Deep Reinforcement Learning

Han Cheol Moon
School of Computer Science and Engineering
Nanyang Technological University
Singapore
`tabularasa8931@gmail.com`
`hancheol001@e.ntu.edu.sg`

Contents

Chapter 1

Introduction

1.1 Introduction

Reinforcement Learning (RL) is the science of decision-making. It focuses on learning to make decisions that maximize a numerical reward signal. Unlike traditional approaches, the learner is not explicitly told which actions to take but must discover the most rewarding actions through experience. In complex scenarios, actions influence not only immediate rewards but also future situations and rewards. RL differs from other machine learning paradigms in several key aspects:

- RL uses training information that *evaluates* actions rather than *instructing* by providing the correct actions.
  - The feedback is evaluative, indicating how good or bad an action was, without explicitly identifying the best or worst action.
  - There is no explicit supervisor; learning is guided by a *reward signal*.
- **Delayed Rewards**: Feedback is not immediate; rewards are often received after a delay, reflecting the long-term consequences of actions.
- Time is crucial: RL deals with sequential decision-making and non-i.i.d. (non-independent and identically distributed) data, where past actions influence future data.
- Actions affect the subsequent data: For instance, a robot moving through a room will see a different view with each step it takes, directly influenced by its previous movements.
- The environment is initially unknown: The agent interacts with the environment and improves its policy through **trial-and-error**.
  - **Planning**: When the environment is known, the agent can compute optimal policies or actions.
  - **Exploration**: The agent gathers more information about the environment by trying new actions.
  - **Exploitation**: The agent leverages existing knowledge to maximize rewards.

1.1.1 Sequential Decision Making

- **Goal**: Select actions that maximize the total future reward.
- Actions may have long-term consequences, requiring foresight and planning.
- Rewards may be delayed.
- It may be better to sacrifice immediate rewards for greater long-term gains.

1.1.2 The Concept of State in a Process

For a gentle introduction to the concept of *State*, we start with an informal notion of the terms *Process* and State. Informally, think of a Process as producing a sequence of random outcomes at discrete time steps that we'll index by a time variable $t = 0, 1, 2, \ldots$. The random outcomes produced by a Process might be key financial/trading/business metrics one cares about, such as prices of financial derivatives or the value of a portfolio held by an investor. To understand and reason about the evolution of these random outcomes of a Process, it is beneficial to focus on the internal representation of the Process at each point in time t, that is fundamentally responsible for driving the outcomes produced by the Process. We refer to this internal representation of the Process at time t as the (random) *State* of the Process at time t and denote it as S_t.

Specifically, we are interested in the probability of the next State S_{t+1}, given the present State S_t and the past States $S_0, S_1, \ldots, S_{t-1}$, *i.e.,* $P[S_{t+1}|S_t, S_{t-1}, \ldots, S_0]$. So to clarify, we distinguish between the internal representation (State) and the output (outcomes) of the Process. The State could be any data type-it could be something as simple as the daily closing price of a single stock, or it could be something quite elaborate like the number of shares of each publicly traded stock held by each bank in the U.S., as noted at the end of each week.

Fully observable environment The agent can directly observe the entire environment state. Here, the agent's state is identical to the environment's state, and the information state is the same:

$$O_t = S_t^a = S_t^w$$

This setting can be formally represented as a **Markov Decision Process (MDP)**.

Partially observable environment The agent can only observe the environment indirectly. For example, a robot using camera vision may not know its exact location, or a trading agent might only observe current prices, not broader market trends. In this case, the agent's state differs from the environment's state, leading to a formal model known as a **Partially Observable Markov Decision Process (POMDP)**.

1.1.3 RL Agent Categories

RL agents can be classified as follows:

- **Value-based**: Policy is implicit, and decisions are based on value functions.
- **Policy-based**: The agent directly learns a policy that maps states to actions.

- **Actor-Critic**: Combines value-based and policy-based methods.
- The set of transition and reward functions is referred to as the **model** of the environment.
- **Model-free**: Model-free RL algorithms learn to make decisions without explicitly modeling the environment's dynamics or transitions.
- **Model-based**: Model-based RL algorithms learn an explicit model of the environment, including transition dynamics and reward structure. The agent builds a model and uses it for planning by simulating possible future trajectories to optimize decisions.

1.1.4 Exploration and Exploitation

- RL is akin to trial-and-error learning.
- The agent should discover a good policy from its experiences in the environment without losing too much reward along the way.
- **Exploration**: The agent explores to gather more information about the environment.
- **Exploitation**: The agent exploits known information to maximize reward. For example,
  1. Restaurant Selection:
     - **Exploitation**: Visit your favorite restaurant.
     - **Exploration**: Try a new restaurant.
  2. Oil Drilling:
     - **Exploitation**: Drill at the best-known location.
     - **Exploration**: Test a new location.

1.2 Markov Chain

- Reachable: $i \to j$
- Communicate: $i \leftrightarrow j$
- Irreducible: $i \leftrightarrow j, \forall i, j$
- Absorbing state: If the only possible transition is to itself. This is also a terminal state.
- Transient state: A state s is called a transient state, if there is another state s', that is reachable from s, but not vice versa.
- Recurrent state: A state that is not transient.
- Periodic: A state is periodic if all of the paths leaving s come back after some multiple steps ($k > 1$).
  - Recurrent state is aperiodic if $k = 1$.
- Ergodicity if a Markov chain follows:
  - Irredicible
  - Recurrent

  - Aperiodic

A *Time-Homogeneous Markov Process* is a Markov Process with the additional property that $P[S_{t+1}|S_t]$ is independent of t.

1.3 Multi-Armed Bandits

Bandit problems are stateless. Each arm has a fixed distribution of rewards. Bandit is another name of a slot machine. It does not depend on which arms were pulled previously. The goal is to explore the reward distributions of all arms and then keep pulling the best one. We only have a single chance of selecting an action in each episode.

You can view the bandit problems as Markov decision processes where all states are terminal. In that case, all decision sequences have a length of 1 and subsequent pulls don't influence each other.

In sum,

- When the lever of a slot machine is pulled it gives a random reward coming from a probability distribution specific to that machine.
- Although the machines look identical, their reward probability distributions are different.
- In each turn, gamblers need to decide whether to play the machine that has given the highest average reward so far, or to try another machine.

In our k-armed bandit problem (or *multi-armed bandit problem*), each of the k actions has an expected or mean reward given that that action is selected: let us call this the *value* of that action. We denote the action selected on time step t as A_t, and the corresponding reward as R_t. The value then of an arbitrary action a, denoted $q_*(a)$, is the expected reward given that a is selected:

$$q_*(a) = \mathbb{E}[R_t | A_t = a].$$

Since we do not know which action is the best, we have to estimate the value of actions a at time step t, $Q_t(a)$.

Let's say there are two slot machines with distributions as follows: Then, we can compute their

# Coins	0	1	2	3
Prob	0.3	0.2	0.1	0.4

Table 1.1: Slot Machine 1

# Coins	0	1	2	3
Prob	0.2	0.3	0.4	0.1

Table 1.2: Slot Machine 2

expected value as follows:

$$\mathbb{E}[S_1] = 0 * 0.3 + 1 * 0.2 + 2 * 0.1 + 3 * 0.4 = 1.6$$
$$\mathbb{E}[S_2] = 0 * 0.2 + 1 * 0.3 + 2 * 0.4 + 3 * 0.1 = 1.4$$

Thus, we can say that the average number of coins we can get from the slot machine 1 is higher than that of the slot machine 2. This tells us that we should chose the slot machine 1 to maximize our rewards.

However, we have no access to the true expected value of the slot machines, so we have to estimate them heuristically.

1.3.1 Action-value Methods

One natural way to estimate this is by averaging the rewards actually received: $Q_t(a)$ is sum of rewards when a taken prior t over number of times a taken prior to t:

$$Q_t(a) = \frac{\sum_{t=1}^{t-1} R_i \cdot \mathbb{1}_{A_i=a}}{\sum_{t=1}^{t-1} \mathbb{1}_{A_i=a}}$$

In other words, the average reward (action value) observed before the n-th selection of this action is $Q_n = (R_1 + \cdots + R_{n-1})/(n-1)$. Note that the Q stands for the quality of the action.

Then, the simplest action selection rule is to select one of the actions with the highest estimated value, which is a *greedy action selection* method represented as follows:

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

This approach always exploits current knowledge to maximize immediate reward (*i.e., exploitation*); it spends no time at all sampling apparently inferior actions to see if they might really be better. A simple alternative is to behave greedily most of the time, but every once in a while, instead select randomly from among all the actions (*i.e., exploration*). This near greedy action selection rule is called ϵ-greedy method. In the limit as the number of steps increases, every action will be sampled an infinite number of times, which ensures all the $Q_t(a)$ converge to $q_*(a)$.

1.3.2 Incremental Implementation of The Action-value Methods

$$\begin{aligned} Q_{n+1} &= \frac{1}{n}\sum_{i=1}^{n} R_i \\ &= \frac{1}{n}\left(R_n + \sum_{i=1}^{n-1} R_i\right) \\ &= \frac{1}{n}\left(R_n + (n-1)\frac{1}{n-1}\sum_{i=1}^{n-1} R_i\right) \\ &= \frac{1}{n}\left(R_n + (n-1)Q_n\right) \\ &= Q_n + \frac{1}{n}\Big[R_n - Q_n\Big] \end{aligned}$$

- Q_n: Old estimate
- Q_{n+1}: New estimate
- R_n: New reward

This is an incremental formulas for updating averages with small, constant computation required to process each new reward. This update rule can be expressed in a general form:

$$NewEstimate \leftarrow OldEstimate + StepSize \underbrace{\Big[Target - OldEstimate\Big]}_{error}.$$

The target is presumed to indicate a desirable direction in which to move, though it may be noisy. So, we adjust our current estimate, Q_n, in the direction of the error that we calculate based on the latest observed reward, $R_n - Q_n$, with a step size of $1/n$ and obtain a new estimate, Q_{n+1} .

```python
def incremental_average(rewards):
    """
    Given a list of rewards, compute the incremental average Q_n
    after each new reward using:

        Q_{n+1} = Q_n + (1/n) * (R_n - Q_n)

    Args:
        rewards (list of float): A list of reward values R_i.

    Returns:
        list of float: The running averages Q_1, Q_2, ..., Q_n.
    """
    Q_values = []
    Q_current = 0.0  # Initial estimate (can be set to zero or any prior)

    for i, R in enumerate(rewards, start=1):
        # i is the current step (n), R is R_n
        # Update Q_{n+1} using the incremental formula
        Q_current = Q_current + (1.0 / i) * (R - Q_current)

        # Store the updated estimate
        Q_values.append(Q_current)

    return Q_values

# Example usage:
rewards_example = [10, 20, 15, 5, 30]
running_estimates = incremental_average(rewards_example)

for step, (reward, Q) in enumerate(zip(rewards_example, running_estimates), start=1):
    print(f"Step {step}, Reward = {reward}, Running Average = {Q:.2f}")
```

We can implement the multi-armed bandit problems with the exploration as follows:

```python
class Bandit:
    def __init__(self, arms=10):
        self.rates = np.random.rand(arms) # Stationary distribution

    def play(self, arm):
        rate = self.rates[arm]
        if rate > np.random.rand():
            return 1
        else:
            return 0


class Agent:
    def __init__(self, epsilon, action_size=10):
        self.epsilon = epsilon
        self.Qs = np.zeros(action_size) # Q for each slot machine
        self.ns = np.zeros(action_size) # # Trials for each machine

    def update(self, action, reward):
        self.ns[action] += 1
        self.Qs[action] += (reward - self.Qs[action]) / self.ns[action] # Incremental approach

    def get_action(self):
        if np.random.rand() < self.epsilon:
            return np.random.randint(0, len(self.Qs))
        return np.argmax(self.Qs)

steps = 1000
epsilon = 0.1

bandit = Bandit()
agent = Agent(epsilon)
total_reward = 0
total_rewards = []
rates = []

for step in range(steps):
    action = agent.get_action()
    reward = bandit.play(action)
    agent.update(action, reward)
    total_reward += reward

    total_rewards.append(total_reward)
    rates.append(total_reward / (step + 1))

print(total_reward)

plt.ylabel('Total reward')
plt.xlabel('Steps')
plt.plot(total_rewards)
plt.show()

plt.ylabel('Rates')
plt.xlabel('Steps')
plt.plot(rates)
plt.show()
```

Due to the randomness introduced by the exploration, we should estimate the average return:

```
runs = 200
steps = 1000
epsilon = 0.1
all_rates = np.zeros((runs, steps))  # (2000, 1000)

for run in range(runs):
    bandit = Bandit()
    agent = Agent(epsilon)
    total_reward = 0
    rates = []

    for step in range(steps):
        action = agent.get_action()
        reward = bandit.play(action)
        agent.update(action, reward)
        total_reward += reward
        rates.append(total_reward / (step + 1))

    all_rates[run] = rates

avg_rates = np.average(all_rates, axis=0)

plt.ylabel('Rates')
plt.xlabel('Steps')
plt.plot(avg_rates)
plt.show()
```

Chapter 2

Markov Decision Process

Example To help conceptualize Finite Markov Processes, let us consider a simple example of changes in inventory at a store. Assume you are the store manager and that you are tasked with controlling the ordering of inventory from a supplier. Let us focus on the inventory of a particular type of bicycle. Assume that each day there is random (non-negative integer) demand for the bicycle with the probabilities of demand following a Poisson distribution (with Poisson parameter $\lambda \in \mathbb{R}_{\geq 0}$), i.e. demand i for each $i = 0, 1, 2, \ldots$ occurs with probability

$$f(i) = \frac{e^{-\lambda}\lambda^i}{i!}$$

Denote $F : \mathbb{Z}_{\geq 0} \to [0, 1]$ as the Poisson cumulative probability distribution function, i.e.,

$$F(i) = \sum_{j=0}^{i} f(j)$$

- Assume you have storage capacity for at most $C \in \mathbb{Z}_{\geq 0}$ bicycles in your store.
- Each evening at 6pm when your store closes, you have the choice to order a certain number of bicycles from your supplier (including the option to not order any bicycles on a given day).
- The ordered bicycles will arrive 36 hours later (at 6am the day after the day after you order—we refer to this as delivery lead time of 36 hours).
- Denote the State at 6pm store-closing each day as (α, β), where α is the inventory in the store (referred to as On-Hand Inventory at 6pm) and β is the inventory on a truck from the supplier (that you had ordered the previous day) that will arrive in your store the next morning at 6am (β is referred to as On-Order Inventory at 6pm). Due to your storage capacity constraint of at most C bicycles, your ordering policy is to order $C - (\alpha + \beta)$ if $\alpha + \beta < C$ and to not order if $\alpha + \beta \geq C$. The precise sequence of events in a 24-hour cycle is:

In sum,

- Observe the (α, β) *State* at 6pm store-closing (call this state S_t).
- Immediately order according to the ordering policy described above.

- Receive bicycles at 6am if you had ordered 36 hours ago.
- Open the store at 8am.
- Experience random demand from customers according to demand probabilities stated above (number of bicycles sold for the day will be the minimum of demand on the day and inventory at store opening on the day).
- Close the store at 6pm and observe the state (this state is S_{t+1}).

If we let this process run for a while, in steady-state, we ensure that $\alpha + \beta \leq C$. So to model this process as a Finite Markov Process, we shall only consider the steady-state (finite) set of states

$$\mathcal{S} = \{(\alpha, \beta) | \alpha \in \mathbb{Z}_{\geq 0}, \beta \in \mathbb{Z}_{\geq 0}, 0 \leq \alpha + \beta \leq C\}$$

So restricting ourselves to this finite set of states, our order quantity equals $C - (\alpha + \beta)$ when the state is (α, β).

If the current state S_t is (α, β), there are only $\alpha + \beta + 1$ possible next states S_{t+1} as follows[1]:

$$(\alpha + \beta - i, C - (\alpha + \beta)) \text{ for } i = 0, 1, \ldots, \alpha + \beta$$

with transition probabilities governed by the Poisson probabilities of demand as follows:

$$\mathcal{P}((\alpha, \beta), (\alpha + \beta - i, C - (\alpha + \beta))) = f(i) \text{ for } 0 \leq i \leq \alpha + \beta - 1$$
$$\mathcal{P}((\alpha, \beta), (0, C - (\alpha + \beta))) = \sum_{j=\alpha+\beta}^{\infty} f(j) = 1 - F(\alpha + \beta - 1)$$

- $P((\alpha, \beta), (\alpha + \beta - i, C - (\alpha + \beta)))$ is the probability of transitioning from state (α, β) to $(\alpha + \beta - i, C - (\alpha + \beta))$ when demand equals i. The probability $f(i)$ is based on the Poisson demand distribution.
- If demand exceeds $\alpha + \beta$, the system transitions to a state where all on-hand inventory is depleted:

$$\mathbb{P}((\alpha, \beta), (0, C - (\alpha + \beta))) = 1 - F(\alpha + \beta - 1)$$

- $1 - F(\alpha + \beta - 1)$ gives the probability that the demand is greater than or equal to $\alpha + \beta$ (*i.e.,* demand is large enough to deplete the entire on-hand inventory).

2.1 Markov Reward Process

The reason we covered Markov Processes is that we want to make our way to Markov Decision Processes (the framework for Reinforcement Learning algorithms) by adding incremental features to Markov Processes. Now we cover an intermediate framework between Markov Processes and Markov Decision Processes, known as Markov Reward Processes. We essentially just include the notion of a numerical reward to a Markov Process each time we transition from one state to the

[1]Since $\alpha + \beta = C$, there are only $C + 1$ states. Also, the number of bicycles is the sum of the number of bicycles in the inventory and the truck. Thus, the next state's bicycles is $\alpha + \beta + i$

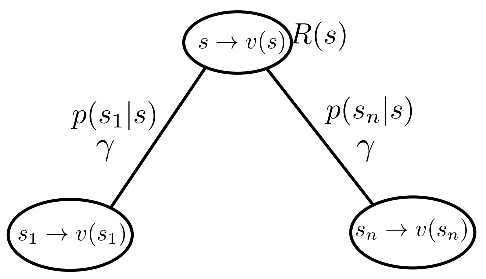


Figure 2.1: MRP diagram

next. These rewards are random, and all we need to do is to specify the probability distributions of these rewards as we make state transitions.

The main purpose of Markov Reward Processes is to calculate how much reward we would accumulate (in expectation, from each of the non-terminal states) if we let the process run indefinitely, bearing in mind that future rewards need to be discounted appropriately (otherwise, the sum of rewards could blow up to ∞). In order to solve the problem of calculating expected accumulative rewards from each non-terminal state, we will first set up some formalism for Markov Reward Processes, develop some (elegant) theory on calculating rewards accumulation, write plenty of code (based on the theory), and apply the theory and code to the simple inventory example (which we will embellish with rewards equal to negative of the costs incurred at the store).

We'd want to identify non-terminal states with large expected returns and those with small expected returns. This, in fact, is the main problem involving a Markov Reward Process—to compute the *Expected Return* associated with each non-terminal state in the Markov Reward Process. Formally, we are interested in computing the *Value Function*, which is given by

$$\begin{aligned} v(s) &= \mathbb{E}[G_t|S_t = s] \\ &= \mathbb{E}\left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1}|S_t = s\right] \\ &= \sum_{s',r} p(s', r|s)[r + \gamma v(s')]. \end{aligned}$$

It is easier to understand the equation with a diagram (*c.f.,* Fig. 2.1) Note that a MRP can be obtained when we fix a policy π for your MDP.

In matrix form,

$$v = PR + \gamma P v.$$

We can solve this system by

$$v = (I - \gamma P)^{-1} PR$$

An MRP is fully characterized by a $\langle S, P, R, \gamma \rangle$ tuple, where S is a set of states, P is a transition probability matrix, R is a reward function, and $\gamma \in [0, 1]$ is a discount factor. I will explain more about the discount factor later.

2.2 Markov Decision Process

The general framework of MDPs (representing environments as MDPs) allows us to model virtually any complex sequential decision-making problem under uncertainty in a way that RL agents can interact with and learn to solve solely through experience.

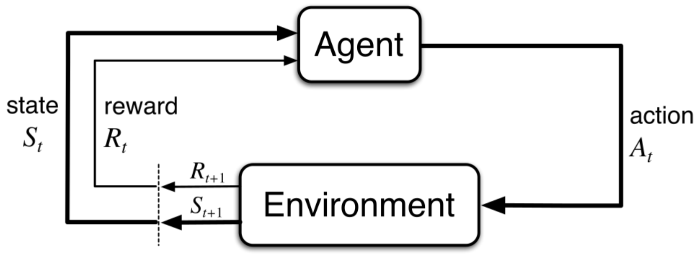


Figure 2.2: The agent-environment interaction in a Markov decision process.

Components of RL:

- An agent
- A policy
- A reward signal: what is good in an immediate sense.
- A value function: what is good in the long run.
- A model of the environment: This allows inferences to be made about how the environment will behave.

An MDP is fully characterized by a $\langle S, A, P, R, \gamma \rangle$ tuple, where S is a set of states, P is a transition probability matrix, R is a reward function, and $\gamma \in [0, 1]$ is a discount factor.

Definition 1 (Markov Property) *A state S_t is* ***Markov*** *if and only if*

$$P[S_{t+1}|S_t, A_t] = P[S_{t+1}|S_t, A_t, S_{t-1}, A_{t-1}, ...]$$

- Actions: a mechanism to influence the environment
- State: specific configurations of the environment

Definition 2 (Transition Function)

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

- The way the environment changes as a response to actions is referred to as the state-transition probabilities, or more simply, the transition function, and is denoted by $T(s, a, s')$.
- $\sum_{r\in\mathcal{R}} \sum_{s'\in\mathcal{S}} p(s', r|s, a) = 1, \quad \forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s)$
- $p(s'|s, a) = P(S_t = s'|S_{t-1} = s, A_{t-1} = a) = \sum_{r\in\mathcal{R}} p(s', r|s, a)$

Definition 3 (Reward Hypothesis) *All goals can be described by the maximization of expected cumulative reward.*

- All goals and purposes can be well thought of as the maximization of the expected value of the cumulative sum of a received scalar signal (*i.e.,* reward).

Definition 4 (Reward Function)

$$\begin{aligned} r(s,a) &= \mathbb{E}[R_t|S_{t-1}=s, A_{t-1}=a] \\ &= \sum_{r\in\mathcal{R}} r \sum_{s'\in\mathcal{S}} p(s',r|s,a) \end{aligned}$$

- $R_t \in \mathcal{R} \subset \mathbb{R}$.
  - Note that negative reward is still reward.
- The expected reward function is defined as a function that takes in a state-action pair.
- It is the expectation of reward at time step t, given the state-action pair in the previous time step.
- The reward function can also be defined as a function that takes a full transition tuple s, a, s'.

$$\begin{aligned} r(s,a,s') &= \mathbb{E}[R_t|S_{t-1}=s, A_{t-1}=a, S_t=s] \\ &= \sum_{r\in\mathcal{R}} r\frac{p(s',r|s,a)}{p(s'|s,a)} \end{aligned}$$

Definition 5 (Discount Factor, γ)

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-1} R_T$$

- The sum of all rewards obtained during the course of an episode is referred to as the *return*, G_t.
- Episodic tasks: $G_t = R_{t+1} + R_{t+2} + \cdots + R_T$.
  - $G_t = R_{t+1} + \gamma G_{t+1}$
- Continuing tasks: $G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$.
  - $\gamma = 0$: myopic evaluation
  - $\gamma = 1$: far-sighted evaluation
  - Uncertainty about the future that may not be fully observed
  - Mathematically convenient to discount rewards.
  - Avoid infinite returns in cyclic Markov processes.

A Simple Example:

- Environment:
  - The grid world has 3 states: $S1$, $S2$, and $S3$.
  - The agent can take actions $A = \{left, right\}$.
  - The rewards are as follows:
    - $R_{left} = -1$: penalty for moving left
    - $R_{right} = +1$: reward for moving right
  - The discount factor γ: 0.9.
- Episode:
  1. The agent starts in $S1$.
  2. If the agent moves left, it incurs a penalty ($R_{left} = -1$), and the episode ends.
  3. If the agent moves right, it receives a reward ($R_{right} = +1$), transitions to $S2$, and the episode continues.
  4. In $S2$, the agent faces the same choices.
  5. The episode ends when the agent either moves left or reaches $S3$.
- Return Calculation:
  - Let's calculate the return for a sample trajectory: $\{S1, right, S2, right, S3\}$.
  - $G = R_{right} + \gamma R_{right} + \gamma^2 R_{right} = 2.71$
  - So, the return for this trajectory is 2.71.

2.3 Policies and Value Functions

- Policies are universal plans, which provides all possible plans for all states.
  - Plans are not enough to fully describe an environment in stochastic environments.
    - What if an agent intends to move right, but ends up going left. Which action does the agent take?
  - Policies are the per-state action descriptions.
  - Policy can be stochastic or deterministic.
  - A policy is a function that prescribes actions to take for a given non-terminal state.

Formally, a policy is a mapping from states to probabilities of selecting each possible action. It can be defined

$$\pi(a|s)$$

2.3.1 The State-Value Functions

Almost all reinforcement learning algorithms involve estimating *value functions*, functions of states (or of state-action pairs) that estimate how good it is for the agent to be in a given state (or how good it is to perform a given action in a given state). The notion of "how good" here is defined in terms of future rewards that can be expected, or, to be precise, in terms of **expected return**. Sure, we want high returns, but high in expectation (on average). Of course, the rewards the agent can expect to receive in the future depend on what actions it will take. Accordingly, value functions are defined with respect to particular ways of action, called policies.

Definition 6 (The State-Value Function, V) *The state-value function $v(s)$ for policy π of an Markov Reward Process is the expected return starting from state s*

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\ &= \mathbb{E}_\pi\left[\sum_{k=0}^{\infty}\gamma^k R_{t+1+k}\middle|S_t = s\right], \quad \forall s \in \mathcal{S} \end{aligned}$$

- The value of a state s is the expectation over policy π.
- Value function is the expected return.
  - Reward: one-step signal that an agent receives.
  - Return: total discounted rewards, which is the sum of rewards obtained from time t onward.
- If we are given a policy and the MDP, we should be able to calculate the expected return starting from every single state.

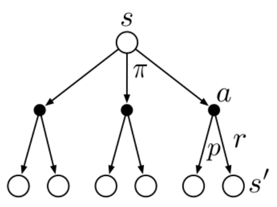
The state-value function satisfies the *Bellman equation*, which expresses the relationship between the value of a state and the values of its successor states:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] \tag{2.1}$$

$$= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] \tag{2.2}$$

$$= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')], \ \forall s \in \mathcal{S}. \tag{2.3}$$

For better understanding, let's look at the §2.3.1. Starting from state s, the root node at the top, the agent could take any of some set of actions (three actions are shown in the diagram), based on its policy π. From each of these, the environment could respond with one of several next states, s' (two states are shown in the figure), along with a reward, r, depending on its dynamics given by the function p. The Bellman equation averages over all the possibilities, weighting each by its probability of occurring. **It states that the value of the start state must equal (discounted) the value of the expected next state, plus the reward expected along the way**. The derivation of Eq. (2.3) is given in Appendix A.1. The state-value function is essential for understanding the value of different states in a Markov Decision Process (MDP) and is often used in various reinforcement learning algorithms, such as value iteration and policy iteration.



2.3.2 The Action-Value Function

- Another critical question that we often need to ask is not merely about the value of a state but the value of taking action a at a state s.
- Which action is better under each policy?
- The action-value function, also known as Q-function or $Q^{\pi}(s, a)$, captures precisely this.
  - The expected return if the agent follows policy π after taking action a in state s.

Definition 7 (The Action-Value Function, Q) *The action-value function $q_{\pi}(s, a)$ for policy π is the expected return starting from state s, tacking action a under policy π*

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

- The Bellman equation for the action-value function is given by

$$\begin{aligned} q_{\pi}(s, a) &= \mathbb{E}_{\pi}[G_t | S_t = s, A_t = a] \\ &= \mathbb{E}_{\pi}[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a] \\ &= \sum_{s', r} p(s', r | s, a)[r + \gamma v_{\pi}(s')] \end{aligned}$$

- Notice that we do not weight over actions because we are interested only in a specific action.
- The action-value function is crucial for understanding the value of different state-action pairs in a Markov Decision Process (MDP) and is used in various reinforcement learning algorithms, such as Q-learning and deep Q-networks (DQN).
- The derivation is given in Appendix A.1

The state-value function can be also expressed by using the action-value function as follows:

$$\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\
&= \sum_{g_t} g_t \cdot P[G_t|S_t = s] \\
&= \sum_{g_t} g_t \cdot P[G_t, S_t = s]/P[S_t = s] \\
&= \sum_{g_t} g_t \cdot \sum_a P[G_t, S_t = s, A_t = a]/P[S_t = s] \\
&= \sum_{g_t} g_t \cdot \sum_a \Big[P[G_t|S_t = s, A_t = a]P[S_t = s, A_t = a]\Big]/P[S_t = s] \\
&= \sum_{g_t} g_t \sum_a P[G_t, A_t = a|S_t = s]P[A_t = a|S_t = s] \\
&= \sum_a \sum_{g_t} g_t P[G_t|S_t = s, A_t = a]P[A_t = a|S_t = s] \\
&= \sum_a q_\pi(s, a)\pi(a|s)
\end{aligned}$$

Note that the expectation is parameerized π as written in $\mathbb{E}_\pi$. We can also simply prove it by the *Law of Total Expectation* [2],

$$\begin{aligned}
v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\
&= \mathbb{E}[\mathbb{E}_\pi[G_t|S_t = s, A_t = a]] \\
&= \sum_a \mathbb{E}_\pi[G_t|S_t = s, A_t = a]P(A_t = a|S_t = s) \\
&= \sum_a q_\pi(s, a)\pi(a|s)
\end{aligned}$$

An intuitive explantation of this derivation is that the expectation depends on an action $a \sim \pi(a|s)$. We want to estimate the expected total return by the sampled action (this is because the total return is the function of a, implicitly). Then, we need to introduce an action variable a and its probability in the expression as the second line of the equation.

2.3.3 The Action-Advantage Function

Definition 8 (The Action-Advantage Function, A)

$$a_\pi(s, a) = q_\pi(s, a) - v_\pi(s)$$

The advantage function describes how much better it is to take action a instead of following policy π. In other words, the advantage of choosing action a over the default action, set by the policy.

2.4 Optimality

Solving a reinforcement learning task means, roughly, finding a policy that achieves a lot of reward over the long run. For finite MDPs, we can precisely define an optimal policy in the

[2] $\mathbb{E}[\mathbb{E}[X|Y]] = \sum_y \Big[\sum_x x \cdot\ p(X = x|Y)\Big]p(Y = y) = \mathbb{E}[X]$

following way. Value functions define a partial ordering over policies. A policy π is defined to be better than or equal to a policy π' if its expected return is greater than or equal to that of π' for all states. In other words, $\pi \geq \pi'$ if and only if $v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in \mathcal{S}$. There is always at least one policy that is better than or equal to all other policies. This is an *optimal policy.* The optimal state-value function, denoted v_* can be defined as

Definition 9 (Optimal State-Value Function) *The optimal state-value function $v_*(s)$ is the maximum value over all policies*

$$v_*(s) = \max_\pi v_\pi(s), \quad \forall s \in \mathcal{S}.$$

- The optimal state-value function can be obtained as follows:

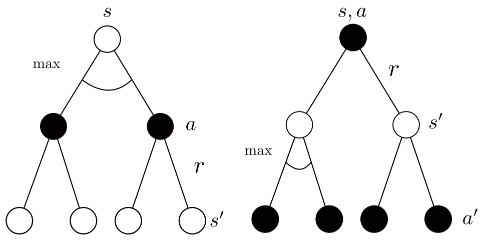
$$\begin{aligned} v_*(s) &= \max_{a \in \mathcal{A}(s)} q_{\pi_*}(s,a) \\ &= \max_a \mathbb{E}_{\pi_*}[G_t | S_t = s, A_t = a] \\ &= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma G_{t+1} | S_t = s, A_t = a] \\ &= \max_a \mathbb{E}_{\pi_*}[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = s, A_t = a] \\ &= \max_a \sum_{r,s'} p(s',r|s,a) \Big[r + \gamma v_*(s') \Big] \end{aligned}$$

Definition 10 (Optimal Action-Value Function) *The optimal action-value function $q_*(s,a)$ is the maximum value over all policies*

$$q_*(s,a) = \max_\pi q_\pi(s,a), \quad \forall s \in \mathcal{S} \textit{ and } a \in \mathcal{A}.$$

- The optimal action-value function can be obtained as follows:

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



- The optimal value function specifies the best possible performance in the MDP.
- The MDP is solved when we know the optimal value function

Theorem 1 (Optimal Policy Theorem)

$$\pi \geq \pi' \quad \textit{if} \quad v_\pi(s) \geq v_{\pi'}(s), \forall s$$

For any Markov Decision Process:

- *There exists an optimal policy π_* that is better than or equal to all other policies, $\pi_* \geq \pi, \forall \pi$*
- *All optimal policies achieve the optimal value function, $v_{\pi_*}(s) = v_*(s)$*
- *All optimal policies achieve the optimal action-value function, $q_{\pi_*}(s,a) = q_*(s,a)$*

An optimal policy can be found by maximizing over $q_*(s,a)$,

$$\pi_*(a|s) = \begin{cases} 1 & \text{if } a = \operatorname{argmax}_{a \in \mathcal{A}} q_*(s,a) \\ 0 & \text{otherwise} \end{cases}$$

- There is always a deterministic optimal policy for any MDP
- If we know $q_*(s,a)$, we immediately have the optimal policy
  - Q-learning: learns Q values first
  - Policy gradient: learns optimal policy without learning Q values

Chapter 3

Dynamic Programming

The term dynamic programming (DP) refers to a collection of algorithms that can be used to compute optimal policies given a perfect model of the environment as a Markov decision process (MDP). Classical DP algorithms are of limited utility in reinforcement learning both because of their assumption of a perfect model and because of their great computational expense, but they are still important theoretically. The key idea of DP, and of reinforcement learning generally, is the use of value functions to organize and structure the search for good policies.

Let π and π' be any pair of deterministic policies such that, for all $s \in \mathcal{S}$,

$$q_\pi(s, \pi'(s)) \geq v_\pi(s).$$

Then, the policy π' must be as good as, or better than π. Equivalently, it must obtain the following inequality for all states $s \in \mathcal{S}$:

$$v'_\pi(s) \geq v_\pi(s).$$

Theorem 2 (Policy Improvement Theorem) $q_\pi(s, \pi'(s)) \geq v_\pi(s)$, *then* $v'_\pi(s) \geq v_\pi(s)$.

Proof.

$$\begin{aligned}
v_\pi(s) &\leq q_\pi(s, \pi'(s)) \\
&= \mathbb{E}[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s, A_t = \pi'(s)] \\
&= \mathbb{E}_{\pi'}[R_{t+1} + \gamma v_\pi(S_{t+1}) | S_t = s] \\
&\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma q_\pi(S_{t+1}, \pi'(S_{t+1})) | S_t = s] \\
&= \mathbb{E}_{\pi'}[R_{t+1} + \gamma \mathbb{E}_{\pi'}[R_{t+2} + \gamma v_\pi(S_{t+2}) | S_{t+1}, A_{t+1} = \pi'(S_{t+1})] \,|\, S_t = s] \\
&= \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 v_\pi(S_{t+2}) |\, S_t = s] \\
&\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 v_\pi(S_{t+3}) |\, S_t = s] \\
&\vdots \\
&\leq \mathbb{E}_{\pi'}[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \ldots \,|\, S_t = s] \\
&= \mathbb{E}_{\pi'}[G_t |\, S_t = s] \\
&= v_{\pi'}(s).
\end{aligned}$$

3.1 Policy Evaluation

- Prediction problem: refers to the problem of **evaluating policies** (sipmly, rating policies), of estimating value functions given a policy (learning to predict returns).
- Control problem: problem of **finding optimal policies**. Usually solved by the pattern of generalized policy iteration (GPI), where the competing processes of policy evaluation and policy improvement progressively move policies towards optimality.
- Policy evaluation: refers to algorithms that solve the prediction problem.
  - Iterative policy evaluation:

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')].$$

    1. Init $v_0(s)$ for all s arbitrarily and to 0 if s is terminal.
    2. Bootstrapping: $v_1(s) \to v_2(s) \to \cdots \to v_N(s)$

A single round of updates, k , involves updating all the state values. The algorithm stops until the changes in state values are sufficiently small in successive iterations.

3.2 Policy Improvement

Our reason for computing the value function for a policy is to help find better policies. The goal of policy improvement is to modify the policy to make it better with respect to the value function. This is done by selecting actions that are known to be better, according to the current value estimates.

- The key is the action-value function, $q_\pi(s, a)$. To improve a policy, we use a state-value function and an MDP to get a one-step look-ahead and determine which of the actions lead to the highest value. We can selects the action that maximizes the action-value function in a greedy way:

$$\begin{aligned}\pi'(s) &= \operatorname*{argmax}_a q_\pi(s, a) \\ &= \operatorname*{argmax}_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')].\end{aligned}$$

- The greedy policy takes the action that looks best in the short term according to v_π.

The process of policy evaluation and improvement is often repeated iteratively until the policy converges to an optimal policy that maximizes the expected return in the given environment. This iterative process is the basis for algorithms like *policy iteration* and *value iteration* in reinforcement learning.

3.3 Value Iteration

One drawback to policy iteration is that each of its iterations involves policy evaluation, which may itself be a protracted iterative computation requiring multiple sweeps through the state set.

In fact, the policy evaluation step of policy iteration can be truncated in several ways without losing the convergence guarantees of policy iteration.

$$v_{k+1}(s) = \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')], \; \forall s \in \mathcal{S}. \tag{3.1}$$

For arbitrary v_0, the sequence v_k can be shown to converge to v_* under the same conditions that guarantee the existence of v_*.

Chapter 4

Monte Carlo Methods

Monte Carlo methods are a class of computational algorithms that rely on random sampling to obtain numerical results. In the context of reinforcement learning (RL), Monte Carlo methods are often used to estimate the value of states or state-action pairs in a Markov Decision Process (MDP).

4.1 Monte Carlo Prediction (Evaluation)

The goal is to estimate the value of a policy, that is, to learn how much total reward to expect from a policy.

The most straightforward approach that comes to mind is one I already men- tioned: it's to run several episodes with this policy collecting hundreds of trajectories, and then calculate averages for every state, just as we did in the bandit environments. This method of estimating value functions is called Monte Carlo prediction (MC).

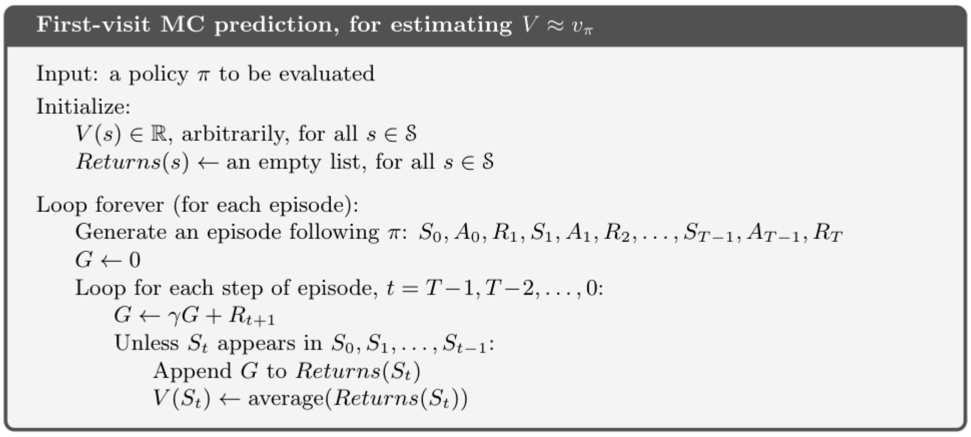
4.1.1 First Visit vs. Every Visit

Suppose we wish to estimate $v_\pi(s)$, the value of a state s under policy π, given a set of episodes obtained by following π and passing through s. Each occurrence of state s in an episode is called a visit to s.

The first-visit MC method estimates $v_\pi(s)$ as the average of the returns following first visits to s, whereas *the every-visit MC* method averages the returns following all visits to s.

- First visit MC treats each trajectory as an i.i.d., sample of $v(s)$.
- For instance, we have two example episodes,
  - $A, 3 \to A, 2 \to B, -4 \to A, 4 \to B, -3$.
  - $B, -2 \to A, 3 \to B, -3$.

- Each number next to the states are reward. Let's say discount factor is 1 and we want to compute $V(A)$, then the first visit MC computes a return by summing all the rewards ($G \leftarrow \gamma G + R_{t+1}$) coming after the first visit to the state A. Therefore, we can't have more than one summation term for each episode for a state. In sum,
  - For the first episode: $3 + 2 - 4 + 4 - 3 = 2$.
  - For the second episode: $3 - 3 = 0$.
  - Thus, $V(A) = (2 + 0)/2 = 1$.
  - It must be noted that if an episode doesn't have an occurrence of 'A', it won't be considered in the average. Hence if a 3rd episode is given like $B, 3 \rightarrow B, 3 \rightarrow \textit{terminate}$ existed, still $V(A) = 1$.



First-visit MC prediction, for estimating $V \approx v_\pi$

```
Input: a policy $\pi$ 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 $Returns(S_t)$
            $V(S_t) \leftarrow$ average($Returns(S_t)$)
```

Figure 4.1: First-visit MC prediction algorithm.

Every visit MC method averages the returns following **all visits** to s.

- Here, we would be creating a new summation term adding all rewards coming after every occurrence of 'A'(including that of A as well).
  - From 1st episode: $(3 + 2 - 4 + 4 - 3) + (2 - 4 + 4 - 3) + (4 - 3) = 2 - 1 + 1$
  - From 2nd episode: $(3 - 3) = 0$
  - As we got 4 summation terms, we will be averaging using $N = 4$ *i.e.,*
  - $V(A) = (2 - 1 + 1 + 0)/4 = 0.5$

4.2 Monte Carlo Control

$$\pi_0 \xrightarrow{E} q_{\pi_0} \xrightarrow{I} \pi_0 \xrightarrow{E} q_{\pi_0} \xrightarrow{I} \cdots \xrightarrow{I} \pi_* \xrightarrow{E} q_{\pi_*},$$

- π_0: arbitrary policy.
- Policy improvement is done by making the policy greedy with respect to the current value function (action-value function).

- For any action-value function q, the corresponding greedy policy is the one that, for each $s \in \mathcal{S}$, deterministically chooses an action with maximal action-value:

$$\pi(S) = \operatorname*{argmax}_a q(s, a).$$

- Policy improvement then can be done by constructing each π_{k+1} as the greedy policy with respect to q_{π_k}.
- By Theorem 2, each π_{k+1} is uniformly better than π_k or just as good as π_k.

4.3 On-Policy vs. Off-Policy

On-policy learns about the optimal policy by executing the policy and evaluating and improving it.

- Learning to be great by itself.
- The agent learns from the experiences it gathers while following its current policy.
- The policy is typically denoted by π.
- The learning process is directly tied to the policy being used for action selection.
- Examples of on-policy algorithms include REINFORCE (Monte Carlo Policy Gradient) and Proximal Policy Optimization (PPO).

Off-policy learns about the optimal policy by using data generated by another policy.

- Learning from others.
- The agent can learn from historical data, which may have been generated by an older policy.
- The policy used for learning is not necessarily the one used for action selection.
- The data collected can be more efficiently reused for learning new policies.
- Examples of off-policy algorithms include Q-learning, Deep Q Networks (DQN), and Actor-Critic methods.

Exploration-Exploitation Trade-off

- On-policy algorithms often struggle with exploration because they are committed to following their current policy, which may not explore the state space efficiently.
- Off-policy algorithms can be more sample-efficient in exploration as they can learn from a mixture of data from different policies.

Data Efficiency

- Off-policy algorithms can potentially be more data-efficient since they can reuse past experiences collected by any policy.
- On-policy algorithms typically require more samples to update the policy effectively.

Stability and Convergence

- On-policy algorithms can be more stable because they are directly optimizing the policy they are following.
- Off-policy algorithms might face challenges related to the stability of learning, especially when dealing with function approximation, as is often the case with deep learning.

Both on-policy and off-policy algorithms have their strengths and weaknesses, and the choice between them depends on the specific requirements and characteristics of the problem at hand.

4.4 Exploration-Exploitation Trade-off

The more episodes are collected, the better, but there is a problem. If the algorithm for policy improvement always updates the policy greedily, meaning it takes only actions leading to immediate reward, actions and states not on the greedy path will not be sampled sufficiently, and potentially better rewards would stay hidden from the learning process.

Essentially, we are forced to make a choice between making the best decision given the current information or start exploring and finding more information. This is also known as the Exploration vs. Exploitation Dilemma.

We are looking for something like a middle ground between those. Full-on exploration would mean that we would need a lot of time to collect the needed information, and full-on exploitation would make the agent stuck into a local reward maximum. There are two approaches to ensure all actions are sampled sufficiently called on-policy and off-policy methods.

4.4.1 On Policy

On-policy methods solve the exploration vs exploitation dilemma by including randomness in the form of a policy that is soft, meaning that non-greedy actions are selected with some probability. These policies are called ϵ-greedy policies as they select random actions with an ϵ probability and follow the optimal action with $1-\epsilon$ probability

Since the probability of selecting from the action space randomly is ϵ, the probability of selecting any particular non-optimal (non-greedy) action is $\epsilon/|\mathcal{A}(s)|$. The probability of following the optimal action will always be slightly higher, however, because we have a $1-\epsilon$ probability of selecting it outright and $\epsilon/|\mathcal{A}(s)|$ probability of selecting it from sampling the action space [1]

$$P(a_t^*) = 1 - \epsilon + \epsilon/|\mathcal{A}(s)|.$$

[1]Since the greedy (or optimal) action can be selected by either 1-ϵ or $\epsilon/|\mathcal{A}(s)|$, $a_1 = \epsilon/\mathcal{A}(s), a_2 = \epsilon/\mathcal{A}(s), \cdots, a_{best} = 1-\epsilon+\epsilon/\mathcal{A}(s), a_n = \epsilon/\mathcal{A}(s)$

It is also worth noting that because the optimal action will be sampled more often than the others making on-policy algorithms will generally converge faster but they also have the risk of trapping the agent into a local optimum of the function.

4.4.2 Off Policy

All learning control methods face a dilemma: They seek to learn action values conditional on subsequent *optimal* behavior, but they need to behave non-optimally in order to explore all actions (to find the optimal actions). How can they learn about the optimal policy while behaving according to an exploratory policy? The on-policy approach in the preceding section is actually a compromise. It learns action values not for the optimal policy, but for a near-optimal policy that still explores. A more straightforward approach is to **leverage two policies**, one that is learned about and that becomes the optimal policy, and one that is more exploratory and is used to generate behavior. The policy being learned about is called the *target policy*, and the policy used to generate behavior is called the *behavior policy.* In this case we say that learning is from data "off" the target policy, and the overall process is termed *off-policy learning.*

Suppose we wish to estimate v_π or q_π, but all we have are episodes following another policy b, where $b \neq \pi$. In this case, π is the target policy, b is the behavior policy, and both policies are considered fixed and given.

In order to use episodes from b to estimate values for π, we require that every action taken under π is also taken, at least occasionally, under b. That is, we require that $\pi(a|s) > 0$ implies $b(a|s) > 0$. This is called the assumption of *coverage.* It follows from coverage that b must be stochastic in states where it is not identical to π. The target policy π, on the other hand, may be deterministic, and, in fact, this is a case of particular interest in control applications. In control, the target policy is typically the deterministic greedy policy with respect to the current estimate of the action-value function. This policy becomes a deterministic optimal policy while the behavior policy remains stochastic and more exploratory, for example, an ϵ-greedy policy.

Almost all off-policy methods utilize *importance sampling*, a general technique for **estimating expected values under one distribution given samples from another**. We apply importance sampling to off-policy learning by weighting returns according to the relative probability of their trajectories occurring under the target and behavior policies, called the *importance-sampling ratio.* Given a starting state S_t, the probability of the subsequent state-action trajectory, $A_t, S_{t+1}, A_{t+1}, \cdots, S_T$, occurring under any policy π is

$$\begin{aligned} P(A_t, S_{t+1}, A_{t+1}, \cdots, S_T) &= \pi(A_t|S_t)p(S_{t+1}|S_t, A_t)\pi(A_{t+1}|S_{t+1}) \cdots (S_T|S_{T-1}, A_{T-1}) \\ &= \prod_{k=t}^{T-1} \pi(A_k|S_k)p(S_{k+1}|S_k, A_k), \end{aligned}$$

where p here is the state-transition probability function. Thus, the relative probability ρ of the trajectory under the target (π) and behavior policies (b) is

$$\begin{aligned} \rho_{t:T-1} &= \frac{\prod_{k=t}^{T-1} \pi(A_k|S_k)p(S_{k+1}|S_k, A_k)}{\prod_{k=t}^{T-1} b(A_k|S_k)p(S_{k+1}|S_k, A_k)} \\ &= \prod_{k=t}^{T-1} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}. \end{aligned}$$

The importance sampling ratio ends up depending only on the two polices and the sequence, not on the MDP's state-transition probability.

Recall that we wish to estimate the expected returns under the target policy, but all we have are returns G_t from the behavior policy, which results in

$$\mathbb{E}[G_t|S_t = s] = v_b(s).$$

So, the ratio $\rho_{t:T-1}$ transforms the returns

$$\mathbb{E}[\rho_{t:T-1}G_t|S_t = s] = v_\pi(s).$$

4.5 Temporal-Difference Learninig

One of the central ideas of reinforcement learning is temporal-difference (TD) learning. TD learning is a combination of Monte Carlo ideas and dynamic programming (DP) ideas. Like Monte Carlo methods, TD methods can learn directly from raw experience without a model of the environment's dynamics. Like DP, TD methods update estimates based in part on other learned estimates, without waiting for a final outcome (they bootstrap).

One of the main drawbacks of MC methods is the fact that the agent has to wait until the end of an episode when it can obtain the actual G_t. Then, it uses that return as a target for $V(S_t)$.

The core of TD learning is the Temporal Difference (TD) error, denoted as δ_t, which represents the difference between the estimated value of a state at time t and the estimated value at time $t+1$. It is calculated as the difference between the reward at time $t+1$ and the discounted estimate of the value function at time t.

Constant-α MC:

$$V(S_t) \leftarrow V(S_t) + \alpha\Big[G_t - V(S_t)\Big]$$

One-step TD (or TD(0)):

$$V(S_t) \leftarrow V(S_t) + \alpha\Big[\underbrace{R_{t+1} + \gamma V(S_{t+1}) - V(S_t)}_{\text{TD error, } \delta}\Big]$$

To clarify the difference, let's consider a simple example of a grid-world navigation task.

MC Method The agent explores the grid-world and reaches a target goal (S_9). The episode ends, and the agent receives a reward of $+1$. At the end of the episode, the agent updates the value of each visited state based on the observed return. If the trajectory is $S_1 \to S_2 \to \cdots \to S_9$, then the value update for each visited state would be based on the return of $+1$.

TD Learning The agent starts at S_1, moves to S_2, and receives a reward of -0.1 for the movement. The agent updates the value of S_1 based on the observed TD error, which is the immediate reward plus the discounted estimate of the next state. If the agent moves right from S_1 to S_2, the update for S_1 would be based on $R_{move} + \gamma V(S_2)$.

- Monte Carlo:

  - Pros: Unbiased estimate of value functions.
  - Cons: Requires complete episodes, might be computationally expensive.

- Temporal Difference:
  - Pros: Updates at each time step, more sample-efficient.
  - Cons: Potential bias due to bootstrapping.

In summary, in the grid-world example, Monte Carlo methods would update values based on complete episodes, whereas TD learning would update values at each time step. The choice depends on the nature of the task, the availability of information, and the computational resources.

```python
import numpy as np
import random

def random_walk_environment(state):
    """
    Given the current state (1..5), this environment transitions left or right
    with equal probability and returns the next state, as well as the reward.

    Args:
        state (int): Current non-terminal state, 1 <= state <= 5.

    Returns:
        next_state (int): Next state in {0,1,2,3,4,5,6}.
        reward (float): Reward for the transition.
    """
    # Action: move left (prob=0.5) or right (prob=0.5)
    action = random.choice([-1, +1])
    next_state = state + action

    # If next_state == 6, reward = +1; if next_state == 0, reward = 0
    # For states 1..5, reward = 0
    if next_state == 6:
        reward = 1.0
    else:
        reward = 0.0

    return next_state, reward


def td_zero_random_walk(num_episodes=1000, alpha=0.1, gamma=1.0):
    """
    Perform TD(0) to evaluate the value function V(s) for the random walk
    example.

    Args:
        num_episodes (int): Number of episodes to run.
        alpha (float): Step-size / learning rate.
        gamma (float): Discount factor (set to 1.0 in the classic random walk).

    Returns:
        V (np.array): Learned value function estimates V(s) for states s in
    [0..6].
                      Terminal states 0 and 6 stay at V=0 by definition.
    """
    # We track V(s) for states 0..6, but 0 and 6 are terminal.
    # Initialize with zeros (you could initialize them differently).
    V = np.zeros(7)

    for episode in range(num_episodes):
        # Start each episode in the middle state (often chosen as 3 in the random walk)
        state = 3

        # Continue until we reach a terminal state
        while state not in [0, 6]:
            next_state, reward = random_walk_environment(state)

            # TD(0) update:
            # V[s] <- V[s] + alpha * [R + gamma * V[s'] - V[s]]
            V[state] = V[state] + alpha * (reward + gamma * V[next_state] - V[state])

            state = next_state

    return V

if __name__ == "__main__":
    # Set random seed for reproducibility (optional)
    random.seed(42)
    np.random.seed(42)

    # Run TD(0) for 1000 episodes
    num_episodes = 1000
    alpha = 0.1
    V_estimates = td_zero_random_walk(num_episodes, alpha)

    print(f"Value Function Estimates after {num_episodes} episodes:")
    for s in range(7):
        print(f"V({s}) = {V_estimates[s]:.3f}")
```

4.5.1 Q-learning: Off-policy TD Control

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big]$$

- Off-policy. For instance, we can sample an action A_t from a ϵ-greedy policy.
- Update Q for each step.
- Maximization bias in Q-learning

4.5.2 Sarsa: On-policy TD Control

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \Big[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\Big]$$

- On-policy agorithm, unlike Q-learning, we sample both A_t and A_{t+1} from another policy like ϵ-greedy policy.

4.5.3 Double Q-learning

Randomly select Q_1 or Q_2.

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q_2(S_{t+1}, a^*) - Q_1(S_t, A_t)\Big],$$

where a^* is

$$a^* = \operatorname*{argmax}_a Q_1(s', a).$$

$$Q_2(S_t, A_t) \leftarrow Q_2(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q_1(S_{t+1}, a^*) - Q_2(S_t, A_t)\Big],$$

where a^* is

$$a^* = \operatorname*{argmax}_a Q_2(s', a).$$

Chapter 5

Deep Reinforcement Learning

Three challenges in DRL:

- Sequential feedback
- Evaluative feedback
- Sampled feedback

Select:

1. A value function to approximate.
2. A neural network architecture
    - e.g., value (single output node) or action (multiple output nodes)
3. What to optimize
4. Policy evaluation algorithm
5. Exploration strategy
6. A loss function
7. Optimization method

Some considerations:

- Non-stationary target
- Data correlated with time
    - Samples in a batch are correleated, given that most of these samples come from the same trajectory and policy. It breaks the i.i.d. assumptions.

Chapter 6

Deep Q-Network

Q-learning is a popular model-free reinforcement learning algorithm that is used to find an optimal action-selection policy for a given finite Markov decision process (MDP). It's a form of temporal difference learning, where the agent learns from the consequences of its actions in the environment.

The main goal of Q-learning is to learn a policy, represented by the action-value function $Q(s, a)$, which estimates the expected cumulative future rewards of taking action a in state s.

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t) \Big]$$

6.1 A Few Considerations

The I.I.D. assumption is violated.

6.1.1 Non-Stationary Target

Policy changes over time

6.1.2 Correlated Samples

Samples are drawn from the same trajectory.

6.2 Solutions

6.2.1 Using Target Networks

A straightforward way to make target values more stationary is to have a separate network that we can fix for multiple steps and reserve it for calculating more stationary targets. The network with this purpose in DQN is called the target network.

6.2.2 Experience Replay

Having a replay buffer allows the agent two critical things.

- First, the training process can use a more diverse mini-batch for performing updates.
- Second, the agent no longer has to fit the model to the same small mini-batch for multiple iterations. Adequately sampling a sufficiently large replay buffer yields a slow-moving target, so the agent can now sample and train on every time step with a lower risk of divergence.

6.3 Double DQN

Q-learning tends to overestimate action- value functions. Our DQN agent is no different; we're using the same off-policy TD target, after all, with that max operator. The crux of the problem is simple: We're taking the max of estimated values. Estimated values are often off-center, some higher than the true values, some lower, but the bottom line is that they're off. The problem is that we're always taking the max of these values, so we have a preference for higher values, even if they aren't correct. Our algorithms show a positive bias, and performance suffers.

One way to better understand positive bias and how we can address it when using function approximation is by unwrapping the max operator in the target calculations. The max of a Q-function is the same as the Q-function of the argmax action.

- You create two action-value functions, Q_A and Q_B.
- You flip a coin to decide which action-value function to update. For example, Q_A on heads, Q_B on tails.
- If you got a heads and thus get to update Q_A: You select the action index to evaluate from Q_B, and evaluate it using the estimate Q_A predicts. Then, you proceed to update Q_A as usual, and leave Q_B alone.
- If you got a tails and thus get to update Q_B, you do it the other way around: get the index from Q_A, and get the value estimate from Q_B. Q_B gets updated, and Q_A is left alone.

Instead of adding this overhead that's a detriment to training speed, we can perform double learning with the other network we already have, which is the target network. However, instead of training both the online and target networks, we continue training only the online network, but use the target network to help us, in a sense, cross-validate the estimates.

We want to be cautious as to which network to use for action selection and which network to use for action evaluation. Initially, we added the target network to stabilize training by avoiding chasing a moving target. To continue on this path, we want to make sure we use the network we're training, the online network, for answering the first question. In other words, we use the online network to find the index of the best action. Then, we use the target network to ask the second question, that is, to evaluate the previously selected action.

6.4 Dueling DDQN

The dueling network is an improvement that applies only to the net- work architecture and not the algorithm. That is, we won't make any changes to the algorithm, but the only modifications go into the network architecture.

Chapter 7

Policy Gradient

We will use a gradient ascent algorithm:

$$\begin{aligned} J(\theta) &= \mathbb{E}_{\tau\sim\pi_\theta(\tau)}[r(\tau)] \\ &= \int \pi_\theta(\tau) r(\tau) \end{aligned}$$

It is a expected reward under the policy π_θ.

$$\theta \leftarrow \theta + \eta \nabla_\theta J(\theta)$$

Note that by using REINFORCE algorithm which can be expressed as follows:

$$\nabla_\theta \pi_\theta(\tau) = \pi_\theta \frac{\nabla_\theta \pi_\theta(\tau)}{\pi_\theta(\tau)} = \pi_\theta \nabla_\theta \log \pi_\theta(\tau)$$

We can express $J(\theta)$ as follows:

$$\begin{aligned} \nabla_\theta J(\theta) = \int \nabla_\theta \pi_\theta(\tau) r(\tau) &= \int \pi_\theta \nabla_\theta \log \pi_\theta(\tau) r(\tau) \\ &= \mathbb{E}_{\tau\sim\pi_\theta(\tau)}[\nabla_\theta \log \pi_\theta(\tau) r(\tau)] \end{aligned}$$

Note that $\nabla_\theta \log \pi_\theta(\tau)$ is the maximum loglikelihood of trajectory, because gradient is the maximum direction of the function. This expectation can be estimated by Monte-Carlo method.

We just sample trajectories using current policy and adjust the likelihood of trajectories by episodic rewards.

$$\pi_\theta(\tau) = p_\theta(s_1, a_1, s_2, a_2, \cdots, s_T, a_T) = p(s_1) \prod_{t=1}^{T} \pi_\theta(a_t|s_t) p(s_{t+1}|s_t, a_t)$$

Note that the product term looks causing gradient explosion or vanishing problems for a long sequence problem, but it turns into a multiplication as below by the log-derivative trick, so it can avoid the issues.

In general, the agent has no access to $p(s_1)$ and $p(s_{t+1}|s_t, a_t)$ (we don't know trainsition robability.).

$$\begin{aligned}
\nabla_\theta \log \pi_\theta(\tau) &= \nabla_\theta \left[\log p(s_1) + \sum_{t=1}^{T} (\log \pi_\theta(a_t|s_t) + \log p(s_{t+1}|s_t, a_t)) \right] \\
&= \sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \\
r(\tau) &= \sum_{t=1}^{T} r(s_t, a_t)
\end{aligned}$$

By using the above equations, we can re-express the gradient of the cost function as

$$\begin{aligned}
\nabla_\theta J(\theta) &= \int \nabla_\theta \pi_\theta(\tau) r(\tau) = \int \pi_\theta \nabla_\theta \log \pi_\theta(\tau) \\
&= \mathbb{E}_{\tau \sim \pi_\theta(\tau)} [\nabla_\theta \log \pi_\theta(\tau) r(\tau)] \\
&= \mathbb{E}_{\tau \sim \pi_\theta(\tau)} \left[\left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_t|s_t) \right) \left(\sum_{t=1}^{T} r(s_t, a_t) \right) \right]
\end{aligned}$$

By using Monte-Carlo method, we can replace the expectation by sampling multiple trajectories in practice:

$$\nabla_\theta J(\theta) \approx \frac{1}{N} \sum_{i=1}^{N} \left[\left(\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(\mathbf{a}_{i,t}|\mathbf{s}_{i,t}) \right) \left(\sum_{t=1}^{T} r(\mathbf{s}_{i,t}, \mathbf{a}_{i,t}) \right) \right]$$

7.0.1 Causality

$$\begin{aligned}
\nabla_\theta J(\theta) &\approx \frac{1}{N} \sum_{i=1}^{N} \left[\sum_{t=1}^{T} \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) \sum_{t=1}^{T} r(s_{i,t}, a_{i,t}) \right] \\
&= \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \left[\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) \sum_{t'=1}^{T} r(s_{i,t'}, a_{i,t'}) \right] \\
&= \frac{1}{N} \sum_{i=1}^{N} \sum_{t=1}^{T} \left[\nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}) \sum_{t'=t}^{T} r(s_{i,t'}, a_{i,t'}) \right] \quad \text{by causality}
\end{aligned}$$

7.1 Natural Policy Gradient

reference: Nathan Ratliff, Information Geometry and Natural Gradients.

7.1.1 KL-divergence between perturbed distributions

Let $p(x;\theta)$ be some family of probability distributions over x parameterized by a vector of real numbers θ. We're interested in knowing how much the distribution changes when we perturb the parameter vector from a fixed θ_t to some new value $\theta_t + \delta\theta$. As a measure of change in probability distribution, we can use the KL-divergence measure. Specifically, we want to measure $D_{KL}(p(x;\theta_t)||p(x;\theta_t+\delta\theta)$, but we want to write it in a form amenable to the gradient-based update formulation. We can do this by taking it's second-order Taylor expansion around θ_t. During the derivation, we'll find that a lot of terms in the expansion disappear leaving us with a very simple expression that's perfect for our purposes.

Looking first at the full KL-divergence, we see that the term we want to

$$\begin{aligned} D_{KL}(p(x;\theta_t)||p(x;\theta_t+\delta\theta)) &= \int p(x;\theta_t)\log\frac{p(x;\theta_t)}{p(x;\theta_t+\delta\theta)}dx \\ &= \int p(x;\theta_t)\log p(x;\theta_t)dx - \int p(x;\theta_t)\log p(x;\theta_t+\delta\theta)dx \end{aligned}$$

Note that the second-order Taylor series expansion is

$$f(\theta) \approx f(\theta_t) + \nabla f(\theta_t)^T\delta\theta + \frac{1}{2}\delta\theta^T\nabla^2 f(\theta_t)\delta\theta,$$

where $\theta = \theta_t + \delta\theta$, or equivalently $\delta\theta = \theta - \theta_t$. Applying that expansion to the pertinent term in the KL-divergence expression, we get

$$\log p(x;\theta_t+\delta\theta) \approx \log p(x;\theta_t) + \left(\frac{\nabla p(x;\theta_t)}{p(x;\theta_t)}\right)^T\delta\theta + \frac{1}{2}\delta\theta^T(\nabla^2\log p(x;\theta_t))\delta\theta.$$

Plugging this second-order Taylor expansion back into the above expression for the D_{KL} gives

$$\begin{aligned} &D_{KL}(p(x;\theta_t)||p(x;\theta_t+\delta\theta)) \\ &\quad= \int p(x;\theta_t)\log p(x;\theta_t)dx - \int p(x;\theta_t)\log p(x;\theta_t+\delta\theta)dx \\ &\quad\approx \int p(x;\theta_t)\log p(x;\theta_t)dx \\ &\qquad - \int p(x;\theta_t)\left(\log p(x;\theta_t) + \left(\frac{\nabla p(x;\theta_t)}{p(x;\theta_t)}\right)^T\delta\theta + \frac{1}{2}\delta\theta^T(\nabla^2\log p(x;\theta_t))\delta\theta\right)dx \\ &\quad= \int p(x;\theta_t)\log\frac{p(x;\theta_t)}{p(x;\theta_t)}dx - \underbrace{\int p(x;\theta_t)\left(\frac{\nabla p(x;\theta_t)}{p(x;\theta_t)}\right)^T dx}_{=0} - \frac{1}{2}\delta\theta^T\left(\int p(x;\theta_t)\nabla^2\log p(x;\theta_t)dx\right)\delta\theta \\ &\quad= -\frac{1}{2}\delta\theta^T\left(\int p(x;\theta_t)\nabla^2\log p(x;\theta_t)dx\right)\delta\theta. \end{aligned}$$

$\int\nabla p(x;\theta_t)$ is zero since

$$\int\nabla p(x;\theta_t) = \nabla\int p(x;\theta_t) = \nabla 1 = 0$$

The Hessian can be computed as follows:

$$\begin{aligned}\frac{\partial^2}{\partial\theta_t^i\partial\theta_t^j}[\log p(x;\theta_t)] &= \frac{\partial}{\partial\theta_t^i}\left(\frac{\frac{\partial}{\partial\theta_t^j}p(x;\theta_t)}{p(x;\theta_t)}\right)\\ &= \frac{p(x;\theta_t)\frac{\partial^2}{\partial\theta_t^i\partial\theta_t^j}p(x;\theta_t) - \frac{\partial}{\partial\theta_t^i}p(x;\theta_t)\frac{\partial}{\partial\theta_t^j}p(x;\theta_t)}{p(x;\theta_t)^2}\\ &= \frac{1}{p(x;\theta_t)}\frac{\partial^2}{\partial\theta_t^i\partial\theta_t^j}p(x;\theta_t) - \left(\frac{\frac{\partial}{\partial\theta_t^i}p(x;\theta_t)}{p(x;\theta_t)}\right)\left(\frac{\frac{\partial}{\partial\theta_t^j}p(x;\theta_t)}{p(x;\theta_t)}\right).\end{aligned}$$

The second term is an element of the outer product between $\nabla\log p(x;\theta_t)$ and itself. In matrix form, this becomes

$$\nabla^2\log p(x;\theta_t) = \frac{1}{p(x;\theta_t)}\nabla^2 p(x;\theta_t) - \nabla\log p(x;\theta_t)\nabla\log p(x;\theta_t)^T.$$

Finally, we get

$$\begin{aligned}&D_{KL}(p(x;\theta_t)||p(x;\theta_t+\delta\theta))\\ &\approx -\frac{1}{2}\delta\theta^T\int p(x;\theta_t)\nabla^2\log p(x;\theta_t)dx\delta\theta\\ &= \frac{1}{2}\delta\theta^T\left(\int\nabla^2\log p(x;\theta_t)dx\right)\delta\theta\\ &\quad+\frac{1}{2}\delta\theta^T\left(\int p(x;\theta_t)[\nabla\log p(x;\theta_t)\nabla\log p(x;\theta_t)^T]dx\right)\delta\theta\\ &= \frac{1}{2}\delta\theta^T\underbrace{\left(\int p(x;\theta_t)[\nabla\log p(x;\theta_t)\nabla\log p(x;\theta_t)^T]dx\right)}_{G(\theta_t)}\delta\theta.\end{aligned}$$

The central matrix here $G(\theta_t)$ is known as the **Fisher Information matrix** and can has been thoroughly studied within the field of Information Geometry as the natural Riemannian structure on a manifold of probability distributions. As such it defines a natural norm on perturbations to probability distributions, which was our original motivation for examning the second-order Taylor expansion of the KL-divergence in the first place.

$$\theta_{t+1} = \theta_t - \eta_t G(\theta_t)^{-1}\nabla f(\theta_t).$$

7.2 Proximal Policy Optimization

PPO objective is

$$\theta_{k+1} = \underset{\theta}{\operatorname{argmax}} \underset{s,a\sim\pi_{\theta_k}}{\mathbb{E}}[L(s,a,\theta_k,\theta)],$$

where L is given by

$$L(s,a,\theta_k,\theta) = \min\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)}A^{\pi_{\theta_k}}(s,a),\ \operatorname{Clip}\left(\frac{\pi_\theta(a|s)}{\pi_{\theta_k}(a|s)},1-\varepsilon,1+\varepsilon\right)A^{\pi_{\theta_k}}(s,a)\right).$$

Roughly, ε is a hyperparameter which says how far away the new policy is allowed to go from the old one. A simpler expression of the above expression is

$$L(s,a,\theta_k,\theta) = \min\left(\frac{\pi_\theta\left(a|s\right)}{\pi_{\theta_{\mathrm{k}}}\left(a|s\right)}A^{\pi_{\theta_k}}(s,a), g(\varepsilon, A^{\pi_{\theta_k}}(s,a))\right), \tag{7.1}$$

where

$$g(\varepsilon, A) = \begin{cases}(1+\varepsilon)A & A \geq 0\\ (1-\varepsilon)A & A < 0.\end{cases} \tag{7.2}$$

Positive Advantage: Suppose the advantage for that state-action pair is positive, in which case its contribution to the objective reduces to

$$L(s,a,\theta_k,\theta) = \min\left(\frac{\pi_\theta\left(a|s\right)}{\pi_{\theta_{\mathrm{k}}}\left(a|s\right)}, 1+\varepsilon\right)A^{\pi_{\theta_k}}(s,a). \tag{7.3}$$

Because the advantage is positive, the objective will increase if the action becomes more likely that is, if $\pi_\theta(a|s)$ increases. But the min in this term puts a limit to how much the objective can increase. Once $\pi_\theta(a|s) > (1+\epsilon)\pi_{\theta_k}(a|s)$, the min kicks in and this term hits a ceiling of $(1+\epsilon)A^{\pi_{\theta_k}}(s,a)$. Thus, the new policy does not benefit by going far away from the old policy.

Negative Advantage: Suppose the advantage for that state-action pair is negative, in which case its contribution to the objective reduces to

$$L(s,a,\theta_k,\theta) = \min\left(\frac{\pi_\theta\left(a|s\right)}{\pi_{\theta_{\mathrm{k}}}\left(a|s\right)}, 1-\varepsilon\right)A^{\pi_{\theta_k}}(s,a). \tag{7.4}$$

Because the advantage is negative, the objective will increase if the action becomes less likely—that is, if $\pi_\theta(a|s)$ decreases. But the max in this term puts a limit to how much the objective can increase. Once $\pi_\theta(a|s) < (1-\epsilon)\pi_{\theta_k}(a|s)$, the max kicks in and this term hits a ceiling of $(1-\epsilon)A^{\pi_{\theta_k}}(s,a)$. Thus, again, the new policy does not benefit by going far away from the old policy.

What we have seen so far is that clipping serves as a regularizer by removing incentives for the policy to change dramatically, and the hyperparameter ϵ corresponds to how far away the new policy can go from the old while still profiting the objective.

Chapter 8

Search Algorithm

Reference: John Levine

8.1 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) is a search technique in the field of Artificial Intelligence (AI). It is a probabilistic and heuristic driven search algorithm that combines the classic tree search implementations alongside machine learning principles of reinforcement learning.

It consists of four phases:

- Tree traversal
- Node expansion
- Rollout
- Backpropagation

In tree search, there's always the possibility that the current best action is actually not the most optimal action. In such cases, MCTS algorithm becomes useful as it continues to evaluate other alternatives periodically during the learning phase by executing them, instead of the current perceived optimal strategy. This is known as the "exploration-exploitation trade-off". It exploits the actions and strategies that is found to be the best till now but also must continue to explore the local space of alternative decisions and find out if they could replace the current best.

Exploration helps in exploring and discovering the unexplored parts of the tree, which could result in finding a more optimal path. In other words, we can say that exploration expands the tree's breadth more than its depth. Exploration can be useful to ensure that MCTS is not overlooking any potentially better paths. But it quickly becomes inefficient in situations with large number of steps or repetitions. In order to avoid that, it is balanced out by exploitation. Exploitation sticks to a single path that has the greatest estimated value. This is a greedy approach and this will extend the tree's depth more than its breadth. In simple words, UCB formula applied to trees helps to balance the exploration-exploitation trade-off by periodically exploring relatively unexplored nodes of the tree and discovering potentially more optimal paths than the one it is currently exploiting.

For this characteristic, MCTS becomes particularly useful in making optimal decisions in Artificial Intelligence (AI) problems.

8.1.1 Tree Traversal

In this process, the MCTS algorithm traverses the current tree from the root node using a specific strategy. The strategy uses an evaluation function to optimally select nodes with the highest estimated value. MCTS uses the Upper Confidence Bound (UCB) formula applied to trees as the strategy in the selection process to traverse the tree. It balances the exploration-exploitation trade-off. During tree traversal, a node is selected based on some parameters that return the maximum value. The parameters are characterized by the formula that is typically used for this purpose is given below:

$$\mathrm{UCB1}(S_i) = \underbrace{\bar{V_i}}_{\textit{Exploitation}} + \underbrace{C\sqrt{\frac{\ln N}{n_i}}}_{\textit{Exploration}},$$

- C is a balancing factor between exploitation and exploration.
- N the number of visit of parent node
- n_k the number of visit of node k

Let's say there are two successor nodes. One is visited more times than another one. Then, it means it is exploited more times than the other one. Thus, exploration term of the less exploited one would be higher than the highly visited one. By computing the UCB1 socre, the agent chooses a successor node with higher UCB1 score.

8.1.2 Expanasion

Expand successor node

8.1.3 Rollout (Random Simulation)

Simulation is completely random. In other words, we don't know how an agent reacts to an environment, so each successor state, the agent randomly decides which action to do till the termination.

8.1.4 Backpropagation

Backpropagate rewards (value), and the number of visit at a node.

- t: total score
- $n_k := n_k + 1$

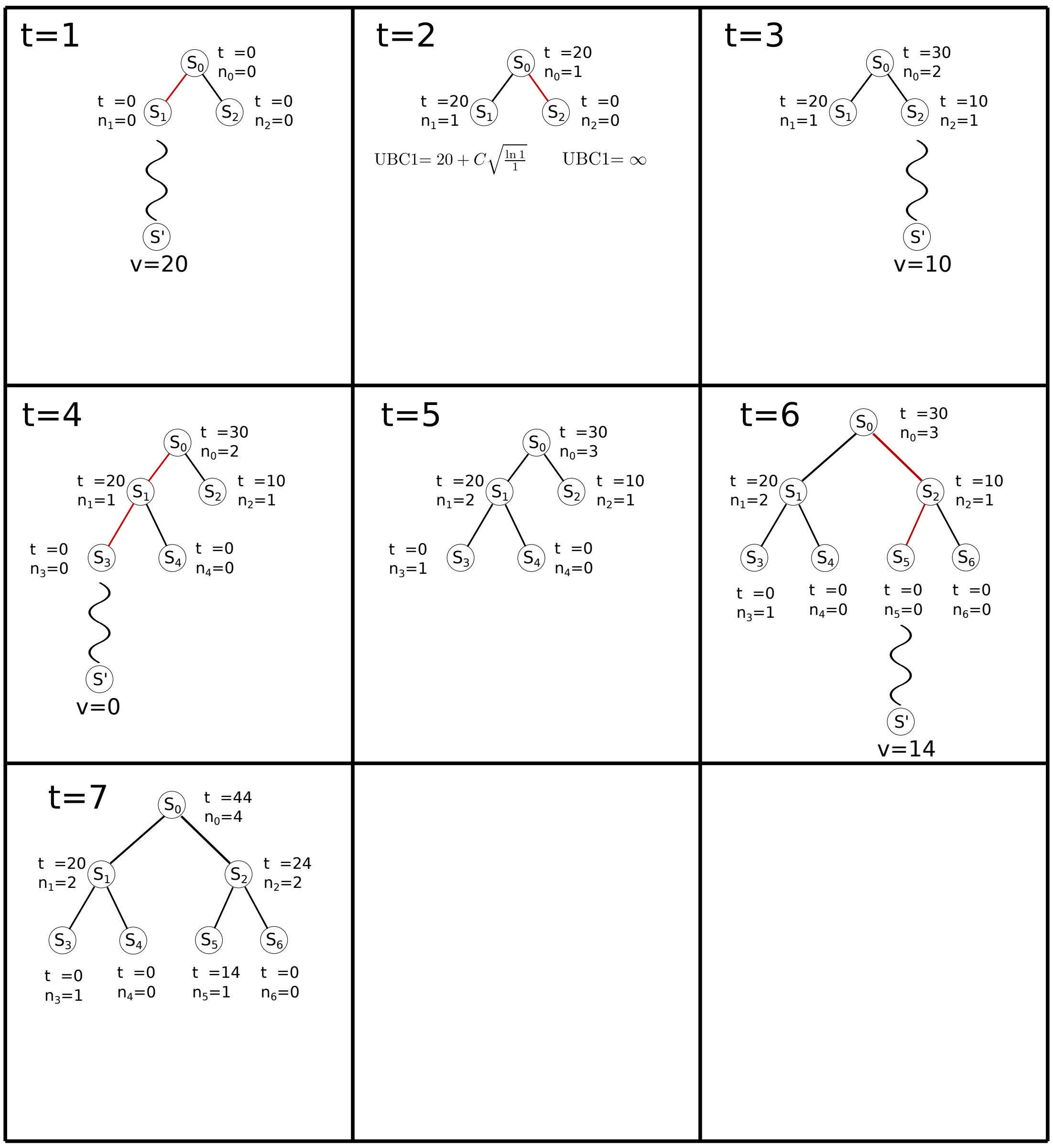
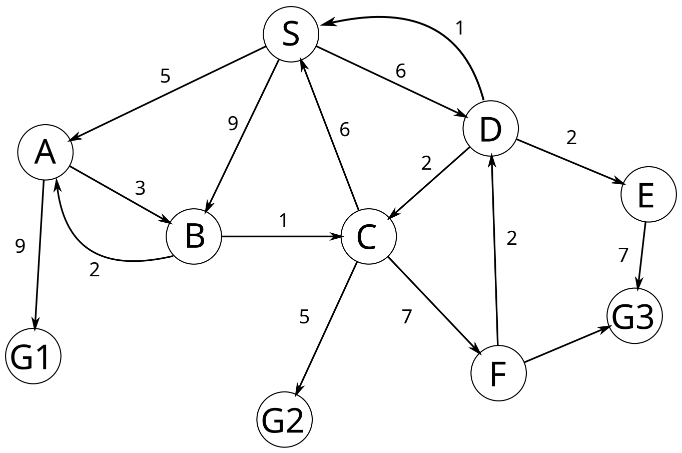


Figure 8.1: MCTS example

8.2 Uniform Cost Search

You have three goal states, G_1, G_2, G_3. Your goal is to reach one of them. UCS is cheapest first search algorithm.



8.3 A^* Search

A^* search algorithms is a variant of Dijkstra's algorithm, which incorporate additional *heuristic* information about a graph. Unlike the Dijkstra's algorithm, you can specify a target node that you are interested in so that you don't have to compute all paths.

In addition, the A^* algorithm is often preferred over Dijkstra's algorithm in scenarios where the cost of reaching a goal from a particular node is not only determined by the actual distance traveled but also influenced by a heuristic estimation of the remaining distance to the goal.

As I said, it exploits heuristics (or estimates). For instance, if we want to get a shortest path to a certain location in a map, then a very reasonable choice is to measure a distance roughly by using Euclidean distance. In Fig. 8.2, we first initialize each node in the graph with our estimates.

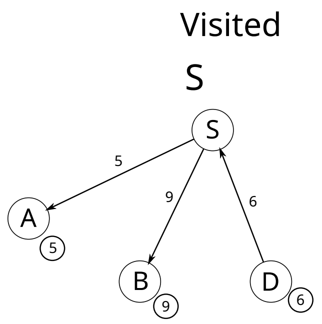
Then, we can explore as shown in §8.3. For each visit, take a shortest path.

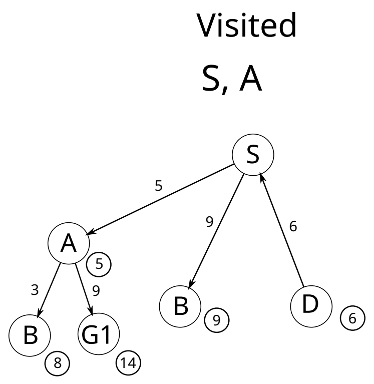
Each number next to the nodes is called A^* score, which is an estimate of cost to get to one of states.

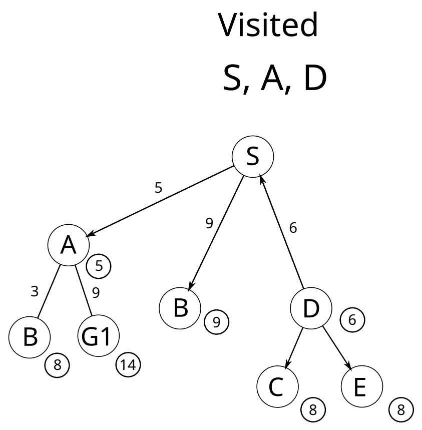
Note that A^* algorithm is typically explained with open set and closed sets, which are just sets for nodes to be evaluated and visited.

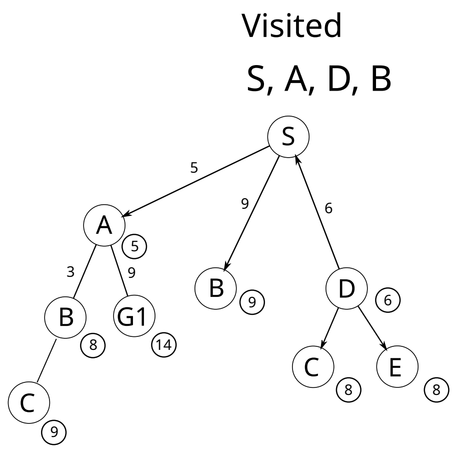
Open Set and Closed Set:

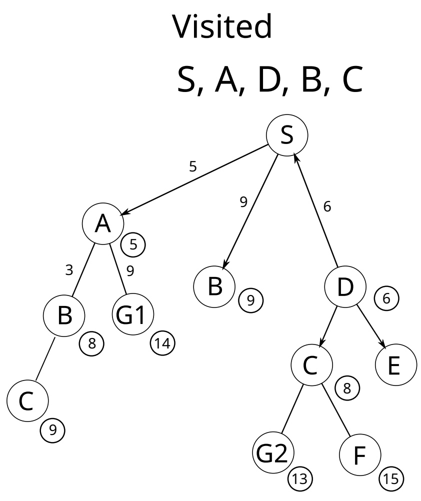
- Open set contains nodes that need to be evaluated,
- Closed set contains nodes that have already been evaluated.

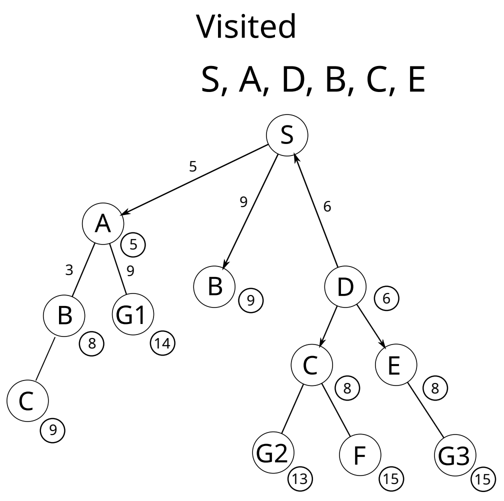

Visited
S


Visited
S, A


Visited
S, A, D


Visited
S, A, D, B


Visited
S, A, D, B, C


Visited
S, A, D, B, C, E

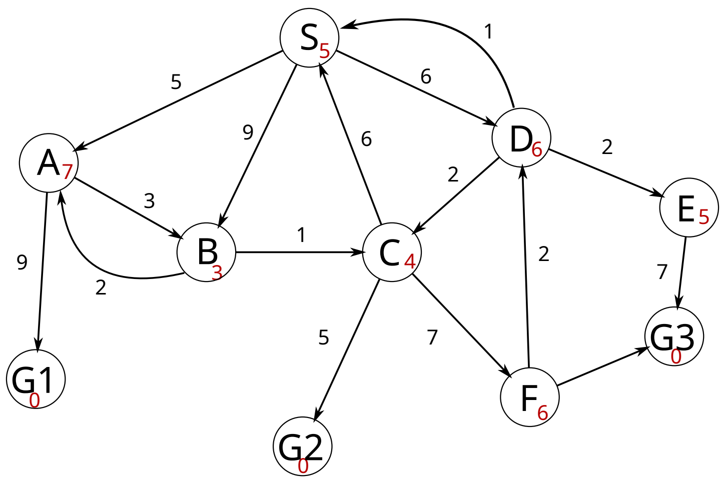
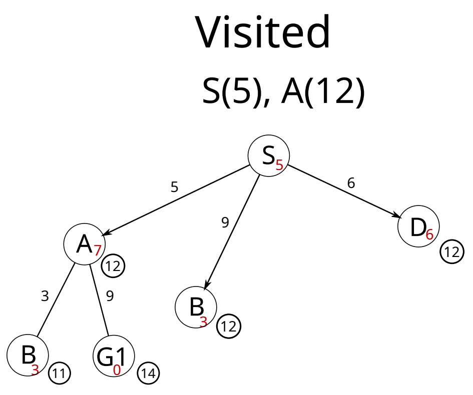
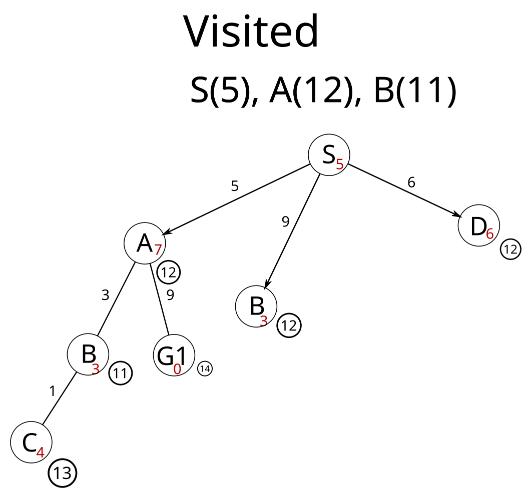
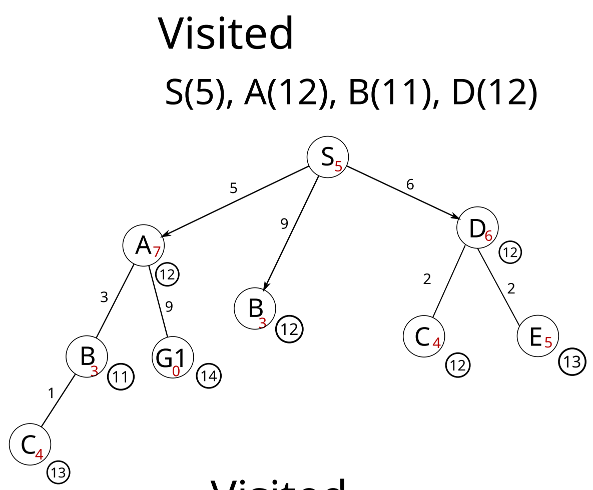
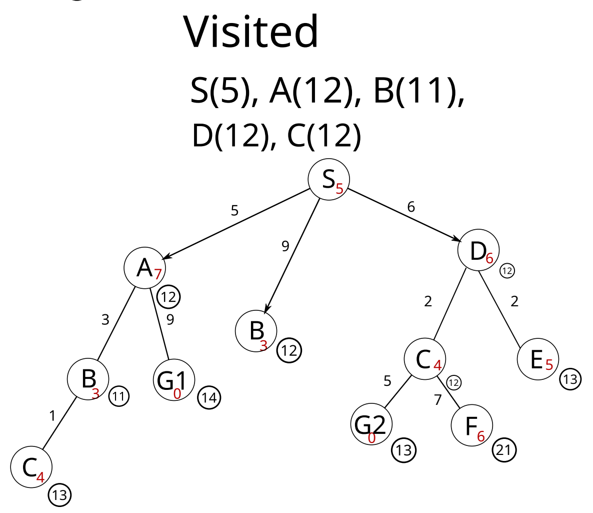


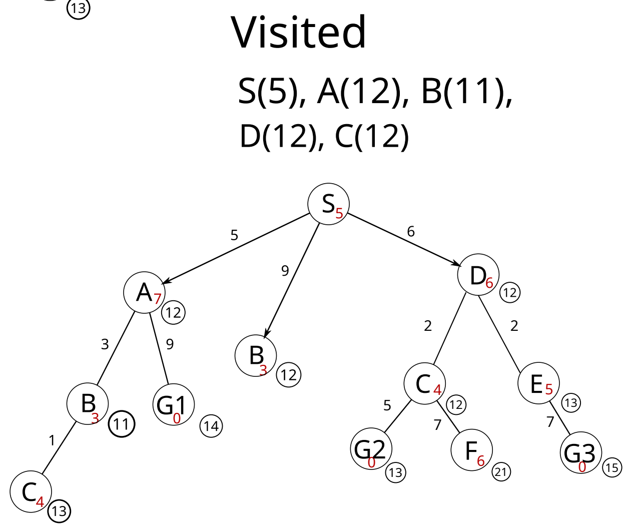
Figure 8.2: Initialize A^* search algorithm with heuristics.


Visited
S(5), A(12)


Visited
S(5), A(12), B(11)


Visited
S(5), A(12), B(11), D(12)


Visited
S(5), A(12), B(11),
D(12), C(12)


Visited
S(5), A(12), B(11),
D(12), C(12)

Bibliography

Appendix

A.1 Bellman Equation

Bellman equation can be derived as follows:

$$
\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] \\
&= \mathbb{E}_\pi[R_{t+1}|S_t = s] + \mathbb{E}_\pi[\gamma G_{t+1}|S_t = s], \quad \text{By Linearity of Expectation.} \\
&= \sum_{r_{t+1}} r_{t+1} P(R_{t+1}|S_t = s) + \mathbb{E}_\pi[\gamma G_{t+1}|S_t = s] \\
&= \sum_{r_{t+1}} r_{t+1} \sum_a P(R_{t+1}|S_t = s, A_t = a) P(A_t = a|S_t = s) + \mathbb{E}_\pi[\gamma G_{t+1}|S_t = s] \\
&= \sum_a \sum_{r_{t+1}} r_{t+1} \sum_{s'} P(R_{t+1}, S_{t+1} = s'|S_t = s, A_t = a) P(A_t = a|S_t = s) + \mathbb{E}_\pi[\gamma G_{t+1}|S_t = s] \\
&= \sum_a \sum_r r \sum_{s'} P(s', r|s, a) \pi(a|s) + \mathbb{E}_\pi[\gamma G_{t+1}|S_t = s] \\
&= \sum_a \sum_{s'} \sum_r r P(s', r|s, a) \pi(a|s) + \mathbb{E}_\pi[\gamma G_{t+1}|S_t = s] \\
&= \sum_a \sum_{s'} \mathcal{P}^a_{ss'} \mathcal{R}^a_{ss'} \pi(a|s) + \gamma \mathbb{E}_\pi[G_{t+1}|S_t = s] \\
&= \sum_a \sum_{s'} \mathcal{P}^a_{ss'} \mathcal{R}^a_{ss'} \pi(a|s) + \gamma \sum_a \mathbb{E}_\pi[G_{t+1}|S_t = s, A_t = a] P(A_t|S_t) \\
&= \sum_a \pi(a|s) \left[\sum_{s'} \mathcal{P}^a_{ss'} \mathcal{R}^a_{ss'} + \gamma \mathbb{E}_\pi[G_{t+1}|S_t = s, A_t = a] \right] \\
&= \sum_a \pi(a|s) \left[\sum_{s'} \mathcal{P}^a_{ss'} \mathcal{R}^a_{ss'} + \gamma \sum_{g_{t+1}} g_{t+1} P(G_{t+1}|S_t = s, A_t = a) \right] \\
&= \sum_a \pi(a|s) \left[\sum_{s'} \mathcal{P}^a_{ss'} \mathcal{R}^a_{ss'} + \gamma \sum_{g_{t+1}} g_{t+1} \frac{P(G_{t+1}, S_t = s, A_t = a)}{P(S_t = s, A_t = a)} \right] \\
&= \sum_a \pi(a|s) \left[\sum_{s'} \mathcal{P}^a_{ss'} \mathcal{R}^a_{ss'} + \gamma \sum_{g_{t+1}} g_{t+1} \frac{\sum_{s'} P(G_{t+1}, S_t = s, S_{t+1} = s', A_t = a)}{P(S_t = s, A_t = a)} \right] \\
&= \sum_a \pi(a|s) \left[\sum_{s'} \mathcal{P}^a_{ss'} \mathcal{R}^a_{ss'} + \gamma \sum_{g_{t+1}} g_{t+1} \frac{\sum_{s'} P(G_{t+1}|s, s', a) P(s, s', a)}{P(s, a)} \right] \\
&= \sum_a \pi(a|s) \left[\sum_{s'} \mathcal{P}^a_{ss'} \mathcal{R}^a_{ss'} + \gamma \sum_{g_{t+1}} g_{t+1} \sum_{s'} P(G_{t+1}|s, s', a) P(s'|s, a) \right] \\
&= \sum_a \pi(a|s) \left[\sum_{s'} \mathcal{P}^a_{ss'} \mathcal{R}^a_{ss'} + \gamma \sum_{s'} P(s'|s, a) \sum_{g_{t+1}} g_{t+1} P(G_{t+1}|s') \right] \quad \text{by Markov Property} \\
&= \sum_a \pi(a|s) \left[\sum_{s'} \mathcal{P}^a_{ss'} \mathcal{R}^a_{ss'} + \gamma \sum_{s'} \mathcal{P}^a_{ss'} v_\pi(s') \right] \\
&= \sum_a \pi(a|s) \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma v_\pi(s') \right] \\
&= \sum_a \pi(a|s) \sum_{r,s'} p(s', r|s, a) \Big[r + \gamma v_\pi(s') \Big]
\end{aligned}
$$

Or simply,

$$\begin{aligned} v_\pi(s) &= \sum_a \pi(a|s) q(s,a) \\ &= \sum_a \pi(a|s) \sum_{r,s'} p(s',r|s,a)\Big[r + \gamma v_\pi(s')\Big] \end{aligned}$$

Reference

Similarly,

$$\begin{aligned} q_\pi(s,a) &= \mathbb{E}_\pi[G_t|S_t=s, A_t=a] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t=s, A_t=a] \\ &= \mathbb{E}_\pi[R_{t+1}|S_t=s, A_t=a] + \gamma \mathbb{E}_\pi[G_{t+1}|S_t=s, A_t=a] \\ &= \sum_r r p(r|s,a) + \gamma \mathbb{E}_\pi[G_{t+1}|S_t=s, A_t=a] \\ &= \sum_r r \sum_{s'} p(s',r|s,a) + \gamma \mathbb{E}_\pi[G_{t+1}|S_t=s, A_t=a] \\ &= \sum_{s',r} r p(s',r|s,a) + \gamma \mathbb{E}[\mathbb{E}_\pi[G_{t+1}|S_t=s, A_t=a, R_{t+1}, S_{t+1}]] \quad \text{By Law of Total Expectation.} \\ &= \sum_{s',r} r p(s',r|s,a) + \gamma \sum_{s',r} \mathbb{E}_\pi[G_{t+1}|S_t=s, A_t=a, R_{t+1}=r, S_{t+1}=s'] p(s',r|s,a) \\ &= \sum_{s',r} p(s',r|s,a)[r + \gamma \mathbb{E}_\pi[G_{t+1}|S_t=s, A_t=a, R_{t+1}=r, S_{t+1}=s'] \\ &= \sum_{s',r} p(s',r|s,a)[r + \gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1}=s'] \quad \text{By Markov Property.} \\ &= \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')] \end{aligned}$$

B.2 Importance Sampling

You have two distributions, $P(A)$ and $P(B)$, and you have a sequence sampled from A. You can estimate an expectation of A,

$$\mathbb{E}[A] = \sum p(a)h(a).$$

Can we use the above equation for estimating an expectation of B? Yes.

$$\mathbb{E}[B] = \sum \frac{p(b)}{p(a)} h(a).$$

The ratio $\frac{p(b)}{p(a)}$ tells us how likely to observe some results under $p(b)$ compared to $p(a)$.

Useful factorization of conditional probability:

$$\begin{aligned}
P[A,B|C] &= \frac{P[A,B,C]}{P[C]} && (1)\\
&= \frac{P[A,B,C]}{P[C]}\frac{P[B,C]}{P[B,C]} && (2)\\
&= \frac{P[A,B,C]}{P[B,C]}\frac{P[B,C]}{P[C]} && (3)\\
&= P[A|B,C]P[B|C] && (4)
\end{aligned}$$

C.3 Fisher Information

Suppose we have a model parameterized by parameter vector θ that models a distribution $p(x;\theta)$. In frequentist statistics, the way we learn θ is to maximize the likelihood of $p(x;\theta)$. To assess the goodness of our estimate of θ we define a **score function** as follows:

$$f(\theta) = \nabla_\theta \log p(x;\theta).$$

The expected value of score function is zero.

$$\begin{aligned}
\mathbb{E}_{p(x;\theta)}[f(\theta)] &= \mathbb{E}_{p(x;\theta)}[\nabla_\theta \log p(x;\theta)]\\
&= \int p(x;\theta)\nabla_\theta \log p(x;\theta)dx\\
&= \int p(x;\theta)\frac{\nabla_\theta p(x;\theta)}{p(x;\theta)}dx\\
&= 0
\end{aligned}$$

The covariance of the score function is given by

$$\mathrm{Cov}[f(\theta), f(\theta)] = \mathbb{E}_{p(x;\theta)}[(f(\theta)-0)(f(\theta)-0)^T] = \mathrm{Var}[f(\theta), f(\theta)].$$

This the definition of Fisher information and it can be written

$$F = \mathbb{E}_{p(x;\theta)}[\nabla \log p(x;\theta)\nabla \log p(x;\theta)^T].$$

Empirically,

$$F = \frac{1}{N}\sum_{i=1}^{N} \nabla \log p(x;\theta)\nabla \log p(x;\theta)^T.$$

D.4 Score Function

In statistics, *the score (or informant) is the gradient of the log-likelihood function with respect to the parameter vector.*

- Evaluated at a particular point of the parameter vector, the score indicates the **steepness of the log-likelihood function and thereby the sensitivity to infinitesimal changes to the parameter values**.

- If the log-likelihood function is continuous over the parameter space, the score will **vanish at a local maximum or minimum**; this fact is used in maximum likelihood estimation to find the parameter values that maximize the likelihood function.

Since the score is a function of the observations that are subject to sampling error, it lends itself to a test statistic known as score test in which the parameter is held at a particular value. Further, the ratio of two likelihood functions evaluated at two distinct parameter values can be understood as a definite integral of the score function.[2]

E.5 Incremental Monte-Carlo

Incremental Mean:

$$\begin{aligned}
\mu_k &= \frac{1}{k}\sum_{j=1}^{k} x_j && (5)\\
&= \frac{1}{k}\Big(x_k + (k-1)\frac{1}{(k-1)}\sum_{j=1}^{k-1} x_j\Big) && (6)\\
&= \frac{1}{k}\Big(x_k + (k-1)\mu_{k-1}\Big) && (7)\\
&= \mu_{k-1} + \frac{1}{k}(x_k - \mu_{k-1}) && (8)
\end{aligned}$$

Incremental MC:

- $N_{n+1}(S_t^n) = N_n(S_t^n) + 1$ for rest $N_{n+1}(s) = N_n(s)$
- $V_{n+1}(S_t) = V_n(S_t) + \frac{G_{t:T}^n - V_n(S_t)}{N_n(S_t)}$ for rest $V_{n+1}(s) = V_n(s)$
- $V_{n+1}(S_t) = V_n(S_t) + \alpha(G_{t:T}^n - V_n(S_t))$ for rest $V_{n+1}(s) = V_n(s)$

F.6 Derivative of Softmax

Softmax function is given by

$$S(x_i) = \frac{e^{x_i}}{\sum_{k=1}^{K} e^{x_k}} \qquad \text{for } i = 1, \dots, K$$

The derivative of softmax function is

$$\frac{\partial S_i}{\partial x_j} = \begin{cases} S_i(1 - S_j) & \text{if } i = j \\ -S_j S_i & \text{if } i \neq j \end{cases}$$

- Diagoal elements: $S_i(1 - S_j)$
- Off-diagonal elements: $-S_j S_i$:

The Jacobian matrix ($j \times i$) for softmax is

$$\frac{\partial S}{\partial x} = \begin{bmatrix} \frac{\partial S_1}{\partial x_1} & \frac{\partial S_1}{\partial x_2} & \dots & \frac{\partial S_1}{\partial x_K} \\ \frac{\partial S_2}{\partial x_1} & \frac{\partial S_2}{\partial x_2} & \dots & \frac{\partial S_K}{\partial x_K} \\ \vdots & \vdots & \dots & \vdots \\ \frac{\partial S_K}{\partial x_1} & \frac{\partial S_K}{\partial x_2} & \dots & \frac{\partial S_K}{\partial x_K} \end{bmatrix}$$

The matrix can be expressed as follows:

$$\begin{bmatrix} S(x_1) - S(x_1)S(x_1) & \dots & 0 - S(x_1)S(x_N) \\ \dots & S(x_j) - S(x_j)S(x_i) & \dots \\ 0 - S(x_N)S(x_1) & \dots & 0 - S(x_N)S(x_N) \end{bmatrix} =$$
$$\begin{bmatrix} S(x_1) & \dots & 0 \\ \dots & S(x_j) & \dots \\ 0 & \dots & S(x_N) \end{bmatrix} - \begin{bmatrix} S(x_1)S(x_1) & \dots & S(x_1)S(x_N) \\ \dots & S(x_j)S(x_i) & \dots \\ S(x_N)S(x_1) & \dots & S(x_N)S(x_N) \end{bmatrix}$$

This can be

```
np.diag(S) - np.outer(S, S)
```

G.7 Policy Gradient Theorem

Function approximation is essential to reinforcement learning, but the standard approach of approximating a value function and determining a policy from it has so far proven theoretically intractable. In this paper we explore an alternative approach in which the policy is explicitly represented by its own function approximator, independent of the value function, and is updated according to the gradient of expected reward with respect to the policy parameters. Williams's REINFORCE method and actor-critic methods are examples of this approach.

We consider the standard reinforcement learning framework, in which a learning agent interacts with a Markov decision process (MDP).

- The state, action, and reward at each time $t \in \{0, 1, 2, \cdots\}$ are denoted $s_t \in \mathcal{S}$, $a_t \in \mathcal{A}$, and $r_t \in \mathbb{R}$.
- The environment's dynamics are characterized by state transition probabilities and expected rewards:

$$\begin{aligned} \mathcal{P}^a_{ss'} &= P(s_{t+1} = s' | s_t = s, a_t = a) \\ \mathcal{R}^a_s &= \mathbb{E}(r_{t+1} | s_t = s, a_t = a), \forall s, s' \in \mathcal{S}, a \in \mathcal{A} \end{aligned}$$

With function approximation, two ways of formultaing the agent's objective are useful: One is the average reward formulation, in which policies are ranked according to their long-term expected reward per-step, $\rho(\pi)$:

$$\rho(\pi) = \lim_{n \to \infty} \frac{1}{n} \mathbb{E}[r_1 + r_2, \cdots, +r_n | \pi] = \sum_s d^\pi(s) \sum_a \pi(a|s) \mathcal{R}^a_s,$$

where $d^{\pi}(s) = \lim_{t\to\infty} P(s_t = s|s_0, \pi)$ is the stationary distribution of states under π, which we assume exists and is independent of s_0 for all policies. Imagine that you can travel along the Markov chain's states forever, and eventually, as the time progresses, the probability of you ending up with one state becomes unchanges. This is the stationary probability that the $s_t = s$ when starting from s_0 and following policy π_θ for t steps. With the average reward formultion, the state-action value function is defined as

$$Q^{\pi}(s,a) = \sum_{t=1}^{\infty} \mathbb{E}[r_t - \rho(\pi)|s_0 = s, a_0 = a, \pi], \forall s \in \mathcal{S}, a \in \mathcal{A}.$$

G.7.1 Proof of Policy Gradient Theorem

$$\begin{aligned} J(\theta) &= \sum_{s\in\mathcal{S}} d^{\pi}(s) V^{\pi}(s) \\ &= \sum_{s\in\mathcal{S}} d^{\pi}(s) \sum_{a\in\mathcal{A}} \pi_\theta(a|s) Q^{\pi}(s,a), \end{aligned}$$

where $d_\pi(s)$ is the stationary distribution of Markov chain for π_θ (on-policy state distribution under π). Imagine that you can travel along the Markov chain's states forever, and eventually, as the time progresses, the probability of you ending up with one state becomes unchanged, this is the stationary probability for π_θ.

$$\begin{aligned} \nabla_\theta V^{\pi}(s) &= \nabla_\theta \sum_{a\in\mathcal{A}} \pi_\theta(a|s) Q^{\pi}(s,a) \\ &= \sum_{a\in\mathcal{A}} \nabla_\theta \pi_\theta(a|s) Q^{\pi}(s,a) + \pi_\theta(a|s) \nabla_\theta Q^{\pi}(s,a) \\ &= \sum_{a\in\mathcal{A}} \nabla_\theta \pi_\theta(a|s) Q^{\pi}(s,a) + \pi_\theta(a|s) \nabla_\theta \sum_{s',r} P(s',r|s,a)(r + V^{\pi}(s')) \\ &= \sum_{a\in\mathcal{A}} \nabla_\theta \pi_\theta(a|s) Q^{\pi}(s,a) + \pi_\theta(a|s) \sum_{s',r} P(s',r|s,a) \nabla_\theta V^{\pi}(s') \quad P(s',r|s,a) \text{ and } r \text{ is not a function of } \theta \\ &= \sum_{a\in\mathcal{A}} \nabla_\theta \pi_\theta(a|s) Q^{\pi}(s,a) + \pi_\theta(a|s) \sum_{s'} P(s'|s,a) \nabla_\theta V^{\pi}(s') \end{aligned}$$

This equation has a recursive from. Let's consider a visitation sequence and transition probability from state s to state x with polict π_θ after k steps asr:

$$\rho^{\pi}(s \to x, k)$$

- This is a state transition probability with a policy π_θ
- When $k = 0$, $\rho^{\pi}(s \to s, k = 0) = 1$
- When $k = 1$, $\rho^{\pi}(s \to s', k = 1) = \sum_a \pi_\theta(a|s) P(s'|s,a)$
- $\rho^{\pi}(s \to x, k+1) = \sum_{s'} \rho^{\pi}(s \to s', k) \rho^{\pi}(s' \to x, 1)$, where s' is the step right behind the state x (intermediate step).

$$
\begin{aligned}
\nabla_\theta V^\pi(s) &= \underbrace{\sum_{a\in\mathcal{A}} \nabla_\theta \pi_\theta(a|s) Q^\pi(s,a)}_{\doteq \phi(s)} + \pi_\theta(a|s) \sum_{s'} P(s'|s,a) \nabla_\theta V^\pi(s') \\
&= \phi(s) + \sum_a \pi_\theta(a|s) \sum_{s'} P(s'|s,a) \nabla_\theta V^\pi(s') \\
&= \phi(s) + \sum_{s'} \sum_a \pi_\theta(a|s) P(s'|s,a) \nabla_\theta V^\pi(s') \\
&= \phi(s) + \sum_{s'} \rho^\pi(s \to s', k=1) \nabla_\theta V^\pi(s') \\
&= \phi(s) + \sum_{s'} \rho^\pi(s \to s', k=1) \left[\phi(s') + \sum_{s''} \rho^\pi(s' \to s'', k=1) \nabla_\theta V^\pi(s'') \right] \\
&= \phi(s) + \sum_{s'} \rho^\pi(s \to s', 1)\phi(s') + \sum_{s''} \sum_{s'} \rho^\pi(s \to s', 1) \rho^\pi(s' \to s'', k=1) \nabla_\theta V^\pi(s'') \\
&= \phi(s) + \sum_{s'} \rho^\pi(s \to s', 1)\phi(s') + \sum_{s''} \rho^\pi(s \to s'', 2) \nabla_\theta V^\pi(s'') \\
&\quad \vdots \\
&= \sum_{x\in\mathcal{S}} \sum_{k=0}^{\infty} \rho^\pi(s \to x, k)\phi(x)
\end{aligned}
$$

We can rewrite the above equation as

$$
\begin{aligned}
\nabla_\theta J(\theta) &= \nabla_\theta V^\pi(s) \\
&= \sum_{x\in\mathcal{S}} \underbrace{\sum_{k=0}^{\infty} \rho^\pi(s_0 \to s, k)}_{\doteq \eta(s)} \phi(x) \\
&= \sum_s \eta(s)\phi(s) \\
&= \underbrace{\sum_s \eta(s)}_{\text{Constant}} \sum_s \underbrace{\frac{\eta(s)}{\sum_s \eta(s)}}_{\text{Normalization}} \phi(s) \\
&\propto \sum_s \frac{\eta(s)}{\sum_s \eta(s)} \phi(s) \\
&= \sum_s d^\pi(s) \sum_{a\in\mathcal{A}} \nabla_\theta \pi_\theta(a|s) Q^\pi(s,a) \\
&= \mathbb{E}_\pi[\nabla_\theta \ln \pi_\theta(a|s) Q^\pi(s,a)]
\end{aligned}
$$