Reinforcement Learning Tutorial

January 9, 2019

1 Gridworld setup

For the majority of this problem set, you will be simulating reinforcement agents in a grid-like environment, called *GridWorld*. This question just serves to set up the environment and make sure that you are able to run everything. You will be coding in python, and the only modules that you need are numpy and matplotlib. To complete the exercises, you will need to obtain the folder `rl_tutorial`, which contains several python files that will help you to complete the exercises. The file `gridworld.py` contains the gridworld environment, along with three example gridworlds, called `GridWorldExample1`, `GridWorldExample2`, and `CliffWorld`. The file `rlagents.py` contains the learning agents. Some of the agents are incomplete, and it will be your task to implement the missing parts. Finally, the file `plotFunctions.py` contains useful plotting functions.

Run the file `gridworld_setup.py` (e.g., type `python gridworld_setup.py` into the terminal). This file initializes the gridworld `GridWorldExample1` and an agent that takes random actions. You should see some text appear in the console, which prints out each action that the agent takes, as well as the reward, coming from the function `run_episode`. Have a look at the code inside this function in the file `rl_agents.py` (it will be useful for later). The agent repeatedly calls `choose_action` and `take_action` until a terminal state is reached. Inside the function `choose_action`, the `policy` function is called, which is set as default to a random policy.

Four figures also appear. The first one is a picture of the gridworld environment. Obstacle squares are colored black, terminal squares are colored gray, and jumps are denoted by arrows (start states are not indicated). The next figure plots the state-value function (v) for each state in the grid (in number and color). Next, is the action-state-value function (q), which plots up to four colors in each state. These indicate the state-action values ($q(s, a)$) of moving to one of the neighboring squares. For jump states, the square in the middle indicates the q value of the jump action. Finally, the last figure is the policy of the agent, which indicates the preferred action of the agent in each state.

2 Dynamic programming (DP)

2.1 Policy evaluation

In this question you will implement the dynamic programming prediction algorithm called *policy evaluation*, in which you need to calculate the state-value function $v_\pi(s)$ for an agent that is following a random policy. Inside the file `rl_agents.py`, you will find a class called `DP_Agent`, with several functions that have yet to be implemented. One of them is called `evaluatePolicy`. Follow the pseudocode shown below and implement policy evaluation for this agent. [*Hint: you do not need to run episodes to evaluate the policy for this agent. All you need from the gridworld environment is the policy transition probabilities and rewards. Note that the function* `initRandomPolicy` *builds the matrix* $\mathcal{P}^\pi$ *for a random policy.*]

After you complete the function, test it by writing a python script in a separate file that initializes the gridworld `GridWorldExample2` and a DP agent. Remember you need to initialise a random policy and *then* evaluate the policy. Plot the state-value function **before** and **after** policy evaluation (use the function `plotStateValue` found in `plotFunctions.py`. It has a parameter to suspend plotting).

Algorithm 1 Policy Evaluation

```
1:  procedure POLICYEVAL($\gamma$)
2:      Initialize $v(s) = 0$ for all states $s$
3:      while loop: $\Delta > 0.0001$
4:          for loop: for all states $s$ in $\mathcal{S}$
5:              $v'(s) \leftarrow \sum_a \pi(a|s) \sum_{s'} P^a_{ss'}[R^a_{ss'} + \gamma v(s')]$
6:              (note: this is equivalent to $v(s) \leftarrow R^\pi_{ss'} + \gamma P^\pi_{ss'} v(s')$)
7:          end loop
8:          $\Delta \leftarrow \max_s |v'(s) - v(s)|$
9:          $v(s) \leftarrow v'(s)$ for all $s$
10:     end loop
```

Figure 1: Policy Evaluation

2.2 Policy iteration

Next, you will implement dynamic programming control, or *policy iteration.* Since you have implemented policy evaluation in the previous question, you must first implement *policy improvement*, followed by a function *policy iteration*, which repeatedly calls policy evaluation and policy improvement until convergence. Use $\gamma = 0.9$. [*Hint: In policy improvement, try to use the value function $v_\pi(s)$ computed in policy evaluation to update the matrix $\mathcal{P}^\pi$ so that it greedily chooses the next state. For policy iteration, you should loop through these two functions until the policy no longer changes.*]

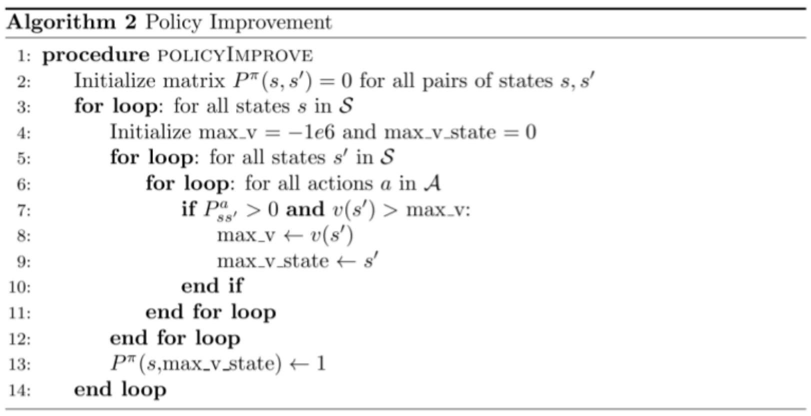
Again, after completing the function, test it by writing a script that initializes the same gridworld as before (`GridWorldExample2`) and a DP agent, and then does policy iteration. Plot the value-function (using `plotStateValue`) as well as the policy of the agent (using `plotPolicyPi`). Compare the value-function for a random policy (previous question) with the one here for the optimal policy.

3 On-policy temporal difference (TD) – SARSA

3.1 TD-Estimation

Now, you will implement state-action-value estimation using temporal difference learning. This means estimating the function $q_\pi(s, a)$ for an agent following a random policy. Again use $\gamma = 0.9$.

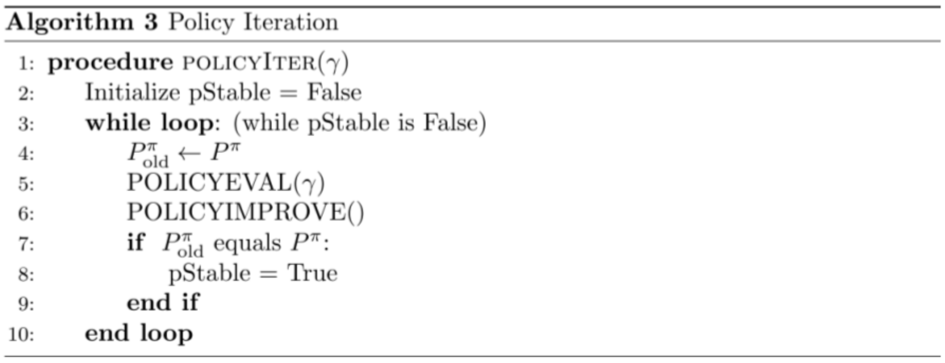
For TD methods, you will need to run sample episodes of the agent in the environment, and then use the TD update rule. Familiarise yourself with the class `RLAgent`, upon which all other agent classes are based. You will need several of these methods in your TD implementation, particularly `reset, choose_action, take_action` and `action_dict`.



Algorithm 2 Policy Improvement

```
1: procedure POLICYIMPROVE
2:     Initialize matrix $P^\pi(s, s') = 0$ for all pairs of states $s, s'$
3:     for loop: for all states $s$ in $\mathcal{S}$
4:         Initialize max_v $= -1e6$ and max_v_state $= 0$
5:         for loop: for all states $s'$ in $\mathcal{S}$
6:             for loop: for all actions $a$ in $\mathcal{A}$
7:                 if $P^a_{ss'} > 0$ and $v(s') >$ max_v:
8:                     max_v $\leftarrow v(s')$
9:                     max_v_state $\leftarrow s'$
10:                end if
11:            end for loop
12:        end for loop
13:        $P^\pi(s$,max_v_state$) \leftarrow 1$
14:    end loop
```

Figure 2: Policy Improvement



Algorithm 3 Policy Iteration

```
1: procedure POLICYITER($\gamma$)
2:     Initialize pStable = False
3:     while loop: (while pStable is False)
4:         $P^\pi_{\text{old}} \leftarrow P^\pi$
5:         POLICYEVAL($\gamma$)
6:         POLICYIMPROVE()
7:         if $P^\pi_{\text{old}}$ equals $P^\pi$:
8:             pStable = True
9:         end if
10:    end loop
```

Figure 3: Policy Iteration

[*Hint: There are two extra parameters to consider in this task: the number of episodes to run, as well as the learning rate, α. You can play around with these, but some sensible options are* 1000 *episodes, and* $\alpha = 0.05$ *(use these for subsequent questions as well).*]

Write a script that tests an agent doing TD-Estimation for `GridWorldExample3`, and plot the state-action-value function $Q(s, a)$, as well as the policy. This time, use the function `plotGreedyPolicyQ` to plot the policy.

Algorithm 4 Temporal Difference Policy Evaluation

```
1: procedure TDPolicyEval($\gamma$, $\alpha$, ntrials)
2:     Initialize $Q(s,a)$ arbitrarily for all states $s$ (e.g., $Q(s,a) = 0$)
3:     for loop: for $i = 1$ to ntrials (number of episodes)
4:         Initialize in starting state $s$
5:         Choose action $a$ from state $s$ according to policy $\pi$
6:         while loop: while terminal state has not been reached
7:             Take action $a$, observe reward $r$ and new state $s'$
8:             Choose action $a'$ from state $s'$ according to policy $\pi$
9:             $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$
10:            $s \leftarrow s', a \leftarrow a'$
11:        end loop (when terminal state is reached)
12:    end loop
```

Figure 4: TD-Estimation

3.2 SARSA

Next, implement TD on-policy control, or SARSA, and test with `GridWorldExample3`. You can use the policy function `epsilongreedyQPolicy` as the policy of the agent. Since, this function automatically uses the Q-values that the agent has learned, you do not need to implement a policy improvement function explicitly. All there is to do is to implement a policy iteration function which repeatedly evaluates the policy until convergence.

Algorithm 5 SARSA Policy Evaluation

```
1: procedure SARSA_PolicyEval($\gamma$, $\alpha$, ntrials)
2:     Initialize $Q(s,a)$ arbitrarily for all states $s$ (e.g., $Q(s,a) = 0$)
3:     for loop: for $i = 1$ to ntrials (number of episodes)
4:         Initialize in starting state $s$
5:         Choose action $a$ from state $s$ derived from Q (e.g., $\epsilon$-greedy)
6:         while loop: while terminal state has not been reached
7:             Take action $a$, observe reward $r$ and new state $s'$
8:             Choose action $a'$ from state $s'$ derived from Q (e.g., $\epsilon$-greedy)
9:             $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a') - Q(s,a)]$
10:            $s \leftarrow s', a \leftarrow a'$
11:        end loop (when terminal state is reached)
12:    end loop
```

Figure 5: SARSA Policy Estimation

Algorithm 6 SARSA Policy Iteration

```
1: procedure SARSA_PolicyIter($\gamma$, $\alpha$, ntrials)
2:     Initialize pStable = False
3:     Initialize $\mathbf{a}_{\text{old}}$ arbitrarily
4:     while loop: (while pStable is False)
5:         SARSA_POLICYEVAL($\gamma$, $\alpha$, ntrials)
6:         if : $\mathbf{a}_{\text{old}}$ equals $\arg\max_a Q(s,a)$
7:             pStable = True
8:         $\mathbf{a}_{\text{old}} = \arg\max_a Q(s,a)$ for all states $s$
9:         end if
10:    end loop
```

Figure 6: SARSA Control

4 Off-policy temporal difference (TD) – Q-learning

Implement off-policy temporal difference learning, or Q-learning. You will need to implement a new function for TD-Estimation, but policy iteration can remain the same as for SARSA. Choose the behavior policy to be ϵ-greedy, and the optimized policy to be greedy (both are already implemented for you inside `RLAgent`).

Test this out and compare it directly with SARSA on another gridworld example called `CliffWorld`. You should notice that SARSA and Q-learning find different policies for this environment. Which one is better and why? Plot the Q-values as well as the policies for the two algorithms. Additionally, run one-hundred episodes of each after learning, and compare the average total return of each episode. Which algorithm receives more reward on average?

5 The lambda return – SARSA(λ)

Implement a SARSA agent which uses the λ-return. To do this, you will only need to reimplement policy evaluation for the previous SARSA agent, but now include eligibility traces in your algorithm (Note that this means all Q values are updated on each step).

Test this on `GridWorldExample2` or `GridWorldExample3`, using $\lambda = 0.9$. Compare it with your previous SARSA algorithm, specifically looking at the number of episodes that it takes to reach a good policy. SARSA(λ) should converge more quickly.

6 Value function approximation – SARSA(λ) mountain car

In this last section, you will implement a mountain car RL agent that uses SARSA(λ) TD learning. In the same directory `rl_tutorial`, you will find the files `mountainworld.py` and `mountaincar_agent.py`. Again, your task will be to implement some functions that the agent will use to achieve the task.

Algorithm 7 Q-learning Policy Evaluation

```
1: procedure Q_PolicyEval($\gamma$, $\alpha$, ntrials)
2:     Initialize $Q(s,a)$ arbitrarily for all states $s$ (e.g., $Q(s,a) = 0$)
3:     for loop: for $i = 1$ to ntrials (number of episodes)
4:         Initialize in starting state $s$
5:         Choose action $a$ from state $s$ derived from Q and behavior policy $\mu$
6:         (e.g., $\epsilon$-greedy)
7:         while loop: while terminal state has not been reached
8:             Take action $a$, observe reward $r$ and new state $s'$
9:             Choose action $a'$ from state $s'$ derived from Q and behavior policy $\mu$
10:            (e.g., $\epsilon$-greedy)
11:            Choose action $a''$ from state $s'$ derived from Q and optimized policy $\pi$
12:            (e.g., greedy)
13:            $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma Q(s',a'') - Q(s,a)]$
14:            $s \leftarrow s', a \leftarrow a'$ (note that behavior policy is followed)
15:        end loop (when terminal state is reached)
16:    end loop
```

Figure 7: Q-learning

Algorithm 9 SARSA(λ) Policy Evaluation

```
1: procedure SARSA_$\lambda$_EVAL($\gamma$, $\alpha$, ntrials,$\lambda$)
2:     Initialize $Q(s,a) = 0$ for all states $s$ and actions $a$
3:     Initialize $E(s,a) = 0$ for all states $s$ and actions $a$
4:     for loop: for $i = 1$ to ntrials (number of episodes)
5:         Initialize in starting state $s$
6:         Choose action $a$ from state $s$ derived from Q (e.g., $\epsilon$-greedy)
7:         while loop: while terminal state has not been reached
8:             Take action $a$, observe reward $r$ and new state $s'$
9:             Choose action $a'$ from state $s'$ derived from Q (e.g., $\epsilon$-greedy)
10:            $\delta_t \leftarrow r + \gamma Q(s',a') - Q(s,a)$
11:            $E(s,a) \leftarrow E(s,a) + 1$
12:            $Q(s,a) \leftarrow Q(s,a) + \alpha\delta_t E(s,a)$
13:            $E(s,a) \leftarrow \gamma\lambda E(s,a)$
14:            $s \leftarrow s', a \leftarrow a'$
15:        end loop (when terminal state is reached)
16:    end loop
```

Figure 8: SARSA(λ)

This agent is placed in an environment consisting of a valley between two hills. The objective of the agent is to move to the top of the right hill, where it will receive a large reward. Otherwise, it receives negative reward for every step that it takes.

The state space of the car agent is its position and velocity, (x, v), and the available actions at any given time are {-1,1,0}, for accelerating backwards, forwards and no acceleration. These actions must compete with the dynamics of the system (somewhat resembling gravity), which pull the car down to the bottom of the valley. They are represented by the following update equations

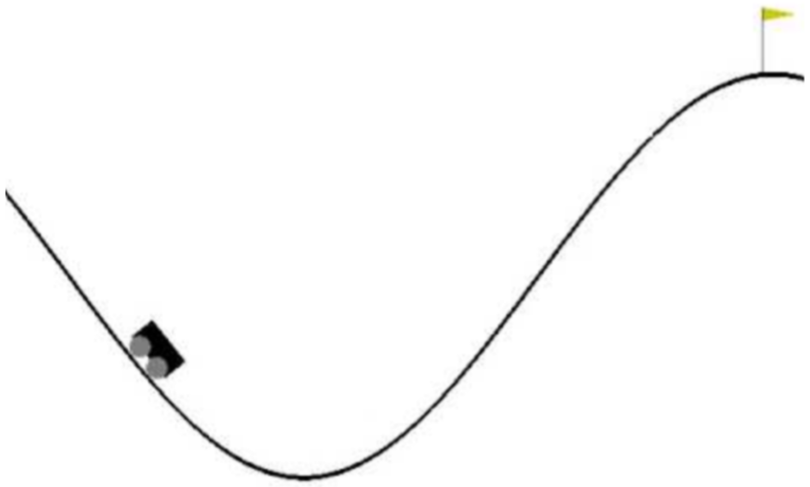


Figure 9: The Mountain Car Problem

(note that time is discrete):

$$\begin{aligned} v_{t+1} &= \text{bound}_v[v_t + 0.001a_t - 0.0025\cos(3x_t)] \\ x_{t+1} &= \text{bound}_x[x_t + v_{t+1}] \end{aligned}$$

where $\text{bound}_v[]$ and $\text{bound}_x[]$ keep the v and x values inside the bounds $[-0.07, 0.07]$ and $[-1.2, 0.6]$, respectively.

Notice that the state space for this environment is continuous, so in order to solve it, we must use value function approximation $\hat{q}(s, a, \mathbf{w})$, where $s = (x, v)$ is the state, $a \in \{-1, 0, 1\}$ is the action, and $\mathbf{w}$ is a vector of weight parameters. We will adopt a very simple linear function

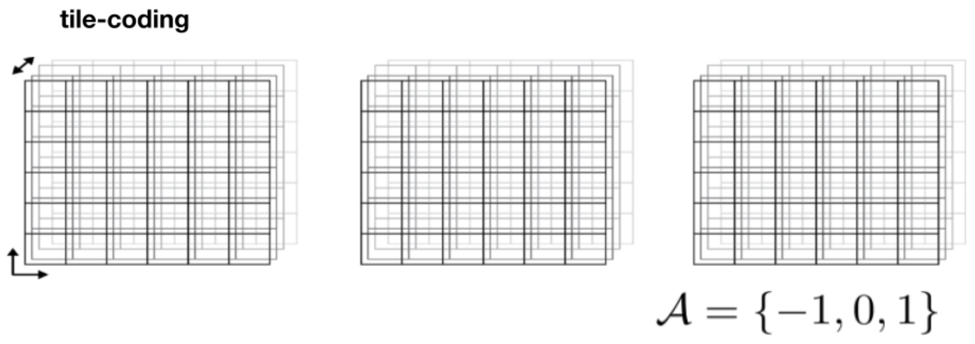


Figure 10: Tile Coding. Shown here for $n = 3$ grids of size $(6, 6)$ and three actions $\{-1, 0, 1\}$.

approximation here, called *tile coding*. In tile coding, we discretize the state space into a set of n overlapping grids, as shown in fig. 10. Each grid has a shape (r, c), but is slightly offset from the others, and there are three sets of grids, one for each action. Each tile of each grid is a separate feature $\phi_i(s, a)$, with the total number of features being: $3 \times n \times r \times c$. Tile coding is a binary code: when the current state of the car falls inside one of the tiles, the corresponding feature is equal to one, otherwise it is zero. Since the feature tiles of an individual grid are not overlapping, only one of them will be active at any given time. This means that for any state that the car is in, there will be exactly n nonzero features. This coding has already been implemented for you in the function `get_features`.

6.1 Implement the control algorithm

Your task for this problem, as for the previous one, is to implement SARSA(λ) with tile-coding value function approximation for the mountain car problem. You will have to implement one function for the mountain car agent – policy evaluation (`evaluatePolicyQ`). Note that this is very similar to the algorithm above for SARSA(λ) in the GridWorld, except that the eligibility traces now must be updated with the feature values, like so:

$$E(S, A) \leftarrow E(S, A) + \phi(S, A)$$

In order to do policy iteration, you can simply repeatedly run `evaluatePolicyQ` for several trials. Try the following parameter ranges: $\lambda \in [0.8, 1.0]$, $\alpha \in [0.001, 0.01]$, $\gamma \in [0.8, 1.0]$ (see solution `testMountainCar.py` – $\alpha = 0.05$, $\gamma = 0.99$, $\lambda = 0.9$, and ntrials= 200).

Algorithm 11 SARSA(λ) Policy Evaluation with Function Approximation

```
procedure SARSA_λ_EVAL($\gamma$, $\alpha$, ntrials,$\lambda$)
    Initialize $w_j = 0$ for all features $\phi_j$
    Initialize $E_j = 0$ for all features $\phi_j$ (eligibility traces)
    for loop: for $i = 1$ to ntrials (number of episodes)
        Initialize in starting state $s$
        Choose action $a$ from state $s$ derived from Q (e.g., $\epsilon$-greedy)
        while loop: while terminal state has not been reached
            Take action $a$, observe reward $r$ and new state $s'$
            Choose action $a'$ from state $s'$ derived from Q (e.g., $\epsilon$-greedy)
            $\delta_t \leftarrow r + \gamma Q(s', a') - Q(s, a)$
            $E_j \leftarrow E_j + \phi_j(s, a)$ for all features $\phi_j(s, a)$
            $Q_j \leftarrow Q_j + \alpha \delta_t E_j$
            $s \leftarrow s', a \leftarrow a'$ (note that behavior policy is followed)
        end loop (when terminal state is reached)
    end loop
```

Figure 11: SARSA(λ) with function approximation