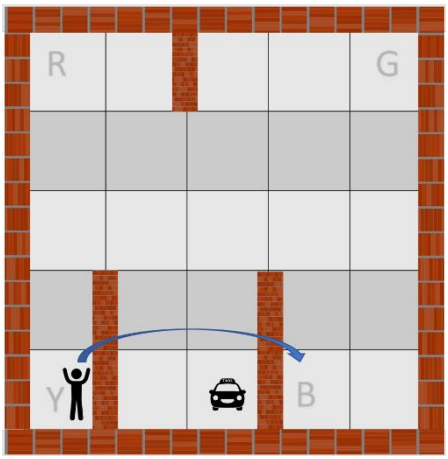
Taxi Driver Agent with Q-Learning and LSPI

CS402 Autonomous Agents

Georgios Nektarios Nikou 2016030125

Problem Description

The taxi problem was first introduced in [Dietterich2000] paper to expose some issues in hierarchical reinforcement learning. The problem is conducted on a 5x5 grid world and the goal is for the taxi to successfully pick up and drop the passenger to the designated location. There are four designated locations in the grid world indicated by R(ed), G(reen), Y(ellow), and B(lue). When the episode starts, the taxi starts off at a random square and the passenger is at a random location. Once the passenger is dropped off, the episode ends. Also as it is shown on the map below there are walls that the taxi cannot pass by.



Action space:

There are 6 discrete deterministic actions

- 0: move south
- 1: move north
- 2: move east
- 3: move west
- 4: pickup passenger
- 5: drop off passenger

Rewards:

- -1 per step unless other reward is triggered.
- +20 delivering passenger.
- -10 executing "pickup" and "drop-off" actions illegally.

State space:

There are 500 discrete states since there are 25 taxi positions, 5 possible locations of the passenger (including the case when the passenger is in the taxi), and 4 destination locations. However only 404 are the reachable discrete states.

Taxi-v3 Implementation by OpenAI Gym

The simulation of the given problem is made with the use of Open AI gym Taxi v3. The

core gym interface is the "env" object, which is the unified environment interface. State space is represented by: (taxi_row, taxi_col, passenger_location, destination).

These are the following methods that can be used:

- observation_space.n: Returns the number of all possible states.
- action_space.n: Returns the number of all possible actions.
- reset(): Resets the environment and returns a random initial state.
- step(action): Step the environment by one timestep. Returns:
  - observation: Observations of the environment
  - reward: If your action was beneficial or not
  - done: Indicates if we have successfully picked up and dropped off a passenger, also called one episode
- render(): Renders one frame of the environment

As for rendering the game is represented as it follows:

- blue: passenger
- magenta: destination
- yellow: empty taxi
- green + underline: full taxi
- other letters (R, G, Y and B): locations for passengers and destinations

Q-Learning

The first reinforcement learning algorithm used for solving the taxi problem is Q-Learning. It is based on the state-action value function Q(s,a). This function denotes the value of taking an action in a state following a policy p. Q-learning is a model-free, off-policy reinforcement learning algorithm that will find the best action to execute, given the current state of the agent. Using the Bellman Equation the Q-Learning equation updating the Q-table for every sample (s,a,r,s') is:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\big(r + \gamma \max_{a' \in \mathcal{A}} Q(s',a') - Q(s,a)\big)$$

If the current state is a terminal one then the Q-table is updated like this:

$$Q(s,a) \leftarrow Q(s,a) + \alpha\big(r - Q(s,a)\big)$$

LSPI Algorithm

This algorithm is the extension of the least-squares temporal-difference (LSTD) algorithm as a control algorithm. The Least Squares Policy Iteration algorithm is implemented with two parts:

- LSTDQ algorithm
- an approximate policy iteration algorithm

The first one is similar to LSTD with difference being that it learns the approximate state-action value function of a fixed policy, thus permitting action selection and policy improvement without a model. Then using the results produced by LSTDQ algorithm an approximate policy-iteration algorithm is formed. These two elements construct the LSPI algorithm. The LSTDQ algorithm is called iteratively until we assume that the policy converges. This is decided by a threshold from which the difference between new and old weights (produced by LSTDQ) is less than it.

The LSTDQ algorithm is described as:

LSTDQ $(D,\ k,\ \phi,\ \gamma,\ \pi)$ // Learns $\widehat{Q}^{\pi}$ from samples

// D : Source of samples (s,a,r,s')
// k : Number of basis functions
// ϕ : Basis functions
// γ : Discount factor
// π : Policy whose value function is sought

$\widetilde{\mathbf{A}} \leftarrow \mathbf{0}$ // $(k \times k)$ matrix
$\widetilde{b} \leftarrow \mathbf{0}$ // $(k \times 1)$ vector

for each $(s,a,r,s') \in D$
$\widetilde{\mathbf{A}} \leftarrow \widetilde{\mathbf{A}} + \phi(s,a)\Big(\phi(s,a) - \gamma\phi\big(s',\pi(s')\big)\Big)^{\mathsf{T}}$
$\widetilde{b} \leftarrow \widetilde{b} + \phi(s,a)r$

$\widetilde{w}^{\pi} \leftarrow \widetilde{\mathbf{A}}^{-1}\widetilde{b}$

return $\widetilde{w}^{\pi}$

The LSPI algorithm is described as:

LSPI $(D,\ k,\ \phi,\ \gamma,\ \epsilon,\ \pi_0)$ // Learns a policy from samples

// D : Source of samples (s, a, r, s')
// k : Number of basis functions
// ϕ : Basis functions
// γ : Discount factor
// ϵ : Stopping criterion
// π_0 : Initial policy, given as w_0 (default: $w_0 = 0$)

$\pi' \leftarrow \pi_0$ // $w' \leftarrow w_0$

repeat
  $\pi \leftarrow \pi'$ // $w \leftarrow w'$
  $\pi' \leftarrow$ **LSTD**$Q\ (D,\ k,\ \phi,\ \gamma,\ \pi)$ // $w' \leftarrow$ **LSTD**$Q\ (D,\ k,\ \phi,\ \gamma,\ w)$
until $(\pi \approx \pi')$ // **until** $(\|w - w'\| < \epsilon)$

return π // **return** w

Implementation

For the implementation of the agent, I used python 3.8 and the gym library in order to access the OpenAI Gym functions. Other libraries that I used include numpy for arrays, matplotlib.pyplot for depicting plots, scipy.linalg for matrix calculation.

About the Q-Learning algorithm I created three functions. First one trains the agent over an iteration of 100000 episodes of the game and returns the Q-table. Next one is the evaluation function that takes as an argument a Q-table and rates through 20000 episodes how well the agent performs with it. Last one is a function about using Q-Learning for an iteration over 20000 episodes with an optional argument for Q-table. About the parameters, after some experimenting and looking for the most optimal combinations, I settle down to this:

- alpha = 0.9
- gamma = 0.6
- epsilon=0.1
- alpha decay = 0.0001

About the LSPI algorithm at first, I created the method for collecting samples of Taxi-v3 episodes that will later be used as input at LSTDQ. There is an option for user to select whether the samples will be collected through a random policy or through Q-Learning agent. I have used 42 gaussian RBFs to calculate the approximate Q^{π} values. Although a certain policy is formed with weights, it doesn't result to a proper policy for the agent. When the evaluation phase starts it yields only massive negative rewards at each episode and it is obvious that it is no different from a random policy. Something I noticed from rendering is that it visits some states very often so I suspect that it is a problem in creating the linear approximation value-function.

Here is the functionality of each one of the modules:

- main.py: It includes the main function where the program is running. Mainly its use is for terminal prints and asking for user input as well as calling the appropriate

functions. Also it includes two functions, one for calculating the rolling average of a matrix given a window size and the other one for plotting the reward results.

- q_learning.py: Contains the Q-Learning methods including the train, simulation and learning functions of the agent. Train method trains the agent over an iteration of 100000 episodes of the game and the Q-table that is formed after is returned.
- lspi.py: Contains the LSTDQ function as well as the LSPI function. More functions are initialize_policy() that sets the starting policy of the agent, act() that picks the action with the highest reward (argmax_a Q(s, a)), collect_samples and read_file are helper functions that write and read each sample respectively from the sample file.
- policy.py: it stores the current policy and has functions that return the best action given the current state.
- rbf.py: it has the class of the gaussian radial basis functions with methods that calculate the weights and matrix calculations of basis functions.

Results

Here are the results for the Q-Learning agent, trained at first and not trained at second.

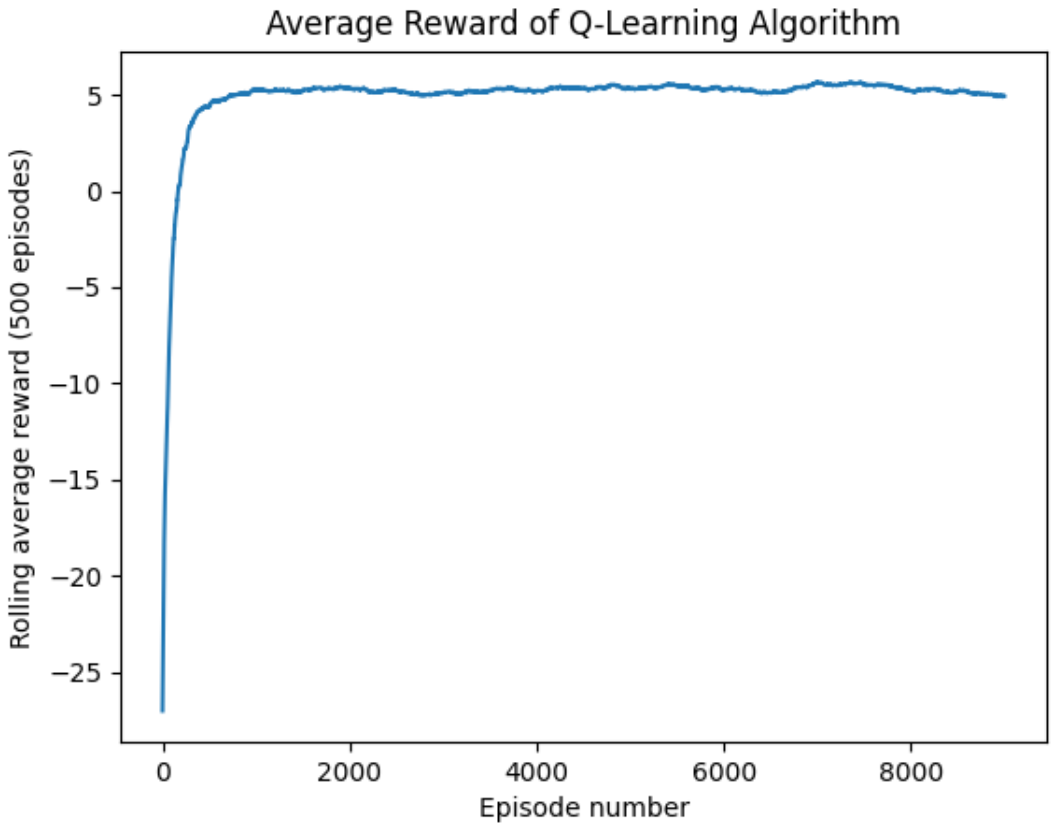


Figure 1 Q-Learning agent with no Q-table at the beginning

At the first graph it is depicted the results of an agent using Q-Learning with no initial Q-table given. It is the rolling average rewards of 10000 episodes with a 500-episode window. As we see at first episodes the random steps, that the agent takes, yield terrible results. But as it reaches 500 episodes the rewards are getting better and finally they are steadying around 5 points.

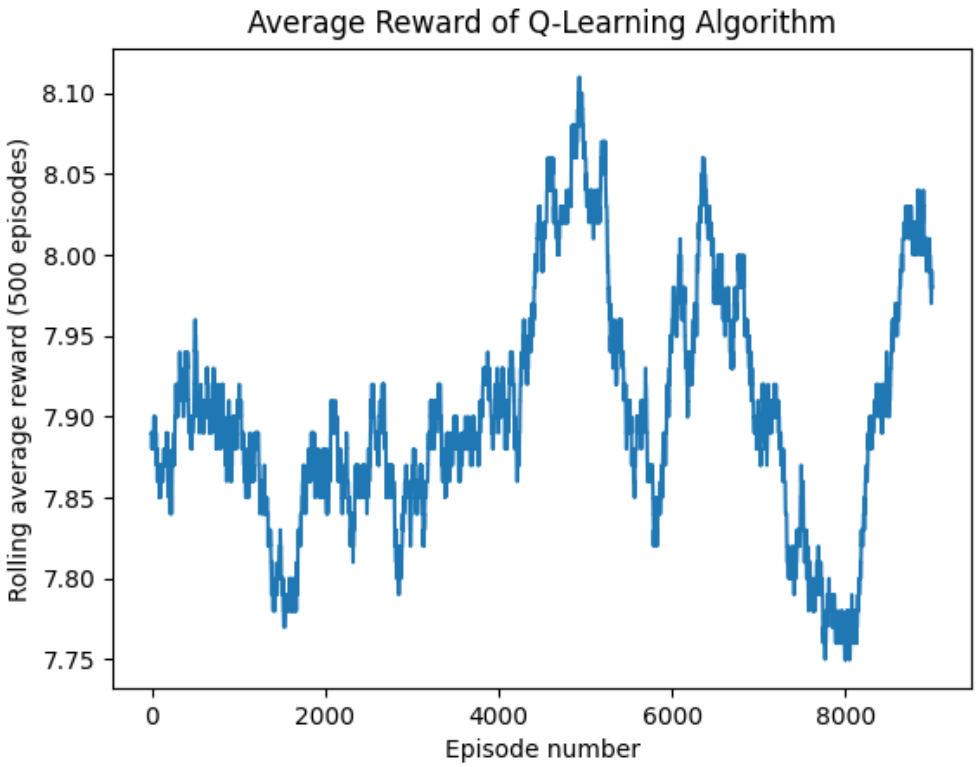


Figure 2 Q-Learning agent trained with 100000 episodes

After a training of 100000 episodes the Q-table is given to the agent it is evaluated. Above it is shown the rolling average of 10000 episodes with a 500-episode window. As we see the results are pretty steady with rolling averages being between (7.75, 8.1). The mean value is 7.92 and the best rolling average is at 8.13.

For the LSPI agent I didn't manage to get some proper results to compare with the Q-Learning one. However here is the table of weights after inputting 5000-episode samples at LSPI of a Q-Learning agent:

[[1.19562800e+003] [8.89488473e-301] [2.78090155e-277] [1.54193259e-255]

[4.94750459e-287] [4.70061686e-287] [6.02530951e-278] [1.19562803e+003]

[7.85455220e-301] [2.45565143e-277] [1.36159044e-255] [4.36885179e-287]

[4.15083969e-287] [5.32059825e-278] [1.19562569e+003] [8.47201194e-301]

[2.64869437e-277] [1.46862738e-255] [4.71229468e-287] [4.47714427e-287]

[5.73885955e-278] [1.19562629e+003] [1.04047415e-300] [3.25294397e-277]

[1.80366699e-255] [5.78731573e-287] [5.49852019e-287] [7.04807198e-278]

[1.19333573e+003] [8.85399092e-297] [2.76811648e-273] [1.53484362e-251]

[4.92475868e-283] [4.67900600e-283] [5.99760844e-274] [1.20872059e+003]

[-1.41854788e-297] [-4.43495573e-274] [-2.45905964e-252] [-7.89023397e-284]

[-7.49649973e-284] [-9.60910716e-275]]

From this matrix I deduce that there are huge differences between the weights. So, a better implementation and calculation of the weight matrix it would cause better results.

References

1. Course's notes and videos
2. https://towardsdatascience.com/reinforcement-learning-with-openai-d445c2c687d2
3. https://medium.com/analytics-vidhya/reinforcement-learning-using-q-learning-to-drive-a-taxi-5720f7cf38df
4. https://towardsdatascience.com/simple-reinforcement-learning-q-learning-fcddc4b6fe56
5. Lagoudakis MG, Parr R (2003a) Least-squares policy iteration. Journal of Machine Learning Research 4:1107–1149
6. https://pythonhosted.org/lspi-python/index.html
7. https://github.com/jeonggwanlee/LSPI