046194 – Practical Exercise 3
RL with Function Approximation

1 General Instructions

1. Exercise is due on June 30th.
2. Python code for the mountain car game is provided in the repository: `https://github.com/tomjur/rl_hw_3_public`.

   Use the following command to get the files:

   `git clone https://github.com/tomjur/rl_hw_3_public.git`

   and you may install all the required dependencies by executing in the project directory:

   `pip install -r requirements.txt`
3. You are required to submit both code and answers to the questions. The code should be submitted with a text file describing the code – what each method does, what question each code piece solves. Submit a single ZIP file that includes the code, its description, and a PDF for the answers and figures.

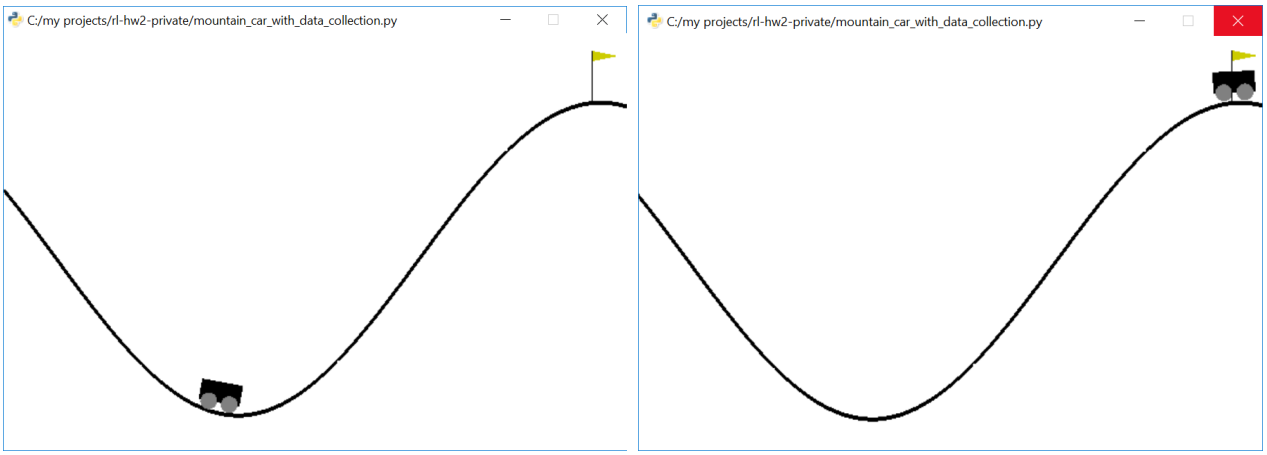
2 Preliminary: the Mountain Car Problem and Feature Engineering

In the git repository you can find a modification of the mountain car environment – a classical benchmark in RL that requires a non-myopic solution. In this game an agent controls a mountain car that should reach the flag at the top of the right mountain top. However, the car is not powerful enough to climb the hill to the top from a resting position. Instead, the car should first climb the left slope, build momentum and use that to reach the original mountain top (Figure 1). In our modification, the starting position and velocity of the car can be reset to a particular initial state manually (vs. the standard version where the car always starts at the bottom of the hill).

1. (5 points) By inspection of the provided code (`mountain_car_with_data_collection.py`), describe the state-space S, action-space A and the rewards $r(s, a)$ of the system.

In this exercise, we will solve mountain car using two different RL algorithms: LSPI and Q-learning. For both, we will use linear function approximation to handle the continuous state space. As features for this domain, we will use radial basis functions (RBFs, which were described in class. Also see `https://en.wikipedia.org/wiki/Radial_basis_function_network`).

1. (5 points) The function `encode_states_with_radial_basis_functions` takes as input a state $s \in \mathbb{R}^2$, and returns its d features $\phi(s) \in \mathbb{R}^d$. Plot the 1st and 2nd features for all states (discretize the state space and use meshgrid to plot `https://numpy.org/doc/stable/reference/generated/numpy.meshgrid.html`). Make sure to focus on the part of $\mathbb{R}^2$ that changes the most (contains the peak of the feature activation).
2. (5 points) In a linear approximation of the value function, what are the advantages of encoding the state space using RBFs features instead of using the states directly as features?



(a) (b)

Figure 1: Mountain car task: 1a (left) shows a possible starting position of the mountain car within the valley. 1b (right) shows the frame when the car crosses the finish line successfully.

3 LSPI

As you saw in class, LSPI is a batch RL method for learning policies from a data-set of examples based on approximate policy iteration. In this question, we will solve mountain-car using LSPI (please refer to the `lspi.py` script). The LSPI solution can be decomposed into the following steps: data collection, running approximate policy iteration, and evaluating the policy corresponding to the learned Q.

1. (10 points) **Episodic update rule:** How would you change the LSPI version shown in class to handle episodes? (hint: consider what happens in terminal states)

2. (5 points) **Data collection:** In the first step a data-set of N samples $D = \{(s_t, a_t, r_t, s_{t+1})\}$ should be generated. This data-set should explore the state-action space well in order for the policy to have good performance for arbitrary states. Use the `reset_specific` method to generate transitions from uniformly distributed states and random actions. We found that 100K samples were sufficient for good performance. What is the mean and standard deviation of the collected states?

3. (5 points) **Q-function linear approximation:** We estimate $\hat{Q}$ using a linear combination of features ($\hat{Q}(s,a) = w^\top \cdot \phi(s,a)$), and the features should be expressive enough to handle the complexity of the problem. Recall that our RBF features are defined for the states and not the actions. Since the action space is discrete, a common approaches is to represent $\phi(s,a)$ using state features for each action independently. That is, we encode the features of the state first $\phi(s)$, and for each action a we will have weights w_a such that

$$\hat{Q}(s,a) = w_a^\top \cdot \phi(s). \tag{1}$$

   To encode $\phi(s,a)$ we use a tiling strategy: First, set $\phi(s,a)$ as an all-zeros vector the size of $|A| \times |\phi(s)|$ (notice that the state features $\phi(s)$ could be tiled $|A|$ times in this vector). Define the elements in $\phi(s,a)$ that correspond to action a to the elements starting in index $a \cdot |\phi(s)|$ to index $(a+1) \cdot |\phi(s)| - 1$. Finally, set the elements of $\phi(s,a)$ corresponding to action a to be $\phi(s)$.

   For example: given $A = \{0,1,2,3\}$ action space of 4 actions, assume we want to find $\phi(s,2)$ and that $\phi(s) = (2,3,1)^\top$ then $\phi(s,2)$ becomes:

$$\phi(s,2) = (0,0,0,0,0,0,2,3,1,0,0,0)^\top$$

And if $a = 0$ and $\phi(s') = (1, 1, 1)^\top$ we get:

$$\phi(s', 0) = (1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0)^\top$$

How many weights are in your linear model?

4. **Evaluation criterion** We define the performance criterion of the model as follows:
   (a) Randomly select 50 starting states, uniformly distributed on the state space with velocity 0.
   (b) Evaluate the success rate of these starting states after each LSPI iteration (success counts when the mountain car reaches the top right peak i.e. the position component of the state > 0.5).
   (c) Use the `game_player.py` script with specific starting conditions.

5. (10 points) **Training the model:** Implement the LSPI algorithm, and run it for 20 iterations (or until the convergence criterion `norm_diff` < 0.00001). After each iteration, evaluate the performance of the greedy policy with respect to the current value function. Repeat 3 for 3 different random seeds. Plot a figure of the average performance vs. iteration.

6. (10 points) **Data dependence of LSPI:** Change the number of samples, and plot the success rate of a greedy policy following the final LSPI value function (after 20 iteration or convergence). Describe your findings.

   Notes:

   - We found the following parameter to work well: $\gamma = 0.999$.
   - When running any policy, make sure to limit the number of actions possible to avoid infinite loops.

4 Q-learning

In this question you are going to implement the online Q-learning algorithm [1]. We have provided you with `q_learn_mountain_car.py` script as a baseline. As opposed to LSPI where you start with a static data-set of transitions, in Q-learning the agent constantly collects data and optimizes its policy according to the newly discovered data. In this question you are going to solve the mountain car using Q-learning with the same feature representation you used in the LSPI question.

Q-learning is an iterative algorithm, where the agent updates it's parameters after each interaction with the environment. In our episodic setting, **the agent starts each episode at a random state, chosen uniformly over the possible positions and velocities in the task**. Each episode is limited to 200 steps.

In order to better track progress we modified the reward function from the previous question. While the goal was not reached the agent get a "reward" of -1, and once the goal is reached it gets a reward of 100. You are required to obtain a cumulative reward of -75.

1. (10 points) **Reward function** according to the reward specification in this question, in how many steps should the car reach the peak to consider the episode successful? Discuss how does setting the reward at the goal state as a relatively high positive value instead of a 0 affect the agent?

2. **Evaluation criterion** We define the performance criterion of the model as follows:
   (a) Randomly select 10 starting states, each starting in position -0.5 (bottom of valley), and with a small uniformly distributed velocity. Run the greedy policy with respect to the current Q function **without exploration**. Measure the average success rate of reaching the top. Use the `is_train=False` flag in the `run_episode` function of the `q_learn_mountain_car.py` script.

3. (20 points) Implement the Q-learning algorithm, and calculate the performance as defined above every 10 episodes. Q-Learning requires a lot of data, thus set the number of episodes to 10K. Repeat 3 for 3 different random seeds. Plot:

   - The total reward in the training episode vs. training episodes.
   - The performance (as defined above) vs. training episodes.
   - The approximate value of the state in the bottom of the hill vs. training episodes.
   - The total Bellman error of the episode, averaged over most recent 100 episodes.

   Describe the information provided by each plot.

4. (5 points) How many iterations were required for the greedy policy to solve the task? At that point, did the Q function converge to a reasonable value? Explain.

   Notes:

   - We found the following parameter to work well:
     - $\gamma = 0.999$
     - ϵ-greedy exploration with $\epsilon = 0.1$.
     - Learning rate $\alpha = 0.05$
   - When running any policy, make sure to limit the number of actions possible to avoid infinite loops. We found that a limit of 200 steps works well.

5. (10 points) **Exploration:** Provide plots of the total reward vs. episode for $\epsilon = 1.0, 0.75, 0.5, 0.3, 0.01$. What is the best value you have found?

5 Bonus

In the Q-learning part above, we let the agent start from a random initial position. Now, consider the case where at each episode, the agent can only start at the bottom of the hill. Does your Q-learning algorithm succeed in this case? Explain.

Modify your algorithm to succeed in this task. Hint: you can modify the exploration strategy. Provide a plot demonstrating your success, and explain your approach.

References

[1] C. J. Watkins and P. Dayan. Q-learning. *Machine learning*, 8(3-4):279–292, 1992.