Reinforcement Learning Lab Session 6

Lars Van Roy

April 2, 2020

General Information

This lab session will cover all the Reinforcement Learning material from courses 1 to 4. It will be graded lab sessions. You are free to use any content you want regarding the questions, but please refrain from using outside code beyond the previous lab sessions.

It should be doable in 4 hours, but the deadline will be set to Thursday 2nd of April, midnight. As usual, for there on, each day of delay will remove 2.5/10 points from your grade.

Submission – You will have to submit both a report and code through Blackboard. Use the code available on the git at `http://github.com/Louis-Bagot/RL_Lab/lab_session6` or `https://github.com/Rajawat23/RL_Lab/lab_session6`.

Make a copy of this LaTeX document from folder **report**, and fill it according to the instructions below. Please to not alter this base document, only add your content.

Question – *Questions will look like this. For questions, write the answer just below where the (your answer here) appears.*

Programming – *Programming exercises will look like this. For programming exercises, read the* `instructions.md` *of the corresponding section, and then fill in the corresponding TODO flags in the code. If this text asks for a plot, copy the plot output of your code in the document, in the figure space provided (in addition to leaving it in the plots folder in the code). If the text asks for an explanation of the results, write it as you would answer a question.*

Contents

1 Bandits

Question 1 – *Explain, in a few lines, the k-armed Bandit problem. Give an example of a real-world application, detailing actions and rewards.*

The k-armed bandits can be used for many real-world applications where there are different options that should be compared with each other. For example different kinds of medications used for a common cause. Each bandit could represent a kind of medicine, each action could be equal to handing a person the medicine, and the reward would be based on the effect of the medicine. For example, the reward could be small/negative if the person was still ill, the reward could be big/positive if the person was cured and the reward could be big and negative in case the person has died.

Question 2 – *Derive the incremental updates of the Sample Average method.*

In order to optimize the computation of these averages, we do not want to store every obtained result continuously, and therefore we want to optimize the formula $Q_{n+1} = \frac{R_1 + R_2 + ... + R_n}{n}$ to the formula $Q_{n+1} = R_n + \frac{1}{n}(R_n - Q_n)$ as can be seen in Formula 1.

$$\begin{aligned} Q_{n+1} &= \frac{R_1 + R_2 + ... + R_n}{n} \\ &= \frac{1}{n}(\sum_{i=1}^{n} R_i) \\ &= \frac{1}{n}(R_n + \sum_{i=1}^{n-1} R_i) \\ &= \frac{1}{n}(R_n + (n-1)\frac{1}{n-1}\sum_{i=1}^{n-1} R_i) \\ &= \frac{1}{n}(R_n + (n-1)Q_n) \\ &= \frac{1}{n}(R_n + nQ_n - Q_n) \\ &= R_n + \frac{1}{n}(R_n - Q_n) \end{aligned} \tag{1}$$

Question 3 – *Explain, with your own words, the difference between Sample Average and Weighted Average methods*

While using Sample Average methods, we calculate average rewards without any difference between newer observations and older observations. This will always converge and is only suitable for constant environments. Weighted Average methods give the option to make the newer observations more valuable than the older observations, via a weight. This weight must be between 0 and 1, and represent the value of the new result in an update of the average. The closer the weight gets to 1, the closer the updated average will be to the new result, when it gets closer to 0, the updated average will get closer to the old average, and only change slowly. Weighted averages are more suitable for constantly changing environments, and will never converge.

Question 4 – *Explain the impact of the hyper-parameters of the following algorithms: ϵ in ϵ-greedy, c in UCB, and α in Gradient Bandit. Use extreme values as examples.*

In *epsilon-greedy* we will try to avoid getting stuck at local maxima that come with exploitation. When purely basing oneself on the greedy algorithm, we will no longer explore as soon as a local maximum is found. To fix this, we will add a *hyper-parameter* ϵ in *ϵ-greedy*, which will represent the chance to continue exploring, even though we have found a local maximum. When ϵ is equal or close to 1, we will continue to explore randomly without any regard for previously found data, when ϵ is very small or equal to zero, it will be close to or equal to the greedy method.

In *UCB* we try to improve the way we randomly select actions in the greedy method. If it is purely random, we do not consider the potential of an action, if you have tried an action many times, the chances are small that the corresponding average reward will change significantly, where if you have only tried an action a couple of times, the average has a very high probability of changing significantly. We will

add a factor the average reward which will allow us to be biased towards actions that have not been chosen as often with that factor being $c\sqrt{\frac{\ln t}{N_t(a)}}$. It is clear to see that whenever we have tried an action many times, N_t will get very big, and the factor will become very small, we can also see that whenever we have only tried an action a few times, the factor will get very big due to the same reasoning. The *hyper-parameter* c decides how much this factor is taken into account. When c is small or close to zero, the *UCB* algorithm will approach or be equal to the greedy method. When c becomes large we will purely base ourselves on the actions that we have chosen the least number of times, and no longer have any (or much) regard for the actual average obtained reward.

In *Gradient Bandit* we want to learn a numerical preference (called $H_t(a)$ for an action a) for each action. In this method we will attempt to increase the probability of choosing an action, based on the current reward compared to the average reward. If the reward is higher than the average reward, we increase the probability of choosing said action, and decrease it otherwise. The *hyper-parameter* α will decide the rate at which these probabilities change. A small alpha will cause the probability to change slowly, to not at all, in case α is equal to zero. In case α gets very big, the rate at which the probabilities change will be massive, one bad reward, and the probability of choosing that action will plummet.

Question 5 – *Show that the Sample Average method erases the bias introduced by the initialization of the Q estimates. What does this mean for Optimistic Greedy? Show that, despite this, Optimistic Greedy does not work on non-stationary problems.*

The Sample average method is a method that will always converge eventually, and the change in initial value will not change this. It might take more or less steps to converge (depending on the actual value it converges towards) as it converges very slowly, but it will converge nevertheless.

Since the added functionality of Optimistic Greedy, being the initialization of Q estimates, did not have any effect on the Sample Average method, it means that we need to use a different method for Optimistic Greedy to compute averages. The one method that would be suitable for this would be Weighted Average, as this will never completely converge.

Despite the fact that Weighted Average is suitable for non-stationary problems, Optimistic Greedy is not. Optimistic Greedy is a method that uses an epsilon value equal to 0, which means that we will always exploit and never explore. The reason that Optimistic Greedy works in stationary situations is that it is forced to explore a little bit due to the initial values, as the optimal action might vary at that point. Once the other ones drop below the currently best action, no other actions will be explored. Over time our current maximum might become a local maximum, but this will never be discovered.

Programming – *Implement a Sample Average and Weighted Average version of ϵ greedy on a Non-Stationary k-armed Bandit problem. In order to see results, run the experiment for $10k$ steps, and paste here the resulting performance plot in the Figure 1 below. Explain your results below.*

(your explanation here)

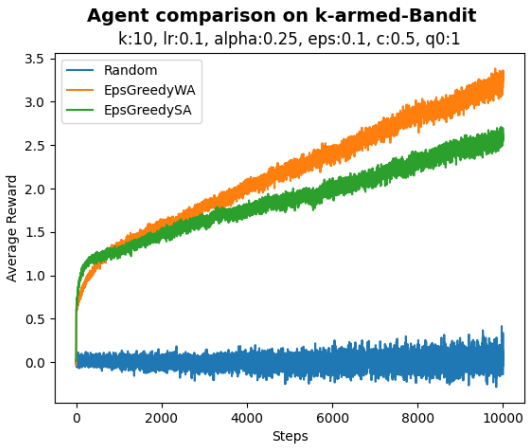

Figure 1: Comparison: ϵ-greedy algorithm with Sample Average versus Weighted Average updates.

As we can clearly see, the *Weighted Average* method performs better over time than the *Sample Average* method does.

This is explainable with the fact that *Sample Average* converges over time, where *Weighted Average* does not. *Sample Average* will converge, and it will get stuck on a maximum, where it will stay indefinitely as the n value will get so big to the point that the average never changes. *Weighted Average* however will never converge, and will always continue changing. The environment will change, but the averages will adapt to this and *Weighted Average* has a far higher chance to continue going for the action with the best reward distribution.

2 Markov Decision Processes

For questions where a drawing (MDP or diagram) is required, you can use whichever of the following methods:

- a (properly cropped and clear) photo of a drawing on paper. Make sure everything is readable.
- a tikz diagram, i.e. the plotting tool for LaTeX (if you know how it works. Don't learn for this report otherwise, tikz takes an eternity)
- Mathcha, which can generate tikz or pngs. (recommended)

Question 1 – *Define a Markov Decision Process, and the goal of the Reinforcement Learning problem.*
A Markov Decision Process is a means to represent environments in a graph format. A process is defined as a 4-tuple (S, A, P_α, R_α) where

- S is a finite set of states
- A is a finte set of actions
- $P_\alpha(s, s')$ is a probability function that gives the probability that action a leads to state s' starting from state s
- $R_\alpha(s, s')$ is a reward function that gives the reward obtained when going from state s' to state s

There could be multiple ways to traverse the graph, as there could be many possible actions that can be performed in each state, we will therefore define a policy function that will make the decisions for us. A policy is a function π that specifies the desired action for each state. This can be updated/modified while traversing the different states and their corresponding actions, until it converges, at which point we can say that the process of choosing an action becomes deterministic. The ultimate goal is to define your policy in such a way that the average reward gained while following the policy is maximal.

Question 2 – *Show that the MDP framework generalizes over Bandits by drawing the Bandits problem as a MDP with reward distributions r_a for each action a. Paste your drawing on Figure 2. Shortly explain your submission.*

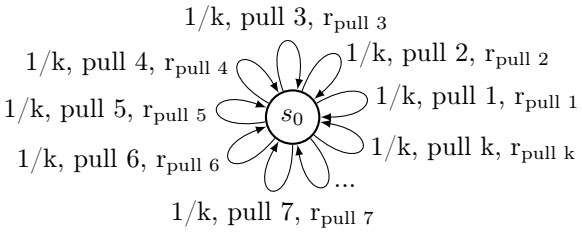

Figure 2: The MDP corresponding to the Bandit problem with reward distributions r_a, $\forall$ actions a

.

The MDP displayed in Figure 2 describes a means in which the bandit problem can be described. Each transition symbolizes the pulling of a single arm, which comes with the reward of that arm, given by the reward distributions r_a, $\forall$ actions a. The chances of each action occurring are uniformly distributed.

Question 3 – *Turn the following statement into a MDP, with states and transitions with actions (named), probabilities and rewards. Paste the graph on Figure 3; pick real values for both rewards and probabilities (no unknowns). Shortly explain your submission after the statement and plot.*

Statement:

You go to the university using Velo - Antwerp's shared bikes. There are three stations where you can drop off your bike on the way: the park, furthest from the university; the cemetery, second furthest; and the university station, right in front of your destination. You want to take the least time possible to go to the university, and you take much longer walking than biking.

At any station, you can either decide to drop off your bike and walk to the university, or continue to the next station. However, it sometimes happens that the stations are full - you cannot drop off your bike there. You can either go back, or, if possible, continue. You notice that the amount of free spots in the first stations often aligns with the amount of free spots in the following stations - or is less. In order to decide whether you should drop off your bike or not, you take note of the last station's number of free spots - it can either be a lot, or a few, or none.

When you have to go back, we assume that people could've come to pick or drop bikes, so the transition doesn't depend on the previous state of the station.

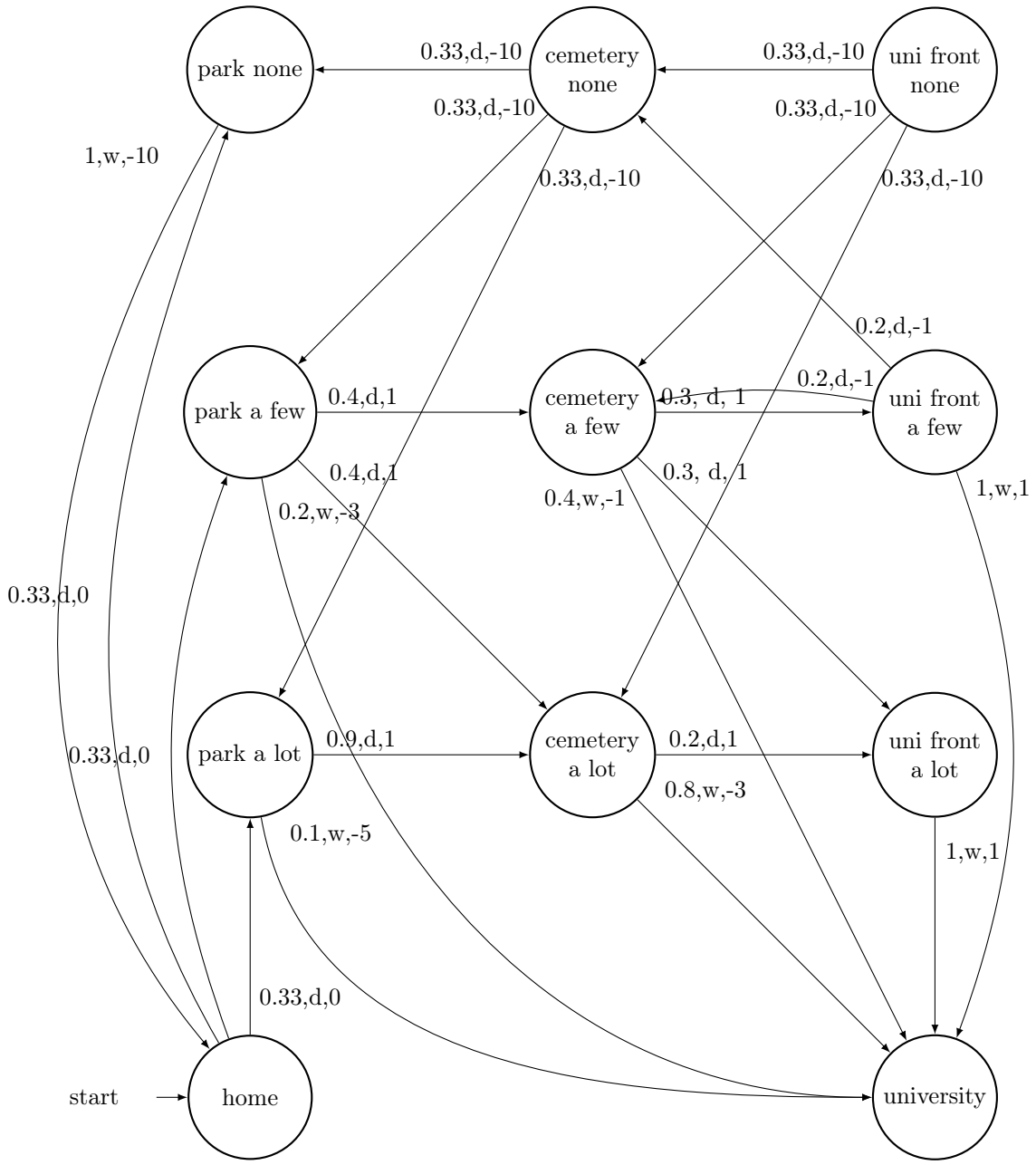

Figure 3: The MDP corresponding to the statement of Question 3

.

There are 5 base states, of which 3 have 3 additional substates. There is home, which represents the start of the path, there is park, which is the first station, cemetery, which is the second station and university front, which is the final station. There is one more state which is university, which is the final destination of the biker. The three stations consist out of three substates, where they can either be in completely filled state, defined by none, a partially filled state, defined by a few and mostly empty, defined by a lot.

The transitions from a degree of filled, to the next station, is considered to be random, that is, the filled degree must at least be as high as the previous degree, and the distribution between the leftover options is assumed to be uniform.

The biker favors moving forward, as long as there are spaces left in the current station, as soon as there are none, the biker will always return, and in case the front of the university is reached, the biker will always go for the university (given that there is any space at all). If there is no space in any of the stations, the biker will go back to the previous station, or in case there is no space at the park, the biker will go back home. There is no specification for what happens if none of the stations have spaces, so the

decision was made to let the biker return home in this case.

In case a station was filled, the degree of filled from the previous station is considered to be uniformly distributed, and therefore, the chances will be 33% for each of the previous station's substations.

The rewards are based on the distance traveled by bike. moving forward from station to station will always result in a reward of 1, moving from the front of the university to the university will also result in a reward of 1. Moving back to a previous station will result in a reward of -10, as this should be avoided at any cost, as this is very counter productive. Early transitions from a station to the university will yield negative rewards, based on the closeness to the university, and the number of free spaces. These are less negative than backtracking (as they are less bad) but are still negative, as continuing forward must be better (in case those stations are not packed).

Question 4 – *RL has been widely popular lately because of its combination with Deep Learning (using Neural Nets as policy or value function approximators), leading to incredible performances on difficult environments like video games. One such game is the first Mario World. Show how the MDP can look like for the frames of the game. Where can stochasticity be involved?*

Based on a frame, a MDP could be created, to determine the next move of Mario. For example, depending on the closeness of the nearest enemy, a very bad score could be given for walking towards said enemy. Depending on the closeness of the nearest coin, a good score could be given for walking towards said coin. Powerups could be given higher scores than coins and good scores could be decreased in case there are enemies nearby.

In case one wishes to combine this with Neural Nets, stochasticity is necessary as this allows us to explore different possibilities, see how each possibility fares, and learn things from those results.

3 Control

In lab session 2 and 3, the Value Iteration algorithm was stopped after a maximum number of iterations. However, this is not the natural stopping condition of the algorithm: it should stop when the value estimates have converged: $V_k = v*$. When implementing this, we define convergence of the V_{k-1}, V_k, V_{k+1}.. stream of vector values as

$$||V_{k+1} - V_k||_2 < \delta$$

Where δ is an arbitrary small constant ($\delta = 0.01$ in the code). The number of iterations of Value Iteration to convergence is a measure of the algorithm's performance.

Policy Iteration alternates evaluating a policy π until convergence to V_π, and updating the policy to be greedy over the new values, $\pi' = greedy(v_\pi)$. We define *convergence in policy* as $\pi' = \pi$ (same action in all states), and *convergence in value* as

$$||V_{\pi'} - V_\pi||_2 < \delta$$

Value Iteration only converges in value, as there is no explicit policy. When comparing convergence speed in value of Value Iteration vs Policy Iteration, make sure to compare the number of single sweeps over the state space! (iterations)

Programming – *Implement Value Iteration on the course's diamond/pit Gridworld environment (course and Lab session 2). You can reuse Environment code from before.*

Programming – *Implement Policy Iteration on the course's diamond/pit Gridworld environment.*

Question 1 – *Discuss and compare the performances of both algorithms. Under what circumstances can one come on top of the other?*

When looking at performance metrics as mentioned in the code, which is based on the number of loops, the Policy Iteration performs worse than the Value Iteration algorithm did, due to Policy Iteration needing additional loops to evaluate the policies. However, these loops are not exactly equal. In the value evaluation loop, Policy Iteration will only need to evaluate one action (being the current policy) per state, where Value Iteration will need to evaluate every action.

In general for complex environments, Policy Iteration will perform better than Value Iteration, Policy Iteration will require us to only compute one action for each states. There might be more loops, since

we only consider one of the actions for each state per iteration, but on the other hand the policy will converge faster than the values do. So as soon as the policy stops changing the Policy Iteration will end while the Value Iteration might still carry on. Whenever we consider more simple environments, Value Iteration will be better, since there will only be a small amount of iterations and in those situations, the additional loops required to updates policies etc, could become significant for the total execution time.

Question 3 – *Explain the fundamental differences between QLearning and the Value Iteration / Policy Iteration algorithms. Can you see why QLearning is more interesting in the general case?*

The fundamental difference between QLearning and Value Iteration / Policy Iteration, is that in QLearning it is assumed that we have zero knowledge regarding state transition probabilities or rewards. In Value Iteration / Policy Iteration we assume that these things are known. To be able to let the agent work on general cases, it is therefore prudent to use QLearning over Policy Iteration / Value Iteration as we will not have the needed information to perform these.

4 Bonus

Programming – ***BONUS, 1.5pts*** *Implement the Gridworld drawn on Figure 4: a river crossing. The actions are up, down, left, right and "do nothing". The agent needs to cross a river from the lower left corner (state S) to the upper right corner (state G). This* 3×5 *Gridworld is divided on the 2nd row by a river that might push the agent right upon entering by one, two or three squares, with corresponding probabilities* $0.2, 0.5$ *and* 0.3*. If the agent is pushed off to the far right in the river, the episode ends with reward* -1*. If the agent reaches the goal, the reward is* $+1$*. Note that it is the transition to the state (i.e. "right" from (0,3) to G=(0,4)) that yields the reward, and states G and red (1,4) are terminal.*

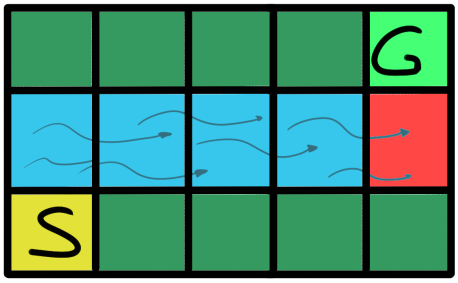

Figure 4: The River Crossing MDP

.

Note: By mistake I originally added the specified environment to the code of the third assignment, this code does work, but in order to be fully correct I also implemented this in the requested bonus directory. At this point I am uncertain as to what I changed in order to implement the grid in the third assignment, so I decided to leave this in. Nothing was changed to the rest of the code, but I want to make the remark, in case you check for differences with the original files.