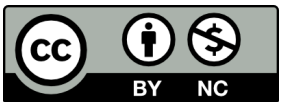
Business 4720

Reinforcement Learning – Tabular Methods

Joerg Evermann



Learning Goals

After reading this chapter, you should be able to:

- Define and explain basic concepts of reinforcement learning, including policy, environment, action, state, reward, return, value functions and model.
- Build a basic k-armed bandit agent and environment in pseudocode or in Python.
- Explain the trade-off between exploration and exploitation and how an ϵ-greedy or ϵ-soft policy can be used in this context.
- Explain the principles behind iterative policy evaluation and policy improvement for solving Markov decision processes using dynamic programming.
- Explain the principles behind MC control, including random starts.
- Explain the principles behind TD control as a bootstrapping method and the differences to MC control.
- Differentiate off-policy learning from on-policy learning.

Sources and Further Reading

The material in this chapter is based on the following sources.

Richard S. Sutton and Andrew G. Barto (2018) *Reinforcement Learning – An Introduction*. 2nd edition, The MIT Press, Cambridge, MA. (SB)
http://incompleteideas.net/book/the-book.html

Chapters 2–7

The Sutton & Barto book is a standard introductory textbook on reinforcement learning and widely used. It is very approachable, but at the same time also detailed and thorough in its exposition. Its focus is on RL prior to the use of neural networks for function approximation, so up to about 2015. While it does not provide Python code itself, the pseudo-code in the book is easily implemented.

Resources

Complete implementations of all examples in this chapter are available in the following GitHub repo:
https://github.com/jevermann/busi4720-rl

The project can be cloned from this URL:
https://github.com/jevermann/busi4720-rl.git

1 Introduction

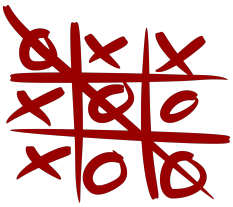
Reinforcement learning (RL) is a type of machine learning in which learning *agents* operate in an *environment* by taking *actions* and receiving *rewards*. The aim is to learn *optimal policies*, that is, those actions for each state that will *maximize* the sum of future rewards. The agent discovers which actions to take and how useful or valuable they are in each state by trying them and observing the reward and the new state.

Initially, agents have little or no knowledge of their environment, so most of the actions will be random exploratory actions. As agents learn more about their environment, they will want to exploit this knowledge by taking the valuable actions, rather than exploring randomly. On the other hand, less exploration may also mean that better actions will not be discovered. In other words, an agent is faced with a trade-off between exploration and exploitation.

What makes RL challenging is both the incomplete knowledge of the environment as well as the stochastic nature of the environment. Taking the same action in the same state will not always yield the same reward, and will not alway put the agent in the same new state. The lack of complete knowledge of the environment also means that typical RL problems cannot be solved by optimization; optimization requires full knowledge of the environment, which for stochastic environments, includes knowledge of any probability distributions. This requirement is not fulfilled in RL problem settings.

The core elements of an RL problem are the following:

- **Policy** π: A deterministic policy π specifies for each state s the action to take, whereas a stochastic policy π specifies for each state s a probability distribution over the possible actions a in state s.
- **Reward** R: The reward is received from the environment after each action. The reward may be positive, negative, or zero. In designing RL problems, the reward function is critical to inducing the correct learning behaviour and having the RL agent solve the right problem.
- **Return** G: The return is the possibly discounted sum of future rewards. The discount factor $0 < \gamma \leq 1$ expresses the fact that immediate rewards are worth more than future rewards. This is due to the uncertain nature of future rewards. In a stochastic environment of which the agent has incomplete knowledge, future rewards may or may not accrue as expected.
- **State value function** v: This function expresses how valuable it is for an agent to be in any particular state s. It is defined as the expected return for each state.
- **Action value function** q: This function expresses how valuable it is for an agent in a particular state s to take a specific action a. It is defined as the expected return for each state and action taken in that state.
- **Model** p: This is a set of probability distributions over rewards and new states for every pair of current state s and action a. It expresses the stochastic behaviour of the environment. If an RL agent had such a model of the environment, an optimal



https://commons.wikimedia.org/wiki/File:Tic_tac_toe.svg

Figure 1: The game of Tic-Tac-Toe

policy can be found using dynamic programming, a type of optimization. RL agents typically do not try to build such a model, but instead focus on learning the state value function, the action value function, or the policy directly.

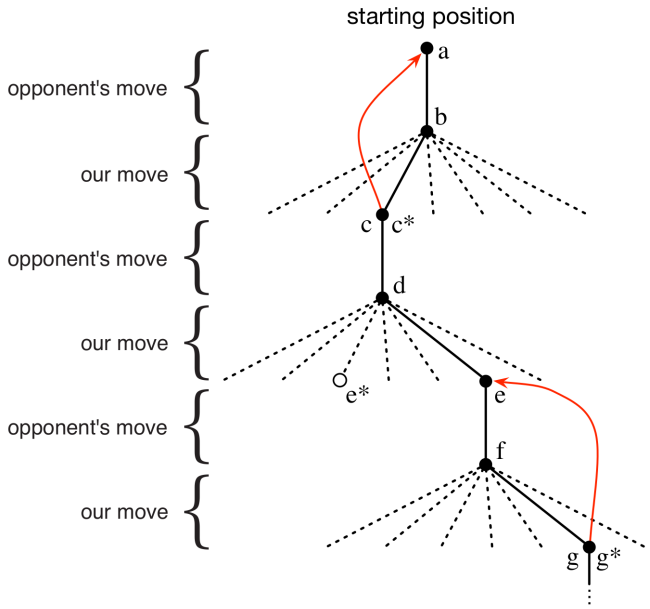
Introductory Example

Consider an RL agent learning the game of Tic-Tac-Toe ("naughts-and-crosses"), as shown in Figure 1. A state is defined as the position of all the X and O on the board; the possible actions in each state are to place an X in a free space (assume the agent plays X). The reward at each step is 0 except it is +1 when the game is won. Clearly, the value of any state with a row of X-X-X is 1 because the reward in this case is 1. The value of any state with a row of O-O-O or a full board is 0 because no future reward can occur.

Figure 2 indicates a sequence of moves by the RL agent and the opponent where a starred state (e.g. c^*) indicates an optimal state. Beginning from state a, the opponent makes the first move and brings the agent to state b. The agent then exploits the knowledge about the environment that is reflected in its policy or state value function and chooses the optimal action to move to state c^*. The opponent's move leads to state d. Now the agent makes an exploratory move and rather than moving to optimal state e^* it moves to state e. The opponent moves the state to f and the following action of the agent is exploiting behaviour again.

Behaviour that exploits knowledge about the environment, that is, the current action value function and policy, is called *greedy* behaviour as it seeks to maximize the value of the next state. In contrast, exploratory behaviour is typically *random* behaviour.

After each greedy action from state s_t, the RL agent updates its value function for the state s_t based on the value of the new state s_{t+1}. The intuition is that if the optimal action from an initial state of low value results in a new state of high value then the value of the initial state should reflect this. In other words, the updated value of the initial state should be closer to that of the final state. This leads to the central *update rule* in *temporal-difference learning*:



Source: SB Figure 1.1

Figure 2: Exploration and exploitation in an RL environment

$$V(S_t) \leftarrow V(S_t) + \alpha \left[V(S_{t+1}) - V(S_t) \right] \tag{1}$$

Here, V is the value function, and α is a *step size* that determines the rate of update or learning. The term $V(S_{t+1}) - V(S_t)$ is called the *error* and the term $V(S_{t+1})$ is called the *update target*.

Applications in Business and Management

There are many applications for reinforcement learning in business and management. Consider the following examples:

- *Dynamic Pricing*: Dynamic pricing involves setting flexible prices for products or services based on current market demands. RL algorithms can help businesses optimize pricing strategies in real-time by learning from consumer behavior and competitor actions, maximizing revenue or market share.
- *Supply Chain Optimization*: In supply chain management, RL can optimize inventory levels, improve logistics, and manage the supply chain network's dynamic environment. By learning from historical data and ongoing operations, RL algorithms can make adjustments to inventory and shipping strategies, reducing costs and improving service levels.

https://commons.wikimedia.org/wiki/File:Antique_one-armed_bandit,_Ventnor,_Isle_of_Wight,_UK.jpg

Figure 3: An ”one-armed bandit” slot machine

- *Customer Interaction Management*: Reinforcement learning can enhance customer relationship management systems by learning to tailor interactions based on customer behavior. This includes optimizing marketing strategies, personalizing recommendations, and improving customer service, all aimed at enhancing customer satisfaction and loyalty.
- *Financial Portfolio Management*: In finance, RL can be used for portfolio management, where the goal is to optimize the allocation of assets in a portfolio over time. RL algorithms can adapt to changes in market conditions, learning to maximize returns or minimize risk based on the investment strategy.
- *Manufacturing Process Optimization*: RL algorithms can be applied to control and optimize manufacturing processes by continuously learning and adapting to new data. This can include adjustments to machine settings, production schedules, and maintenance plans to optimize efficiency and reduce operational costs.

2 K-Armed Bandits

To introduce RL learning, consider the k-armed bandit problem. It is named after the nickname of early slot machines (Figure 3). The RL agent is faced with k such slot machines that give different stochastic rewards. The rewards given by each of the k bandits are initially unknown to the agent. The goal of the agent is to find a policy of which bandit to play in order to maximize the return, that is, the sum of future rewards.

The k-armed bandit problem is a very simple RL problem because it is *stateless*. That is, the agent is only ever in one state and the state does not change. This means that the action value function depends only on the action, not the state-action pair.

Formally, there are k possible actions A_t at time t with stochastic reward R_t. The action value for each action can be defined as the average reward for that action:

$$Q_t(a) = \frac{\sum_{i=1}^{t-1} R_i \times \mathbb{1}_a}{\sum_{i=1}^{t-1} \mathbb{1}_a} \qquad \text{(average reward)}$$

Here $\mathbb{1}_a$ is 1 when action a has been taken and 0 when another action has been taken.

A suitable policy that balances exploitation of existing knowledge and exploration for gathering new knowledge is the *ϵ-greedy policy*. An ϵ-greedy policy is one that with probability ϵ takes a random action and with probability $1 - \epsilon$ takes the optimal action:

$$A_t = \underset{a}{\operatorname{argmax}}\, Q_t(a)$$

An incremental implementation of the action value function simply updates the running average when a new reward is received, as follows:

$$Q_{t+1}(a) = Q_t(a) + \frac{1}{t}\left[R_t(a) - Q_t(a)\right] \tag{2}$$

Note how the form of Equations 1 and 2 is similar. They represent different cases of the general *update rule for estimates*:

$$NewEstimate \leftarrow OldEstimate + StepSize\left[Target - OldEstimate\right]$$

Where $[Target - OldEstimate]$ is the *error* in the estimate.

A complete k-armed bandit algorithm is shown in peusdocode in Figure 4. The corresponding implementation in Python is straightforward[1]. The following code block defines a class `k_bandit_agent` that represents an agent. Initialization specifies the number of bandits k in the environment, the parameter ϵ for the ϵ-greedy policy and the initial value of the action value function, which may be 0 as in the pseudocode in Figure 4. The method `determine_action` is simply the ϵ-greedy policy. The `train` method for each step determines the action to take, then takes that action in the environment and receives a reward. The agent then updates the action value function as in Equation 2.

[1]Complete implementation is available at https://github.com/jevermann/busi4720-rl/blob/main/bandits.py

Initialize, for $a = 1$ to k:

$Q(a) \leftarrow 0$

$N(a) \leftarrow 0$

Loop forever:

$$A \leftarrow \begin{cases} \text{argmax}_a\, Q(a) & \text{with probability } 1-\epsilon \\ \text{a random action} & \text{with probability } \epsilon \end{cases}$$

$R \leftarrow \text{bandit}(A)$

$N(A) \leftarrow N(A) + 1$

$$Q(A) \leftarrow Q(A) + \frac{1}{N(A)}\left[R - Q(A)\right]$$

Figure 4: A simple bandit algorithm (Source: SB)

```python
class k_bandit_agent:
    def __init__(self, k, epsilon, initial_value):
        self.k = k
        self.epsilon = epsilon
        self.env = k_bandit_env(k)

        self.Q = [initial_value] * self.k
        self.N = [.0] * self.k

    def determine_action(self):
        if random.uniform(0,1) < self.epsilon:
            # explore
            action = random.randint(0, self.k-1)
        else:
            # exploit
            action = self.Q.index(max(self.Q))
        return action

    def train(self, steps):
        rewards = []
        for step in range(steps):
            action = self.determine_action()
            reward = self.env.step(action)
            self.N[action] += 1
            self.Q[action] = (reward-self.Q[action])/self.N[action]
            rewards.append(reward)
        return rewards
```

A corresponding environment for the agent to act in is also readily implemented in Python and shown in the following code block. The initialization of the environment randomly sets the mean rewards of each of the k bandits. Each time a bandit is played,

the `step()` method randomly determines a reward from a standard normal distribution with the mean of the k-th bandit.

```
class k_bandit_env:
    def __init__(self, k):
        self.k = k
        self.mean_rewards = []

        for i in range(self.k):
            self.mean_rewards.append(random.normalvariate(0, 1))

    def step(self, action):
        mean = self.mean_rewards[action]
        reward = random.normalvariate(mean, 1)
        return reward
```

Figure 5 shows a comparison of learning behaviour for agents with different parameters ϵ. The horizontal axis shows the index of 1000 steps in the environment and the vertical axis shows the reward received at each step (mean over 1000 runs of the algorithm). The agent with $\epsilon = 0$ (blue line) shows the worst learning behaviour. That agent only exploits and never explores. In other words, it never finds any better or more valuable actions to take, once it has found a good action. In contrast, the agent with $\epsilon = 0.01$ (red line) learns slower in the beginning as it explores more but after 1000 steps has achieved a better mean reward. Finally, the purple line represents an agent with $\epsilon = 0$ but whose action value function has been "optimistically" initialized with values of 5 instead of 0, that is, above the expected reward. This prevents it from assuming the first good action is the best one, which leads to high initial learning. However, with an ϵ of 0, that agent is not capable of further learning later in the sequence of actions taken.

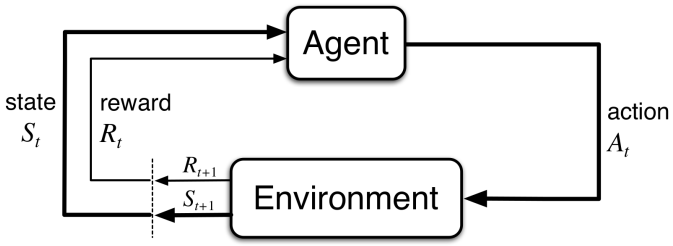
3 Markov Decision Processes and Dynamic Programming

This section introduces the concept of *Markov decision processes*, that is, a sequence of decisions or actions and states that has the Markov property: the reward and next state depend only on the current state and action, not on the state history.

One can think of RL learning as a Markov decision process. Figure 6 shows the RL agent and the environment it is situated in. The environment is at time t in state S_t. The agent takes action A_t in the environment and receives reward R_t. As a result, the environment's state changes to the new state S_{t+1}. This leads to the concept of a *trajectory* as a sequence of states, actions, and rewards:

$$S_0, A_0, R_1, S_1, A_1, R_2, \ldots$$

Figure 5: Learning performance for k-armed bandit agents for different ϵ and initial action-values



Source: SB Figure 3.1

Figure 6: RL agent and environment

3.1 Definitions

The behaviour of the environment is stochastic and can be described through the "p-function" which expresses the state transition and reward probabilities. This is known as the *dynamics* of the environment:

$$p(s', r|s, a) = \Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\} \tag{3}$$

The *return* is formally defined as the possibly discounted sum of future rewards:

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned} \tag{4}$$

The *state value function* of state s under a policy π is defined as the expected value of the return in that state where $\mathbb{E}_\pi$ is the expection when acting according to policy π:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\ &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s\right] \end{aligned} \tag{5}$$

Similarly, the *action value function* of state s and action a for policy π is defined as the expected return of being in state s and taking action a:

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \\ &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s, A_t = a\right] \end{aligned} \tag{6}$$

3.2 Bellman Equations and Iterative Policy Evaluation

Starting with Equation 5 and substituting Equation 4 into it yields:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] \end{aligned}$$

An expectation is the sum of values weighted by their probability. Summing over all possible combinations of action a, next state s' and rewards r and using the dynamics of the environment (Equation 3) and the stochastic policy $\pi(a|s)$ for probabilities of taking action a in state s, then yields:

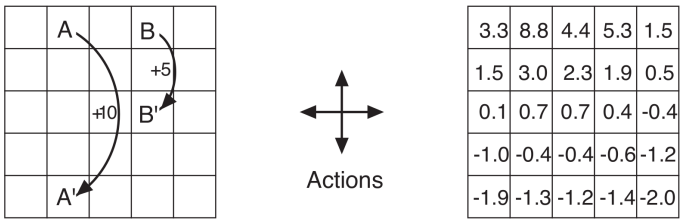
$$v_\pi(s) = \sum_a \sum_{s'} \sum_r \left[r + \gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s']\right] p(s', r|s, a)\pi(a|s)$$

Note that the expectation of G_{t+1} is not replaced by this sum and remains an expectation. Rearringing this slightly:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} \sum_r p(s', r|s, a) \left[r + \gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s']\right]$$

Recognizing that the expectation in the final term on the right is just the state value function (Equation 5) for state s' yields:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_\pi(s')\right] \quad \text{for all } s \in \mathcal{S} \tag{7}$$



Source: SB Figure 3.2

Figure 7: Gridworld example and optimal state value function

Equation 7 is called the *Bellman equation* for the state value function. A similar equation can be derived for the action value function, beginning with Equation 6 and following the same steps.

To illustrate the concept of the state value function, consider the gridworld example in Figure 7. An agent is placed on the grid in the left panel of the figure. The agent can take four possible actions; it can move up, down, left or right. Moving off the grid, for example taking action "up" when in the top row, results in a reward of -1 and the state is unchanged. All other actions yield a reward of 0 with state changes as indicated by the action, except when states A or B are reached. When reaching state A, the agent receives a reward of $+10$ and the next state is A'. When reaching state B, the agent receives a reward of $+5$ and the next state is B'.

The policy π in this example is a random policy; independent of its state, the agent takes each action with equal probability. The discount rate is set at $\gamma = 0.9$, favouring immediate rewards over future ones.

The state values for this problem under the random policy are shown in the right panel of Figure 7. It is clear that being in states A and B is most valuable, as the immediate reward is great. However, the state values are not $+10$ or $+5$ because the agent is likely to incur some negative future rewards. In particular, the state values of the edge states are low or negative because of the probability of falling off the world and incurring a reward of -1.

The Bellman equation (Equation 7 and its equivalent for the action value function) express the fact that the value of a state (or state-action pair) is a function of the values of all other states (or state-action pairs). This suggests an intuitive way to compute the state values iteratively. Beginning with random values for each state, calculate updated values for all states using Equation 7. Then, consider the updated values as the current values, and calculate updated values based on these[2]. Iterate like this until the values do not change any more. It can be proven that this procedure converges to the correct solution and terminates.

This process is called *iterative policy evaluation*. Figure 8 shows this in pseudocode

[2]In fact, it is not even necessary to wait until all states have been iterated over and updated before using updated state values as current ones.

Loop:

$\Delta \leftarrow 0$

Loop for each $s \in \mathcal{S}$:

$v \leftarrow V(s)$

$V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$

$\Delta \leftarrow \max(\Delta, |v - V(s)|)$

until $\Delta < \theta$

Figure 8: Iterative Policy Evaluation (Source: SB)

and the following Python code block shows the straightforward implementation, beginning with initial values of 0 for each state.

```
# Define actions for gridworld
A = list(range(0,4))
# Initialize value function V
V = dict()
for state in States:
    V[state] = 0
# Initialize random policy pi
pi = dict()
for state in States:
    pi[state] = random.choice(A)

def evaluate_policy():
    while True:
        Delta = 0
        for s in States:
            v = V[s]
            V[s] = exp_reward(s, pi[s])
            Delta = max(Delta, abs(v - V[s]))
        print(Delta)
        if Delta < theta:
            break
```

3.3 Bellman Optimality and Iterative Policy Improvement

Maximizing the state value function v or action value function q is finding an optimal policy π, that is, that policy that when following it yields the maximum state value or action value:

$$v_*(s) = \max_\pi v_\pi(s)$$
$$q_*(s, a) = \max_\pi q_\pi(s, a)$$

Intuitively, the value of a state under an optimal policy π_* is equal to the expected return for the the best action from that state:

$$v_*(s) = \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s, a)$$

Substituting the definition of the action value function (Equation 6):

$$= \max_a \mathbb{E}_{\pi_*}[G_t | S_t = s, A_t = a]$$

Substituting the recursive definition of the return (Equation 4):

$$= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a]$$

Noting that the expected future return is just the value of the next state, that is, using Equation 5:

$$= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a]$$

The expectation is the sum over all following states s' and rewards r weighted by their probabilities. Using the dynamics of the environment (Equation 3) that describe the probabilities yields:

$$v_*(s) = \max_a \sum_{s', r} p(s', r | s, a)[r + \gamma v_*(s')]$$

Similarly, the action value under an optimal policy π^* can be derived as:

$$\begin{aligned} q_*(s, a) &= \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') | S_t = s, A_t = a\right] \\ &= \sum_{s', r} p(s', r | s, a)[r + \gamma \max_{a'} q_*(s', a')] \end{aligned}$$

These final expressions are known as the *Bellman optimality* equations for the state and action value functions.

A simple, intuitive way of finding the optimal policy is to find the optimal state value function. The optimal policy is then to take that action that will yield the best following state. However, changing the policy will change the state value function. This intuitive procedure shows that state value function calculation and policy updates should happen alternatingly, until the policy no longer changes.

Because the state value computation was already demonstrated using iterative policy evaluation above, the procedure for *iterative policy improvement* is simple, expressed in Figure 9. The following Python code block illustrates the straightforward implementation:

Loop:

$stable \leftarrow true$

For each $s \in \mathcal{S}$:

$old\_action \leftarrow \pi(s)$

$\pi(s) \leftarrow \text{argmax}_a \sum_{s',r} p(s', r|s, a)[r + \gamma V(s')]$

If $old\_action \neq \pi(s)$ then $stable \leftarrow false$

If $stable$ then

return $V \approx v_*$ and $\pi \approx \pi_*$

else

go to policy evaluation

Figure 9: Iterative Policy Improvement (Source: SB)

```
def improve_policy():
    stable = True
    for s in States:
        old_action = pi[s]
        max_r = -math.inf
        max_a = None
        for action in Actions:
            r = exp_reward(s, action)
            if r > max_r:
                max_r = r
                max_a = action
        pi[s] = max_a
        if old_action != pi[s]:
            stable = False
    return stable
```

Putting both functions, `evaluate_policy()` and `improve_policy()`, together in an iteration will yield the optimal policy:

```
stable = False
while not stable:
    evaluate_policy()
    stable = improve_policy()

print("Optimal Policy:")
print(pi)
```

The procedures of *iterative policy evaluation* and *iterative policy improvement* are an example of the more general approach to optimization called *dynamic programming*.

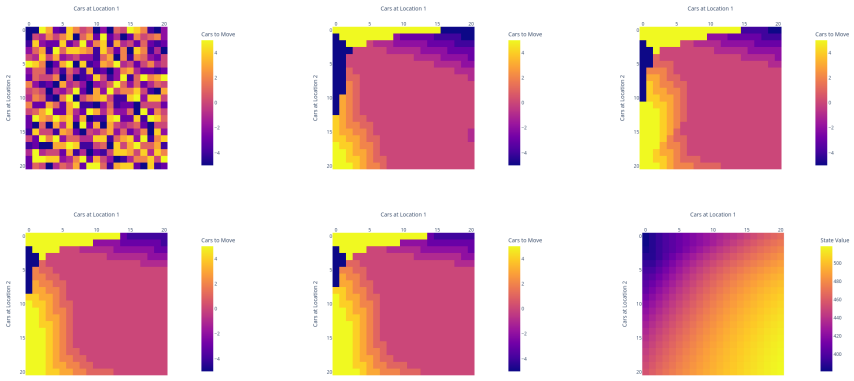


Figure 10: Iterative policy improvement example (exercise 4.2 in SB). Policies and final state value function.

Consider the example of "Jack's Car Rental" (example 4.2 in SB). Jack rents cars at 2 locations. Each location can store 20 cars. The number of daily rental requests and rental returns are Poisson distributed. Jack can move a maximum of 5 cars between the two locations overnight. Each move incurs a reward (cost) of -2 and each satisfied rental request receives a reward of $+10$. Jack is looking for the optimal policy that specifies how many cars to move from location 1 to location 2 every night.

The states in this problem are defined as the number of cars in location 1 and 2, for a total of $20 \times 20 = 400$ possible states and 2 actions, defined in the following Python code block[3]:

```
States = []
for cars1 in range(21):
    for cars2 in range(21):
        States.append((cars1, cars2))

Actions = range(-5, 5+1)
```

Figure 10 shows the policies after each iteration of the iterative policy improvement algorithm in Figure 9, beginning with the random policy in the top left panel of Figure 10. Positive values for the policy indicate cars to move from location 2 to location 1, negative values indicate cars to move from location 1 to location 2. The policy improvement converges to the optimal policy after 4 iterations, with the final state value function shown in the bottom right panel of Figure 10.

[3]Complete implementation available at https://github.com/jevermann/busi4720-rl/blob/main/jacks.py

Input: a policy π to be evaluated

Initialize:

$V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$

$Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):

Generate an episode following $\pi : S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Unless S_t appears in $S_0, S_1, \ldots, S_{t-1}$:

Append G to $Returns(S_t)$

$V(S_t) \leftarrow \text{average}(Returns(S_t))$

Figure 11: First-visit MC prediction (Source: SB)

4 Monte Carlo (MC) Learning

The previous section illustrates how optimal policies and their state value and action value function can be computed *under the assumption that the dynamics of the environment, that is Equation 3, are known*. In practice, this is not the case — the $p(s', r|s, a)$ are unknown, there is no model of the environment. The RL agent must learn V and Q from *experience*, that is, it must act in an environment and generate trajectories (sequences of states, actions, and rewards).

This section assumes *episodic tasks*, that is, problems with a terminal state, a finite trajectory, and finite returns. The agent acts in an environment according to a policy π until it arrives in a terminal state and the episode ends. At that point, the entire sequence of states, actions, and rewards is known and state value functions can be estimated or approximated. Recall that the value of a state is the expected return, that is the expected sum of discounted future rewards. State values can then be approximated as the average of the returns in that state.

A problem arises because the agent can be in the same state multiple times before the episode terminates. Different assumptions can be made. For example, one can assume that it is the first visit of a state that is most influential in determining the outcome of the episode and therefore the state value should be updated with the return at the time of the first visit of a state. Alternatively, one could assume that the last visit of a state is most important, and the value function is updated with the returns at the last time that the state is visited. Yet another alternative is to update the value function for all visits of a state. Figure 11 shows the pseudocode for this process when the state value function is updated only for the first visit of a state, known as *First-Visit Monte Carlo Prediction*.

Initialize for all $s \in \mathcal{S}, a \in \mathcal{A}$

$\pi(s) \in \mathcal{A}(s)$ (arbitrarily)

$Q(s,a) \in \mathbb{R}$ (arbitrarily)

$Returns(s,a) \leftarrow$ empty list

Loop forever (for each episode):

Choose $S_0 \in \mathcal{S}, A_0 \in \mathcal{A}(S_0)$ randomly

Generate an episode following $\pi : S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0$

Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1, \ldots, S_{t-1}, A_{t-1}$:

Append G to $Returns(S_t, A_t)$

$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$

$\pi(S_t) \leftarrow \underset{a}{\text{argmax}}\, Q(S_t, a)$

Figure 12: First visit MC control with exploring starts (Source: SB)

Note the computation of the return G backwards from the end of the episode. The line "unless S_t appears in $S_o, S_1, \ldots, S_{t-1}$" ensures the first-visit property and the final two lines approximate the state value by the average of the returns for that state.

Monte Carlo Control

While Monte Carlo prediction is useful in approximating the state value function, obtaining the optimal policy is done with *Monte Carlo control*. Instead of approximating $V(S)$ from the returns of an episode, MC control approximates $Q(S, A)$. Figure 12 shows the first-visit MC control algorithm as pseudocode. It is very similar to the first-visit MC prediction algorithm in Figure 11. Consider the final three lines: The main change is that returns are assigned not to states, but to pairs of states and actions. The action value function for a state-action pair is approximated as the average over all the returns assigned to it. The optimal policy in some state is that action for wich the action value is maximal.

There is one additional difference between Figures 11 and 12 in that the initial states and actions of each episode are chosen randomly. This is necessary because the policy π in Figure 12 is deterministic and greedy. Without any random influence, there would be no exploration. Forcing episodes to begin with random states and actions is called *exploring starts* and it ensures that every state-action pair is visited at least once (assuming sufficiently many episodes are generated).

As an example of an episodic RL problem that can be usefully learned with MC meth-

ods, consider the game of Blackjack (example 5.3 in SB)[4]. In this game, cards have values A, 2, 3, . . . , 10 where A is the ace and 10 includes face cards. An ace can count as 1 point or 11 points. An ace that is counted as 11 is called a "usable ace". The dealer's initial card is shown. The player can take two possible actions: take another card ("hit") or do not take a card ("stick"). Once the player sticks, the dealer takes cards. The dealer sticks on a sum of 17 or more. When the player's or the dealer's sum of cards is over 21 they are "bust", that is, they lose the game. The dealer's policy is deterministic, they stick on 17 points or more.

For Blackjack, the states are defined as a combination of the player's current sum of cards, the initial card the dealer is showing, and whether the player has a usable ace (one that can be converted from an 11 to a 1). Actions are to hit or stick. The following Python code block shows how this can be readily implemented:

```
gamma = 1.0
# Define states
States = []
for ace in [0,1]:
    for dealer_showing in range(1,11):
        for hand_sum in range(12, 22):
            States.append((ace,dealer_showing,hand_sum))

# Define actions
Actions = (0, 1)
```

Initially, the policy is a random policy, action value functions are 0 for all state-action pairs, and the list of returns for each state-action pair is empty:

```
# Initialize policy
pi = dict()
for s in States:
    pi[s] = random.randint(0,1)

# Initialize action value function
Q = dict()
for s in States:
    for a in Actions:
        Q[(s, a)] = 0

# Initialize returns
Returns = dict()
for s in States:
    for a in Actions:
        Returns[(s, a)] = []
```

The following code block shows the generation of an episode under policy `pi` from initial state `s0` and with initial action `a0`. The function `step()` calls the environment.

[4] A complete implementation is available at https://github.com/jevermann/busi4720-rl/blob/main/blackjack_es.py

It includes the dealer drawing cards after the player uses the ”stick” action in a state.

```
def generate_episode(pi, s0, a0):
    terminal = False
    s = s0
    a = a0
    states = [s0]
    actions = [a0]
    rewards = [math.nan]
    while terminal is False:
        sprime, r, terminal = step(s, a)
        rewards.append(r)
        if not terminal:
            aprime = pi[sprime]
            states.append(sprime)
            actions.append(aprime)
            s = sprime
            a = aprime

    return states, actions, rewards, len(rewards)
```

The core of MC control is implemented in the following Python code block, which is very much analogous to the pseudocode in Figure 12. Every episode begins in a random state and with a random action. An episode is generated and the sequence of states, actions, returns, and the length of the episode `T` are returned. Returns are computed from the end of the episode and assigned to the first-visit of a state-action pair. The policy is updated based on the new approximate value of the Q function.

```
# Learn the Q function
for e in range(0, 1000000+1):
    s0 = random.choice(States)
    pi0 = random.choice(Actions)
    S, A, R, T = generate_episode(pi, s0, pi0)
    G = 0
    for t in reversed(range(0, T-1)):
        G = gamma*G + R[t+1]
        if (t == 0) or ((S[t], A[t]) not in zip(S[0:t-1], A[0:t-1])):
            Returns[(S[t], A[t])].append(G)
            Q[(S[t], A[t])] = mean(Returns[(S[t], A[t])])
            if Q[(S[t], 1)] > Q[(S[t], 0)]:
                pi[S[t]] = 1
            else:
                pi[S[t]] = 0
```

Figure 13 shows the resulting policy (left panels) and state value function (right panels) after 1,000,000 learning episodes. The top two plots in Figure 13 are with a usable ace, the bottom two plots without a usable ace. An action of 1 means to hit, and action 0 is to stand.

For the game of Blackjack, exploring starts with a deterministic policy is a good way to ensure exploration. However, exploring starts are not always possible or realistic.

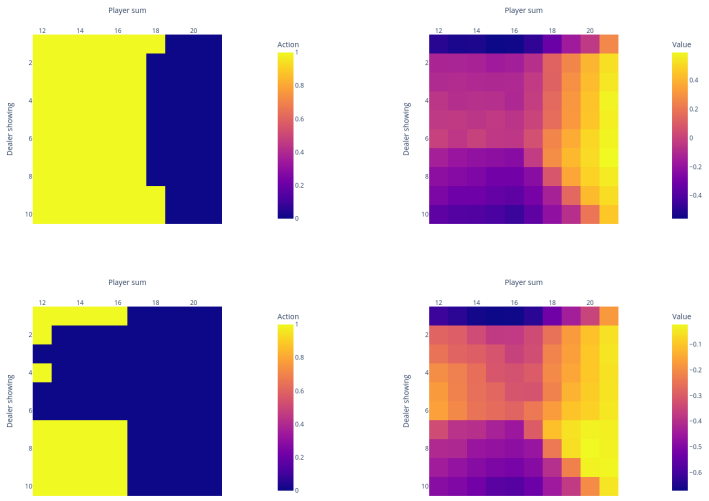


Figure 13: Policies and state value function for the Blackjack example after 1,000,000 episodes. Usable ace on top, no usable ace at bottom

Instead of using a greedy policy, in those cases an ϵ-soft policy can be used to ensure exploration. The policy π now represents not the action to be taken (greedily, deterministically) but the probability with which each action should be chosen. Consequently, π is updated with ϵ-soft probabilities, as shown in the following Python code fragment[5]:

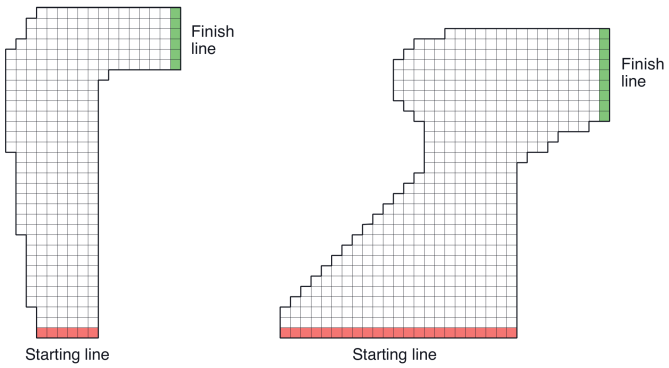
```
Q[(S[t], A[t])] = mean(Returns[(S[t], A[t])])
# Optimal policy (for two actions)
A_star = 1 if Q[(S[t], 1)] > Q[(S[t], 0)] else 0

for a in Actions:
    if a == A_star:
        pi[(S[t],a)] = 1-epsilon+epsilon/len(Actions)
    else:
        pi[(S[t],a)] = epsilon/len(Actions)
```

An example where exploring starts are not realistic is the Racetrack problem (exercise 5.12 in SB). Cars have to follow the right curve of a racetrack, from the starting line to the finish line, as shown in two examples in Figure 14.

The states are defined by the car's position and velocity on the race track, each in two dimensions, horizontally and vertically. The possible actions are to accelerate, coast, or brake in each direction. The rewards are -1 for each step taken and +1 for crossing the finish line, when the episode terminates. When a car moves off the racetrack, it is reset

[5] An implementation of the Blackjack problem with an ϵ-soft policy is available at https://github.com/jevermann/busi4720-rl/blob/main/blackjack_eps.py.



Source: SB Figure 5.5

Figure 14: Racetrack example

to a random position on the starting line and the episode continues. The environment is stochastic: With a small probability, the actions of the agents are ignored, that is, they have no effect.

The following Python code block defines the actions, the action value function and the ϵ-soft policy that samples from the probability distribution over actions in a state[6] .

```
Actions = []
for y in range(-1, 2):
    for x in range(-1, 2):
        Actions.append((y,x))

Q = dict()
def getQ(s, a):
    if (s, a) not in Q:
        return 0
    else:
        return Q[(s, a)]

pi = dict()
def get_action(s):
    weights = []
    for a in Actions:
        if (s, a) in pi:
            weights.append(pi[(s, a)])
    return random.choices(Actions, weights=weights)[0]
```

The following two Python functions manage the returns for each state-action pair:

[6]Complete implementation is available at https://github.com/jevermann/busi4720-rl/blob/main/racetrack.py.

```
Returns = dict()
def getReturns(s, a):
    if (s, a) not in Returns:
        return []
    else:
        return Returns[(s, a)]
def appendReturn(s, a, r):
    if (s, a) not in Returns:
        Returns[(s, a)] = [r]
    else:
        Returns[(s, a)].append(r)
```

The following Python code block represents the core MC control algorithm. Note that the policy `pi` now represents the probabilities of taking an action in a state and is updated accordingly. The remainder of the code is largely unchanged from the Blackjack example, except that the Racetrack example does not need to use exploring starts, that is, random starting states and actions.

```
for e in range(0, 10000+1):
    S, A, R, T = env.generate_episode()
    G = 0
    for t in reversed(range(0, T-1)):
        G = gamma*G + R[t+1]
        if (t == 0) or ((S[t], A[t]) not in zip(S[0:t-1], A[0:t-1])):
            appendReturn(S[t], A[t], G)
            Q[(S[t],A[t])] = mean(getReturns(S[t],A[t]))
            A_star = argmaxQ(S[t])
            for a in Actions:
                if a == A_star:
                    pi[(S[t],a)]=1-eps+eps/len(Actions)
                else:
                    pi[(S[t],a)]=eps/len(Actions)
```

Figure 15 shows visualizations of the trajectory of a car (green line) on the racetrack after 0, 100, 200, and 10,000 episodes. It is clear from these trajectories that the number of steps is reduced as training progresses.

5 Off-Policy MC Learning

In the MC methods described above, the policy used to generate behaviour ("*behaviour policy*"), and the policy that is learned ("*target policy*") are the same. However, the need to keep exploring, that is, to behave sub-optimally, when a better, greedy policy could be available, means that the agent's performance is reduced. *Off-policy learning* is motivated by the need for efficiency and flexibility in learning optimal policies. The main motivation is the ability to learn about the optimal policy independently of the agent's actions. This is useful in environments where exploring all actions is either potentially damaging or costly.

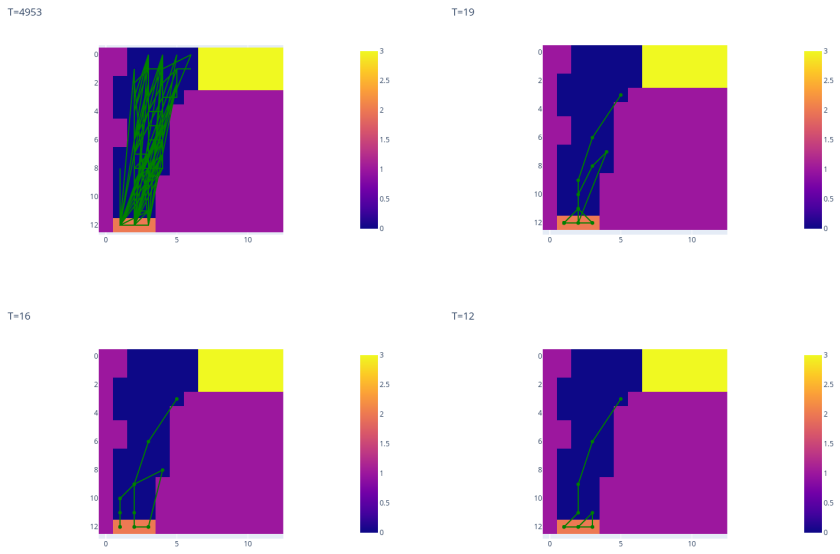


Figure 15: Racetrack trajectory after 0, 100, 200, and 10000 learning episodes:

In on-policy methods, the agent learns from the actions it evaluates and takes, meaning the behaviour policy is the same as the target policy. In contrast, off-policy methods allow the agent to learn a policy different from the one it follows. This is beneficial as it permits the use of historical data from other policies, or data generated from exploratory or less optimal policies, to improve a potentially different and more optimal target policy.

One important requirement in off-policy learning is that the behaviour policy must *cover* the target policy. That is, all behaviour possible under the target policy must be (eventually) generated by the behaviour policy. This requirement is intuitive, as the target policy cannot learn about behaviour, that is, the effects of actions in states, that is never encountered in a generated episode.

Figure 16 shows pseudocode for off-policy MC control. In this learning method, the behaviour policy b is typically an ϵ-soft policy, as presented above, that allows exploratory behaviour. The target policy π is a deterministic, greedy policy, as can be seen in the third-to-last line of Figure 16.

The following Python code blocks show how off-policy MC control can be implemented. The first block defines the two policies, the ϵ-soft behaviour policy `b` and the greedy deterministic policy `pi`.

Initialize for all $s \in \mathcal{S}, a \in \mathcal{A}(s)$:

$Q(s,a) \in \mathbb{R}$ (arbitrarily)

$C(s,a) \leftarrow 0$

$\pi(s) \leftarrow \text{argmax}_a \, Q(s,a)$

Loop forever (for each episode):

Generate an episode following $b : S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$

$G \leftarrow 0; W \leftarrow 1$

Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:

$G \leftarrow \gamma G + R_{t+1}$

$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} \left[G - Q(S_t, A_t)\right]$$

$$\pi(S_t) \leftarrow \underset{a}{\text{argmax}} \, Q(S_t, a)$$

If $A_t \neq \pi(S_t)$ then proceed to next episode

$W \leftarrow W / b(A_t | S_t)$

Figure 16: Off-Policy MC Control (Source: SB)

```
def b(s):
    weights = []
    for a in Actions:
        if (s, a) in Q:
            weights.append(math.exp(Q[(s, a)]))
        else:
            weights.append(0)
    if len(weights) == 0 or sum(weights) == 0:
        return random.choice(Actions)
    else:
        return random.choices(Actions, weights)[0]

def pi(s):
    a = argmaxQ(s)
    if a is None:
        return random.choice(Actions)
    else:
        return a
```

Note that the last line of Figure 16 makes reference to the probabilities of action A_t in state S_t under the behaviour policy b. This probability is computed by the following Python code block:

```
def bprob(a, s):
    if (s, a) not in Q:
        return 1
    weights = []
    for aa in Actions:
        if (s, aa) in Q:
            weights.append(math.exp(Q[(s, aa)]))
    if len(weights) == 0 or sum(weights) == 0:
        return 1
    else:
        return math.exp(Q[(s, a)]) / sum(weights)
```

With these definitions, the learning algorithm is a straightforward implementation of the pseudocode in Figure 16[7]:

```
for e in range(0, 10000+1):
    S, A, R, T = env.generate_episode_b()
    G = 0
    W = 1
    for t in reversed(range(0, T-1)):
        G = gamma*G + R[t+1]
        C[(S[t], A[t])] = getC(S[t], A[t]) + W
        Q[(S[t], A[t])] = getQ(S[t], A[t]) + \
              W/getC(S[t], A[t]) * (G-getQ(S[t],A[t]))
        if A[t] != pi(S[t]):
            break
        else:
            W = W * 1/bprob(A[t], S[t])
```

6 Temporal-Difference (TD) Learning

Temporal Difference (TD) learning is an RL approach that combines ideas from both Monte Carlo (MC) methods and dynamic programming. The primary motivation for temporal difference learning stems from its ability to learn predictions based on other learned predictions, a concept referred to as bootstrapping. Unlike Monte Carlo methods, which wait until the completion of an episode to update the value estimates based on actual returns, TD learning updates estimates based on estimate returns, thus not requiring the episode to terminate before updates can be made. This allows TD learning to make more frequent updates, which can accelerate learning, and to be applied in continuing (non-episodic) environments.

Recall that in MC control, the updates to the action value function use the actual return G_t at time t as the target, which can only be computed at the end of an episode:

[7] Complete implementation using the Racetrack example is available at https://github.com/jevermann/busi4720-rl/blob/main/racetrack_off_policy.py.

Initialize $Q(s, a)$ for all $s \in \mathcal{S}^+$, arbitrarily
Loop for each episode:
  Initialize S
  Choose A from S using policy derived from Q
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose A' from S' using policy derived from Q
    $Q(S, A) \leftarrow Q(S, A) + \alpha \left[R + \gamma Q(S', A') - Q(S, A)\right]$
    $S \leftarrow S'; A \leftarrow A'$
  until S is terminal

Figure 17: TD-control with the SARSA method (Source: SB)

$$Q(S_t, a) \leftarrow Q(S_t, a) + \alpha \left[G_t - Q(S_t, a)\right]$$

Substituting the recursive definition of the return (Equation 4) yields:

$$Q(S_t, a) \leftarrow Q(S_t, a) + \alpha \left[R_{t+1} + \gamma G_{t+1} - Q(S_t, a)\right]$$

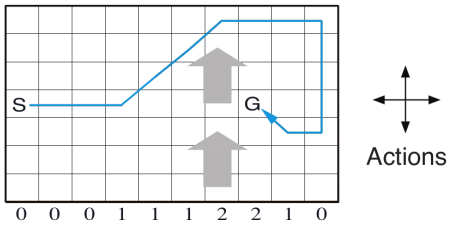
Recognizing that the expected value of G_{t+1} is approximated by the Q value of the optimal action in the next state (Equation 6) yields the TD update:

$$Q(S_t, a) \leftarrow Q(S_t, a) + \alpha \left[R_{t+1} + \gamma Q(S_{t+1}, a^*_{t+1}) - Q(S_t, a)\right] \tag{8}$$

In other words, there is no need to wait until the actual return G_t is known at the end of an episode, as TD learning uses the current approximation or estimate of G_t as the update target. This idea of using estimates to update or compute better estimates is called *bootstrapping*.

Figure 17 shows the pseudocode of a method called "*SARSA*", so-called because it uses information of the current state S, current action A, reward R, next state S' and next action A' in its update. SARSA uses the update function defined in Equation 8. SARSA is an on-policy method, typically using an ϵ-greedy policy based on Q to generate behaviour and ensure exploration.

To illustrate the use of SARSA, consider the "Windyworld" example environment (example 6.5 in SB), shown in Figure 18. An agent has to move from start state S to terminal goal state G in this gridworld. However, there is a wind on some columns of the grid that pushes the agent towards the top (indicated below the columns in Figure 18). Rewards are -1 for every step until termination, so that the agent is encouraged to find the path requiring the least number of actions. There are no penalties for moving off the world (the action is simply ignored) and the problem is not discounted ($\gamma = 1$).



Source: SB Chapter 6

Figure 18: Windyworld example

The following python code blocks define the states, actions, and an ϵ-greedy policy `pi`:

```
# Define states
States = []
for i in range(nrow):
    for j in range(ncol):
        States.append((i, j))

# Define actions
Actions = range(0, 4)
# Initialize Q
Q = dict()
for s in States:
    for a in Actions:
        Q[(s, a)] = random.random()

# Define pi
def pi(s):
    if random.random() < epsilon:
        return random.choice(Actions)
    else:
        return argmaxQ(s)
```

With these definitions, the implementation of SARSA is straightforward from the pseudocode in Figure 17[8] and shown in the following Python code block.

[8] A complete implementation of the Windyworld example is available at https://github.com/jevermann/busi4720-rl/blob/main/windyworld_sarsa.py.

```
for e in range(0, 100):
    terminal = False
    S = windy.reset()
    A = pi(S)
    step = 0
    while terminal is False:
        Sprime, R, terminal = windy.step(A)
        Aprime = pi(Sprime)
        Q[(S,A)] = Q[(S,A)] + alpha*(R + \
            gamma * Q[(Sprime, Aprime)] - Q[(S, A)])
        S = Sprime
        A = Aprime
```

In summary, Temporal Difference learning and Monte Carlo methods differ primarily in when and how the updates to value estimates are made:

- *Update Timing:* TD learning updates values at every time step using current estimates, which means it can start learning from incomplete sequences, making it suitable for non-episodic environments. Monte Carlo methods update only at the end of each episode, using the total accumulated return from the episode.
- *Sampling vs. Bootstrapping:* Monte Carlo methods rely solely on actual returns (full sampling), and do not bootstrap. In contrast, TD methods bootstrap, using existing value estimates to update new estimates.
- *Convergence Properties:* Due to its incremental nature and frequent updates, TD learning can converge faster in practical applications than Monte Carlo methods, which require longer trajectories and may suffer from higher variance in their estimates due to the complete reliance on actual returns.

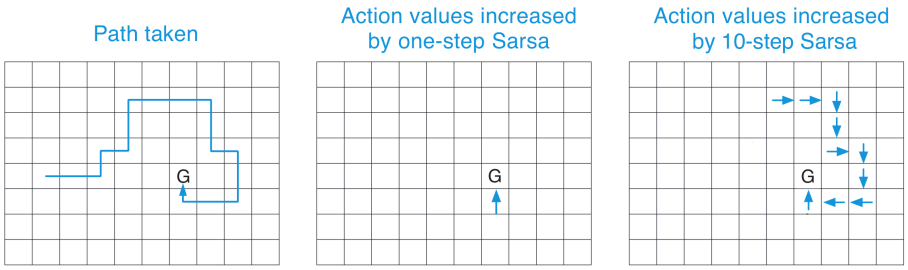
Generalizing TD Learning to N Steps

In developing the SARSA method (Equation 8), the recursive definition of the return (Equation 4) and the definition of the action value as the expected return (Equation 6) were applied once. Hence the SARSA update target $R_{t+1} + Q(S_{t+1}, a_{t+1})$ "looks ahead" by one step to the next reward $R_{t=1}$ and then approximates the remaining portion of G_{t+1} by $Q(S_{t+1}, a_{t+1})$. This is therefore called the "1-step" target, and the update error δ is called the "1-step error":

$$\delta_{TD1} = R_{t+1} + Q(S_{t+1}, a_{t+1}) - Q(S_t, a)$$

This can be extended by applying Equations 4 and 6 a second time. This yields the "2-step error" that looks ahead at the next two rewards, and then approximates the remainder by $Q(S_{t+2}, a_{t+2})$:

$$\delta_{TD2} = R_{t+1} + \gamma R_{t+2} + \gamma^2 Q(S_{t+2}, a_{t+2}) - Q(S_t, a)$$



Source: SB Figure 7.4

Figure 19: SARSA versus n-Step TD Learning (n-step SARSA)

Calculating the 2-step error requires knowledge of two actual rewards, so the 2-step update can be performed only after two steps. Or, viewed from a different perspective, the 2-step update updates not only the value of the most recent state-action pair but the values of the two most recent state-action pairs.

This can be generalized by applying Equations 4 and 6 n-times, yielding the ”n-Step TD error”:

$$\delta_{TDn} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n Q(S_{t+n}, a_{t+1}) - Q(S_t, a)$$

The n-step update can only be performend after n steps have passed so that n actual rewards are available, with the remainder of G_{t+1} being approximated. As an alternative interpretation, the n-step update changes the values of the last n state-action pairs in the trajectory. Figure 19 shows an illustration of this for 10-step SARSA.

7 Off-Policy TD Learning

Temporal Difference learning can be combined with off-policy learning as well. A popular method for this is Q-learning. Q-learning updates the action values in a greedy way, that is, of a greedy policy, regardless of the action taken by the policy being followed (the behavior policy). The differences between SARSA and Q-learning are minor. Importantly, it is not necessary to explicitly encode the behaviour and target policies.

Consider the following aspects of SARSA. The next action A' that is carried out is determined using a (behaviour) policy based on Q and Q is being learned (updated) based on the actually taken action A' (target policy). This makes SARSA an on-policy method.

SARSA (on-policy):

Take action A, observe R, S'

Choose A' from S' using policy derived from Q

$Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma Q(S',A') - Q(S,A)\right]$

$S \leftarrow S'; A \leftarrow A'$

A minor change is sufficient to change on-policy SARSA to off-policy Q-learning, shown in the box below. Here, the next action A that is carried out is also determined using a (behaviour) policy based on Q, but Q is updated not based on the actually taken action A, but based on the optimal action A', that is, the action with the maximum Q value in the following state S'. This difference means that the policy that governs behaviour ("behaviour policy") is different than the policy that is updated or learned ("target policy"), making Q-learning an off-policy method.

Q-learning (off-policy):

Choose A from S using policy derived from Q

Take action A, observe R, S'

$Q(S,A) \leftarrow Q(S,A) + \alpha\left[R + \gamma \max_a Q(S',A') - Q(S,A)\right]$

$S \leftarrow S'$

The corresponding Python implementation for the Windyworld example is also straightforward from the above box[9]:

```
for e in range(0, 1000):
    terminal = False
    S = windy.reset()
    step = 0
    while terminal is False:
        A = pi(S)
        Sprime, R, terminal = windy.step(A)
        Q[(S,A)] = Q[(S,A)] + alpha*(R + \
            gamma * maxQ(Sprime) - Q[(S, A)])
        S = Sprime
```

Figure 20 shows the difference in learning behaviour between SARSA and Q-learning on the Windyworld problem. It plots the the number of required actions to achieving the goal against the number of episodes generated; off-policy Q-learning shows faster learning than SARSA.

[9] A complete implementation is available at https://github.com/jevermann/busi4720-rl/blob/main/windyworld_q_learning.py.

SARSA **Q-Learning**

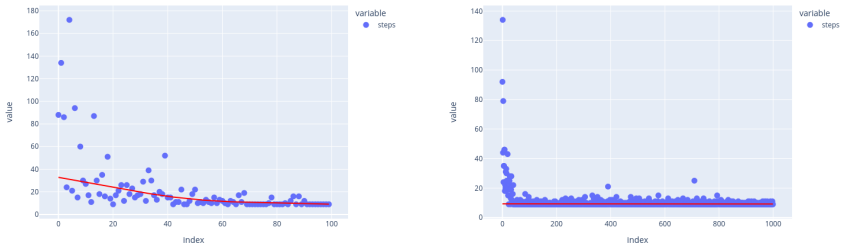


Figure 20: SARSA and Q-Learning Results on Windyworld

8 Review Questions

Introduction

1. What is reinforcement learning and how does it differ from other types of machine learning?
2. What is meant by the term ”model” in the context of reinforcement learning?
3. Discuss the challenges associated with the stochastic nature of the environment in reinforcement learning.
4. Define the following terms and explain their significance in reinforcement learning:
   - Policy (π)
   - Reward (R)
   - Return (G)
   - State value function (v)
   - Action value function (q)
   - Model (p)
5. Explain the difference between a deterministic policy and a stochastic policy. Provide an example of each from any real-world scenario.
6. Describe the general process of updating the value function. Include the update rule equation and explain each component.
7. Explain the significance of the action value function in the reinforcement learning framework. How does it differ from the state value function in terms of utility and information provided to the agent?
8. Discuss how reinforcement learning can be applied to customer interaction management and financial portfolio management. What are the potential benefits and challenges?
9. Consider the manufacturing process optimization application of reinforcement learning. Describe how an RL algorithm could continuously improve the process and what metrics it might focus on.
10. Discuss the implications of the exploration-exploitation trade-off in a non-gaming business scenario. How might a company balance these two aspects effectively?

11. Reflect on the potential impacts of reinforcement learning on customer satisfaction in a service-oriented business. What are the risks and rewards?
12. What role does the discount factor γ play in calculating the return in a reinforcement learning problem? Why might one use a smaller or larger value of γ?

Questions on K-Armed Bandits

13. Why is the k-armed bandit problem considered to be a "stateless" problem in reinforcement learning?
14. Describe the action value function $Q_t(a)$ in the k-armed bandit problem. What does it represent?
15. Explain the concept of the ϵ-greedy policy. How does it balance the exploration-exploitation trade-off?
16. Consider the incremental update formula used in the k-armed bandit problem and explain each term: $Q_{t+1}(a) = Q_t(a) + \frac{1}{t}[R_t(a) - Q_t(a)]$. How does this formula ensure that the estimate becomes more accurate over time?
17. What are the implications of setting a higher or lower ϵ value in terms of long-term gains vs. short-term exploration?
18. Discuss the rationale behind using an "optimistic" initial value for the action-value estimates. How does this approach influence the agent's behavior?

Questions on Markov Decision Processes and Dynamic Programming

19. Define a Markov Decision Process (MDP) and explain the Markov property in this context.
20. Draw a figure of an RL agent and environment interaction; describe the roles of S_t, A_t, and R_t.
21. Explain what is meant by a *trajectory* in the context of reinforcement learning.
22. Define the environment's dynamics using the *p-function* and discuss its importance in MDPs.
23. Define the *state value function* $v_\pi(s)$ and explain how it is used to evaluate a policy π.
24. Similarly, define the *action value function* $q_\pi(s, a)$ and explain its relevance in policy evaluation.
25. Detail how the Bellman equation provides a recursive way to compute the value of a state under a specific policy.
26. Describe how iterative policy evaluation can be used to approximate the state value function before performing policy improvement.
27. Explain the process of *iterative policy improvement* and how it leads to finding an optimal policy.

Questions on Monte Carlo (MC) Learning

28. Explain the fundamental concept of Monte Carlo (MC) learning in the context of reinforcement learning.
29. Define and distinguish between episodic tasks and continuous tasks in reinforcement learning.

30. Explain the difference between first-visit and every-visit MC methods. What are the implications of each approach on the learning process?
31. Describe the process of First-Visit Monte Carlo prediction as pseudocode. How does it update the state value function?
32. What is the purpose of the backward computation of the return G in the first-visit MC prediction algorithm?
33. Explain the concept of "exploring starts" in the context of MC control and why it is necessary.
34. How do Monte Carlo methods handle the trade-off between exploration and exploitation, especially in environments where exploring starts are not feasible?
35. How does the game of Blackjack illustrate the application of Monte Carlo methods in episodic RL problems?
36. Analyze the potential impact of episodic task length on the effectiveness of Monte Carlo methods. What happens as the length of episodes increases?
37. How do Monte Carlo methods adapt when the environment's dynamics change over time?
38. Reflect on how Monte Carlo methods might perform in real-world scenarios such as financial markets or automated driving systems.

Questions on Off-Policy Monte Carlo (MC) Learning

39. Explain the difference between on-policy and off-policy learning methods in the context of Monte Carlo (MC) methods.
40. How does off-policy learning address the exploration-exploitation dilemma differently than on-policy learning?
41. Discuss why it is important for the behavior policy to cover the target policy in off-policy MC learning.
42. How does the update formula in off-policy MC control differ from the one used in on-policy MC control?
43. In the pseudocode for off-policy MC control, why is the episode terminated early if the action taken does not match the action recommended by the target policy?
44. Explain the potential impacts of the choice of behavior policy on the efficiency and effectiveness of off-policy learning.
45. Discuss how off-policy learning can be applied to complex environments where safety or cost constraints limit exploration.
46. Describe a scenario in which off-policy learning would be particularly advantageous over on-policy learning.
47. How can off-policy MC control be adapted to environments where the behavior policy cannot sufficiently cover the target policy?

Questions on Temporal-Difference (TD) Learning

48. What is temporal-difference (TD) learning and how does it differ from Monte Carlo methods?
49. Explain the concept of bootstrapping in the context of TD learning.

50. Discuss the advantages of TD learning in terms of update frequency and applicability to different types of environments.
51. Describe the SARSA algorithm and explain how it uses the TD update formula.
52. How can the SARSA algorithm be adjusted to improve its performance in highly dynamic environments?
53. Discuss how TD learning methods can be adapted to continuous (non-episodic) environments. What are the implications for learning in such environments?
54. Compare the convergence properties of TD learning and Monte Carlo methods. Why might TD learning converge faster in practical applications?
55. Explain the concept of "n-step TD learning." How does it extend the basic idea of TD learning?
56. What are the implications of using different "n" values in n-step TD learning on the performance and speed of learning?
57. Explain how n-step TD learning might provide a more stable learning update compared to one-step TD updates.
58. Provide an example scenario where TD learning might significantly outperform Monte Carlo methods in terms of learning efficiency and accuracy.

Questions on Off-Policy Temporal-Difference (TD) Learning

59. Describe the main difference between the update rules of SARSA and Q-learning. How does this difference define each as either on-policy or off-policy?
60. Explain how the Q-learning update rule ensures that the learning is directed towards the optimal policy.
61. What are the implications of using the $\max_a Q(S', A')$ term in the Q-learning update rule? Discuss how this term influences the policy improvement process.
62. How does Q-learning handle the exploration-exploitation trade-off differently compared to SARSA?
63. How does the initial setting of Q-values influence the learning process and eventual performance in Q-learning? Discuss the impact of optimistic versus pessimistic initialization.