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 on the course website.
3. You are required to submit both code and answers to the questions. The code should be submitted with a textual file describing the code (for instance, what each method does, what question each code piece solves, if you have testers what is being tested etc.) Submit a single ZIP file that includes the code, its description, and a PDF for the answers and figures.

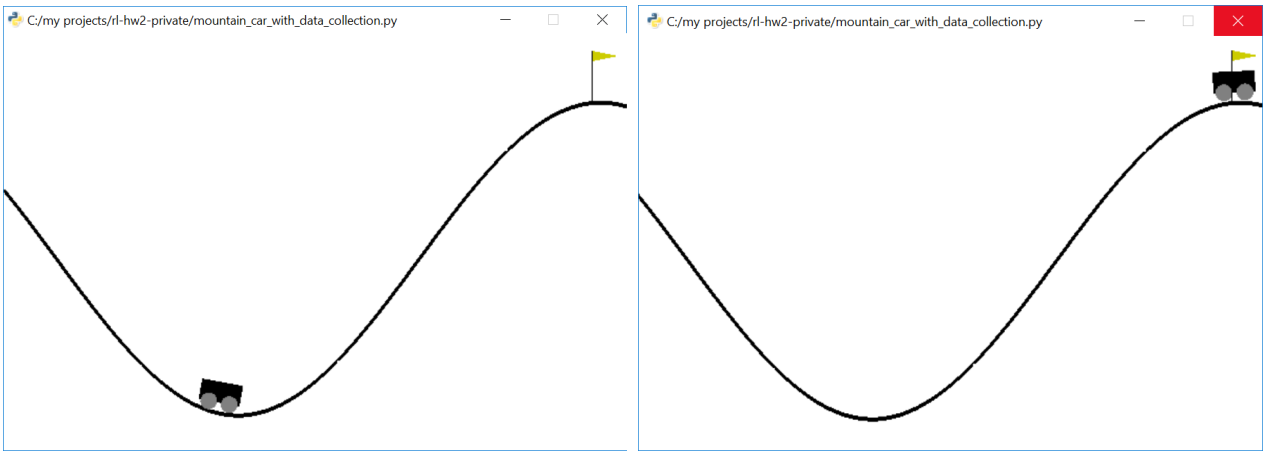
2 Preliminary: the Mountain Car Problem and Feature Engineering

In the course website 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 could be reset to a particular initial state manually which would be useful for data collection for the LSPI algorithm in question [1].

1. (5 points) By inspection of the provided code, describe the state-space S, action-space A and the rewards $r(s, a)$ of the system.
2. (5 points) **Value function approximation:** In this exercise we consider an approximation of the value function as a linear combination of features of the state. Useful features for this problem are, for instance, radial-basis functions (RBFs see `https://en.wikipedia.org/wiki/Radial_basis_function_network`). What are the advantages of encoding the state space using RBF features instead of using the states directly as features?
3. (5 points) **Data standardization:** next, read the Wikipedia section discussing data standardization `https://en.wikipedia.org/wiki/Feature_scaling#Standardization`. What are the benefits to standardizing the data? Specifically relate your answer to RBF kernels (hint, how do you select μ and σ of these kernels?)

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. The LSPI solution



(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.

can be decomposed into the following steps: data collection, representation (feature) selection, learning the Q function, 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 at specific states. Describe your data collection process. How many samples were used? What portion of the data provides positive rewards? We found that 100K samples were sufficient for convergence.

3. (5 points) **Deciding on Q-function features:** The next step is to decide on a feature representation $\phi(s, a)$ for your Q-values in order to estimate $\hat{Q}$. Since the model is linear ($\hat{Q}(s, a) = w^\top \cdot \phi(s, a)$) the features should be expressive enough to handle the complexity of the problem. Since the action-space is small and discrete, and only the state space in continuous, one common approaches is to represent the state using features for each action independently. That is, we encode the features of the state first $e(s)$, and for each action a we will have weights w_a such that

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

   (a) **Define $e(s)$:** Select a method of encoding s to $e(s)$. We recommend you follow:
      i. Standardize the data.
      ii. Encode using kernel function (we recommend RBF kernels but other method could also work).
      iii. Add a bias term to $e(s)$ (a bias term is a feature fixed to 1). For example: let $e(s) = (2, 3, 4)^\top$ then the features with bias are $(2, 3, 4, 1)^\top$. If you choose to use a bias term, explain its role.

   Describe the state encoding process you selected. If you selected to use data standardization explain why (and why does it help to design kernel functions?). Explain your choice of kernel function and provide numeric values when needed. For instance, if RBF kernels were selected specify the μ and σ of each kernel.

   (b) **Define $\phi(s, a)$:** So far you have an encoding of the state $e(s)$. We now show that the state-action representation in (1) can be written as $\hat{Q}(s, a) = w^\top \cdot \phi(s, a)$, with an appropriately chosen $\phi(s, a)$. This technical trick greatly simplifies the code for training the weights.

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

For example: given $A = \{0,1,2,3\}$ action space of 4 actions, assume we want to find $\phi(s,2)$ and that $e(s) = (2,3,1)^\top$ 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 $e(s') = (1,1,1)^\top$ 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. **Training the model:** So far we defined the model we are going to train, it is time to define how to train it. To fill in the missing pieces define $\gamma = 0.999$, and describe your stopping criterion.

5. (10 points) **Testing the model:** We define the performance criterion of the model as follows:

   (a) Randomly select 10 starting states (from the same distribution used by the `reset` method.

   (b) Evaluate the success rate of these starting states after each LSPI iteration (success counts when the mountain car reaches the top right peak).

   (c) The game is considered solved if 9 out of 10 starting positions result in a successful episode.

   (d) Repeat this evaluation 5 times (5 random seeds used to sample states), but keep the initial states used for evaluation fixed. Report the **average** of these 5 runs after every iteration.

   Plot a graph of the success rate after every LSPI iteration. How many LSPI iterations are needed (on average) for your model to converge?

   Notes:

   - The process of making LSPI work should be taken as an iterative process rather than the linear flow presented here. For instance, you might start with too few data and don't converge, or too much data and your model takes to long to train, so you may need to return to the data sampling portion after defining the subsequent parts. Alternatively, you might decide that some aspect of your features could be improved etc.
   - When running any policy, make sure to limit the number of actions possible to avoid infinite loops. Moreover, create an upper bound on the number of LSPI iterations to avoid infinite loops caused by bugs in the stopping criterion.

6. (10 points) **Sample size of LSPI:** Let N denote the number of samples required for model convergence that you found in the previous question. Plot the **average** success rates after every LSPI iteration for the following sample sizes (on the same plot): N, $N/2$, $N/10$, $2N$, and $10N$, for 5 repetitions of the experiment (if one model takes less LSPI iterations to converge, simply repeat a value of 100%). Describe the results.

4 Q-learning

In this question you are going to implement the online Q-learning algorithm [2]. 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.

Note: if you decide to change the feature for any reason, please explain why and explain the reasoning for the new feature representation as you did in the LSPI question.

1. (5 points) **Online data collection with ϵ-greedy policies:** Q-learning is an iterative algorithm that includes two steps in each iteration: data-collection and model training, we denote such iteration as a *training cycle*. We would now focus on the data-collection aspect. The goal of the data collection is to gather transitions that are more similar to the current (implicit) policy.

   In order to discover new information, we use an ϵ-greedy policy to collect samples. In ϵ-greedy the agent takes a random action with probability $\epsilon \in [0, 1]$, and otherwise takes the action with highest estimated Q-value. In more advanced versions, the exploration parameter ϵ has a schedule that reduces the value of ϵ over time. Implement an ϵ-greedy mountain-car agent that takes as input a policy, and a value for ϵ, and returns all the collected transitions. Describe the value of ϵ you use, and schedule if applicable. Also specify the number of games the agent plays in every training cycle.

2. (5 points) **Training:** As stated earlier, the training process in Q-learning interleaves data collection with training the policy, and each training step is comprised of one (or more) batched-update steps. For each batched-update we sample a batch of transitions from the collected experience, and update w according to the Bellman error on the sampled batch with a gradient step.

   - Implement the update rule of Q-learning.
   - State your batch size, and how many update steps your model takes per one training cycle.
   - How many samples (with repetitions) does your model use per one training cycle? What is the ratio of that number to the size of the collected experience?
   - Why do we need more than one training cycles and not just use a single iteration with many update steps? (a training cycle is data collection followed by several model updates)
   - Specify your learning rate, or learning rate schedule if applicable.

3. (10 points) **Testing:** Repeat the testing instructions from the previous question (the performance of the model is measured by the success rates of 5 experiments). Plot the success rate over training cycles. You don't need to plot the success rate after every training cycle, you can evaluate the success rate once every several cycles (but state so in the report).

4. (10 points) **Different ϵ values:** Let ϵ_t denote your selected schedule for the values of ϵ, in this question we are going to compare the performance of different values of ϵ. Fix all other design choices other than ϵ, repeat the experiments from the **testing** question for $\epsilon_t^1 = 0.5 \cdot \epsilon_t$ and $\epsilon_t^2 = 2 \cdot \epsilon_t$. Compare and analyze the results.

5. (15 points) **Sample efficiency:** Count and compare the average number of samples for both LSPI and Q-learning. Explain the results. What does LSPI assume about the environment that Q-learning does not?

5 Policy Gradients - Bonus

In this question you are going to solve mountain car with policy gradient methods. Unlike the methods from previous questions that learned a representation of the Q-values using the Bellman Equation and had an implicit policy, in policy gradient methods, we explicitly have a representation of the policy $\pi(a|s)$, and we search for optimal policy in the space of policy weights directly.

1. (2 points) As a warm-up, suggest a modification to the linear approximations $\phi(s, a)^\top \cdot w$ from previous questions in order to obtain a stochastic policy $\pi(a|s)$ (hint: the answer is simple and includes a *softmax* activation).

2. (5 points) What is the REINFORCE update rule of your policy weights (the linear vector w)? Explain how you derived it. Are you considering an episodic or discounted future? Which variance reduction method did you use?

3. (5 points) Similarly to Q-learning, policy gradient methods also work in update cycles that include data-collection and model updates, but since we are learning policies instead of Q-values we use a different model update. Since we don't have a representation to the Q-values we don't update according to the Bellman error, instead, each policy gradient defines an advantage estimate and try to increase the likelihood of actions associated with high advantage.

   Implement the REINFORCE algorithm. Specify your choice of γ, learning rate, and number of episodes per data-collection. Plot the success rate of 5 experiments over training cycles (as defined in previous questions).

4. (3 points) **Comparison to Q-learning:** It is interesting to compare RL algorithms according the how many samples each algorithm requires in order to achieve a good performance measure, here we are going to compare Q-learning to REINFORCE.

   Plot the success rate of each algorithm as follows: the X axis should indicate the number of samples observed by the algorithm, the Y axis should correspond to the success rate, and finally you should draw two curves (one for Q-learning and one for REINFORCE). The curves should track the average success rate of 5 runs of each algorithm, see the previous questions on how to plot these exactly. Explain the results, are these the results you expected?

5. (5 points) **Natural Policy Gradient:** The Natural policy gradient is an extension of the vanilla policy gradient that changes each element in the weight vector (in our case w) according to the Fisher information matrix. Follow the previous instructions for REINFORCE for this algorithm as well; Define the new update rule now, specify the model's hyper-parameters, and provide a graph comparing the sample efficiency of Q-learning, REINFORCE and Natural policy gradients.

References

[1] M. G. Lagoudakis and R. Parr. Least-squares policy iteration. *Journal of machine learning research*, 4(Dec):1107–1149, 2003.

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