Mini-project 1: Dealing with sparse rewards in the Mountain Car environment

Instructions on submitting the project

What should you submit?

Please submit your Python code and your project report in a single zip file via the moodle page. The file name should have the structure `MP1_MountainCar_NameMember1_NameMember2.zip`. The zip file should contain the following:

1. **Report** (in pdf format): Document all your solutions in a project report of at most 5 pages (one per group). This report will be the basis of your mini-project grade. Write clearly and concisely and present figures carefully (axis labels, legends, etc.). Make the plots easily understandable by adding reference points, if needed. Make sure you explain and interpret all your plots.
2. **Jupyter notebook** (in ipynb format): The Jupyter notebook should contain code for reproducing your results. Your code will not be graded, but it will be used for the fraud-detection interviews, so make sure it is well-documented and readable.

If you work in teams both of you should submit the same zip file. On moodle, make sure you press the submit button before the deadline.

When should you submit?

You have two options:

1. Submit by Monday, 27th May (before 23:55) and be available for the fraud detection interview between 29th and 31st May.
2. Submit by Sunday, 2nd June (before 23:55) and be available for the fraud detection interview on 4th and 5th June.

1 Introduction

1.1 Mountain Car Environment

The environment considered in this project is the Mountain Car environment implemented in Gymnasium (documentation). A car is placed in a valley at a random location. There are 3 possible actions: accelerate to the left, accelerate to the right, do nothing. We provide an introduction to interacting with the Mountain Car environment in the tutorial notebook. The goal is to reach the top of the right hill. The car is underpowered, so it needs to gain momentum by going back and forth before being able to reach the goal. The reward at each step is -1 except when the goal is reached where it is 0. The main challenge of this environment is the sparse reward function. Before finding the goal for the first time, the car will have no clue on what to do. We will see how this issue can be tackled.

1.2 Agents

We will implement a model-free algorithm (DQN) and a model-based one (Dyna). The goal is to understand how these two algorithms work, and to compare them. All agents in this project will follow the same structure to facilitate simulations. Concretely, an agent should be implemented as a Python class and have the following functions:

- `observe(self, state, action, next_state, reward)` : called upon observing a new transition of the environment.

- `select_action(self, state)` : pick an action from the given state.
- `update(self)` : called after each environment step. This is where all the training takes place.

2 First steps

We will start by gaining intuition on the task at hand by running a random agent. Create a `RandomAgent`, which selects actions randomly. Run the agent on the environment until the episode is either truncated or terminated. Store the episode and render (visualize) it.

all episodes last 200 steps

Now run the agent for 100 episodes. At every episode initialize the environment with a new randomly sampled seed. Plot the duration of each episode as a scatter plot. What do you find? Explain.

3 DQN

We'll begin by implementing the DQN algorithm, starting with a basic version. Subsequently, we'll introduce modifications to improve its performance.

3.1 Implementation

Create a `DQNAgent` class that implements a DQN agent that uses a feed-forward network to estimate Q-values and an ϵ-greedy policy to take actions. Since DQN is an off-policy RL algorithm, make sure your class has a replay buffer that allows to add observed transitions (s, a, r, s') and sample batches of transitions for the network updates.
Your class should have *at least* the following attributes:

SARSA ??

- Q : a multi-layer perception (MLP) that takes a state s as input and outputs $Q(s, a)$ for each action a. The number and size of hidden layers are parameters that can be tuned. You can start with an MLP with two hidden layers, each layer having 64 neurons.
- γ : discount factor for the Q-values. A good default value is 0.99.
- ϵ : for the ϵ-greedy policy. Note that ϵ can vary during the training, starting with a high value and ending with a low value. You can for example implement a schedule where ϵ is initialized at 0.9 and exponentially decays until it reaches a minimum value of 0.05.
- `replay_buffer`: data structure to store past episodes for training. It should have a fixed capacity (the capacity of storing 10000 latest transitions is a good default to start with).
- `batch_size`: size of the batch for one training step. You can use a batch size of 64.
- `optimizer`: Optimizer for learning. AdamW is a good default.

Feel free to add more class attributes that you find useful. *Hint:* Be careful in handling your gradients, and think about how to deal with terminal states.

3.2 No auxiliary reward

a faire mais easy

Train the `DQNAgent` for 1000 episodes and report its loss and average cumulative reward per episode. Does your agent solve the task?

The main challenge of this task is the sparse reward problem discussed in the introduction. Let's explore ways to address this issue and help our agent to learn faster.

3.3 Heuristic reward function

We will now create an auxiliary reward function to assist the agent in learning better behavior. Try to think of a reward that would help the agent solve the task. Which states location-wise should the agent strive for? Implement and add this reward to the environment reward in the simulation.

Run your `DQNAgent` with auxiliary reward for 3000 episodes. Does it successfully solve the task? What happens if you choose very small/very large auxiliary rewards? Experiment with the parameters of your auxiliary reward to improve the performance. With a good set of parameters, it should solve the task in less than a thousand episodes.

Report your results using the following plots:

- Episodes duration: Plot the duration of each episode with a scatter plot.
- Reward and Cumulative reward: Plot the accumulated reward per episode. You can use an averaging window to smoothen the curve. In a separate plot, show the cumulative reward over episodes (reward accumulated up to episode n). In both cases, separately depict the environment reward, the auxiliary reward and their sum. Interpret your plots and explain your choice of auxiliary reward.
- Cumulative number of successes: For each episode n, plot the number of times the goal has been achieved up to episode n. A perfect agent (one that solves the task in each episode) would show a straight line going up.
- Training loss : plot the training loss computed in the update function of the agent, for each step. Is the loss behaving as you would expect? Explain.

3.4 Non domain-specific reward

Heuristic reward functions work well if they are well-designed, but have the undesirable property that they require knowledge about the specific environment that they are designed for and do not necessarily generalize to different environments. Now, you will implement a reward function that is not tied to a specific goal and encourages the agent to explore new parts of the environment by rewarding the agent highly for reaching less-frequently encountered areas and giving lower rewards for reaching well-known states. This *intrinsic* reward aims to motivate the agent to explore until it eventually reaches the goal state. To implement this intrinsic reward in a discrete environment, we could track the count of each state and provide a reward for reaching a state inversely proportional to its count.

Unfortunately, however, our environment is continuous, so count-based methods cannot be applied directly. Instead, you will implement Random Network Distillation (RND) [Burda et al., 2018]. This technique involves a target network (with fixed random weights) and a predictor network (with adaptable weights), both MLPs taking a state as input and outputting a single number. They are both initialized randomly and independently. The target network is fixed, while the predictor network will be trained to match the target network on the states that are encountered. After some update steps, the predictor's outputs should be close to the target's outputs on states that have been encountered frequently, but not on states that have never or only rarely been encountered. With this in mind, at any point s' that the agent reaches, we define the RND intrinsic reward as the squared difference between the predictor and the target applied to s', with some normalization.
Implement this intrinsic reward:

- Add a predictor and target network to the `DQNAgent` class.
- In the update function, train the predictor network to match the output of the target on the next state s' using MSE.
- Add a function to compute the RND reward given the next state s'. Make sure you add the following normalizations:
  - Before evaluating the predictor and target networks for s', nomalize s' using a running estimate of the mean and std of the states: $\frac{s'-mean}{std}$.

– Normalize the squared difference between predictor and target using a running estimate of its mean and std: $\frac{r - mean_{init}}{std_{init}}$ and clamp the resulting value between -5 and 5 to obtain the RND reward.

Why are the two normalizations necessary, and what would happen if we neglect to implement them? Explain. *Hint:* The running mean and std estimates will be inaccurate during the first few updates; only start computing your RND reward after a few initial steps that you use to obtain initial estimates of the running mean and std.

too high is good no ? (too low => like random)

- The balance between environment rewards and your RND reward should be regulated by a `reward_factor` parameter that is multiplied with your RND reward. Explain what range of values makes sense to use for the `reward_factor` parameter (reward balancing parameter). What would happen if we set the `reward_factor` parameter too high or too low?

- Run the `DQNAgent` with Random Network Distillation auxiliary reward and plot the results as before (duration of episodes, reward and cumulative reward, cumulative number of successes and training loss plots). Comment on the results and compare them with your heuristic reward. What trends can you observe? Think about how the two reward functions compute auxiliary rewards and explain your observations.

- *Bonus*: Another way of implementing a prediction-based reward method is to train a predictor that takes current state s and action a as inputs and tries to predict the next state s' (forward dynamics prediction). Explain why RND is a superior choice compared to training a forward dynamics prediction model for stochastic environments.

4 Dyna

Let's now implement a model-based approach to address the problem at hand. Dyna uses observations to construct a model of the environment and uses this model to derive a policy. However, it is important to note that Dyna is designed for discrete tabular state spaces. Therefore, our first step involves discretizing the state space before applying Dyna.

4.1 State discretization

In the mountain car environment, the state ranges from $[-1.2, -0.07]$ to $[0.6, 0.07]$. To create a discrete state space, we divide the state space into bins. The bin sizes should differ for the two state dimensions.

use the dynamics equations and project on the grid right ?

4.2 Model building

Do i use a NN for each ?

During training, the agent builds a model of the environment consisting of two key components: an estimation of transition probabilities $\hat{P}_{s,a}(s')$ and an estimation of the reward for each state-action pair $\hat{R}(s, a)$. $\hat{P}_{s,a}(s')$ represents the expected probability of transitioning to state s' after taking action a in state s, while $\hat{R}(s, a)$ is the expected reward when taking action a in state s. These estimates are updated with each new observation.

4.3 Implementation

Create a `DynaAgent` class with at least the following attributes:

- `discr_step` : size of the bins for discretizing the states. Should be a vector of size 2 (one step size for each state dimension). You can start with a discretization step of 0.025 for the first dimension and 0.005 for the second.

- γ : discount factor for the Q-values. A good default value is 0.99.

- ϵ : for the ϵ-greedy policy. Note that ϵ can vary during the training, starting with a high value and ending with a low value. You can for example implement a schedule where ϵ is initialized at 0.9 and exponentially decays until it reaches a minimum value of 0.05.

optimization steps

- k : number of updates to perform in the `update` function.
- $\hat{P}$: 3-D array of size $(n_{\text{states}}, n_{\text{actions}}, n_{\text{states}})$. The value at index (s, a, s') should be the estimated probability of reaching state s' after taking action a in state s. For each (s, a), initialize this array with uniform distribution over next states s'.
- $\hat{R}$: 2-D array of size $(n_{\text{states}}, n_{\text{actions}})$. At index (s, a) should be the expected reward after taking action a in state s.
- Q : 2-D array of size $(n_{\text{states}}, n_{\text{actions}})$. Contains the estimated Q-values for each state-action pair. Initialize this array with zeros.

Feel free to add more class attributes that you find useful.
After a transition (s, a, r, s'), yourf `DynaAgent` should do the following:

1. Discretize the states s and s' observed in the given transition.
2. Update the model of the transition probabilities $\hat{P}_{s,a}(s')$ and the expected reward $R(s, a)$ for taking action a in state s.
3. Update the Q-value at state s using the formula:

$$Q(s, a) \leftarrow \hat{R}(s, a) + \gamma \sum_{s'} \hat{P}_{s,a}(s') \max_{a'} Q(s', a')$$

4. Perform further updates for the Q-values of k randomly chosen state-action pairs that were already encountered:

$$Q(s_k, a_k) \leftarrow \hat{R}(s_k, a_k) + \gamma \sum_{s'} \hat{P}_{s_k,a_k}(s') \max_{a'} Q(s', a')$$

Like your DQN agent, you Dyna agent uses an ϵ-greedy policy on the Q-values, with a suitable schedule to decrease ϵ.

4.4 Results

Run the `DynaAgent` on the environment with no auxiliary reward for 3000 episodes.

- Does it solve the task?
- Play with the size of the discretization steps. What happens if you use overly large bins? On the other hand, why would we refrain from using too small bins? Give an intuitive explanation, backed up by experimental results.
- Create the same plots as before. Instead of the training loss plot the Q-value update step instead. Comment on trends observed in the plots.
- Try to understand why the `DynaAgent` manages to solve the task even without any auxiliary reward. Explain your reasoning.
- When running the agent for a sufficiently large number of episodes, do you see a pattern in the episode duration plot? Try to find an explanation for this pattern.
- In addition to the previous plots, create a new one illustrating the estimation of the Q-values after learning. The x-axis represents the car's position, and the y-axis its velocity. On a given position and velocity (i.e state s), the plot should show $\max_a Q(s, a)$. Do not show values for states that were never encountered What pattern do you see? Comment on the plot.
- On top of the plot showing Q-values, plot a curved line showing some episode trajectories. Choose key episodes that show the progression of the agent at solving the task. Show each of these trajectories in separate plots. Can you relate what you see to the patterns in the episode duration plot? Explain.
- *Bonus*: Create plots showing the progression of learned Q-values throughout training. Create separate plots showing $\max_a Q(s, a)$ (same as before) for snapshots of `DynaAgent` at different points in training.

4.5 Comparison with DQN

We now want to compare the 3 agents that we have trained: the DQN with heuristic rewards, the DQN with RND reward, and the Dyna agent.

- Create a plot where you compare the *training* performance of each agent: plot the environment rewards achieved by each of the three agents during training. Comment on the results.
- Now compare their *testing* performance on 1000 new episodes, each one initialized with a randomly sampled seed as before: Set $\epsilon = 0$ and run each of the trained agents on testing environments. Make sure you are comparing the agents on the same set of seeds. Plot their performance and comment.

5 Conclusion

Compare DQN and Dyna in the way they approach the problem, and in the resulting behavior.