, the agent receives a reward of $+1$ at every time step until either:

  (i) $|x| > 2.4$ m,

  (ii) $|\theta| > 12°$, or

  (iii) 500 steps have elapsed

  after which the episode terminates. **Note**: By setting `sutton_barto_reward=True`, this reward is changed to 0 per step and -1 upon termination.

- **MountainCar-v0** (Figure 2b): Mountain car is another classic benchmark in which an under-powered car must drive up a steep hill. Due to insufficient engine power, the car cannot reach the goal at the top by driving forward alone. Instead, the car must first build momentum by rocking back and forth between the two slopes.

  Again, we can formulate this as an RL problem. In this case, the state $s = \begin{bmatrix} p & v \end{bmatrix}^\top \in \mathbb{R}^2$ includes car's position $p \in [-1.2, \, 0.6]$ and velocity $v \in [-0.07, \, 0.07]$. There are three available actions $\mathcal{A} = \{\text{push left}, \ \text{coast}, \ \text{push right}\}$ that correspond to three different throttle settings. The agent receives a reward of -1 at every time step until it reaches the goal $p \geq 0.5$,

or until 200 steps have elapsed, at which point the episode terminates.

What is included: As part of this problem set, you are provided with the following files:

- `qlearning.py`: Defines a Q-learning agent
- `policy_gradient.py`: Defines the policy gradient agent
- `main.py`: Used for training and evaluation

You can run training using the following command:

```
$ python main.py --env CartPole-v1 --algorithm qlearning
```

What is included: We are using the `gymnasium` API (`https://gymnasium.farama.org/index.html`) for this problem. The full `gymnasium` package requires dependencies that may not be supported for some operating systems. To install `gymnasium` **only** with the classic control environments, run:

```
$ pip install "gymnasium[classic_control]"
```

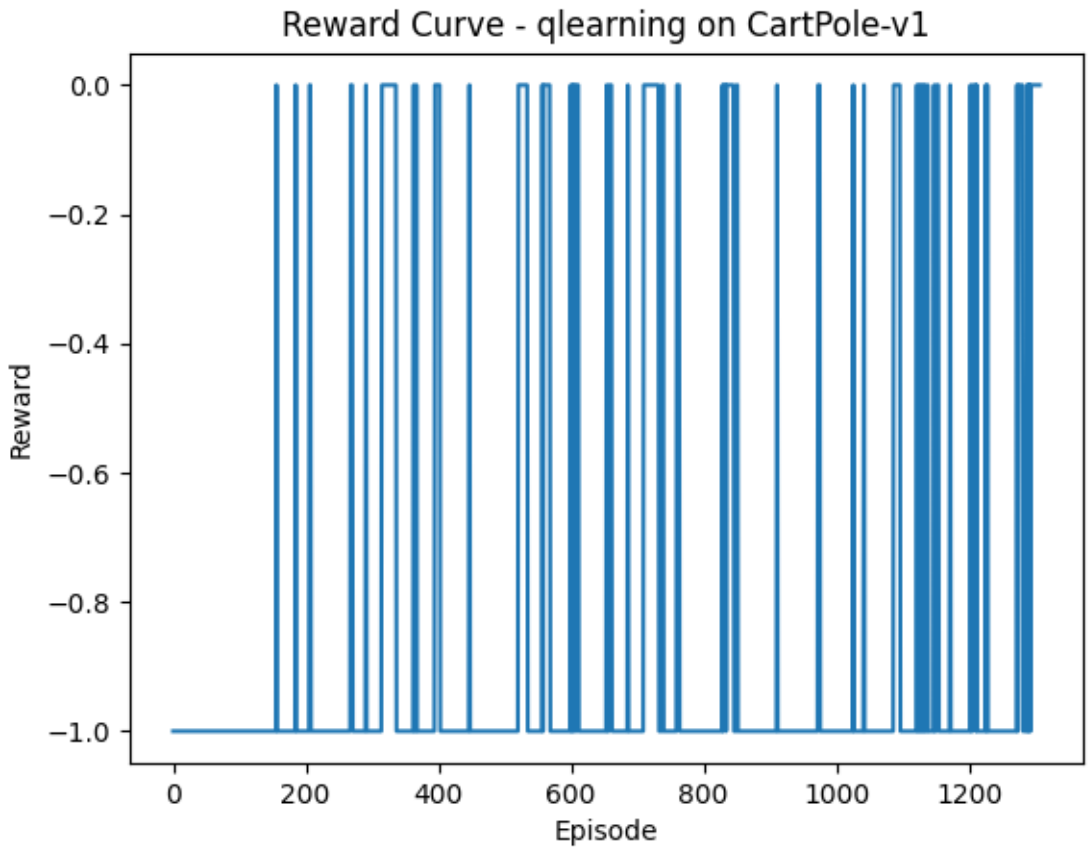
What to hand in: Your `qlearning.py` and `policy_gradient.py` files, along with a PDF that includes all requested plots and written responses. Plots should be embedded directly in the PDF and clearly labeled.

(a) **[10pts]** Complete the epsilon-greedy policy in `QLearningPolicy.sample_action()` and the Q-learning update in `__call__`. The policy should select random actions with probability ϵ and greedy actions otherwise. The Q-learning update should apply:

$$Q(s, a) \leftarrow r + \gamma \max_{a'} Q(s', a')$$

for each transition. The network's MSE loss should be minimized over a batch of replayed experiences. The code includes a replay buffer and decaying ϵ.

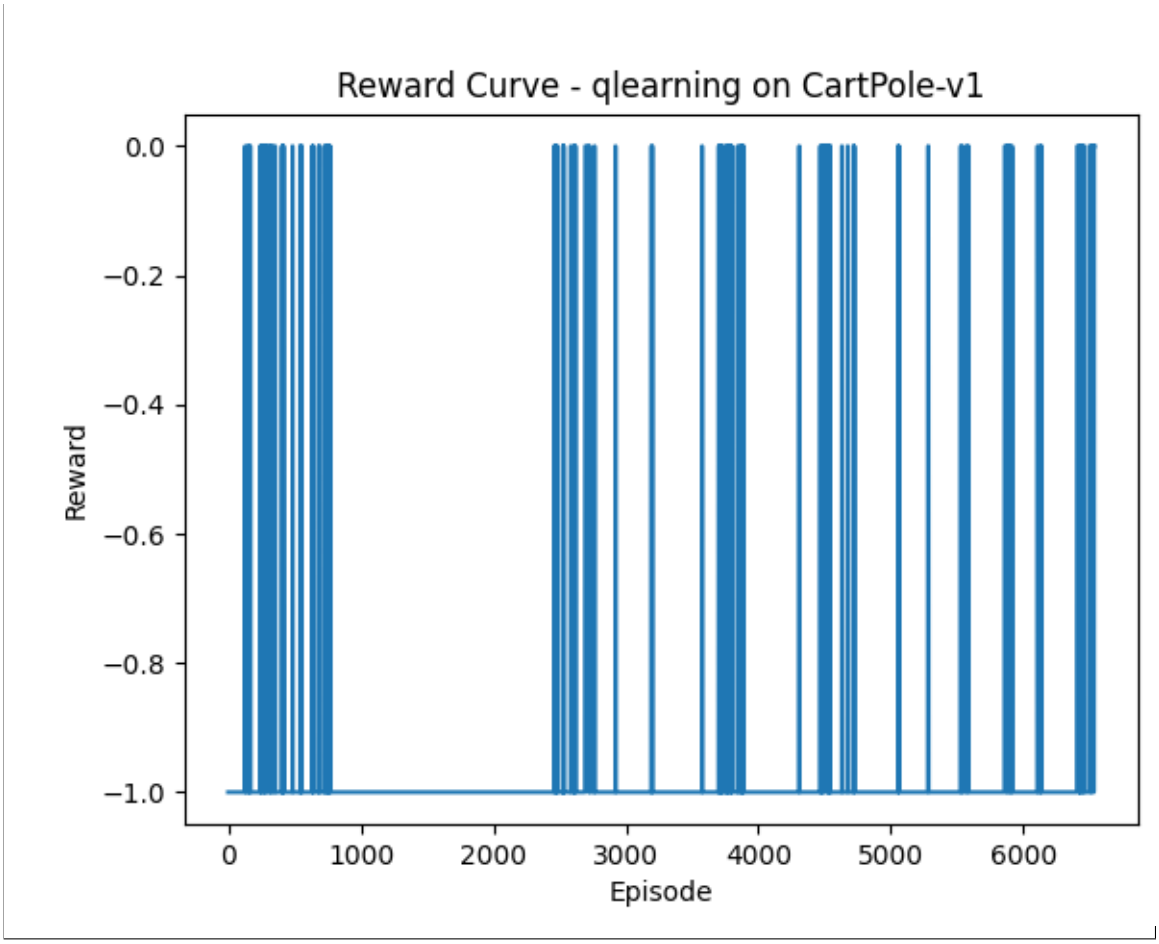
(b) **[2pts]** Train your Q-learning agent on `CartPole-v1` for a sufficient number of episodes. Record and plot the total episode reward vs. episode number. Submit this plot. Briefly describe your observations: does reward improve over time?



Reward does indeed seem to generally improve over time as does average episode length.

(c) **[3pts]** Set the replay buffer capacity to 100 instead of the default $10,000$. Train your Q-learning agent and compare the resulting reward curve to the default setting. How does the smaller buffer affect learning speed and final performance? Explain why this happens.

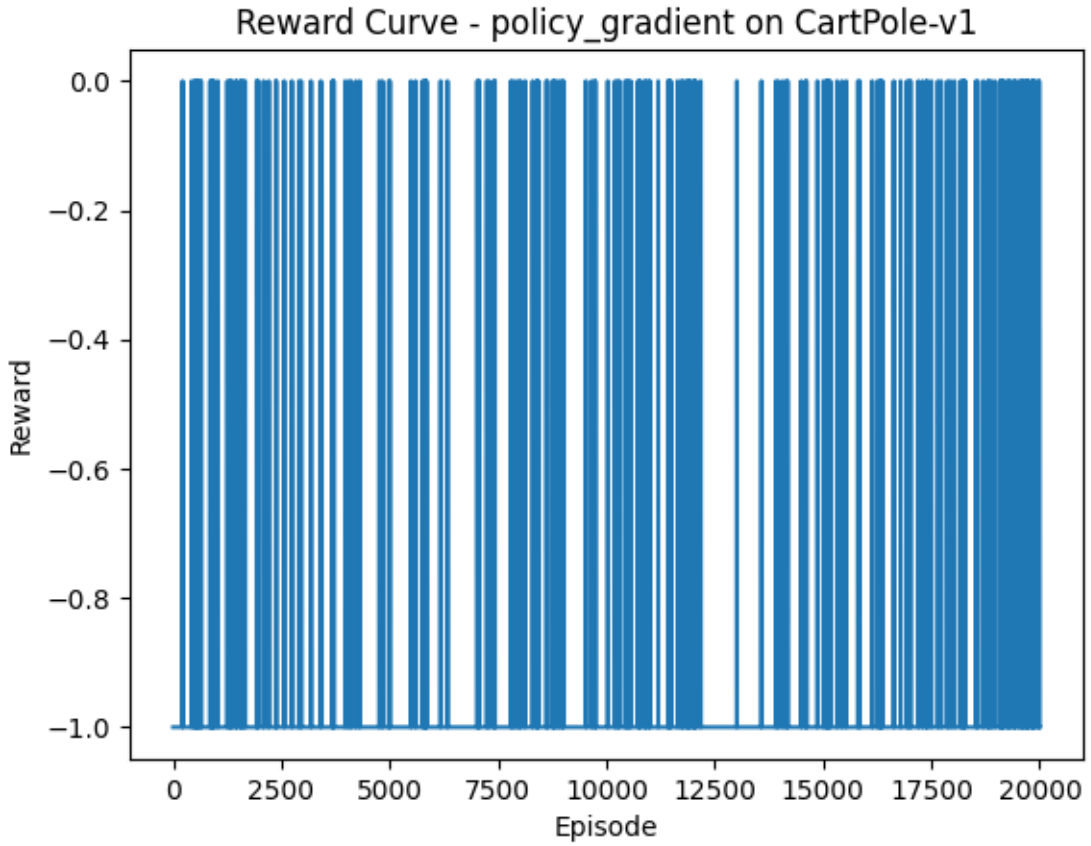
The default I used was 1000 and compared to the 100 replay buffer capacity, The learning speed with the smaller replay buffer is dramatically slower (6500 episodes as opposed to 1300), which makes sense given the additional instability introduced by limiting the size of the replay buffer so dramatically. This is likely a result of having many similar/correlated samples, leading to oscillations and divergence. Final performance is similar due to our stopping policy.



(d) **[10pts]** In `PolicyGradientPolicy.sample_action`, sample from $\pi_\theta(a \mid s)$ and store the log-probabilities. In `__call__`, at episode end, compute discounted returns $G_t = R_t + \gamma R_{t+1} + \cdots$ and update the policy using the loss:

$$L = -\sum_t G_t \log \pi_\theta(a_t \mid s_t)$$

(e) **[2pts]** Train your REINFORCE agent and plot the total episode reward over time. Submit the plot. Comment on convergence behavior and any noise in the curve.



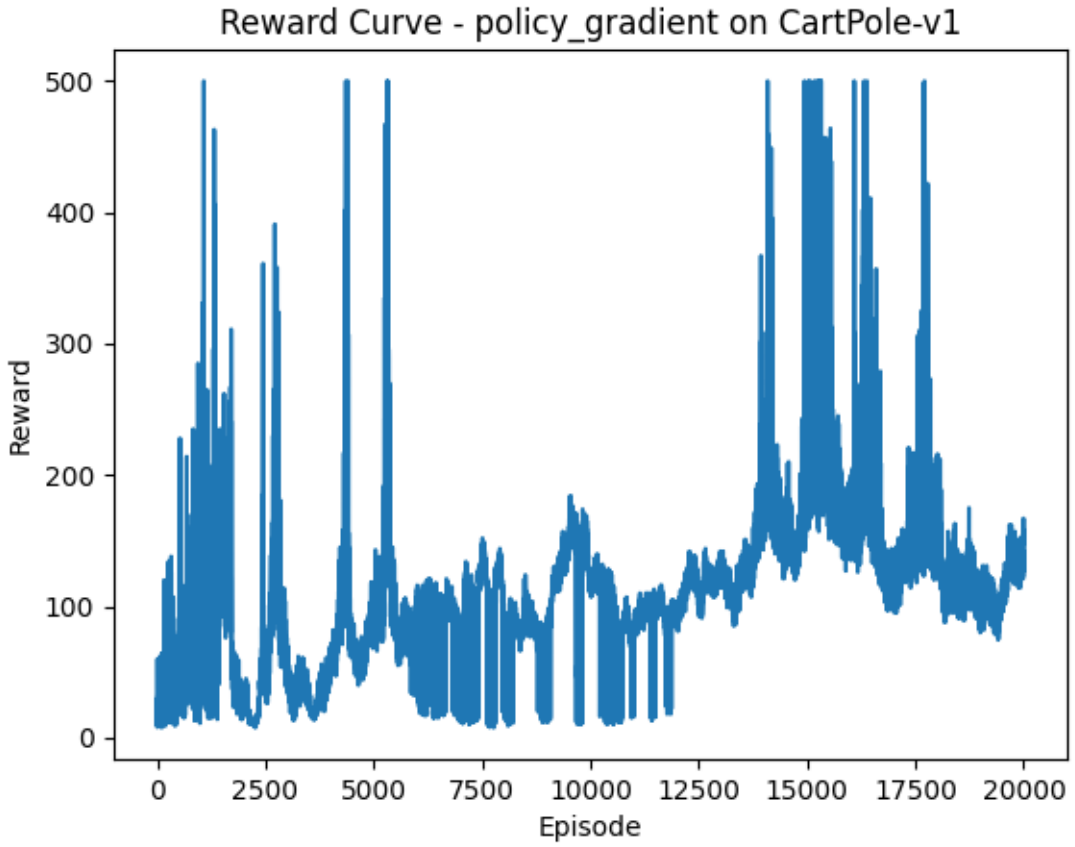
Final test results are:

- Test Episode 1: Reward = 257.0
- Test Episode 2: Reward = 388.0
- Test Episode 3: Reward = 203.0
- Test Episode 4: Reward = 228.0
- Test Episode 5: Reward = 304.0

It seems to converge quite slowly, though there is a clear improvement over time, especially towards the final 2 thousand episodes or so.

(f) **[3pts]** Run the policy gradient agent on `CartPole-v1` with both reward settings: (1) `sutton_barto_reward=True` and (2) `False`. Compare the learning curves and final reward. Which reward scheme leads to better learning and why?

The run with `sutton_barto_reward=True` can be seen in the previous sub-part and the `False` setting can be seen below.



It achieves final test results of

- Test Episode 1: Reward = 133.0
- Test Episode 2: Reward = 152.0
- Test Episode 3: Reward = 149.0
- Test Episode 4: Reward = 171.0
- Test Episode 5: Reward = 146.0

We see that without sutton barto, the reward curve is shaped fundamentally differently since it is no longer the case that the agent is only rewarded negatively for terminating, but is rather positively rewarded for each step taken, including the termination step. It seems that the model learns slower and in a more unstable fashion.

(g) **[5pts]** In `shape_reward`, implement a reward shaping strategy for `MountainCar-v0`. You may use velocity-, position-, or potential-based shaping. Train the policy with and without shaping. Describe the shaping function and whether it helps. Justify your design and discuss observed effects.

Without reward shaping, it seems that the mountain car never leaves the initial valley, even over 20k episodes, i.e. it never solves the scenario. In particular, the policy net gradients seem to go to 0 within several thousand episodes.

I added to each reward $k\cdot + \frac{1}{2}k \cdot v^2$ to mimic the physical energy of the system in terms of potential energy with the first term and kinetic energy with the second. This is essentially to reward the agent for achieving the higher energy levels that are necessary to reach the top right while also directly encouraging it to move towards the top right in its exploration. I experimented a little with k values and also eventually realized in my exploration that the potential energy term was doing most of the heavy lifting just due to the scale factors, so I ended up removing the kinetic energy term and focusing just on the potential energy. With $k = 15$, I observed much better progress and solutions within 5k episodes.

4 Time and Collaboration Accounting

(a) **[1pt]** Did you work with anyone on this problem set? If so, who?

N/A

(b) **[1pt]** How long did you spend on this problem set?

5-10 hours.