Homework 4 [**50 pts**]

Due Friday, December 13 at 5pm

For this homework assignment, you will implement Q-learning to solve the CartPole-v1 task (`https://gymnasium.farama.org/environments/classic_control/cart_pole/`) from the OpenAI Gymnasium. Starter code is provided.

1 Background

The CartPole-v1 task corresponds to balancing a pole on top of a cart that can be moved back and forth. The state space is a tuple of four real numbers:

1. The cart's position (in the range $[-4.8, 4.8]$)
2. The cart's velocity (in the range $(-\infty, \infty)$)
3. The pole's angular position (in the range $(-0.41887903, 0.41887903)$, i.e. $(-24°, 24°)$ in radians)
4. The pole's angular velocity (in the range $(-\infty, \infty)$)

The actions available to the agent are 0: "move the cart left" or 1: "move the cart right". The game ends once the cart's position leaves the range $(-2.4, 2.4)$ or once the pole's angle leaves the range $(-0.2095, 0.2095)$. The agent receives a reward of $+1$ for each timestep that the game continues.

The fact that this is a continuous state space complicates Q learning, so I have provided code which discretizes the state space by splitting up the cart position into 30 buckets of width 0.32, the cart speed into 24 buckets of width 0.2 (with speeds outside the range $[-1.4, 1.4]$ grouped in the first or last bucket), and so on.

Once this discretization is done, there are a finite number $(30 \times 24 \times 30 \times 30)$ of possible states and two possible actions to take from each state. Therefore, there are exactly 1,296,000 possible arguments to the Q function, $Q(s, a)$. We therefore can represent the Q function as a table, `q_table`, which has dimensions $(30 \times 24 \times 30 \times 30 \times 2)$.

Recall that if we knew the Q function for the optimal policy, then we would know the optimal action to take from any state:

$$a^*(s) = \arg\max_a Q^*(s, a)$$

Therefore, our goal will be to iteratively update `q_table` until hopefully `q_table`$(s, a) \approx Q^*(s, a)$.

To do this, we will repeatedly simulate playing the CartPole game. One "episode" is one play of the game, which results in a sequence of states, actions, and rewards $\{s_t\}$, $\{a_t\}$, and $\{r_t\}$.

To implement Q learning, we begin with an initial guess at the Q function (I implemented this for you in the starter code), and we gradually improve it by using the update equation discussed in class:

$$\texttt{q\_table}(s_t, a_t) \leftarrow (1 - \texttt{lr})\texttt{q\_table}(s_t, a_t) + \texttt{lr}\Big(r_t + \gamma \max_a \texttt{q\_table}(s_{t+1}, a)\Big) \tag{1}$$

where `lr` $\in (0, 1)$ is a learning rate hyperparameter chosen by you, and $\gamma \in (0, 1]$ is the MDP discount factor (which is also chosen by you for this assignment).

By any method of training, you will visit various states s and play various actions a from those states. In order to get the value of `q_table`(s, a) just right for a particular state s and action a using these updates, it will require visiting that state and playing that action *at least* one time, and most likely a lot more than that. But since there are over 1 million possible combinations of states and actions, this means that millions, tens of millions, or possibly even hundreds of millions of timesteps would need to be simulated in order to get every value in the `q_table` just right.

Luckily, we don't really need to get all of the values correct—we only need to get a good estimate of the Q function at states and actions that are likely to arise when using a good policy! If playing well means we will never end up in a particular state, then as long as we play well, who cares what the Q function is for that state?

Of course, at the beginning of training, we do not know which states and which values to focus on because our Q function is bad, but over time, as our Q function improves, we can trust it more. This motivates what is called an "ϵ-greedy" approach to exploring the state space.

In each episode that we simulate, we need to choose which action to play from each state. The ϵ-greedy approach is parametrized by $\epsilon \in [0, 1]$, and it says that we should do one of two things: with probability ϵ, we will play a *random* action (i.e. 0 with probability 0.5 or 1 with probability 0.5); alternatively, with probability $1 - \epsilon$, we play the action indicated by our Q function, i.e. $\arg\max_a \texttt{q\_table}(s_t, a)$.

At the beginning of training, we should not trust our Q function estimate, so in the early episodes, ϵ should be very close to 1. This makes it so that we usually play random actions which help us explore the state space and figure out what moves are good or bad.

Later on, as training progresses, we should gradually decrease ϵ towards zero, because now our Q function is fairly accurate and we want to focus on improving its estimates for the important states, and we care less about exploring.

2 The assignment

For the actual assignment, you will need to write code that does the following:

1) [**25 pts**]: Simulate `n_episodes` episodes of the CartPole game using an ϵ-greedy strategy to select which action to play. After each action taken, update `q_table`(s_t, a_t) using the equation (1).

2) [**10 pts**]: Select and tune the hyperparameters `n_episodes` (the number of episodes to train for), `gamma` (the MDP discount factor), `epsilon` (the ϵ-greedy parameters—this should be equal to 1 for the first episode and decrease towards 0 as you train), and `lr` (the learning rate in $(0, 1)$).

3) [**15 pts**]: Whatever your implementation and hyperparameter selection, after training for **at most 50,000 episodes**, an agent that plays according to your estimated Q function should be

able to keep the pole upright for at least 250 time steps at least 1 time out of 3. (Use the provided `simulate_and_render` function to simulate running your agent. This function also renders the game as .mp4 files so that you can see your agent in action).

Things to be submitted:

1. Your Python code.
2. One paragraph describing the hyperparameters that you used (again, `n_episodes` should be at most 50,000).
3. Report the number of timesteps that your agent remains upright out of three trials simulated with `simulate_and_render` (if all three attempts earn less than 250 reward, then you have either selected poor hyperparameters or there is a bug in your training code).

Notes:

- During training, you may want to end a given episode early (e.g. after 500 or 1000 timesteps) to prevent a single episode from taking too long.
- As a starting point, I'd suggest using `gamma` ≈ 0.98, `lr` ≈ 0.1, and decrease `epsilon` smoothly from 1 to 0 across the training episodes. However, you can likely get faster training / better performance using different values.
- If implemented correctly, training for 50,000 episodes should take no more than 3-5 minutes.