Tutorial 1: Tabular Methods in MDP

mingfei.sun@manchester.ac.uk

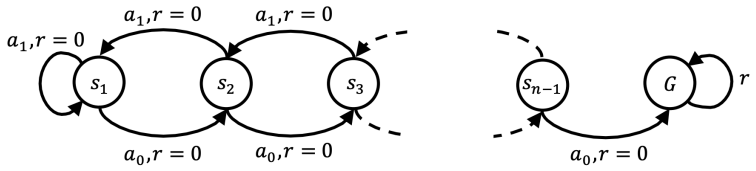
February 12, 2025

In this tutorial, we will familiarize with tabular methods for reinforcement learning. In Part I, we will discuss how to evaluate and find optimal policy, both when the model of the world is available and when it is not. The exercises marked with BONUS are optional – they cover and provide guidance for deriving some fundamental results, and you can try solving them after the tutorial or instead of Part II if you are more interested in theory of reinforcement learning. In Part II, we will implement value iteration and Q-learning algorithms on some simple gridworld environments and Pacman.

Part I

Exercise 1. Simple MDP.

Consider a simple n-state MDP shown in the Figure 1. The agent starts at state s_1, and has two actions available in each of the states s_i, with reward 0. Taking any action from state G results in a reward $r > 0$ and the agent stays in state G. The actions are deterministic and always succeed. Assume a discount factor $\gamma < 1$.



(i) What is optimal action at any state s_i? Find the optimal value function for all states s_i and the goal state G. Does the optimal policy depend on the discount factor γ? .

(ii) How will the optimal value function change with the addition of a constant factor β to all rewards? How about optimal policy?

(iii) How will the optimal value function and policy change if all rewards are both shifted by a constant factor β and scaled by a constant factor α?

$$r \to \alpha(r + \beta)$$

Exercise 2. Policy evaluation.

Consider the Bellman equation for a state value of policy π at state $s \in S$ with actions $a \in A$ (considering infinite horizon only):

$$V^{\pi}(s) = \sum_{a \in A} \pi(a|s) \sum_{s' \in S} P(s'|s,a)[R(s',s,a) + \gamma V^{\pi}(s')]. \tag{1}$$

(i) Define v_i^{π}, r_i^{π} and $P_{i,j}^{\pi}$, such that equation 1 can be expressed in a matrix form:

$$\mathbf{v}^{\pi} = \mathbf{r}^{\pi} + \gamma P^{\pi} \mathbf{v}^{\pi}. \tag{2}$$

(ii) BONUS. Show the linear equation (2) always has a unique analytical solution: $\mathbf{v}^{\pi} = (I - \gamma P^{\pi})^{-1} \mathbf{r}^{\pi}$. HINT: you will need to show that P^{π} is a row-stochastic matrix, the absolute value of all eigenvalues λ_i of a row-stochastic matrix satisfy property $|\lambda_i| \leq 1$ and $||\cdot||_{\infty}$ norm of a row-stochastic matrix is 1, and matrix $I - \gamma P^{\pi}$ is nonsingular for any $0 < \gamma < 1$.

(iii) BONUS. Show that iterative policy evaluation algorithm with tolerance level ϵ always terminates. Show the solution obtained this way, $\mathbf{v}' = \mathbf{v}^{(k)}$ such that $||\mathbf{v}^{(k)} - \mathbf{v}^{(k-1)}||_{\infty} < \epsilon$, will have an error bound by $||\mathbf{v}^{\pi} - \mathbf{v}'||_{\infty} \leq \frac{\epsilon\gamma}{1-\gamma}$. HINT: Recall the properties of contractions on Banach spaces. You will need to show that Bellman operator $B^{\pi}\mathbf{v} = \mathbf{r}^{\pi} + \gamma P^{\pi}\mathbf{v}$ is a strict contraction on a vector space $\mathbb{R}^{|S|}$ equipped with the $||\cdot||_{\infty}$ norm.

Exercise 3. Policy and value iteration.

Recall the policy and value iteration algorithms. Policy iteration consists of two steps: policy evaluation and policy improvement. Value iteration integrates the two steps into one update rule:

$$V^{(k+1)}(s) = \max_{a}[R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V^{(k)}(s')]. \tag{3}$$

(i) For an MDP with an infinite horizon and static transition probability, when is the optimal policy $\pi^*(s) = \arg\max_{\pi} V^{\pi}(s)$ deterministic? Can the optimal policy be stochastic? Under what conditions? Explain your answer.

(ii) BONUS. What is the complexity of policy evaluation and policy improvement steps of policy iteration? How does it compare to the analytical solution which involves computing a matrix inverse?

(iii) BONUS. Prove the policy improvement theorem. If $\pi'(a|s)$ is a greedy policy with respect to $Q^{\pi}(s,a) = R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a) V^{\pi}(s)$, it follows that $V^{\pi'}(s) \geq V^{\pi}(s)$, for each $s \in S$.

(iv) BONUS. Prove convergence of value iteration algorithm to an optimal value function $V^*(s)$. HINT: Similar to the proof of policy evaluation convergence, you will need to show that Bellman optimality operator, $B^*V(s) = \max_a[R(s,a) + \gamma \sum_{s' \in S} P(s'|s,a)V(s')]$, is a strict contraction on a vector space $\mathbb{R}^{|S|}$ equipped with the $||\cdot||_{\infty}$ norm.

Exercise 4. Control without a model.

Recall constant-α MC and TD(0) control algorithms. The two stochastic methods are iteratively estimating the value function:

$$\hat{V}^{\pi}(s_t) \to \hat{V}^{\pi}(s_t) + \alpha(\hat{R}_t - \hat{V}^{\pi}(s_t)), \tag{4}$$

relying on the following estimates of return which are used as a target:

$$\hat{R}_t^{MC} = \sum_{i=t}^{H} \gamma^{i-t} r_t, \tag{5}$$

$$\hat{R}_t^{TD} = r_t + \gamma \hat{V}^{\pi}(s_{t+1}), \tag{6}$$

where r_t are rewards observed following a target policy π and $\alpha > 0$ is the update size.

(i) How do Monte Carlo and TD(0) estimators of return compare in terms of bias and variance? Explain your answer.

(ii) Why is it necessary to use an ϵ-greedy policy? Recall the difference between target policy and behaviour policy. Under what conditions can a policy be used as a behavioural policy?

(iii) Why can Q-learning be considered an off-policy method? What are some advantages of off-policy methods?

(iv) BONUS. Prove the ϵ-greedy policy improvement theorem. If π' is an ϵ-greedy policy w.r.t. q_π, where π is an ϵ-soft policy, it follows that $V^{\pi'}(s) \geq V^{\pi}(s)$, for each $s \in S$. HINT: The proof is very similar to the proof of policy improvement theorem from the previous exercise.

Exercise 5. Importance sampling.

Consider the importance sampling estimator:

$$\hat{s}_q = \frac{1}{N} \sum_{i}^{N} f(x_i) \frac{p(x_i)}{q(x_i)}, \; x_i \sim q, \tag{7}$$

where $p(x)$ and $q(x)$ are known and $s_p = \mathbb{E}_{x \sim p}[f(x)]$ is the estimated quantity.

(i) Under which conditions is $\hat{s}_q$ an unbiased estimator? Identify some of the problems with the importance sampling estimator and propose solutions.

(ii) BONUS. Let us consider the variance of the importance sample estimator. When is it maximized? Show that the choice of $q(x)$ minimizing the variance is $q^*(x) = p(x)|f(x)|/Z$, where Z is the normalization constant. Because sampling from q^* is usually infeasible, an alternative approach is to use weighted or self-normalized importance sampling estimator:

$$\hat{s}_{WIS} = \frac{\sum_i^N w_i f(x_i)}{\sum_i^N w_i}, \; w_i = \frac{p(x_i)}{q(x_i)}, \; x_i \sim q. \tag{8}$$

Show $\hat{s}_{WIS}$ is a biased estimator, however it does converge to s_p as $N \to \infty$.

Part II: Coding Exercises

Setup Instructions

Download the code from this link: `https://github.com/mingfeisun/COMP64202-RL/tree/master/Labs/Lab1-MDP/code`. Try running the following command

```
python gridworld.py -m
```

If this works, you're all set! If not, read below or ask for help. If you are using MacOS and the pop-up window is not showing anything, check if you have installed the tk package correctly. The provided codes have been tested on MacOS with Python3.11 and python-tk@3.11. Installation commands are following:

```
brew install python@3.11
brew install python-tk@3.11
```

Python The coding exercises are based on `Python 3`. If you have Python, you can check which version you have installed by running

```
python -V
```

Also check `python3 -V`. If you don't have any python version installed, we recommend using Anaconda. If you have Python 3 with Anaconda, you can create a virtual environment (see below).

Installing Anaconda If you don't have any Python installed, we recommend using Anaconda. Follow the instructions here: `https://conda.io/en/latest/miniconda.html`.

Exercise 6. Value Iteration.

In this exercise, you will implement the value iteration algorithm on a simple gridworld environment. To have a look at the environment and play manually, execute

```
python gridworld.py -m
```

You can control many aspects of the simulation. A full list of options is available by running

```
python gridworld.py -h
```

In value iteration, we want to learn the optimal value function. Hence we need a way to save the values for each state that we encounter. Before starting the exercise, think about or discuss the following:

> What is a good way to implement this? How would you initialise the values? What if you don't know upfront what the state space is, how would you go about implementing this?

You should use the `Counter` class in `util.py` to save the state values, which is a dictionary with a default value of zero.

(i) **Algorithm.** First, you will edit the file `valueIterationAgents.py`. Value iteration is an offline planner, and hence you have access to the state transitions and environment dynamics. The `ValueIterationAgent` has a model of the environment via a MarkovDecisionProcess (see `mdp.py`) that is used to estimate state values before ever actually acting.

To implement the value iteration, complete the `valueIteration()` function in `valueIterationAgents.py`. This function should compute the values for all states.

Note that to compute the optimal values, the agent does *not* interact with the environment. Your value iteration agent is an offline planner, not a reinforcement learning agent, and so the relevant training option is the number of iterations of value iteration it should run (option -i).

(ii) **Action Selection.** Now that you have a way to compute the optimal values for each state, you need an agent that acts optimally given those values. For this, you first need to complete `computeQValueFromValues(state, action)`: This function returns the Q-value of the (state, action) pair given by the value function given by `self.values`. Using this, implement action selection via `computeActionFromValues(state)`: This function computes the best action according to the value function given by `self.values`.

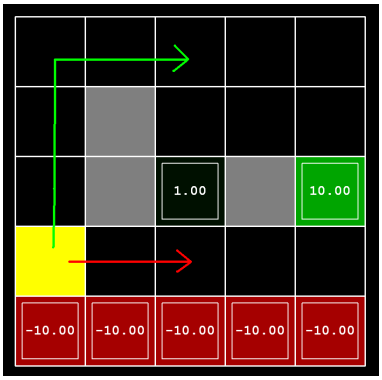
(iii) **Evaluation.** To run the algorithm, execute:

```
python gridworld.py -a value -i 100 -k 10
```

This will show you the values of the algorithm you implemented for each state. Press a key to cycle through state values, Q-values, and policies.

Please check with us if the values are correct before continuing.

Exercise 7. CLIFFWORLD MDP.



In this exercise, we look at the Cliff World environment and learn a policy using the value iteration algorithm you just implemented. We are interested in the effect of different parameters on the optimal policy. There are two terminal states with positive payoffs +1 and +10. We distinguish between two types of paths: (1) paths that "risk the cliff" and travel near the bottom row of the grid; these paths are shorter but risk earning a large negative payoff, and are represented by the red arrow in the figure below. (2) paths that "avoid the cliff" and travel along the top edge of the grid. These paths are longer but are less likely to incur huge negative payoffs. To play with the environment manually, execute

```
python gridworld.py -g DiscountGrid -m
```

You will see that the agent starts in the bottom left corner, and that there are two terminal states - one with a reward of 1, and one with a reward of 10. The bottom row consists of a cliff which the agent can fall down, which gives a reward of -10 and terminates the episode. When testing this, you might notice that sometimes the agent takes random actions instead of the one you chose. This is just part of life in a grid world!

Before you start the exercise, think about and discuss the following:

> What is a good strategy for the agent to maximise its return? How does the reward in non-terminal states influence the optimal policy? What effect do different discount factors have on the optimal policy? How does the randomness of the environment change the optimal behaviour?

(i) **Varying MDP parameters.** Here are the optimal policy types you should attempt to produce:

(a) Prefer the close exit (+1), risking the cliff (-10)

(b) Prefer the close exit (+1), but avoiding the cliff (-10)

(c) Prefer the distant exit (+10), risking the cliff (-10)

(d) Prefer the distant exit (+10), avoiding the cliff (-10)

(e) Avoid both exits and the cliff (so an episode should never terminate)

To do so, you can vary the following settings:

- The discount factor, using `-d`
- The reward given to the agent at each time step, using `-r`
- The probability of selecting a random action, using `-n`

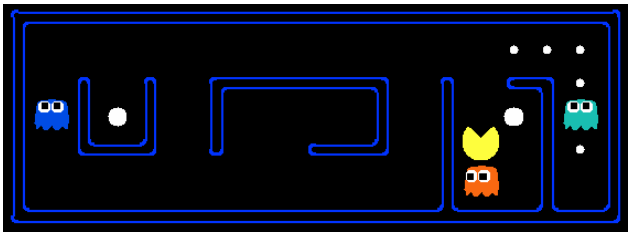
Your setting of the parameter values for each part should have the property that, if your agent followed its optimal policy without being subject to any noise, it would exhibit the given behavior.

To test different settings, use

```
python gridworld.py -g DiscountGrid -a value -i 100 -d 0.9 -r 0.0 -n 0.2
```

Please put your answers for each setting into the file `analysis.py`, using the functions question3a() through question3e(). These should each return a 3-item tuple of (discount, noise, living reward), unless if a particular behavior is not achieved for any setting of the parameters. In this case return the string 'NOT POSSIBLE'.

Exercise 8. PACMAN MDP.



Time to learn Pacman! In this exercise, you will implement tabular Q-Learning for Pacman. To have a look at the environment and play manually, execute

```
python pacman.py
```

For the purpose of this exercise we will use a tiny version of Pacman, which you can test using

```
python pacman.py --layout smallGrid
```

If the above works you can start the exercise below, otherwise have a look at the setup instructions (`setup_instructions.md`) or let us know and we will help you set up.

Before starting the exercise, think about / discuss the following:

Since we are implementing *tabular* Q-Learning, you need a way to save the Q-Values for all state-action pairs. What do you think is a good way to implement this? What do you do with actions that are invalid given a state?

You will edit the class `QLearningAgent` in the file `qlearningAgents.py`.

(i) **Tabular Q-Values and Action Selection.** You are not given the action and state space upfront. Instead, you have to extend the Q-Table on the fly and initialise the values for new state-action pairs once you see them. For this, you should again use the class `Counter` in `util.py`. Have a look at what it is doing, and use it in the `__init__` function of your Q-Learning agent. Once you have initialised yout Q-Table, implement the following functions: - `getQValue`, which returns a Q-value for a given state and action - `computeValueFromQValues`, which returns the maximum Q-value of all legal actions in the given state - `computeActionFromQValues`, which computes the best action to take in a state - `getAction`, which selects an action given the current state using epsilon-greedy action selection.

(ii) **Update Function.** Once you have the above functionality, we have to update the Q-values in order to learn how to select the best action. For this, you need to adapt the function `update` and implement the Q-learning update function.

(iii) **Evaluation.** Once you are done, you should be able to execute:

```
python pacman.py -p PacmanQAgent -x 2000 -n 2010 -l smallGrid
```

and let your agent learn! (This will train for 2000 games and evaluate for 10.) Once Pacman is done training, he should win very reliably in test games (at least 90% of the time). Flags you can use: - `-n 2010` Number of games to play (when to switch between training and testing is determined by numTraining) - `-x 2000` How many episodes are training (suppressed output) - `-a numTraining=2000` Use this option instead of the above, if you want to watch training - `-a epsilon=0.05, alpha=0.2, gamma=0.8` Set different learning parameters

If you want to watch 10 training games to see what's going on, use the command

```
python pacman.py -p PacmanQAgent -n 10 -l smallGrid -a numTraining=10
```

(iv) **Scaling Things Up.** Now, what happens if you run the same code on larger grids (options: `mediumGrid`, `smallClassic`, `mediumClassic`), like

```
python pacman.py -p PacmanQAgent -x 2000 -n 2010 -l mediumGrid
```

Why does this fail? What are strategies to scale your algorithm to larger grids?