Reinforcement Learning

SoS Report

Shiv Narang
23B1075
Mentor: Mr. Satush Parikh

Contents

Plan of Action

- Week 1:Introducon to mul-armed Bandits ,Coin-tossing game ,-first and -greedy algorithms.Graph of E[r] versus t;Definion of regret,GLIE sampling
- Week 2:UCB,KL-UCB,Thompson Sampling algorithms,Hoeffding's Inequality ,"KL" Inequality
- Week 3:Markov Decision Problems,Upper bound regret of UCB; Opmal policy,MDP planning problem,Bellman equaons
- Week 4:Connuing and episodic tasks,Infinite discounted,applicaon of MDPs,Banach's Fixed-point Theorem;Bellman opmality.
- Week 5:Linear programming and it's applicaons to MDP planning,Acon value funcon,Policy Improvements,Bellman operator,Proof of policy
- Week 6:Complexity bounds for MDP planning,Howard's PI and Batch switching PI,The reinforcement learning problem
- Week 7:Predicon and control problems,Ergodic MDPs,Model based algorithm for acng opmally in the limit
- Week 8:Monte Carlo methods for predicon,Maximum likelihood esmates and least squares esmates,Bootstrapping
- Week 9:Tile coding Control with funcon approximaons,Decision me Planning

Althpugh this plan of action was tentative and i dont know initially what are major topics so i cover these topics in depth till midsem

- Introduction to the topic
- Multi-armed Bandits
- markov Decision process
- The Agent-Environment Interface
- Monte Carlo Methods
- Dynamic Programming

1 Introduction to the topic

Perhaps the idea that we learn by interacting with our environment takes precedence For us if we think about what learning looks like. When a baby plays, waves, or looks around, has no obvious teacher, but has a direct sensorimotor connection with the environment. Using this relationship provides a wealth of information about cause and effect, the consequences of actions and what should be done about them Goals are achieved. Without a doubt, such communication is an important part of our lives knowledge of our environment and ourselves.

Reinforcement learning is learning what needs to be done - how to impose conditions on tasks - in order to maximize the statistical reward signal. The student is not told what actions to perform, but discovers actions that are more useful with effort

1.1 What is it?

Reinforcement gaining knowledge of is different from supervised studying that's gaining knowledge of from a schooling set of categorised examples supplied via a knowledgeable outside manager. The object of this sort of gaining knowledge of is for the machine to extrapolate, or generalize, its responses so that it acts efficiently in situations no longer gift within the schooling set. In interactive issues it is frequently impractical to gain examples of desired behavior that are both accurate and consultant of all the conditions wherein the agent has to act. In uncharted territory—where one would count on getting to know to be maximum useful—an agent have to be able to examine from its own revel in.

Reinforcement gaining knowledge of is likewise distinct from what machine getting to know researchers call unsupervised studying, which is normally approximately finding structure hidden in collections of unlabeled records. The phrases supervised mastering and unsupervised mastering could seem to exhaustively classify system getting to know paradigms, but they do no longer.

One of the demanding situations that stand up in reinforcement mastering, and not in other types of studying, is textbfthe alternate-off between exploration and exploitation. To reap quite a few praise, a reinforcement studying agent ought to decide upon actions that it has attempted in the past and determined to be effective in generating praise. But to find out such moves, it has to strive actions that it has now not decided on earlier than. The agent has to emphtake advantage of what it has already experienced so that it will acquire reward, however it additionally has to emphdiscover to be able to make better motion alternatives within the future.

Another key characteristic of reinforcement gaining knowledge of is that it explicitly considers the complete problem of a goal-directed agent interacting with an unsure environment. This is in contrast to many tactics that keep in mind subproblems

1.2 History of Reinforcement Learning

The history of reinforcement learning has two main threads, both long and rich, that were pursued independently before intertwining in modern reinforcement learning. One thread concerns learning by trial and error that started in the psychology of animal learning. This thread runs through some of the earliest work in artificial intelligence and led to the revival of reinforcement learning in the early 1980s. The other thread concerns the problem of

optimal control and its solution using value functions and dynamic programming. For the most part, this thread did not involve learning. Although the two threads have been largely independent, the exceptions revolve around a third, less distinct thread concerning temporal-difference methods such as used in the tic-tac-toe example in this chapter. All three threads came together in the late 1980s to produce the modern field of reinforcement learning as we present it in this book. The thread focusing on trial-and-error learning is the one with which we are most familiar and about which we have the most to say in this brief history. Before doing that, however, we briefly discuss the optimal control thread.

1.3 Examples

Some of the possible applications where Reinforcement Learning can be applied are:

- A master chess player makes a move. The choice is informed both by planning—anticipating possible replies and counter-replies and by immediate, intuitive judgements of the desirability of particular positions and moves.
- A gazelle calf struggles to its feet minutes after being born. Half an hour later it is running at 20 miles per hour.
- A mobile robot decides whether it should enter a new room in search of more trash to collect or start trying to find its way back to its battery recharging station. It makes its decision based on the current charge level of its battery and how quickly and easily it has been able to find the recharger in the past.

1.4 Components of RL

In addition to agent and environment, four main sub-components of the reinforcement learning process can be identified: structure, reward signals, objective functions, and in other words, environment example of encounter

- **Policy**: A policy describes the way a subject user behaves at a given time. It is a map from perceived environmental conditions to the actions to be taken when in those conditions.
- **Compensation**: The compensation rule describes the objectives of the energy learning problem. At each time step, the environment sends a number as a reward to the reinforcing learning. The agent's sole objective is to maximize the total reward in the long run.
- **Value Function**: The value function determines what is optimal in the long run. The state value is the amount of reward that the agent can accumulate in the future starting from that situation.
  The primary reward is directly provided by the environment, but it results from sequentially calculated and recalculated values sequence of an agent observes in its entirety life.
- **Model**: This is something that simulates the behavior of the environment, or more generally, it can be theories about how the environment will behave. Models are used for planning, in which we determine each course of action by considering possible future scenarios before actually experiencing them

	AI Planning	SL	UL	RL	IL
Optimization	Y			Y	Y
Learns from experience		Y	Y	Y	Y
Generalization	Y	Y	Y	Y	Y
Delayed Consequences	Y			Y	Y
Exploration				Y	

Table 1: Different Machine Learning Paradigms

1.5 How is RL different from other ml

Reinforcement Learning is different from other Machine Learning types. This will become clear based on the following factors as is shown in the table

Let us go through the terms used in the table briefly:

- **SL:** Supervised Learning
- **UL:** Unsupervised Learning
- **IL:** Imitation learning
- **Optimization:** Goal is to find an optimal way to make decisions.
- **Delayed Consequences:** Decisions can impact things later.
- **Exploration:** Learning about the things by making decisions.

2 Multi-armed Bandit Problem

In this problem, you repeatedly face a choice among k different options or actions. After each choice, you receive a numerical reward drawn from a stationary probability distribution dependent on the action selected. The objective is to maximize the expected total reward over a specified time period.

We denote the action selected at time step t as A_t and the corresponding reward as R_t. The value of an arbitrary action a, denoted $q_*(a)$, is the expected reward given that a is selected:

$$q_*(a) = E[R_t \mid A_t = a]$$

We denote the estimated value of action a at time step t as $Q_t(a)$.

2.1 Estimating Action Values

One common method to estimate the value of an action is by averaging the rewards actually received when that action was taken:

$$Q_t(a) = \frac{\text{sum of rewards when } a \text{ was taken prior to } t}{\text{number of times } a \text{ was taken prior to } t} = \frac{\sum_{i=1}^{t-1} R_i \cdot \mathbb{1}_{A_i=a}}{\sum_{i=1}^{t-1} \mathbb{1}_{A_i=a}} \tag{1}$$

where $\mathbb{1}_{\text{predicate}}$ is an indicator function that is 1 if the predicate is true and 0 otherwise.

2.2 Greedy Action Selection

The simplest action selection rule is to choose one of the actions with the highest estimated value. This greedy approach is expressed as:

$$A_t = \arg\max_a Q_t(a) \tag{2}$$

where $\arg\max_a$ denotes the action a that maximizes the expression.

2.3 ϵ-Greedy Action Selection

An alternative to the greedy method is the ϵ-greedy approach, where most of the time the action with the highest estimated value is chosen, but with a small probability ϵ, a random action is selected. This helps ensure exploration of all actions.

2.4 Incremental Estimation Techniques

Incremental methods allow for efficient updating of estimates. The general form of an incremental implementation is:

`NewEstimate` $\leftarrow$ `OldEstimate + StepSize [Target - OldEstimate]`

Figure 1 shows pseudocode for a complete bandit algorithm using incrementally computed sample averages and ϵ-greedy action selection.

A simple bandit algorithm

Initialize, for $a = 1$ to k:
$Q(a) \leftarrow 0$
$N(a) \leftarrow 0$

Loop forever:
$A \leftarrow \begin{cases} \arg\max_a Q(a) & \text{with probability } 1-\varepsilon \quad \text{(breaking ties randomly)} \\ \text{a random action} & \text{with probability } \varepsilon \end{cases}$
$R \leftarrow bandit(A)$
$N(A) \leftarrow N(A) + 1$
$Q(A) \leftarrow Q(A) + \frac{1}{N(A)}\left[R - Q(A)\right]$

Figure 1: An incremental implementation of a simple bandit algorithm [1]

2.5 Upper Confidence Bound (UCB) Action Selection

A more advanced method for action selection is the Upper Confidence Bound (UCB) approach, which balances exploration and exploitation by considering both the estimated value of actions and the uncertainty in those estimates. Actions are selected according to:

$$A_t = \arg\max_a \left[Q_t(a) + c\sqrt{\frac{\ln t}{N_t(a)}} \right] \tag{3}$$

2.6 Contextual Bandit Problems

Consider a scenario where a slot machine changes the color of its display to reflect changes in action values. By learning a policy that associates colors with actions—for example, selecting arm 1 if the display is red and arm 2 if it is green—we can improve performance.

This type of problem, where the decision-making process depends on additional context or state information, is known as a contextual bandit problem. It involves both trial-and-error learning to determine the best actions and the association of these actions with specific contexts.

2.7 Exploration vs. Exploitation Dilemma

In multi-armed bandit problems, there is a fundamental trade-off between exploration (trying out different actions to gather more information) and exploitation (selecting the best-known action to maximize reward). Effective strategies must balance these two aspects to optimize performance over time.

2.8 Applications of Multi-armed Bandit Algorithms

Multi-armed bandit algorithms have a wide range of applications, including online advertising, clinical trials, adaptive routing, and recommendation systems. Their ability to handle uncertainty and make decisions based on incomplete information makes them valuable in various fields.

3 The Agent-Environment Interface

Markov Decision Processes (MDPs) are a classical framework for sequential decision-making, where actions influence not just immediate rewards but also future states and rewards. MDPs provide a structured approach to the problem of learning from interaction to achieve a goal. The learner and decision-maker is called the agent. The entity it interacts with, comprising everything outside the agent, is called the environment. The environment generates rewards, numerical values that the agent seeks to maximize over time through its actions. This relationship is illustrated in Figure 2.

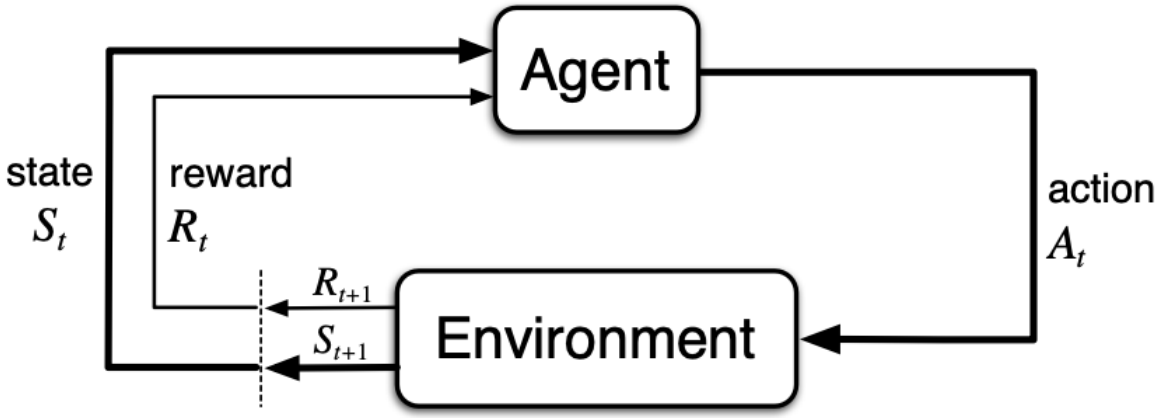

Figure 2: The agent-environment interaction in a Markov Decision Process

3.1 Agent-Environment Interaction

The agent and environment interact at each of a sequence of discrete time steps, $t = 0, 1, 2, 3, \ldots$. At each time step t, the agent receives some representation of the environment's state, $S_t \in S$, and on that basis selects an action, $A_t \in A(s)$.

For particular values of random variables, $s' \in S$ and $r \in R$, there is a probability of those values occurring at time t, given particular values of the preceding state and action:

$$p(s', r \mid s, a) = Pr\{S_t = s', R_t = r \mid S_{t-1} = s, A_{t-1} = a\} \tag{4}$$

3.2 Markov Property

The function p defines the dynamics of the MDP. The state must include all information about past agent-environment interactions that make a difference for the future. If it does, the state is said to have the Markov property.

3.3 Expected Rewards

We can compute the expected rewards for state-action pairs as a two-argument function $r : S \times A \to R$:

$$r(s, a) = E[R_t \mid S_{t-1} = s, A_{t-1} = a] = \sum_{r \in R} r \sum_{s' \in S} p(s', r \mid s, a) \tag{5}$$

and the expected rewards for state-action-next-state triples as a three-argument function $r : S \times A \times S \to R$:

$$r(s, a, s') = E[R_t \mid S_{t-1} = s, A_{t-1} = a, S_t = s'] = \sum_{r \in R} r \frac{p(s', r \mid s, a)}{p(s' \mid s, a)} \tag{6}$$

3.4 Agent-Environment Boundary

The general rule is that anything that cannot be changed arbitrarily by the agent is considered part of the environment. The agent-environment boundary represents the limit of the agent's absolute control, not of its knowledge.

3.5 Return and Episodes

The goal is to maximize the expected return, denoted G_t, defined as a specific function of the reward sequence. In the simplest case, the return is the sum of the rewards:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T \tag{7}$$

where T is the final time step. This is applicable when there is a natural notion of a final time step, known as episodes. Each episode ends in a special state called the terminal state, followed by a reset to a standard starting state.

3.6 Continuing Tasks and Discounting

In many cases, the agent-environment interaction does not break into episodes but continues indefinitely. These are called continuing tasks. The agent maximizes the sum of the discounted rewards received over the future:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{8}$$

where γ is the discount rate, $0 \leq \gamma \leq 1$. If $\gamma = 0$, the agent focuses only on immediate rewards.

3.7 Policies and Value Functions

A policy is a mapping from states to probabilities of selecting each possible action. If the agent is following policy π, $\pi(a \mid s)$ is the probability that $A_t = a$ if $S_t = s$.

The value function of a state s under a policy π, denoted $v_\pi(s)$, is the expected return when starting in s and following π thereafter:

$$v_\pi(s) = E_\pi[G_t \mid S_t = s] = E_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s \right] \tag{9}$$

This is called the state-value function for policy π.

Similarly, the value of taking action a in state s under policy π, denoted $q_\pi(s, a)$, is the expected return starting from s, taking action a, and thereafter following policy π:

$$q_\pi(s, a) = E_\pi[G_t \mid S_t = s, A_t = a] = E_\pi \left[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a \right] \tag{10}$$

This is called the action-value function for policy π.

The Bellman equation for v_π is:

$$v_\pi(s) = \sum_a \pi(a \mid s) \sum_{s',r} p(s', r \mid s, a)[r + \gamma v_\pi(s')] \tag{11}$$

3.8 Optimal Policies and Value Functions

A policy π is considered better than or equal to another policy π' if its expected return is greater than or equal to that of π' for all states. Formally, $\pi \geq \pi'$ if and only if $v_\pi(s) \geq v_{\pi'}(s)$ for all $s \in S$. There exists at least one policy that is better than or equal to all other policies, known as an optimal policy. Although there may be multiple optimal policies, they share the same state-value function, denoted v_*:

$$v_*(s) = \max_\pi v_\pi(s)$$

for all $s \in S$. Optimal policies also share the same optimal action-value function.

4 Markov Decision Processes

A Markov Decision Process (MDP) is a mathematical framework used to describe an environment in decision-making problems where outcomes are partly random and partly under the control of a decision maker. MDPs provide a formalism for modeling sequential decision-making problems where actions taken by an agent affect both the immediate reward and the next state of the system.

4.1 MDP Components

An MDP is defined by the following components:

- **States** (S): A finite set of states representing all possible situations in the environment.
- **Actions** (A): A finite set of actions available to the agent.
- **Transition Probability** (P): A transition probability function $P(s'|s, a)$ which defines the probability of transitioning to state s' from state s after taking action a.
- **Reward** (R): A reward function $R(s, a, s')$ which defines the immediate reward received after transitioning from state s to state s' due to action a.
- **Discount Factor** (γ): A discount factor $\gamma \in [0, 1]$ which models the importance of future rewards.

4.2 Policy

A policy π defines the behavior of an agent, specifying the action a to take in state s. Formally, a policy π is a mapping from states to probabilities of selecting each possible action, $\pi : S \to \mathcal{P}(A)$.

4.3 Value Functions

Value functions are used to evaluate the goodness of states and state-action pairs under a policy. They estimate the expected return (cumulative reward) starting from a given state or state-action pair.

4.3.1 State-Value Function

The state-value function $v_\pi(s)$ under a policy π is defined as the expected return when starting from state s and following policy π thereafter:

$$v_\pi(s) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) \middle| s_0 = s \right] \tag{12}$$

4.3.2 Action-Value Function

The action-value function $q_\pi(s, a)$ under a policy π is defined as the expected return when starting from state s, taking action a, and thereafter following policy π:

$$q_\pi(s, a) = \mathbb{E}_\pi \left[\sum_{t=0}^{\infty} \gamma^t R(s_t, a_t, s_{t+1}) \middle| s_0 = s, a_0 = a \right] \tag{13}$$

4.4 Bellman Equations

The Bellman equations provide a recursive decomposition of the value functions, breaking them down into immediate reward plus the discounted value of the successor state.

4.4.1 Bellman Equation for v_π

The Bellman equation for the state-value function v_π is:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s'} P(s'|s, a)[R(s, a, s') + \gamma v_\pi(s')] \tag{14}$$

4.4.2 Bellman Equation for q_π

The Bellman equation for the action-value function q_π is:

$$q_\pi(s, a) = \sum_{s'} P(s'|s, a)[R(s, a, s') + \gamma \sum_{a'} \pi(a'|s') q_\pi(s', a')] \tag{15}$$

4.5 Optimal Policies

An optimal policy π_* is one that maximizes the expected return from all states. The corresponding optimal state-value function v_* and optimal action-value function q_* are defined as:

$$v_*(s) = \max_\pi v_\pi(s) \tag{16}$$

$$q_*(s, a) = \max_\pi q_\pi(s, a) \tag{17}$$

4.6 Bellman Optimality Equations

The Bellman optimality equations characterize the optimal value functions. For the optimal state-value function v_* and the optimal action-value function q_*, the equations are:

$$v_*(s) = \max_a \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma v_*(s')] \tag{18}$$

$$q_*(s,a) = \sum_{s'} P(s'|s,a)[R(s,a,s') + \gamma \max_{a'} q_*(s',a')] \tag{19}$$

4.7 Solving MDPs

To solve an MDP means to find an optimal policy π_*. This can be done using various algorithms, including dynamic programming methods like policy iteration, value iteration, and asynchronous dynamic programming, as well as other approaches like Monte Carlo methods and temporal-difference learning.

5 Monte Carlo Methods

Monte Carlo methods rely solely on experience—sample sequences of states, actions, and rewards from actual or simulated interaction with an environment. Learning from actual experience is notable because it requires no prior knowledge of the environment's dynamics, yet can still achieve optimal behavior. Learning from simulated experience is also powerful; although a model is required, it needs only to generate sample transitions rather than the complete probability distributions of all possible transitions, as required in dynamic programming (DP).

Monte Carlo methods sample and average returns for each state-action pair, much like the bandit methods sample and average rewards for each action. The main difference is that there are multiple states, each acting like a different bandit problem (similar to an associative-search or contextual bandit), and the different bandit problems are interrelated. The return after taking an action in one state depends on the actions taken in later states in the same episode.

5.1 Monte Carlo Prediction

To estimate $v_\pi(s)$, the value of a state s under policy π, we use 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. State s may be visited multiple times in the same episode; the first time it is visited in an episode is called the first visit to s. The first-visit MC method estimates $v_\pi(s)$ as the average of the returns following first visits to s, while the every-visit MC method averages the returns following all visits to s. First-visit MC is illustrated in procedural form in Figure 3.

An important feature of Monte Carlo methods is that the estimates for each state are independent. The estimate for one state does not rely on the estimates of any other state, unlike in DP.

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

Input: a policy π to be evaluated
Initialize:
    $V(s) \in \mathbb{R}$, arbitrarily, for all $s \in \mathcal{S}$
    $Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Loop forever (for each episode):
    Generate an episode following π: $S_0, A_0, R_1, S_1, A_1, R_2, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    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 3: First-visit Monte Carlo method

5.2 Monte Carlo Estimation of Action Values

Without a model, state values alone are insufficient. One must explicitly estimate the value of each action to use the values in policy decisions. A primary goal of Monte Carlo methods is to estimate q_*, the optimal action-value function.

The complication is that many state-action pairs may never be visited. If π is a deterministic policy, it will observe returns for only one of the actions from each state. With no returns to average, the Monte Carlo estimates of the other actions will not improve with experience. For policy evaluation to work for action values, continual exploration is necessary. One way to ensure this is by specifying that episodes start in a state-action pair and that every pair has a nonzero probability of being selected as the start. This guarantees that all state-action pairs will be visited an infinite number of times in the limit of infinite episodes. This is known as the assumption of exploring starts.

5.3 Monte Carlo Control

Monte Carlo control methods alternate between policy evaluation and policy improvement, starting with an arbitrary policy π_0 and aiming to achieve the optimal policy and optimal action-value function. The Monte Carlo ES (Exploring Starts) algorithm is presented in Figure 4.

5.4 Monte Carlo Control without Exploring Starts

To ensure all actions are selected infinitely often, the agent must continue to select them. There are two approaches to ensuring this: on-policy methods and off-policy methods. On-policy methods evaluate or improve the policy used to make decisions, whereas off-policy methods evaluate or improve a policy different from that used to generate the data.[2]

In on-policy control methods, the policy is generally soft, meaning $\pi(a \mid s) > 0$ for all $s \in S$ and all $a \in A(s)$, but gradually shifts closer to a deterministic optimal policy. The

Monte Carlo ES (Exploring Starts), for estimating $\pi \approx \pi_*$

Initialize:
$\pi(s) \in \mathcal{A}(s)$ (arbitrarily), for all $s \in \mathcal{S}$
$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
$Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Loop forever (for each episode):
Choose $S_0 \in \mathcal{S}$, $A_0 \in \mathcal{A}(S_0)$ randomly such that all pairs have probability > 0
Generate an episode from S_0, A_0, following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
$G \leftarrow 0$
Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
$G \leftarrow \gamma G + R_{t+1}$
Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:
Append G to $Returns(S_t, A_t)$
$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$
$\pi(S_t) \leftarrow \arg\max_a Q(S_t, a)$

Figure 4: Monte Carlo ES (Exploring Starts) algorithm

on-policy method shifts the policy to an ϵ-greedy policy, as shown in Figure 5.

On-policy first-visit MC control (for ε-soft policies), estimates $\pi \approx \pi_*$

Algorithm parameter: small $\varepsilon > 0$
Initialize:
$\pi \leftarrow$ an arbitrary ε-soft policy
$Q(s,a) \in \mathbb{R}$ (arbitrarily), for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$
$Returns(s,a) \leftarrow$ empty list, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$

Repeat forever (for each episode):
Generate an episode following π: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
$G \leftarrow 0$
Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
$G \leftarrow \gamma G + R_{t+1}$
Unless the pair S_t, A_t appears in $S_0, A_0, S_1, A_1 \ldots, S_{t-1}, A_{t-1}$:
Append G to $Returns(S_t, A_t)$
$Q(S_t, A_t) \leftarrow \text{average}(Returns(S_t, A_t))$
$A^* \leftarrow \arg\max_a Q(S_t, a)$ (with ties broken arbitrarily)
For all $a \in \mathcal{A}(S_t)$:

$$\pi(a|S_t) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{A}(S_t)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(S_t)| & \text{if } a \neq A^* \end{cases}$$

Figure 5: On-policy Monte Carlo method

5.5 Off-policy Prediction via Importance Sampling

Off-policy methods use two policies: the target policy, which is learned about and becomes the optimal policy, and the behavior policy, which is more exploratory and generates behavior. Learning from data "off" the target policy is known as off-policy learning.

Almost all off-policy methods utilize importance sampling, a technique for estimating expected values under one distribution given samples from another. Importance sampling applies 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, $A_t, S_{t+1}, A_{t+1}, \ldots, S_T$, under any policy π is:

$$Pr\{A_t, S_{t+1}, A_{t+1}, \ldots, S_T \mid S_t, A_{t:T-1} \sim \pi\} = \prod_{k=t}^{T-1} \pi(A_k \mid S_k) p(S_{k+1} \mid S_k, A_k)$$

Thus, the relative probability of the trajectory under the target and behavior policies (the importance sampling ratio) is:

$$p_{t:T-1} = \frac{\prod_{k=t}^{T-1} \pi(A_k \mid S_k) p(S_{k+1} \mid S_k, A_k)}{\prod_{k=t}^{T-1} b(A_k \mid S_k) p(S_{k+1} \mid S_k, A_k)} = \prod_{k=t}^{T-1} \frac{\pi(A_k \mid S_k)}{b(A_k \mid S_k)} \tag{20}$$

Although the trajectory probabilities depend on the MDP's transition probabilities, which are generally unknown, they cancel out as they appear identically in both the numerator and the denominator. The importance sampling ratio depends only on the two policies and the sequence, not on the MDP.

To estimate $v_\pi(s)$, we scale the returns by the ratios and average the results:

$$V(s) = \frac{\sum_{t \in J(s)} \rho_{t:T-1} G_t}{|J(s)|} \tag{21}$$

When importance sampling is done as a simple average, it is called ordinary importance sampling.

A notable alternative is weighted importance sampling, which uses a weighted average defined as:

$$V(s) = \frac{\sum_{t \in J(s)} \rho_{t:T-1} G_t}{\sum_{t \in J(s)} \rho_{t:T-1}} \tag{22}$$

or zero if the denominator is zero.

5.6 Incremental Implementation

For a sequence of returns $G_1, G_2, \ldots, G_{n-1}$, all starting in the same state and each with a corresponding random weight W_i, we estimate:

$$V_n = \frac{\sum_{k=1}^{n-1} W_k G_k}{\sum_{k=1}^{n-1} W_k}$$

and keep it up-to-date as we obtain an additional return G_n. We must maintain the cumulative sum C_n of the weights given to the first n returns. The update rule for V_n is:

$$V_{n+1} = V_n + \frac{W_n}{C_n}[G_n - V_n]$$

and

$$C_{n+1} = C_n + W_{n+1}$$

Figure 6 contains a complete episode-by-episode incremental algorithm for Monte Carlo policy evaluation.

Off-policy MC prediction (policy evaluation) for estimating $Q \approx q_\pi$

Input: an arbitrary target policy π
Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
    $Q(s,a) \in \mathbb{R}$ (arbitrarily)
    $C(s,a) \leftarrow 0$

Loop forever (for each episode):
    $b \leftarrow$ any policy with coverage of π
    Generate an episode following b: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
    $G \leftarrow 0$
    $W \leftarrow 1$
    Loop for each step of episode, $t = T-1, T-2, \ldots, 0$, while $W \neq 0$:
        $G \leftarrow \gamma G + R_{t+1}$
        $C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
        $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$
        $W \leftarrow W \frac{\pi(A_t|S_t)}{b(A_t|S_t)}$

Figure 6: Off-policy Monte Carlo policy evaluation algorithm

5.7 Off-policy Monte Carlo Control

Off-policy control methods follow the behavior policy while learning about and improving the target policy. These techniques require that the behavior policy has a nonzero probability of selecting all actions that might be selected by the target policy (coverage). To explore all possibilities, the behavior policy must be soft, meaning it selects all actions in all states with nonzero probability. Figure 7 shows the pseudocode for this.[1]

6 Dynamic Programming

Dynamic programming (DP) refers to a collection of algorithms that can compute optimal policies given a perfect model of the environment as a Markov decision process (MDP). DP algorithms are obtained by turning Bellman equations into assignments, i.e., update rules for improving approximations of the desired value functions.

6.1 Bellman Optimality Equations

The Bellman optimality equations are fundamental to DP. They provide a way to express the value of a state or state-action pair in terms of the values of successor states. For

Off-policy MC control, for estimating $\pi \approx \pi_*$

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
$Q(s,a) \in \mathbb{R}$ (arbitrarily)
$C(s,a) \leftarrow 0$
$\pi(s) \leftarrow \text{argmax}_a\, Q(s,a)$ (with ties broken consistently)

Loop forever (for each episode):
$b \leftarrow$ any soft policy
Generate an episode using b: $S_0, A_0, R_1, \ldots, S_{T-1}, A_{T-1}, R_T$
$G \leftarrow 0$
$W \leftarrow 1$
Loop for each step of episode, $t = T-1, T-2, \ldots, 0$:
$G \leftarrow \gamma G + R_{t+1}$
$C(S_t, A_t) \leftarrow C(S_t, A_t) + W$
$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{W}{C(S_t, A_t)} [G - Q(S_t, A_t)]$
$\pi(S_t) \leftarrow \text{argmax}_a\, Q(S_t, a)$ (with ties broken consistently)
If $A_t \neq \pi(S_t)$ then exit inner Loop (proceed to next episode)
$W \leftarrow W \frac{1}{b(A_t|S_t)}$

Figure 7: Off-policy Monte Carlo control algorithm

the optimal state value function $v_*(s)$ and the optimal action value function $q_*(s,a)$, the equations are:

$$v_*(s) = \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')] \tag{23}$$

$$q_*(s,a) = \sum_{s',r} p(s', r|s, a)[r + \gamma \max_{a'} q_*(s', a')] \tag{24}$$

6.2 Policy Evaluation (Prediction)

The initial approximation, v_0, is chosen arbitrarily (except that the terminal state, if any, must be given value 0), and each successive approximation is obtained by using the Bellman equation for v_π (Equation 23) as an update rule:

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')] \tag{25}$$

Indeed, the sequence $\{v_k\}$ can be shown in general to converge to v_π as $k \to \infty$ under the same conditions that guarantee the existence of v_π. This algorithm is called iterative policy evaluation. A complete in-place version of iterative policy evaluation is shown in pseudocode in Figure 8. The pseudocode tests the quantity $\max_{s \in S} |v_{k+1}(s) - v_k(s)|$ after each sweep and stops when it is sufficiently small.

6.3 Policy Improvement

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

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

Iterative Policy Evaluation, for estimating $V \approx v_\pi$

Input π, the policy to be evaluated
Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$ arbitrarily, for $s \in \mathcal{S}$, and $V(terminal)$ to 0

Loop:
  $\Delta \leftarrow 0$
  Loop for each $s \in \mathcal{S}$:
    $v \leftarrow V(s)$
    $V(s) \leftarrow \sum_a \pi(a|s) \sum_{s',r} p(s', r \,|\, s, a)\big[r + \gamma V(s')\big]$
    $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Figure 8: Iterative Policy Evaluation

Then the policy π' must be as good as, or better than π. That is, it must obtain greater or equal expected return from all states $s \in S$:

$$v_{\pi'}(s) \geq v_\pi(s)$$

In other words, consider the new greedy policy π', given by:

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

where $\arg\max_a$ denotes the value of a at which the expression that follows is maximized (with ties broken arbitrarily). The process of making a new policy that improves on an original policy, by making it greedy with respect to the value function of the original policy, is called policy improvement. In particular, the policy improvement theorem carries through as stated for the stochastic case.

6.4 Policy Iteration

Once a policy, π, has been improved using v_π to yield a better policy, π', we can then compute $v_{\pi'}$ and improve it again to yield an even better π''. We can thus obtain a sequence of monotonically improving policies and value functions. Because a finite MDP has only a finite number of deterministic policies, this process must converge to an optimal policy and the optimal value function in a finite number of iterations. This way of finding an optimal policy is called policy iteration. A complete algorithm is given in Figure 9.

6.5 Value Iteration

Value iteration is an algorithm that can be written as a particularly simple update operation that combines the policy improvement and truncated policy evaluation steps:

$$v_{k+1}(s) = \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_k(s')] \tag{26}$$

Figure 10 shows a complete algorithm with this kind of termination condition. Value

Policy Iteration (using iterative policy evaluation) for estimating $\pi \approx \pi_*$

1. Initialization
   $V(s) \in \mathbb{R}$ and $\pi(s) \in \mathcal{A}(s)$ arbitrarily for all $s \in \mathcal{S}$; $V(terminal) \doteq 0$

2. Policy Evaluation
   Loop:
   $\Delta \leftarrow 0$
   Loop for each $s \in \mathcal{S}$:
   $v \leftarrow V(s)$
   $V(s) \leftarrow \sum_{s',r} p(s', r \mid s, \pi(s)) \big[r + \gamma V(s')\big]$
   $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
   until $\Delta < \theta$ (a small positive number determining the accuracy of estimation)

3. Policy Improvement
   policy-stable $\leftarrow$ *true*
   For each $s \in \mathcal{S}$:
   old-action $\leftarrow \pi(s)$
   $\pi(s) \leftarrow \arg\max_a \sum_{s',r} p(s', r \mid s, a) \big[r + \gamma V(s')\big]$
   If *old-action* $\neq \pi(s)$, then *policy-stable* $\leftarrow$ *false*
   If *policy-stable*, then stop and return $V \approx v_*$ and $\pi \approx \pi_*$; else go to 2

Figure 9: Policy Iteration

Value Iteration, for estimating $\pi \approx \pi_*$

Algorithm parameter: a small threshold $\theta > 0$ determining accuracy of estimation
Initialize $V(s)$, for all $s \in \mathcal{S}^+$, arbitrarily except that $V(terminal) = 0$

Loop:
| $\Delta \leftarrow 0$
| Loop for each $s \in \mathcal{S}$:
| $v \leftarrow V(s)$
| $V(s) \leftarrow \max_a \sum_{s',r} p(s', r \mid s, a) \big[r + \gamma V(s')\big]$
| $\Delta \leftarrow \max(\Delta, |v - V(s)|)$
until $\Delta < \theta$

Output a deterministic policy, $\pi \approx \pi_*$, such that
$\pi(s) = \arg\max_a \sum_{s',r} p(s', r \mid s, a) \big[r + \gamma V(s')\big]$

Figure 10: Value Iteration

iteration effectively combines, in each of its sweeps, one sweep of policy evaluation and one sweep of policy improvement. Faster convergence is often achieved by interposing multiple policy evaluation sweeps between each policy improvement sweep.

6.6 Asynchronous Dynamic Programming

A major drawback to the DP methods that we have discussed so far is that they involve operations over the entire state set of the MDP, that is, they require sweeps of the state set. If the state set is very large, then even a single sweep can be prohibitively expensive.

Asynchronous DP algorithms are in-place iterative DP algorithms that are not organized in terms of systematic sweeps of the state set. These algorithms update the values of states in any order whatsoever, using whatever values of other states happen to be available. The values of some states may be updated several times before the values of others are updated once.

Asynchronous algorithms also make it easier to intermix computation with real-time interaction. To solve a given MDP, we can run an iterative DP algorithm at the same time that an agent is actually experiencing the MDP. The agent's experience can be used to determine the states to which the DP algorithm applies its updates. At the same time, the latest value and policy information from the DP algorithm can guide the agent's decision making.

6.7 Generalized Policy Iteration

We use the term generalized policy iteration (GPI) to refer to the general idea of letting policy-evaluation and policy-improvement processes interact, independent of the granularity and other details of the two processes. Almost all reinforcement learning methods are well described as GPI. That is, all have identifiable policies and value functions, with the policy always being improved with respect to the value function and the value function always being driven toward the value function for the policy, as suggested by the diagram to the right. If both the evaluation process and the improvement process stabilize, that is, no longer produce changes, then the value function and policy must be optimal. The value function stabilizes only when it is consistent with the current policy, and the policy stabilizes only when it is greedy with respect to the current value function. Thus, both processes stabilize only when a policy has been found that is greedy with respect to its own evaluation function.

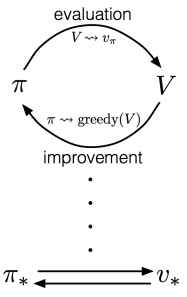

Figure 11: Generalized Policy Iteration

7 START OF END TERM REPORT

8 nstep Bootstrapping

This topic was really intresting for me. I studied this topic from the book by Sutton and Barto and watched a playlist of Stanford University for this course. So basically, this method combines the Monte Carlo method and the dynamic programming algorithm. It strikes a balance between the variance reduction of MC methods and the bias reduction of DP methods, thereby providing more efficient and stable learning algorithms.

8.1 Concept

In N-step bootstrapping, the update of value estimates is based on rewards received over the next n steps, followed by the bootstrapped value from the estimated value function at the n-th step. This approach generalizes the concept of Temporal-Difference (TD) learning by considering multiple steps, leading to a more flexible and robust update rule.

8.2 Algorithm

The general idea of N-step bootstrapping can be outlined as follows:

1. **Initialization**: Initialize the value function $V(s)$ arbitrarily for all states s.
2. **Interaction with Environment**:
   - At each time step t, take an action a_t based on the current policy π and observe the reward r_{t+1} and the next state s_{t+1}.
3. **N-step Return Calculation**:
   - For each time step t, calculate the N-step return $G_t^{(n)}$:

     $$G_t^{(n)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n V(s_{t+n})$$

     Here, γ is the discount factor.
4. **Update Value Function**:
   - Update the value function $V(s_t)$ using the N-step return:

     $$V(s_t) \leftarrow V(s_t) + \alpha \left(G_t^{(n)} - V(s_t) \right)$$

     Here, α is the learning rate.

8.3 n-step TD Prediction

Monte Carlo methods perform an update for each state based on the entire sequence of observed rewards from that state until the end of the episode. The update of one-step TD methods, on the other hand, is based on just the one next reward, bootstrapping from the value of the state one step later as a proxy for the remaining rewards. One kind of intermediate method, then, would perform an update based on an intermediate number of rewards: more than one, but less than all of them until termination.

Methods in which the temporal difference extends over n steps are called n-step TD methods. The target for an arbitrary n-step update is the n-step return:

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n_1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n}) \tag{27}$$

for all n,t such that $n \geq 1$ and $0 \leq t < T - n$. If $t + n \geq T$ (if the n-step return extends to or beyond termination), then all the missing terms are taken as zero, and the n-step return defined to be equal to the ordinary full return ($G_{t:t+n} = Gtift + n \geq T$).

The natural state-value learning algorithm for using n-step returns is thus:

$$V_{t+n}(S_t) = V_{t+n-1}(S_t) + \alpha[G_{t:t+n} - V_{t+n-1}(S_t)] \tag{28}$$

Complete pseudocode is given in the box **??**.

The worst error of the expected n-step return is guaranteed to be less than or equal to γ^n times the worst error under V_{t+n-1}

$$\max_s |E_\pi[G_{t:t+n}|S_t = s] - v_\pi(s)| \leq \gamma^n \max_s |V_{t+n-1}(s) - v_\pi(s)| \tag{29}$$

This is called the error reduction property of n-step returns.

9 Fast Reinforcement Learning

I have covered this topic exclusivly from Stanford YouTube playlist. This topic says that RL can be sometimes slow, so we require this algorithm. I spent a full week on this topic as it has 3 videos in the playlist, and now I am confident in this topic. Major thing that I learned, I will try to explain in summary, so let's get started.

9.1 Introduction

Fast Reinforcement Learning focuses on accelerating the learning process while maintaining or improving the performance of the agent. Key techniques in this area include efficient exploration, function approximation, and leveraging parallelism.

9.2 Efficient Exploration

Efficient exploration is crucial in RL as it helps the agent discover rewarding states and actions quickly. Several strategies can be employed to achieve efficient exploration:

- **Exploration-Exploitation Trade-off**: Balancing exploration of new actions with exploitation of known rewarding actions using techniques like ϵ-greedy or Upper Confidence Bound (UCB).
- **Intrinsic Motivation**: Encouraging exploration through intrinsic rewards, such as novelty or curiosity-driven approaches.
- **Reward Shaping**: Modifying the reward function to provide intermediate rewards that guide the agent towards the goal.

9.3 Function Approximation

Function approximation methods are used to generalize from limited experiences and reduce the computational cost of learning. Common function approximation techniques include:

- **Linear Function Approximation**: Using linear models to approximate the value function or policy.
- **Neural Networks**: Leveraging deep learning to approximate complex value functions and policies, as seen in Deep Q-Networks (DQN) and Deep Deterministic Policy Gradients (DDPG).
- **Tile Coding**: Dividing the state space into tiles and approximating the value function within each tile.

9.4 Parallelism and Distributed Learning

Parallelism and distributed learning techniques can significantly speed up the training process by leveraging multiple processors or machines. Key approaches include:

- **Asynchronous Methods**: Algorithms like Asynchronous Advantage Actor-Critic (A3C) that use multiple parallel agents to explore the environment and update shared parameters asynchronously.
- **Distributed Experience Replay**: Distributing the experience replay buffer across multiple machines to improve data efficiency and training speed.
- **Parameter Server Architectures**: Using parameter servers to manage and update model parameters across multiple workers.

9.5 Challenges and Future Directions

While significant progress has been made, fast reinforcement learning faces several challenges:

- **Scalability**: Ensuring algorithms scale effectively with the complexity of the environment and the size of the state-action space.
- **Stability**: Maintaining stability and convergence of learning algorithms, especially when using function approximation and parallelism.
- **Generalization**: Developing methods that generalize well to unseen environments and tasks.

10 Value Function Approximation with Function Parameterization

The last topic for my learning was this topic on value function in which I spent a week learning from the web and the book by Sutton and Barto. I also took references from YouTube videos. Overall, to summarize this topic, I created a short summary for each topic and subtopic.

10.1 Approaches to Value Function Approximation

Modern methods in machine learning and statistics often assume a fixed training dataset for multiple iterations. However, reinforcement learning requires online learning capabilities where the agent updates its knowledge as it interacts with the environment or its model. This necessitates techniques that efficiently learn from newly acquired data and adapt to changing target functions.

10.2 Objective for Value Function Prediction

To define which states are of greater importance, we introduce a state distribution $\mu(s) \geq 0$ with $\sum_s \mu(s) = 1$, representing the significance of the error in each state s. The error in state s is quantified by the squared difference between the approximate value $\hat{v}(s, \mathbf{w})$ and the true value $v_\pi(s)$. By weighting this error by $\mu(s)$, we derive the mean squared value error objective function, denoted $\overline{VE}$:

$$\overline{VE}(\mathbf{w}) = \sum_{s \in S} \mu(s)[v_\pi(s) - \hat{v}(s, \mathbf{w})]^2 \tag{30}$$

10.3 Gradient-Based Methods for Optimization

In gradient descent techniques, the weight vector is represented as $\mathbf{w}=(w_1, w_2, \ldots, w_d)_T$, and the approximate value function $\hat{v}(s, \mathbf{w})$ is differentiable with respect to $\mathbf{w}$ for all $s \in S$.

Stochastic gradient descent (SGD) methods update the weight vector after each example, moving in the direction that reduces the error for that example:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)]\nabla\hat{v}(S_t, \mathbf{w}_t) \tag{31}$$

SGD is called "stochastic" because the update is based on individual examples that may be randomly selected. Gradient descent methods typically converge to a locally optimal solution. Pseudocode for a full algorithm is illustrated in Figure 12.

Bootstrapping methods, which differ from true gradient descent, consider only part of the gradient and are thus termed semi-gradient methods. These methods, while potentially less robust in convergence, offer benefits such as faster learning and the ability to learn continuously online. Pseudocode for semi-gradient TD(0) is shown in Figure 13.

State aggregation simplifies function approximation by grouping states together, estimating a single value for each group.

Gradient Monte Carlo Algorithm for Estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop forever (for each episode):
  Generate an episode $S_0, A_0, R_1, S_1, A_1, \ldots, R_T, S_T$ using π
  Loop for each step of episode, $t = 0, 1, \ldots, T-1$:
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[G_t - \hat{v}(S_t,\mathbf{w})\big]\nabla\hat{v}(S_t,\mathbf{w})$

Figure 12: Pseudocode for gradient-based Monte Carlo state-value prediction.

Semi-gradient TD(0) for estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  Initialize S
  Loop for each step of episode:
    Choose $A \sim \pi(\cdot|S)$
    Take action A, observe R, S'
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})\big]\nabla\hat{v}(S,\mathbf{w})$
    $S \leftarrow S'$
  until S is terminal

Figure 13: Pseudocode for semi-gradient TD(0).

10.4 Linear Function Approximation

For each state s, there is a corresponding real-valued vector $\mathbf{x}(\text{s}) = (x_1(s), x_2(s), \ldots, x_d(s))^T$, matching the dimensionality of $\mathbf{w}$. Linear methods approximate the state-value function as the inner product of $\mathbf{w}$ and $\mathbf{x}(\text{s})$:

$$\hat{v}(s, \mathbf{w}) = \mathbf{w}^T\mathbf{x}(s) = \sum_{i=1}^{d} w_i x_i(s) \tag{32}$$

Here, $\mathbf{x}(\text{s})$ is the feature vector for state s. The gradient of the approximate value function with respect to $\mathbf{w}$ is:

$$\nabla\hat{v}(s, \mathbf{w}) = \mathbf{x}(s)$$

The semi-gradient TD(0) algorithm converges under linear approximation, with the update rule at each time step t:

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha[R_{t+1}\mathbf{x}_t - \mathbf{x}_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})^T\mathbf{w}_t] \tag{33}$$

$$E[\mathbf{w}_{t+1}|\mathbf{w}_t] = \mathbf{w}_t + \alpha(\mathbf{b} - \mathbf{A}\mathbf{w}_t) \tag{34}$$

where $\mathbf{b} = E[R_{t+1}\mathbf{x}_t] \in \mathbb{R}^d$ and $A = E[\mathbf{x}_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})^T] \in \mathbb{R}^{d\times d}$. If the system converges, it will converge to the weight vector $\mathbf{w}_{TD}$, where

$$\mathbf{w}_{TD} = \mathbf{A}^{-1}\mathbf{b} \tag{35}$$

References

[1] Sutton Barto. *Reinforcement Learning*.

[2] Shivaram Kalyanakrishnan. CS747 IIT Bombay - Foundations of Intelligent and Learning Agents. Lecture slides, 2022. Course: CS747 Foundations of Intelligent and Learning Agents.