Foundations of Reinforcement Learning with Applications in Finance

Ashwin Rao, Tikhon Jelvis

1 Monte-Carlo and Temporal-Difference for Control

In chapter **??**, we covered MC and TD algorithms to solve the *Prediction* problem. In this chapter, we cover MC and TD algorithms to solve the *Control* problem. As a reminder, MC and TD algorithms are Reinforcement Learning algorithms that only have access to an individual experience (at a time) of next state and reward when the AI agent performs an action in a given state. The individual experience could be the result of an interaction with a real environment or could be served by a simulated environment (as explained at the state of Chapter **??**). It also pays to remind that RL algorithms overcome the Curse of Dimensionality and the Curse of Modeling by learning an appropriate function approximation of the Value Function from a stream of individual experiences. Hence, large-scale Control problems that are typically seen in the real-world are often tackled by RL.

1.1 Refresher on *Generalized Policy Iteration* (GPI)

We shall soon see that all RL Control algorithms are based on the fundamental idea of *Generalized Policy Iteration* (introduced initially in Chapter **??**), henceforth abbreviated as GPI. The exact ways in which the GPI idea is utilized in RL algorithms differs from one algorithm to another, and they differ significantly from how the GPI idea is utilized in DP algorithms. So before we get into RL Control algorithms, it's important to ground on the abstract concept of GPI. We now ask you to re-read Section **??** in Chapter **??**.

To summarize, the key concept in GPI is that we can evaluate the Value Function for a policy with *any* Policy Evaluation method, and we can improve a policy with *any* Policy Improvement method (not necessarily the methods used in the classical Policy Iteration DP algorithm). The word *any* does not simply mean alternative algorithms for Policy Evaluation and/or Policy Improvements - the word *any* also refers to the fact that we can do a "partial" Policy Evaluation or a "partial" Policy Improvement. The word "partial" is used quite generically here - any set of calculations that simply take us *towards* a complete Policy Evaluation or *towards* a complete Policy Improvement qualify. This means GPI allows us to switch from Policy Evaluation to Policy Improvements without doing a complete Policy Evaluation or complete Policy Improvement (for instance, we don't have to take Policy Evaluation calculations all the way to convergence). Figure 1.1 illustrates Generalized Policy Iteration as the shorter-length arrows (versus the longer-length arrows seen in Figure **??** for the usual Policy Iteration algorithm). Note how these shorter-length arrows don't go all the way to either the "value function line" or the "policy line" but the shorter-length arrows do go some part of the way towards the line they are meant to go towards at that stage in the algorithm.

As has been our norm in the book so far, our approach to RL Control algorithms is to first cover the simple case of Tabular RL Control algorithms to illustrate the core concepts in a simple and intuitive manner. In many Tabular RL Control algorithms (especially Tabular TD Control), GPI consists of the Policy Evaluation step for just a single state (versus for all states in usual Policy Iteration) and the Policy Improvement step is also done for

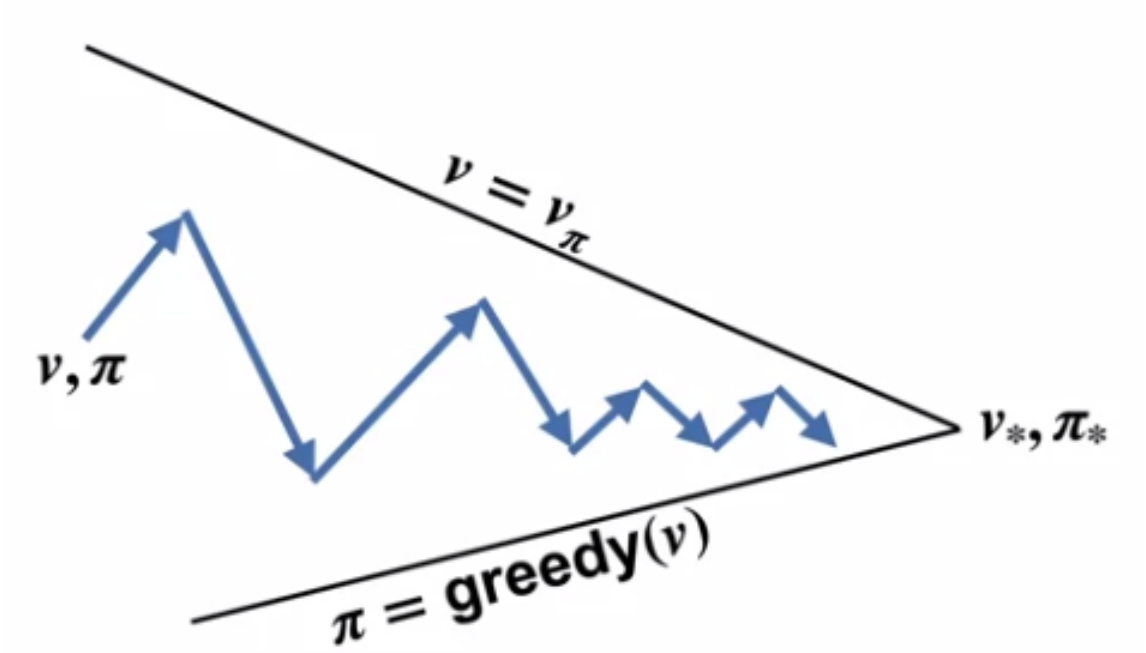

Figure 1.1: Progression Lines of Value Function and Policy in Generalized Policy Iteration (Image Credit: Coursera Course on Fundamentals of RL)

just a single state. So essentially these RL Control algorithms are an alternating sequence of single-state policy evaluation and single-state policy improvement (where the single-state is the state produced by sampling or the state that is encountered in a real-world environment interaction). Similar to the case of Prediction, we first cover Monte-Carlo (MC) Control and then move on to Temporal-Difference (TD) Control.

1.2 GPI with Evaluation as Monte-Carlo

Let us think about how to do MC Control based on the GPI idea. The natural idea that emerges is to do Policy Evaluation with MC (this is basically MC Prediction), followed by greedy Policy Improvement, then MC Policy Evaluation with the improved policy, and so on ... . This seems like a reasonable idea, but there is a problem with doing greedy Policy Improvement. The problem is that the Greedy Policy Improvement calculation (Equation 1.1) requires a model of the state transition probability function $\mathcal{P}$ and the reward function $\mathcal{R}$, which is not available in an RL interface.

$$\pi'_D(s) = \arg\max_{a \in \mathcal{A}} \{\mathcal{R}(s,a) + \gamma \sum_{s' \in \mathcal{N}} \mathcal{P}(s,a,s') \cdot V^\pi(s')\} \text{ for all } s \in \mathcal{N} \tag{1.1}$$

However, we note that Equation 1.1 can be written more succinctly as:

$$\pi'_D(s) = \arg\max_{a \in \mathcal{A}} Q^\pi(s,a) \text{ for all } s \in \mathcal{N} \tag{1.2}$$

This view of Greedy Policy Improvement is valuable because instead of doing Policy Evaluation for calculating V^π (MC Prediction), we can instead do Policy Evaluation to calculate Q^π (with MC Prediction for the Q-Value Function). With this modification to Policy

Evaluation, we can keep alternating between Policy Evaluation and Policy Improvement until convergence to obtain the Optimal Value Function and Optimal Policy. Indeed, this is a valid MC Control algorithm. However, this algorithm is not practical as each Policy Evaluation (MC Prediction) typically takes very long to converge (as we have noted in Chapter **??**) and the number of iterations of Evaluation and Improvement until GPI convergence will also be large. More importantly, this algorithm simply modifies the Policy Iteration DP/ADP algorithm by replacing DP/ADP Policy Evaluation with MC Q-Value Policy Evaluation - hence, we simply end up with a slower version of the Policy Iteration DP/ADP algorithm. Instead, we seek an MC Control Algorithm that switches from Policy Evaluation to Policy Improvement without requiring Policy Evaluation to converge (this is essentially the GPI idea).

So the natural GPI idea here would be to do the usual MC Prediction updates (of the Q-Value estimate) at the end of an episode, then improve the policy at the end of that episode, then perform MC Prediction updates (with the improved policy) at the end of the next episode, and so on ... . Let's see what this algorithm looks like. Equation 1.2 tells us that all we need to perform the requisite greedy action (from the improved policy) at any time step in any episode is an estimate of the Q-Value Function. For ease of understanding, for now, let us just restrict ourselves to the case of Tabular Every-Visit MC Control with equal weights for each of the *Return* data points obtained for any (state, action) pair. In this case, we can simply perform the following two updates at the end of each episode for each (S_t, A_t) pair encountered in the episode (note that at each time step t, A_t is based on the greedy policy derived from the current estimate of the Q-Value function):

$$\begin{aligned} Count(S_t, A_t) &\leftarrow Count(S_t, A_t) + 1 \\ Q(S_t, A_t) &\leftarrow Q(S_t, A_t) + \frac{1}{Count(S_t, A_t)} \cdot (G_t - Q(S_t, A_t)) \end{aligned} \tag{1.3}$$

It's important to note that $Count(S_t, A_t)$ is accumulated over the set of all episodes seen thus far. Note that the estimate $Q(S_t, A_t)$ is not an estimate of the Q-Value Function for a single policy - rather, it keeps updating as we encounter new greedy policies across the set of episodes.

So is this now our first Tabular RL Control algorithm? Not quite - there is yet another problem. This problem is more subtle and we illustrate the problem with a simple example. Let's consider a specific state (call it s) and assume that there are only two allowable actions a_1 and a_2 for state s. Let's say the true Q-Value Function for state s is: $Q_{true}(s, a_1) = 2, Q_{true}(s, a_2) = 5$. Let's say we initialize the Q-Value Function estimate as: $Q(s, a_1) = Q(s, a_2) = 0$. When we encounter state s for the first time, the action to be taken is arbitrary between a_1 and a_2 since they both have the same Q-Value estimate (meaning both a_1 and a_2 yield the same max value for $Q(s, a)$ among the two choices for a). Let's say we arbitrarily pick a_1 as the action choice and let's say for this first encounter of state s (with the arbitrarily picked action a_1), the return obtained is 3. So $Q(s, a_1)$ updates to the value 3. So when the state s is encountered for the second time, we see that $Q(s, a_1) = 3$ and $Q(s, a_2) = 0$ and so, action a_1 will be taken according to the greedy policy implied by the estimate of the Q-Value Function. Let's say we now obtain a return of -1, updating $Q(s, a_1)$ to $\frac{3-1}{2} = 1$. When s is encountered for the third time, yet again action a_1 will be taken according to the greedy policy implied by the estimate of the Q-Value Function. Let's say we now obtain a return of 2, updating $Q(s, a_1)$ to $\frac{3-1+2}{3} = \frac{4}{3}$. We see that as long as the returns associated with a_1 are not negative enough to make the estimate $Q(s, a_1)$ negative, a_2 is "locked out" by a_1 because the first few occurrences of a_1 happen to yield

an average return greater than the initialization of $Q(s, a_2)$. Even if a_2 was chosen, it is possible that the first few occurrences of a_2 yield an average return smaller than the average return obtained on the first few occurrences of a_1, in which case a_2 could still get locked-out prematurely.

This problem goes beyond MC Control and applies to the broader problem of RL Control - updates can get biased by initial random occurrences of returns (or return estimates), which in turn could prevent certain actions from being sufficiently chosen (thus, disallowing accurate estimates of the Q-Values for those actions). While we do want to *exploit* actions that seem to be fetching higher returns, we also want to adequately *explore* all possible actions so we can obtain an accurate-enough estimate of their Q-Values. This is essentially the Explore-Exploit dilemma of the famous Multi-Armed Bandit Problem. In Chapter **??**, we will cover the Multi-Armed Bandit problem in detail, along with a variety of techniques to solve the Multi-Armed Bandit problem (which are essentially creative ways of resolving the Explore-Exploit dilemma). We will see in Chapter **??** that a simple way of resolving the Explore-Exploit dilemma is with a method known as ϵ-greedy, which essentially means we must be greedy ("exploit") a certain $(1 - \epsilon)$ fraction of the time and for the remaining (ϵ) fraction of the time, we explore all possible actions. The term "certain fraction of the time" refers to probabilities of choosing actions, which means an ϵ-greedy policy (generated from a Q-Value Function estimate) will be a stochastic policy. For the sake of simplicity, in this book, we will employ the ϵ-greedy method to resolve the Explore-Exploit dilemma in all RL Control algorithms involving the Explore-Exploit dilemma (although you must understand that we can replace the ϵ-greedy method by the other methods we shall cover in Chapter **??** in any of the RL Control algorithms where we run into the Explore-Exploit dilemma). So we need to tweak the Tabular MC Control algorithm described above to perform Policy Improvement with the ϵ-greedy method. The formal definition of the ϵ-greedy stochastic policy π' (obtained from the current estimate of the Q-Value Function) for a Finite MDP (since we are focused on Tabular RL Control) is as follows:

$$\text{Improved Stochastic Policy } \pi'(s, a) = \begin{cases} \frac{\epsilon}{|\mathcal{A}|} + 1 - \epsilon & \text{if } a = \arg\max_{b \in \mathcal{A}} Q(s, b) \\ \frac{\epsilon}{|\mathcal{A}|} & \text{otherwise} \end{cases}$$

where $\mathcal{A}$ denotes the set of allowable actions and $\epsilon \in [0, 1]$ is the specification of the degree of exploration.

This says that with probability $1 - \epsilon$, we select the action that maximizes the Q-Value Function estimate for a given state, and with probability ϵ, we uniform-randomly select each of the allowable actions (including the maximizing action). Hence, the maximizing action is chosen with probability $\frac{\epsilon}{|\mathcal{A}|} + 1 - \epsilon$. Note that if ϵ is zero, π' reduces to the deterministic greedy policy π'_D that we had defined earlier. So the greedy policy can be considered to be a special case of ϵ-greedy policy with $\epsilon = 0$.

But we haven't yet actually proved that an ϵ-greedy policy is indeed an improved policy. We do this in the theorem below. Note that in the following theorem's proof, we re-use the notation and inductive-proof approach used in the Policy Improvement Theorem (Theorem **??**) in Chapter **??**. So it would be a good idea to re-read the proof of Theorem **??** in Chapter **??** before reading the following theorem's proof.

Theorem 1.2.1. *For a Finite MDP, if π is a policy such that for all $s \in \mathcal{N}, \pi(s, a) \geq \frac{\epsilon}{|\mathcal{A}|}$ for all $a \in \mathcal{A}$, then the ϵ-greedy policy π' obtained from Q^π is an improvement over π, i.e., $\boldsymbol{V}^{\pi'}(s) \geq \boldsymbol{V}^\pi(s)$ for all $s \in \mathcal{N}$.*

Proof. We've previously learnt that for any policy π', if we apply the Bellman Policy Operator $\boldsymbol{B}^{\pi'}$ repeatedly (starting with $\boldsymbol{V}^{\pi}$), we converge to $\boldsymbol{V}^{\pi'}$. In other words,

$$\lim_{i \to \infty} (\boldsymbol{B}^{\pi'})^i (\boldsymbol{V}^{\pi}) = \boldsymbol{V}^{\pi'}$$

So the proof is complete if we prove that:

$$(\boldsymbol{B}^{\pi'})^{i+1}(\boldsymbol{V}^{\pi}) \geq (\boldsymbol{B}^{\pi'})^i (\boldsymbol{V}^{\pi}) \text{ for all } i = 0, 1, 2, \ldots$$

In plain English, this says we need to prove that repeated application of $\boldsymbol{B}^{\pi'}$ produces a non-decreasing sequence of Value Functions $[(\boldsymbol{B}^{\pi'})^i (\boldsymbol{V}^{\pi}) | i = 0, 1, 2, \ldots]$.

We prove this by induction. The base case of the proof by induction is to show that $\boldsymbol{B}^{\pi'}(\boldsymbol{V}^{\pi}) \geq \boldsymbol{V}^{\pi}$

$$\begin{aligned}
\boldsymbol{B}^{\pi'}(\boldsymbol{V}^{\pi})(s) &= (\boldsymbol{\mathcal{R}}^{\pi'} + \gamma \cdot \boldsymbol{\mathcal{P}}^{\pi'} \cdot \boldsymbol{V}^{\pi})(s) \\
&= \boldsymbol{\mathcal{R}}^{\pi'}(s) + \gamma \cdot \sum_{s' \in \mathcal{N}} \boldsymbol{\mathcal{P}}^{\pi'}(s, s') \cdot \boldsymbol{V}^{\pi}(s') \\
&= \sum_{a \in \mathcal{A}} \pi'(s, a) \cdot (\mathcal{R}(s, a) + \gamma \cdot \sum_{s' \in \mathcal{N}} \mathcal{P}(s, a, s') \cdot \boldsymbol{V}^{\pi}(s')) \\
&= \sum_{a \in \mathcal{A}} \pi'(s, a) \cdot Q^{\pi}(s, a) \\
&= \sum_{a \in \mathcal{A}} \frac{\epsilon}{|\mathcal{A}|} \cdot Q^{\pi}(s, a) + (1 - \epsilon) \cdot \max_{a \in \mathcal{A}} Q^{\pi}(s, a) \\
&\geq \sum_{a \in \mathcal{A}} \frac{\epsilon}{|\mathcal{A}|} \cdot Q^{\pi}(s, a) + (1 - \epsilon) \cdot \sum_{a \in \mathcal{A}} \frac{\pi(s, a) - \frac{\epsilon}{|\mathcal{A}|}}{1 - \epsilon} \cdot Q^{\pi}(s, a) \\
&= \sum_{a \in \mathcal{A}} \pi(s, a) \cdot Q^{\pi}(s, a) \\
&= \boldsymbol{V}^{\pi}(s) \text{ for all } s \in \mathcal{N}
\end{aligned}$$

The line with the inequality above is due to the fact that for any fixed $s \in \mathcal{N}$, $\max_{a \in \mathcal{A}} Q^{\pi}(s, a) \geq \sum_{a \in \mathcal{A}} w_a \cdot Q^{\pi}(s, a)$ (maximum Q-Value greater than or equal to a weighted average of all Q-Values, for a given state) with the weights $w_a = \frac{\pi(s,a) - \frac{\epsilon}{|\mathcal{A}|}}{1 - \epsilon}$ such that $\sum_{a \in \mathcal{A}} w_a = 1$ and $0 \leq w_a \leq 1$ for all $a \in \mathcal{A}$.

This completes the base case of the proof by induction.

The induction step is easy and is proved as a consequence of the monotonicity property of the $\boldsymbol{B}^{\pi}$ operator (for any π), which is defined as follows:

$$\text{Monotonicity Property of } \boldsymbol{B}^{\pi} : \boldsymbol{X} \geq \boldsymbol{Y} \Rightarrow \boldsymbol{B}^{\pi}(\boldsymbol{X}) \geq \boldsymbol{B}^{\pi}(\boldsymbol{Y})$$

Note that we proved the monotonicity property of the $\boldsymbol{B}^{\pi}$ operator in Chapter **??**. A straightforward application of this monotonicity property provides the induction step of the proof:

$$(\boldsymbol{B}^{\pi'})^{i+1}(\boldsymbol{V}^{\pi}) \geq (\boldsymbol{B}^{\pi'})^i (\boldsymbol{V}^{\pi}) \Rightarrow (\boldsymbol{B}^{\pi'})^{i+2}(\boldsymbol{V}^{\pi}) \geq (\boldsymbol{B}^{\pi'})^{i+1}(\boldsymbol{V}^{\pi}) \text{ for all } i = 0, 1, 2, \ldots$$

This completes the proof. □

We note that for any ϵ-greedy policy π, we do ensure the condition that for all $s \in \mathcal{N}$, $\pi(s, a) \geq \frac{\epsilon}{|\mathcal{A}|}$ for all $a \in \mathcal{A}$. So we just need to ensure that this condition holds true for the initial choice of π (in the GPI with MC algorithm). An easy way to ensure this is to choose the initial π to be a uniform choice over actions (for each state), i.e., for all $s \in \mathcal{N}$, $\pi(s, a) = \frac{1}{|\mathcal{A}|}$ for all $a \in \mathcal{A}$.

1.3 GLIE Monte-Control Control

So to summarize, we've resolved two problems - firstly, we replaced the state-value function estimate with the action-value function estimate and secondly, we replaced greedy policy improvement with ϵ-greedy policy improvement. So our MC Control algorithm does GPI as follows:

- Do Policy Evaluation with the Q-Value Function with Q-Value updates at the end of each episode.
- Do Policy Improvement with an ϵ-greedy Policy (readily obtained from the Q-Value Function estimate at any time step for any episode).

So now we are ready to develop the details of the Monte-Control algorithm that we've been seeking. For ease of understanding, we first cover the Tabular version and then we will implement the generalized version with function approximation. Note that an ϵ-greedy policy enables adequate exploration of actions, but we will also need to do adequate exploration of states in order to achieve a suitable estimate of the Q-Value Function. Moreover, as our Control algorithm proceeds and the Q-Value Function estimate gets better and better, we reduce the amount of exploration and eventually (as the number of episodes tend to infinity), we want to have ϵ (degree of exploration) tend to zero. In fact, this behavior has a catchy acronym associated with it, which we define below:

Definition 1.3.1. We refer to *Greedy In The Limit with Infinite Exploration* (abbreviated as GLIE) as the behavior that has the following two properties:

1. All state-action pairs are explored infinitely many times, i.e., for all $s \in \mathcal{N}$, for all $a \in \mathcal{A}$, and $Count_k(s,a)$ denoting the number of occurrences of (s,a) pairs after k episodes:
$$\lim_{k \to \infty} Count_k(s,a) = \infty$$
2. The policy converges to a greedy policy, i.e., for all $s \in \mathcal{N}$, for all $a \in \mathcal{A}$, and $\pi_k(s,a)$ denoting the ϵ-greedy policy obtained from the Q-Value Function estimate after k episodes:
$$\lim_{k \to \infty} \pi_k(s,a) = \mathbb{I}_{a=\text{arg max}_{b \in \mathcal{A}} Q(s,b)}$$

A simple way by which our method of using the ϵ-greedy policy (for policy improvement) can be made GLIE is by reducing ϵ as a function of number of episodes k as follows:

$$\epsilon_k = \frac{1}{k}$$

So now we are ready to describe the Tabular MC Control algorithm we've been seeking. We ensure that this algorithm has GLIE behavior and so, we refer to it as *GLIE Tabular Monte-Carlo Control*. The following is the outline of the procedure for each episode (terminating trace experience) in the algorithm:

- Generate the trace experience (episode) with actions sampled from the ϵ-greedy policy π obtained from the estimate of the Q-Value Function that is available at the start of the trace experience. Also, sample the first state of the trace experience from a uniform distribution of states in $\mathcal{N}$. This ensures infinite exploration of both states and actions. Let's denote the contents of this trace experience as:
$$S_0, A_0, R_1, S_1, A_1, \ldots, R_T, S_T$$

and define the trace experience return G_t associated with (S_t, A_t) as:

$$G_t = \sum_{i=t+1}^{T} \gamma^{i-t-1} \cdot R_i = R_{t+1} + \gamma \cdot R_{t+2} + \gamma^2 \cdot R_{t+3} + \ldots \gamma^{T-t-1} \cdot R_T$$

- For each state S_t and action A_t in the trace experience, perform the following updates at the end of the trace experience:

$$Count(S_t, A_t) \leftarrow Count(S_t, A_t) + 1$$

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \frac{1}{Count(S_t, A_t)} \cdot (G_t - Q(S_t, A_t))$$

- Let's say this trace experience is the k-th trace experience in the sequence of trace experiences. Then, at the end of the trace experience, set:

$$\epsilon \leftarrow \frac{1}{k}$$

We state the following important theorem without proof.

Theorem 1.3.1. *The above-described GLIE Tabular Monte-Carlo Control algorithm converges to the Optimal Action-Value function: $Q(s, a) \to Q^*(s, a)$ for all $s \in \mathcal{N}$, for all $a \in \mathcal{A}$. Hence, GLIE Tabular Monte-Carlo Control converges to an Optimal (Deterministic) Policy π^*.*

The extension from Tabular to Function Approximation of the Q-Value Function is straightforward. The update (change) in the parameters $\boldsymbol{w}$ of the Q-Value Function Approximation $Q(s, a; \boldsymbol{w})$ is as follows:

$$\Delta \boldsymbol{w} = \alpha \cdot (G_t - Q(S_t, A_t; \boldsymbol{w})) \cdot \nabla_{\boldsymbol{w}} Q(S_t, A_t; \boldsymbol{w}) \tag{1.4}$$

where α is the learning rate in the stochastic gradient descent and G_t is the trace experience return from state S_t upon taking action A_t at time t on a trace experience.

Now let us write some code to implement the above description of GLIE Monte-Carlo Control, generalized to handle Function Approximation of the Q-Value Function. As you shall see in the code below, there are a couple of other generalizations from the algorithm outline described above. Let us start by understanding the various arguments to the below function `glie_mc_control`.

- `mdp: MarkovDecisionProcess[S, A]` - This represents the interface to an abstract Markov Decision Process. Note that this interface doesn't provide any access to the transition probabilities or reward function. The core functionality available through this interface are the two `@abstractmethods` `step` and `actions`. The `step` method only allows us to access an individual experience of the next state and reward pair given the current state and action (since it returns an abstract `Distribution` object). The `actions` method gives us the allowable actions for a given state.
- `states: NTStateDistribution[S]` - This represents an arbitrary distribution of the non-terminal states, which in turn allows us to sample the starting state (from this distribution) for each trace experience.
- `approx_0: QValueFunctionApprox[S, A]` - This represents the initial function approximation of the Q-Value function (that is meant to be updated, in an immutable manner, through the course of the algorithm).

- `gamma: float` - This represents the discount factor to be used in estimating the Q-Value Function.
- `epsilon_as_func_of_episodes: Callable[[int], float]` - This represents the extent of exploration (ϵ) as a function of the number of trace experiences done so far (allowing us to generalize from our default choice of $\epsilon(k) = \frac{1}{k}$).
- `episode_length_tolerance: float` - This represents the $tolerance$ that determines the trace experience length T (the minimum T such that $\gamma^T < tolerance$).

`glie_mc_control` produces a generator (`Iterator`) of Q-Value Function estimates at the end of each trace experience. The code is fairly self-explanatory. The method `simulate_actions` of `mdp: MarkovDecisionProcess` creates a single sampling trace (i.e., a trace experience). At the end of each trace experience, the `update` method of `FunctionApprox` updates the Q-Value Function (creates a new Q-Value Function without mutating the currrent Q-Value Function) using each of the returns (and associated state-actions pairs) from the trace experience. The ϵ-greedy policy is derived from the Q-Value Function estimate by using the function `epsilon_greedy_policy` that is shown below and is quite self-explanatory.

```
from rl.markov_decision_process import epsilon_greedy_policy, TransitionStep
from rl.approximate_dynamic_programming import QValueFunctionApprox
from rl.approximate_dynamic_programming import NTStateDistribution

def glie_mc_control(
    mdp: MarkovDecisionProcess[S, A],
    states: NTStateDistribution[S],
    approx_0: QValueFunctionApprox[S, A],
    gamma: float,
    epsilon_as_func_of_episodes: Callable[[int], float],
    episode_length_tolerance: float = 1e-6
) -> Iterator[QValueFunctionApprox[S, A]]:
    q: QValueFunctionApprox[S, A] = approx_0
    p: Policy[S, A] = epsilon_greedy_policy(q, mdp, 1.0)
    yield q

    num_episodes: int = 0
    while True:
        trace: Iterable[TransitionStep[S, A]] = \
            mdp.simulate_actions(states, p)
        num_episodes += 1
        for step in returns(trace, gamma, episode_length_tolerance):
            q = q.update([((step.state, step.action), step.return_)])
        p = epsilon_greedy_policy(
            q,
            mdp,
            epsilon_as_func_of_episodes(num_episodes)
        )
        yield q
```

The implementation of `epsilon_greedy_policy` is as follows:

```
from rl.policy import DeterministicPolicy, Policy, RandomPolicy

def greedy_policy_from_qvf(
    q: QValueFunctionApprox[S, A],
    actions: Callable[[NonTerminal[S]], Iterable[A]]
) -> DeterministicPolicy[S, A]:
    def optimal_action(s: S) -> A:
        _, a = q.argmax((NonTerminal(s), a) for a in actions(NonTerminal(s)))
        return a
    return DeterministicPolicy(optimal_action)
```

```
def epsilon_greedy_policy(
    q: QValueFunctionApprox[S, A],
    mdp: MarkovDecisionProcess[S, A],
    epsilon: float = 0.0
) -> Policy[S, A]:
    def explore(s: S, mdp=mdp) -> Iterable[A]:
        return mdp.actions(NonTerminal(s))
    return RandomPolicy(Categorical(
        {UniformPolicy(explore): epsilon,
         greedy_policy_from_qvf(q, mdp.actions): 1 - epsilon}
    ))
```

The above code is in the file rl/monte_carlo.py.

Note that `epsilon_greedy_policy` returns an instance of the class `RandomPolicy`. `RandomPolicy` creates a policy that randomly selects one of several specified policies (in this case, we need to select between the greedy policy of type `DeterministicPolicy` and the `UniformPolicy`). The implementation of `RandomPolicy` is shown below and you can find it's code in the file rl/policy.py.

```
@dataclass(frozen=True)
class RandomPolicy(Policy[S, A]):
    policy_choices: Distribution[Policy[S, A]]

    def act(self, state: NonTerminal[S]) -> Distribution[A]:
        policy: Policy[S, A] = self.policy_choices.sample()
        return policy.act(state)
```

Now let us test `glie_mc_control` on the simple inventory MDP we wrote in Chapter **??**.

```
from rl.chapter3.simple_inventory_mdp_cap import SimpleInventoryMDPCap

capacity: int = 2
poisson_lambda: float = 1.0
holding_cost: float = 1.0
stockout_cost: float = 10.0
gamma: float = 0.9

si_mdp: SimpleInventoryMDPCap = SimpleInventoryMDPCap(
    capacity=capacity,
    poisson_lambda=poisson_lambda,
    holding_cost=holding_cost,
    stockout_cost=stockout_cost
)
```

First let's run Value Iteration so we can determine the true Optimal Value Function and Optimal Policy

```
from rl.dynamic_programming import value_iteration_result
true_opt_vf, true_opt_policy = value_iteration_result(fmdp, gamma=gamma)
print("True Optimal Value Function")
pprint(true_opt_vf)
print("True Optimal Policy")
print(true_opt_policy)
```

This prints:

```
True Optimal Value Function
{NonTerminal(state=InventoryState(on_hand=0, on_order=0)): -34.894855194671294,
 NonTerminal(state=InventoryState(on_hand=0, on_order=1)): -27.66095964467877,
 NonTerminal(state=InventoryState(on_hand=0, on_order=2)): -27.99189950444479,
 NonTerminal(state=InventoryState(on_hand=1, on_order=0)): -28.66095964467877,
```

```
 NonTerminal(state=InventoryState(on_hand=1, on_order=1)): -28.99189950444479,
 NonTerminal(state=InventoryState(on_hand=2, on_order=0)): -29.991899504444792}
True Optimal Policy
For State InventoryState(on_hand=0, on_order=0): Do Action 1
For State InventoryState(on_hand=0, on_order=1): Do Action 1
For State InventoryState(on_hand=0, on_order=2): Do Action 0
For State InventoryState(on_hand=1, on_order=0): Do Action 1
For State InventoryState(on_hand=1, on_order=1): Do Action 0
For State InventoryState(on_hand=2, on_order=0): Do Action 0
```

Now let's run GLIE MC Control with the following parameters:

```
from rl.function_approx import Tabular
from rl.distribution import Choose
from rl.chapter3.simple_inventory_mdp_cap import InventoryState

episode_length_tolerance: float = 1e-5
epsilon_as_func_of_episodes: Callable[[int], float] = lambda k: k ** -0.5
initial_learning_rate: float = 0.1
half_life: float = 10000.0
exponent: float = 1.0

initial_qvf_dict: Mapping[Tuple[NonTerminal[InventoryState], int], float] = {
    (s, a): 0. for s in si_mdp.non_terminal_states for a in si_mdp.actions(s)
}
learning_rate_func: Callable[[int], float] = learning_rate_schedule(
    initial_learning_rate=initial_learning_rate,
    half_life=half_life,
    exponent=exponent
)
qvfs: Iterator[QValueFunctionApprox[InventoryState, int] = glie_mc_control(
    mdp=si_mdp,
    states=Choose(si_mdp.non_terminal_states),
    approx_0=Tabular(
        values_map=initial_qvf_dict,
        count_to_weight_func=learning_rate_func
    ),
    gamma=gamma,
    epsilon_as_func_of_episodes=epsilon_as_func_of_episodes,
    episode_length_tolerance=episode_length_tolerance
)
```

Now let's fetch the final estimate of the Optimal Q-Value Function after `num_episodes` have run, and extract from it the estimate of the Optimal State-Value Function and the Optimal Policy.

```
from rl.distribution import Constant
from rl.dynamic_programming import V
import itertools
import rl.iterate as iterate

num_episodes = 10000
final_qvf: QValueFunctionApprox[InventoryState, int] = \
    iterate.last(itertools.islice(qvfs, num_episodes))

def get_vf_and_policy_from_qvf(
    mdp: FiniteMarkovDecisionProcess[S, A],
    qvf: QValueFunctionApprox[S, A]
) -> Tuple[V[S], FiniteDeterministicPolicy[S, A]]:
    opt_vf: V[S] = {
        s: max(qvf((s, a)) for a in mdp.actions(s))
        for s in mdp.non_terminal_states
    }
```

```
    opt_policy: FiniteDeterministicPolicy[S, A] = \
        FiniteDeterministicPolicy({
            s.state: qvf.argmax((s, a) for a in mdp.actions(s))[1]
            for s in mdp.non_terminal_states
        })
    return opt_vf, opt_policy

opt_vf, opt_policy = get_vf_and_policy_from_qvf(
    mdp=si_mdp,
    qvf=final_qvf
)
print(f"GLIE MC Optimal Value Function with {num_episodes:d} episodes")
pprint(opt_vf)
print(f"GLIE MC Optimal Policy with {num_episodes:d} episodes")
print(opt_policy)
```

This prints:

```
GLIE MC Optimal Value Function with 10000 episodes
{NonTerminal(state=InventoryState(on_hand=0, on_order=0)): -34.76212336633032,
 NonTerminal(state=InventoryState(on_hand=0, on_order=1)): -27.90668364332291,
 NonTerminal(state=InventoryState(on_hand=0, on_order=2)): -28.306190508518398,
 NonTerminal(state=InventoryState(on_hand=1, on_order=0)): -28.548284937363526,
 NonTerminal(state=InventoryState(on_hand=1, on_order=1)): -28.864409885059185,
 NonTerminal(state=InventoryState(on_hand=2, on_order=0)): -30.23156422557605}
GLIE MC Optimal Policy with 10000 episodes
For State InventoryState(on_hand=0, on_order=0): Do Action 1
For State InventoryState(on_hand=0, on_order=1): Do Action 1
For State InventoryState(on_hand=0, on_order=2): Do Action 0
For State InventoryState(on_hand=1, on_order=0): Do Action 1
For State InventoryState(on_hand=1, on_order=1): Do Action 0
For State InventoryState(on_hand=2, on_order=0): Do Action 0
```

We see that this reasonably converges to the true Value Function (and reaches the true Optimal Policy) as produced by Value Iteration.

The code above is in the file rl/chapter11/simple_inventory_mdp_cap.py. Also see the helper functions in rl/chapter11/control_utils.py which you can use to run your own experiments and tests for RL Control algorithms.

1.4 SARSA

Just like in the case of RL Prediction, the natural idea is to replace MC Control with TD Control using the TD Target $R_{t+1} + \gamma \cdot Q(S_{t+1}, A_{t+1}; \boldsymbol{w})$ as a biased estimate of G_t when updating $Q(S_t, A_t; \boldsymbol{w})$. This means the parameters update in Equation (1.4) gets modified to the following parameters update:

$$\Delta \boldsymbol{w} = \alpha \cdot (R_{t+1} + \gamma \cdot Q(S_{t+1}, A_{t+1}; \boldsymbol{w}) - Q(S_t, A_t; \boldsymbol{w})) \cdot \nabla_{\boldsymbol{w}} Q(S_t, A_t; \boldsymbol{w}) \tag{1.5}$$

Unlike MC Control where updates are made at the end of each trace experience (i.e., episode), a TD control algorithm can update at the end of each atomic experience. This means the Q-Value Function Approximation is updated after each atomic experience (*continuous learning*), which in turn means that the ϵ-greedy policy will be (automatically) updated at the end of each atomic experience. At each time step t in a trace experience, the

current ϵ-greedy policy is used to sample A_t from S_t and is also used to sample A_{t+1} from S_{t+1}. Note that in MC Control, the same ϵ-greedy policy is used to sample all the actions from their corresponding states in the trace experience, and so in MC Control, we were able to generate the entire trace experience with the currently available ϵ-greedy policy. However, here in TD Control, we need to generate a trace experience incrementally since the action to be taken from a state depends on the just-updated ϵ-greedy policy (that is derived from the just-updated Q-Value Function).

Just like in the case of RL Prediction, the disadvantage of the TD Target being a biased estimate of the return is compensated by a reduction in the variance of the return estimate. Also, TD Control offers a better speed of convergence (as we shall soon illustrate). Most importantly, TD Control offers the ability to use in situations where we have incomplete trace experiences (happens often in real-world situations where experiments gets curtailed/disrupted) and also, we can use it in situations where we never reach a terminal state (*continuing trace*).

Note that Equation (1.5) has the entities

- **S**tate S_t
- **A**ction A_t
- **R**eward R_t
- **S**tate S_{t+1}
- **A**ction A_{t+1}

which prompted this TD Control algorithm to be named SARSA (for **S**tate-**A**ction-**R**eward-**S**tate-**A**ction). Following our convention from Chapter **??**, we depict the SARSA algorithm in Figure 1.2 with states as elliptical-shaped nodes, actions as rectangular-shaped nodes, and the edges as samples from transition probability distribution and ϵ-greedy policy distribution.

Now let us write some code to implement the above-described SARSA algorithm. Let us start by understanding the various arguments to the below function `glie_sarsa`.

- `mdp: MarkovDecisionProcess[S, A]` - This represents the interface to an abstract Markov Decision Process. We want to remind that this interface doesn't provide any access to the transition probabilities or reward function. The core functionality available through this interface are the two `@abstractmethods` `step` and `actions`. The `step` method only allows us to access a sample of the next state and reward pair given the current state and action (since it returns an abstract `Distribution` object). The `actions` method gives us the allowable actions for a given state.
- `states: NTStateDistribution[S]` - This represents an arbitrary distribution of the non-terminal states, which in turn allows us to sample the starting state (from this distribution) for each trace experience.
- `approx_0: QValueFunctionApprox[S, A]` - This represents the initial function approximation of the Q-Value function (that is meant to be updated, after each atomic experience, in an immutable manner, through the course of the algorithm).
- `gamma: float` - This represents the discount factor to be used in estimating the Q-Value Function.
- `epsilon_as_func_of_episodes: Callable[[int], float]` - This represents the extent of exploration (ϵ) as a function of the number of episodes.
- `max_episode_length: int` - This represents the number of time steps at which we would curtail a trace experience and start a new one. As we've explained, TD Control doesn't require complete trace experiences, and so we can do as little or as large

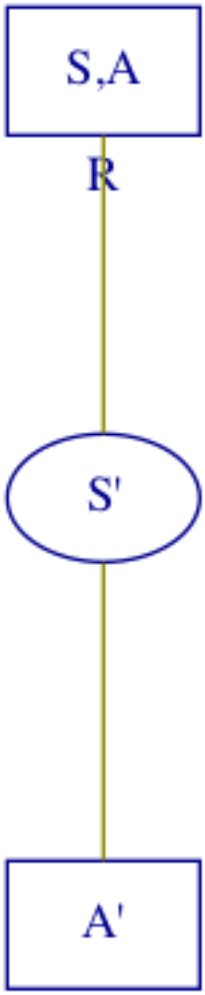

Figure 1.2: Visualization of SARSA Algorithm

a number of time steps in a trace experience (`max_episode_length` gives us that control).

`glie_sarsa` produces a generator (`Iterator`) of Q-Value Function estimates at the end of each atomic experience. The `while True` loops over trace experiences. The inner `while` loops over time steps - each of these steps involves the following:

- Given the current `state` and `action`, we obtain a sample of the pair of `next_state` and `reward` (using the `sample` method of the `Distribution` obtained from `mdp.step(state, action)`.
- Obtain the `next_action` from `next_state` using the function `epsilon_greedy_action` which utilizes the ϵ-greedy policy derived from the current Q-Value Function estimate (referenced by `q`).
- Update the Q-Value Function based on Equation (1.5) (using the `update` method of `q: QValueFunctionApprox[S, A]`). Note that this is an immutable update since we produce an `Iterable` (generator) of the Q-Value Function estimate after each time step.

Before the code for `glie_sarsa`, let's understand the code for `epsilon_greedy_action` which returns an action sampled from the ϵ-greedy policy probability distribution that is derived from the Q-Value Function estimate, given as input a non-terminal state, a Q-Value Function estimate, the set of allowable actions, and ϵ.

```
from operator import itemgetter
from Distribution import Categorical

def epsilon_greedy_action(
    q: QValueFunctionApprox[S, A],
    nt_state: NonTerminal[S],
    actions: Set[A],
    epsilon: float
) -> A:
    greedy_action: A = max(
        ((a, q((nt_state, a))) for a in actions),
        key=itemgetter(1)
    )[0]
    return Categorical(
        {a: epsilon / len(actions) +
         (1 - epsilon if a == greedy_action else 0.) for a in actions}
    ).sample()

def glie_sarsa(
    mdp: MarkovDecisionProcess[S, A],
    states: NTStateDistribution[S],
    approx_0: QValueFunctionApprox[S, A],
    gamma: float,
    epsilon_as_func_of_episodes: Callable[[int], float],
    max_episode_length: int
) -> Iterator[QValueFunctionApprox[S, A]]:
    q: QValueFunctionApprox[S, A] = approx_0
    yield q
    num_episodes: int = 0
    while True:
        num_episodes += 1
        epsilon: float = epsilon_as_func_of_episodes(num_episodes)
        state: NonTerminal[S] = states.sample()
        action: A = epsilon_greedy_action(
            q=q,
            nt_state=state,
            actions=set(mdp.actions(state)),
```

```
            epsilon=epsilon
        )
        steps: int = 0
        while isinstance(state, NonTerminal) and steps < max_episode_length:
            next_state, reward = mdp.step(state, action).sample()
            if isinstance(next_state, NonTerminal):
                next_action: A = epsilon_greedy_action(
                    q=q,
                    nt_state=next_state,
                    actions=set(mdp.actions(next_state)),
                    epsilon=epsilon
                )
                q = q.update([(
                    (state, action),
                    reward + gamma * q((next_state, next_action))
                )])
                action = next_action
            else:
                q = q.update([((state, action), reward)])
            yield q
            steps += 1
            state = next_state
```

The above code is in the file rl/td.py.

Let us test this on the simple inventory MDP we tested GLIE MC Control on (we use the same `si_mdp: SimpleInventoryMDPCap` object and the same parameter values that were set up earlier when testing GLIE MC Control).

```
from rl.chapter3.simple_inventory_mdp_cap import InventoryState

max_episode_length: int = 100
epsilon_as_func_of_episodes: Callable[[int], float] = lambda k: k ** -0.5
initial_learning_rate: float = 0.1
half_life: float = 10000.0
exponent: float = 1.0
gamma: float = 0.9

initial_qvf_dict: Mapping[Tuple[NonTerminal[InventoryState], int], float] = {
    (s, a): 0. for s in si_mdp.non_terminal_states for a in si_mdp.actions(s)
}
learning_rate_func: Callable[[int], float] = learning_rate_schedule(
    initial_learning_rate=initial_learning_rate,
    half_life=half_life,
    exponent=exponent
)
qvfs: Iterator[QValueFunctionApprox[InventoryState, int]] = glie_sarsa(
    mdp=si_mdp,
    states=Choose(si_mdp.non_terminal_states),
    approx_0=Tabular(
        values_map=initial_qvf_dict,
        count_to_weight_func=learning_rate_func
    ),
    gamma=gamma,
    epsilon_as_func_of_episodes=epsilon_as_func_of_episodes,
    max_episode_length=max_episode_length
)
```

Now let's fetch the final estimate of the Optimal Q-Value Function after `num_episodes * max_episode_length` updates of the Q-Value Function, and extract from it the estimate of the Optimal State-Value Function and the Optimal Policy (using the function `get_vf_and_policy_from_qvf` that we had written earlier).

```
import itertools
import rl.iterate as iterate
```

```
num_updates = num_episodes * max_episode_length
final_qvf: QValueFunctionApprox[InventoryState, int] = \
    iterate.last(itertools.islice(qvfs, num_updates))
opt_vf, opt_policy = get_vf_and_policy_from_qvf(
    mdp=si_mdp,
    qvf=final_qvf
)
print(f"GLIE SARSA Optimal Value Function with {num_updates:d} updates")
pprint(opt_vf)
print(f"GLIE SARSA Optimal Policy with {num_updates:d} updates")
print(opt_policy)
```

This prints:

```
GLIE SARSA Optimal Value Function with 1000000 updates
{NonTerminal(state=InventoryState(on_hand=0, on_order=0)): -35.05830797041331,
 NonTerminal(state=InventoryState(on_hand=0, on_order=1)): -27.8507256742493,
 NonTerminal(state=InventoryState(on_hand=0, on_order=2)): -27.735579652721434,
 NonTerminal(state=InventoryState(on_hand=1, on_order=0)): -28.984534974043097,
 NonTerminal(state=InventoryState(on_hand=1, on_order=1)): -29.325829885558885,
 NonTerminal(state=InventoryState(on_hand=2, on_order=0)): -30.236704327526777}
GLIE SARSA Optimal Policy with 1000000 updates
For State InventoryState(on_hand=0, on_order=0): Do Action 1
For State InventoryState(on_hand=0, on_order=1): Do Action 1
For State InventoryState(on_hand=0, on_order=2): Do Action 0
For State InventoryState(on_hand=1, on_order=0): Do Action 1
For State InventoryState(on_hand=1, on_order=1): Do Action 0
For State InventoryState(on_hand=2, on_order=0): Do Action 0
```

We see that this reasonably converges to the true Value Function (and reaches the true Optimal Policy) as produced by Value Iteration (whose results were displayed when we tested GLIE MC Control).

The code above is in the file rl/chapter11/simple_inventory_mdp_cap.py. Also see the helper functions in rl/chapter11/control_utils.py which you can use to run your own experiments and tests for RL Control algorithms.

For Tabular GLIE MC Control, we stated a theorem for theoretical guarantee of convergence to the true Optimal Value Function (and hence, true Optimal Policy). Is there something analogous for Tabular GLIE SARSA? This answers in the affirmative with the added condition that we reduce the learning rate according to the Robbins-Monro schedule. We state the following theorem without proof.

Theorem 1.4.1. *Tabular SARSA converges to the Optimal Action-Value function,* $Q(s,a) \to Q^*(s,a)$ *(hence, converges to an Optimal Deterministic Policy* π^**), under the following conditions:*

- *GLIE schedule of policies* $\pi_t(s,a)$
- *Robbins-Monro schedule of step-sizes* α_t*:*

$$\sum_{t=1}^{\infty} \alpha_t = \infty$$

$$\sum_{t=1}^{\infty} \alpha_t^2 < \infty$$

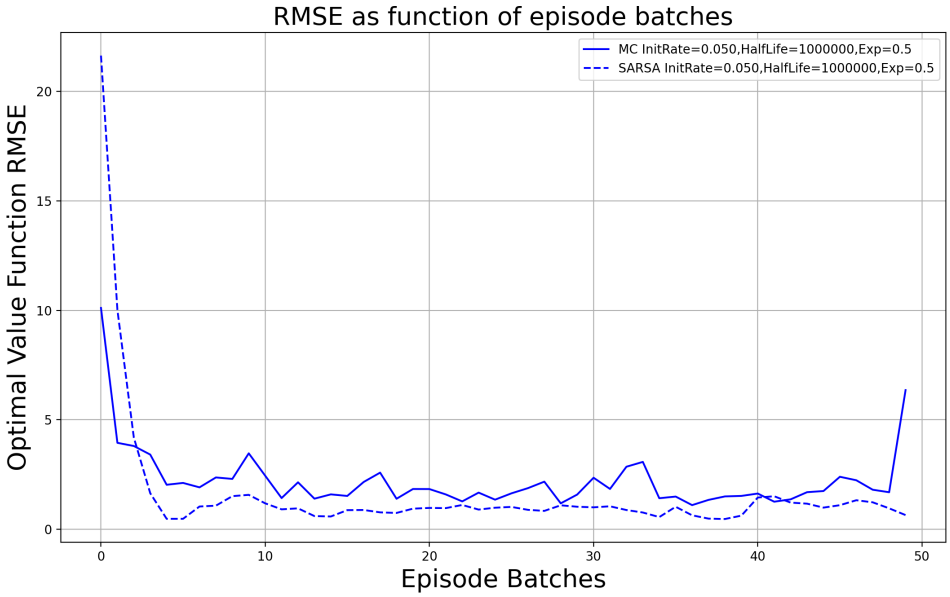

Figure 1.3: GLIE MC Control and GLIE SARSA Convergence for SimpleInventoryMDP-Cap

Now let's compare GLIE MC Control and GLIE SARSA. This comparison is analogous to the comparison in Section **??** in Chapter **??** regarding their bias, variance and convergence properties. GLIE SARSA carries a biased estimate of the Q-Value Function compared to the unbiased estimate of GLIE MC Control. On the flip side, the TD Target $R_{t+1} + \gamma \cdot Q(S_{t+1}, A_{t+1}; \boldsymbol{w})$ has much lower variance than G_t because G_t depends on many random state transitions and random rewards (on the remainder of the trace experience) whose variances accumulate, whereas the TD Target depends on only the next random state transition S_{t+1} and the next random reward R_{t+1}. The bad news with GLIE SARSA (due to the bias in it's update) is that with function approximation, it does not always converge to the Optimal Value Function/Policy.

As mentioned in Chapter **??**, because MC and TD have significant differences in their usage of data, nature of updates, and frequency of updates, it is not even clear how to create a level-playing field when comparing MC and TD for speed of convergence or for efficiency in usage of limited experiences data. The typical comparisons between MC and TD are done with constant learning rates, and it's been determined that practically GLIE SARSA learns faster than GLIE MC Control with constant learning rates. We illustrate this by running GLIE MC Control and GLIE SARSA on `SimpleInventoryMDPCap`, and plot the root-mean-squared-errors (RMSE) of the Q-Value Function estimates as a function of batches of episodes (i.e., visualize how the RMSE of the Q-Value Function evolves as the two algorithms progress). This is done by calling the function `compare_mc_sarsa_ql` which is in the file rl/chapter11/control_utils.py.

Figure 1.3 depicts the convergence for our implementations of GLIE MC Control and GLIE SARSA for a constant learning rate of $\alpha = 0.05$. We produced this Figure by using data from 500 episodes generated from the same `SimpleInventoryMDPCap` object we had created earlier (with same discount factor $\gamma = 0.9$). We plotted the RMSE after each batch of 10 episodes, hence both curves shown in the Figure have 50 RMSE data points plotted.

Firstly, we clearly see that MC Control has significantly more variance as evidenced by the choppy MC Control RMSE progression curve. Secondly, we note that the MC Control RMSE curve progresses quite quickly in the first few episode batches but is slow to converge after the first few episode batches (relative to the progression of SARSA). This results in SARSA reaching fairly small RMSE quicker than MC Control. This behavior of GLIE SARSA outperforming the comparable GLIE MC Control (with constant learning rate) is typical in most MDP Control problems.

Lastly, it's important to recognize that MC Control is not very sensitive to the initial Value Function while SARSA is more sensitive to the initial Value Function. We encourage you to play with the initial Value Function for this `SimpleInventoryMDPCap` example and evaluate how it affects the convergence speeds.

More generally, we encourage you to play with the `compare_mc_sarsa_ql` function on other MDP choices (ones we have created earlier in this book, or make up your own MDPs) so you can develop good intuition for how GLIE MC Control and GLIE SARSA algorithms converge for a variety of choices of learning rate schedules, initial Value Function choices, choices of discount factor etc.

1.5 SARSA(λ)

Much like how we extended TD Prediction to TD(λ) Prediction, we can extend SARSA to SARSA(λ), which gives us a way to tune the spectrum from MC Control to SARSA using the λ parameter. Recall that in order to develop TD(λ) Prediction from TD Prediction, we first developed the n-step TD Prediction Algorithm, then the Offline λ-Return TD Algorithm, and finally the Online TD(λ) Algorithm. We develop an analogous progression from SARSA to SARSA(λ).

So the first thing to do is to extend SARSA to 2-step-bootstrapped SARSA, whose update is as follows:

$$\Delta \boldsymbol{w} = \alpha \cdot (R_{t+1} + \gamma \cdot R_{t+2} + \gamma^2 \cdot Q(S_{t+2}, A_{t+2}; \boldsymbol{w}) - Q(S_t, A_t; \boldsymbol{w})) \cdot \nabla_{\boldsymbol{w}} Q(S_t, A_t; \boldsymbol{w})$$

Generalizing this to n-step-bootstrapped SARSA, the update would then be as follows:

$$\Delta \boldsymbol{w} = \alpha \cdot (G_{t,n} - Q(S_t, A_t; \boldsymbol{w})) \cdot \nabla_{\boldsymbol{w}} Q(S_t, A_t; \boldsymbol{w})$$

where the n-step-bootstrapped Return $G_{t,n}$ is defined as:

$$\begin{aligned} G_{t,n} &= \sum_{i=t+1}^{t+n} \gamma^{i-t-1} \cdot R_i + \gamma^n \cdot Q(S_{t+n}, A_{t+n}; \boldsymbol{w}) \\ &= R_{t+1} + \gamma \cdot R_{t+2} + \ldots + \gamma^{n-1} \cdot R_{t+n} + \gamma^n \cdot Q(S_{t+n}, A_{t+n}; \boldsymbol{w}) \end{aligned}$$

Instead of $G_{t,n}$, a valid target is a weighted-average target:

$$\sum_{n=1}^{N} u_n \cdot G_{t,n} + u \cdot G_t \text{ where } u + \sum_{n=1}^{N} u_n = 1$$

Any of the u_n or u can be 0, as long as they all sum up to 1. The λ-Return target is a special case of weights u_n and u, defined as follows:

$$u_n = (1 - \lambda) \cdot \lambda^{n-1} \text{ for all } n = 1, \ldots, T - t - 1$$

$$u_n = 0 \text{ for all } n \geq T - t \text{ and } u = \lambda^{T-t-1}$$

We denote the λ-Return target as $G_t^{(\lambda)}$, defined as:

$$G_t^{(\lambda)} = (1-\lambda) \cdot \sum_{n=1}^{T-t-1} \lambda^{n-1} \cdot G_{t,n} + \lambda^{T-t-1} \cdot G_t$$

Then, the Offline λ-Return SARSA Algorithm makes the following updates (performed at the end of each trace experience) for each (S_t, A_t) encountered in the trace experience:

$$\Delta \boldsymbol{w} = \alpha \cdot (G_t^{(\lambda)} - Q(S_t, A_t; \boldsymbol{w})) \cdot \nabla_{\boldsymbol{w}} Q(S_t, A_t; \boldsymbol{w})$$

Finally, we create the SARSA(λ) Algorithm, which is the online "version" of the above λ-Return SARSA Algorithm. The calculations/updates at each time step t for each trace experience are as follows:

$$\delta_t = R_{t+1} + \gamma \cdot Q(S_{t+1}, A_{t+1}; \boldsymbol{w}) - Q(S_t, A_t; \boldsymbol{w})$$

$$\boldsymbol{E}_t = \gamma\lambda \cdot \boldsymbol{E}_{t-1} + \nabla_{\boldsymbol{w}} Q(S_t, A_t; \boldsymbol{w})$$

$$\Delta \boldsymbol{w} = \alpha \cdot \delta_t \cdot \boldsymbol{E}_t$$

with the eligiblity traces initialized at time 0 for each trace experience as $\boldsymbol{E}_0 = \nabla_{\boldsymbol{w}} V(S_0; \boldsymbol{w})$. Note that just like in SARSA, the ϵ-greedy policy improvement is automatic from the updated Q-Value Function estimate after each time step.

We leave the implementation of SARSA(λ) in Python code as an exercise for you to do.

1.6 Off-Policy Control

All control algorithms face a tension between wanting to learn Q-Values contingent on *subsequent optimal behavior* versus wanting to explore all actions. This almost seems contradictory because the quest for exploration deters one from optimal behavior. Our approach so far of pursuing an ϵ-greedy policy (to be thought of as an *almost optimal* policy) is a hack to resolve this tension. A cleaner approach is to use two separate policies for the two separate goals of wanting to be optimal and wanting to explore. The first policy is the one that we learn about (which eventually becomes the optimal policy) - we call this policy the *Target Policy* (to signify the "target" of Control). The second policy is the one that behaves in an exploratory manner, so we can obtain sufficient data for all actions, enabling us to adequately estimate the Q-Value Function - we call this policy the *Behavior Policy*.

In SARSA, at a given time step, we are in a current state S, take action A, after which we obtain the reward R and next state S', upon which we take the next action A'. The action A taken from the current state S is meant to come from an exploratory policy (behavior policy) so that for each state S, we have adequate occurrences of all actions in order to accurately estimate the Q-Value Function. The action A' taken from the next state S' is meant to come from the target policy as we aim for *subsequent optimal behavior* ($Q^*(S, A)$ requires optimal behavior subsequent to taking action A). However, in the SARSA algorithm, the behavior policy producing A from S and the target policy producing A' from S' are in fact the same policy - the ϵ-greedy policy. Algorithms such as SARSA in which the behavior policy is the same as the target policy are refered to as On-Policy Algorithms to indicate

the fact that the behavior used to generate data (experiences) does not deviate from the policy we are aiming for (target policy, which drives towards the optimal policy).

The separation of behavior policy and target policy as two separate policies gives us algorithms that are known as Off-Policy Algorithms to indicate the fact that the behavior policy is allowed to "deviate off" from the target policy. This separation enables us to construct more general and more powerful RL algorithms. We will use the notation π for the target policy and the notation μ for the behavior policy - therefore, we say that Off-Policy algorithms estimate the Value Function for target policy π while following behavior policy μ. Off-Policy algorithms can be very valuable in real-world situations where we can learn the target policy π by observing humans or other AI agents who follow a behavior policy μ. Another great practical benefit is to be able to re-use prior experiences that were generated from old policies, say $\pi_1, \pi_2, \ldots$. Yet another powerful benefit is that we can learn multiple policies $\mu_1, \mu_2, \ldots$ while following one behavior policy π. Let's now make the concept of Off-Policy Learning concrete by covering the most basic (and most famous) Off-Policy Control Algorithm, which goes by the name of Q-Learning.

1.6.1 Q-Learning

The best way to understand the (Off-Policy) Q-Learning algorithm is to tweak SARSA to make it Off-Policy. Instead of having both the action A and the next action A' being generated by the same ϵ-greedy policy, we generate (i.e., sample) action A (from state S) using an exploratory behavior policy μ and we generate the next action A' (from next state S') using the target policy π. The behavior policy can be any policy as long as it is exploratory enough to be able to obtain sufficient data for all actions (in order to obtain an adequate estimate of the Q-Value Function). Note that in SARSA, when we roll over to the next (new) time step, the new time step's state S is set to be equal to the previous time step's next state S' and the new time step's action A is set to be equal to the previous time step's next action A'. However, in Q-Learning, we only set the new time step's state S to be equal to the previous time step's next state S'. The action A for the new time step will be generated using the behavior policy μ, and won't be equal to the previous time step's next action A' (that would have been generated using the target policy π).

This Q-Learning idea of two separate policies - behavior policy and target policy - is fairly generic, and can be used in algorithms beyond solving the Control problem. However, here we are interested in Q-Learning for Control and so, we want to ensure that the target policy eventually becomes the optimal policy. One straightforward way to accomplish this is to make the target policy equal to the deterministic greedy policy derived from the Q-Value Function estimate at every step. Thus, the update for Q-Learning Control algorithm is as follows:

$$\Delta \boldsymbol{w} = \alpha \cdot \delta_t \cdot \nabla_{\boldsymbol{w}} Q(S_t, A_t; \boldsymbol{w})$$

where

$$\begin{aligned}\delta_t &= R_{t+1} + \gamma \cdot Q(S_{t+1}, \arg\max_{a \in \mathcal{A}} Q(S_{t+1}, a; \boldsymbol{w}); \boldsymbol{w}) - Q(S_t, A_t; \boldsymbol{w}) \\ &= R_{t+1} + \gamma \cdot \max_{a \in \mathcal{A}} Q(S_{t+1}, a; \boldsymbol{w}) - Q(S_t, A_t; \boldsymbol{w})\end{aligned}$$

Following our convention from Chapter **??**, we depict the Q-Learning algorithm in Figure 1.4 with states as elliptical-shaped nodes, actions as rectangular-shaped nodes, and the edges as samples from transition probability distribution and action choices.

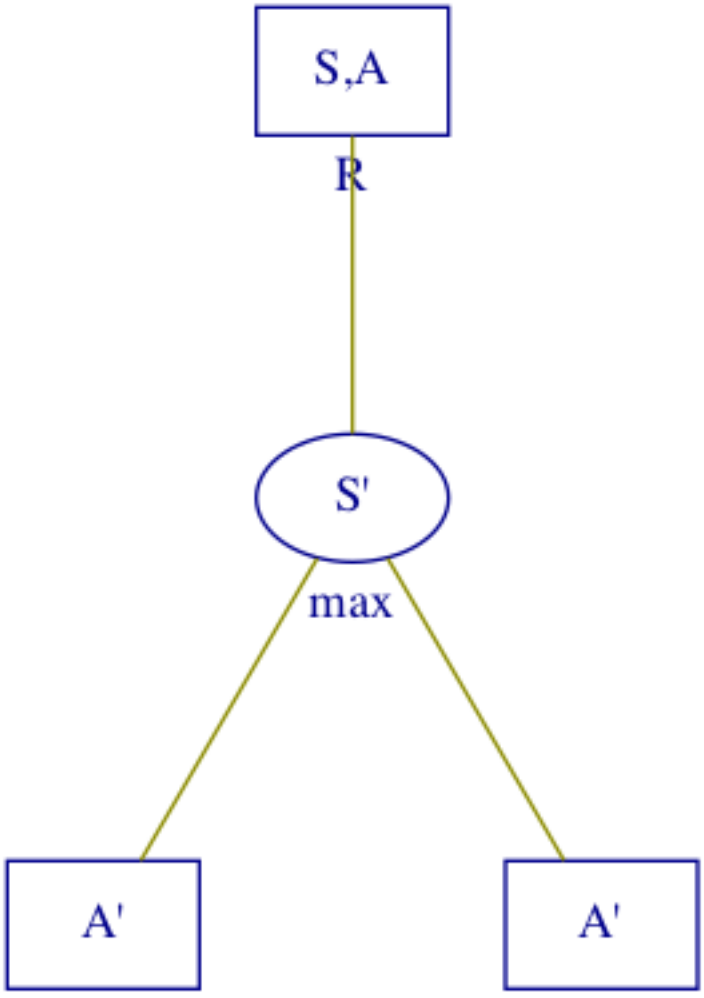

Figure 1.4: Visualization of Q-Learning Algorithm

Although we have highlighted some attractive features of Q-Learning (on account of being Off-Policy), it turns out that Q-Learning when combined with function approximation of the Q-Value Function leads to convergence issues (more on this later). However, Tabular Q-Learning converges under the usual appropriate conditions. There is considerable literature on convergence of Tabular Q-Learning and we won't go over those convergence theorems in this book - here it suffices to say that the convergence proofs for Tabular Q-Learning require infinite exploration of all (state, action) pairs and appropriate stochastic approximation conditions for step sizes.

Now let us write some code for Q-Learning. The function `q_learning` below is quite similar to the function `glie_sarsa` we wrote earlier. Here are the differences:

- `q_learning` takes as input the argument `policy_from_q: PolicyFromQType`, which is a function with two arguments - a Q-Value Function and a `MarkovDecisionProcess` object - and returns the policy derived from the Q-Value Function. Thus, `q_learning` takes as input a general behavior policy whereas `glie_sarsa` uses the ϵ-greedy policy as it's behavior (and target) policy. However, you should note that the typical descriptions of Q-Learning in the RL literature specialize the behavior policy to be the ϵ-greedy policy (we've simply chosen to describe and implement Q-Learning in it's more general form of using an arbitrary user-specified behavior policy).
- `glie_sarsa` takes as input `epsilon_as_func_of_episodes: Callable[[int], float]` whereas `q_learning` doesn't require this argument (Q-Learning can converge even if it's behavior policy has an unchanging ϵ, and any ϵ specification in `q_learning` would be built into the `policy_from_q` argument).
- As explained above, in `q_learning`, the `action` from the `state` is obtained using the specified behavior policy `policy_from_q` and the "next action" from the `next_state` is implicitly obtained using the deterministic greedy policy derived from the Q-Value Function estimate `q`. In `glie_sarsa`, both `action` and `next_action` were obtained from the ϵ-greedy policy.
- As explained above, in `q_learning`, as we move to the next time step, we set `state` to be equal to the previous time step's `next_state` whereas in `glie_sarsa`, we not only do this but we also set `action` to be equal to the previous time step's `next_action`.

```
PolicyFromQType = Callable[
    [QValueFunctionApprox[S, A], MarkovDecisionProcess[S, A]],
    Policy[S, A]
]

def q_learning(
    mdp: MarkovDecisionProcess[S, A],
    policy_from_q: PolicyFromQType,
    states: NTStateDistribution[S],
    approx_0: QValueFunctionApprox[S, A],
    gamma: float,
    max_episode_length: int
) -> Iterator[QValueFunctionApprox[S, A]]:
    q: QValueFunctionApprox[S, A] = approx_0
    yield q
    while True:
        state: NonTerminal[S] = states.sample()
        steps: int = 0
        while isinstance(state, NonTerminal) and steps < max_episode_length:
            policy: Policy[S, A] = policy_from_q(q, mdp)
            action: A = policy.act(state).sample()
            next_state, reward = mdp.step(state, action).sample()
            next_return: float = max(
                q((next_state, a))
```

```
            for a in mdp.actions(next_state)
        ) if isinstance(next_state, NonTerminal) else 0.
        q = q.update([((state, action), reward + gamma * next_return)])
        yield q
        steps += 1
        state = next_state
```

The above code is in the file rl/td.py. Much like how we tested GLIE SARSA on `SimpleInventoryMDPCap`, the code in the file rl/chapter11/simple_inventory_mdp_cap.py also tests Q-Learning on `SimpleInventoryMDPCap`. We encourage you to leverage the helper functions in rl/chapter11/control_utils.py to run your own experiments and tests for Q-Learning. In particular, the functions for Q-Learning in rl/chapter11/control_utils.py employ the common practice of using the ϵ-greedy policy as the behavior policy.

1.6.2 Windy Grid

Now we cover an interesting Control problem that is quite popular in the RL literature - how to navigate a "Windy Grid." We have added some bells and whistles to this problem to make it more interesting. We want to evaluate SARSA and Q-Learning on this problem. Here's the detailed description of this problem:

We are given a grid comprising of cells arranged in the form of m rows and n columns, defined as $\mathcal{G} = \{(i, j) \mid 0 \leq i < m, 0 \leq j < n\}$. A subset of $\mathcal{G}$ (denoted $\mathcal{B}$) are uninhabitable cells known as *blocks*. A subset of $\mathcal{G} - \mathcal{B}$ (denoted $\mathcal{T}$) is known as the set of goal cells. We have to find a least-cost path from each of the cells in $\mathcal{G} - \mathcal{B} - \mathcal{T}$ to any of the cells in $\mathcal{T}$. At each step, we are required to make a move to a non-block cell (we cannot remain stationary). Right after we make our move, a random vertical wind could move us one cell up or down, unless limited by a block or limited by the boundary of the grid.

Each column has it's own random wind specification given by two parameters $0 \leq p_1 \leq 1$ and $0 \leq p_2 \leq 1$ with $p_1 + p_2 \leq 1$. The wind blows downwards with probability p_1, upwards with probability p_2, and there is no wind with probability $1 - p_1 - p_2$. If the wind makes us bump against a block or against the boundary of the grid, we incur a cost of $b \in \mathbb{R}^+$ in addition to the usual cost of 1 for each move we make. Thus, here the cost includes not just the time spent on making the moves, but also the cost of bumping against blocks or against the boundary of the grid (due to the wind). Minimizing the expected total cost amounts to finding our way to a goal state in a manner that combines minimization of the number of moves with the minimization of the hurt caused by bumping (assume discount factor of 1 when minimizing this expected total cost). If the wind causes us to bump against a wall or against a boundary, we bounce and settle in the cell we moved to just before being blown by the wind (note that the wind blows immediately after we make a move). The wind will never move us by more than one cell between two successive moves, and the wind is never horizontal. Note also that if we move to a goal cell, the process ends immediately without any wind-blow following the movement to the goal cell. The random wind for all the columns is specified as a sequence $[(p_{1,j}, p_{2,j}) \mid 0 \leq j < n]$.

Let us model this problem of minimizing the expected total cost while reaching a goal cell as a Finite Markov Decision Process.

State Space $\mathcal{S} = \mathcal{G} - \mathcal{B}$, Non-Terminal States $\mathcal{N} = \mathcal{G} - \mathcal{B} - \mathcal{T}$, Terminal States are $\mathcal{T}$.

We denote the set of all possible moves {UP, DOWN, LEFT, RIGHT} as:

$\mathcal{A} = \{(1, 0), (-1, 0), (0, -1), (0, 1)\}$.

The actions $\mathcal{A}(s)$ for a given non-terminal state $s \in \mathcal{N}$ is defined as: $\{a \mid a \in \mathcal{A}, s+a \in \mathcal{S}\}$ where $+$ denotes element-wise addition of integer 2-tuples.

For all $(s_r, s_c) \in \mathcal{N}$, for all $(a_r, a_c) \in \mathcal{A}((s_r, s_c))$, if $(s_r + a_r, s_c + a_c) \in \mathcal{T}$, then:

$$\mathcal{P}_R((s_r, s_c), (a_r, a_c), -1, (s_r + a_r, s_c + a_c)) = 1$$

For all $(s_r, s_c) \in \mathcal{N}$, for all $(a_r, a_c) \in \mathcal{A}((s_r, s_c))$, if $(s_r + a_r, s_c + a_c) \in \mathcal{N}$, then:

$$\mathcal{P}_R((s_r, s_c), (a_r, a_c), -1 - b, (s_r + a_r, s_c + a_c)) = p_{1,s_c+a_c} \cdot \mathbb{I}_{(s_r+a_r-1,s_c+a_c) \notin \mathcal{S}} + p_{2,s_c+a_c} \cdot \mathbb{I}_{(s_r+a_r+1,s_c+a_c) \notin \mathcal{S}}$$

$$\mathcal{P}_R((s_r, s_c), (a_r, a_c), -1, (s_r + a_r - 1, s_c + a_c)) = p_{1,s_c+a_c} \cdot \mathbb{I}_{(s_r+a_r-1,s_c+a_c) \in \mathcal{S}}$$

$$\mathcal{P}_R((s_r, s_c), (a_r, a_c), -1, (s_r + a_r + 1, s_c + a_c)) = p_{2,s_c+a_c} \cdot \mathbb{I}_{(s_r+a_r+1,s_c+a_c) \in \mathcal{S}}$$

$$\mathcal{P}_R((s_r, s_c), (a_r, a_c), -1, (s_r + a_r, s_c + a_c)) = 1 - p_{1,s_c+a_c} - p_{2,s_c+a_c}$$

Discount Factor $\gamma = 1$

Now let's write some code to model this problem with the above MDP spec, and run Value Iteration, SARSA and Q-Learning as three different ways of solving this MDP Control problem.

We start with the problem specification in the form of a Python class `WindyGrid` and write some helper functions before getting into the MDP creation and DP/RL algorithms.

```
'''
Cell specifies (row, column) coordinate
'''
Cell = Tuple[int, int]
CellSet = Set[Cell]
Move = Tuple[int, int]
'''
WindSpec specifies a random vertical wind for each column.
Each random vertical wind is specified by a (p1, p2) pair
where p1 specifies probability of Downward Wind (could take you
one step lower in row coordinate unless prevented by a block or
boundary) and p2 specifies probability of Upward Wind (could take
you onw step higher in column coordinate unless prevented by a
block or boundary). If one bumps against a block or boundary, one
incurs a bump cost and doesn't move. The remaining probability
1- p1 - p2 corresponds to No Wind.
'''
WindSpec = Sequence[Tuple[float, float]]

possible_moves: Mapping[Move, str] = {
    (-1, 0): 'D',
    (1, 0): 'U',
    (0, -1): 'L',
    (0, 1): 'R'
}

@dataclass(frozen=True)
class WindyGrid:

    rows: int  # number of grid rows
    columns: int  # number of grid columns
    blocks: CellSet  # coordinates of block cells
    terminals: CellSet  # coordinates of goal cells
    wind: WindSpec  # spec of vertical random wind for the columns
    bump_cost: float  # cost of bumping against block or boundary

    @staticmethod
    def add_move_to_cell(cell: Cell, move: Move) -> Cell:
        return cell[0] + move[0], cell[1] + move[1]

    def is_valid_state(self, cell: Cell) -> bool:
        '''
```

```
        checks if a cell is a valid state of the MDP
        '''
        return 0 <= cell[0] < self.rows and 0 <= cell[1] < self.columns \
            and cell not in self.blocks

    def get_all_nt_states(self) -> CellSet:
        '''
        returns all the non-terminal states
        '''
        return {(i, j) for i in range(self.rows) for j in range(self.columns)
                if (i, j) not in set.union(self.blocks, self.terminals)}

    def get_actions_and_next_states(self, nt_state: Cell) \
            -> Set[Tuple[Move, Cell]]:
        '''
        given a non-terminal state, returns the set of all possible
        (action, next_state) pairs
        '''
        temp: Set[Tuple[Move, Cell]] = {(a, WindyGrid.add_move_to_cell(
            nt_state,
            a
        )) for a in possible_moves}
        return {(a, s) for a, s in temp if self.is_valid_state(s)}
```

Next we write a method to calculate the transition probabilities. The code below should be self-explanatory and mimics the description of the problem above and the mathematical specification of the transition probabilities given above.

```
from rl.distribution import Categorical

    def get_transition_probabilities(self, nt_state: Cell) \
            -> Mapping[Move, Categorical[Tuple[Cell, float]]]:
        '''
        given a non-terminal state, return a dictionary whose
        keys are the valid actions (moves) from the given state
        and the corresponding values are the associated probabilities
        (following that move) of the (next_state, reward) pairs.
        The probabilities are determined from the wind probabilities
        of the column one is in after the move. Note that if one moves
        to a goal cell (terminal state), then one ends up in that
        goal cell with 100% probability (i.e., no wind exposure in a
        goal cell).
        '''
        d: Dict[Move, Categorical[Tuple[Cell, float]]] = {}
        for a, (r, c) in self.get_actions_and_next_states(nt_state):
            if (r, c) in self.terminals:
                d[a] = Categorical({((r, c), -1.): 1.})
            else:
                down_prob, up_prob = self.wind[c]
                stay_prob: float = 1. - down_prob - up_prob
                d1: Dict[Tuple[Cell, float], float] = \
                    {((r, c), -1.): stay_prob}
                if self.is_valid_state((r - 1, c)):
                    d1[((r - 1, c), -1.)] = down_prob
                if self.is_valid_state((r + 1, c)):
                    d1[((r + 1, c), -1.)] = up_prob
                d1[((r, c), -1. - self.bump_cost)] = \
                    down_prob * (1 - self.is_valid_state((r - 1, c))) + \
                    up_prob * (1 - self.is_valid_state((r + 1, c)))
                d[a] = Categorical(d1)
        return d
```

Next we write a method to create the `MarkovDecisionProcess` for the Windy Grid.

```
from rl.markov_decision_process import FiniteMarkovDecisionProcess
```

```
    def get_finite_mdp(self) -> FiniteMarkovDecisionProcess[Cell, Move]:
        '''
        returns the FiniteMarkovDecision object for this windy grid problem
        '''
        return FiniteMarkovDecisionProcess(
            {s: self.get_transition_probabilities(s) for s in
             self.get_all_nt_states()}
        )
```

Next we write methods for Value Iteration, SARSA and Q-Learning

```
from rl.markov_decision_process import FiniteDeterministicPolicy
from rl.dynamic_programming import value_iteration_result, V
from rl.chapter11.control_utils import glie_sarsa_finite_learning_rate
from rl.chapter11.control_utils import q_learning_finite_learning_rate
from rl.chapter11.control_utils import get_vf_and_policy_from_qvf
    def get_vi_vf_and_policy(self) -> \
            Tuple[V[Cell], FiniteDeterministicPolicy[Cell, Move]]:
        '''
        Performs the Value Iteration DP algorithm returning the
        Optimal Value Function (as a V[Cell]) and the Optimal Policy
        (as a FiniteDeterministicPolicy[Cell, Move])
        '''
        return value_iteration_result(self.get_finite_mdp(), gamma=1.)

    def get_glie_sarsa_vf_and_policy(
        self,
        epsilon_as_func_of_episodes: Callable[[int], float],
        learning_rate: float,
        num_updates: int
    ) -> Tuple[V[Cell], FiniteDeterministicPolicy[Cell, Move]]:
        qvfs: Iterator[QValueFunctionApprox[Cell, Move]] = \
            glie_sarsa_finite_learning_rate(
                fmdp=self.get_finite_mdp(),
                initial_learning_rate=learning_rate,
                half_life=1e8,
                exponent=1.0,
                gamma=1.0,
                epsilon_as_func_of_episodes=epsilon_as_func_of_episodes,
                max_episode_length=int(1e8)
            )
        final_qvf: QValueFunctionApprox[Cell, Move] = \
            iterate.last(itertools.islice(qvfs, num_updates))
        return get_vf_and_policy_from_qvf(
            mdp=self.get_finite_mdp(),
            qvf=final_qvf
        )

    def get_q_learning_vf_and_policy(
        self,
        epsilon: float,
        learning_rate: float,
        num_updates: int
    ) -> Tuple[V[Cell], FiniteDeterministicPolicy[Cell, Move]]:
        qvfs: Iterator[QValueFunctionApprox[Cell, Move]] = \
            q_learning_finite_learning_rate(
                fmdp=self.get_finite_mdp(),
                initial_learning_rate=learning_rate,
                half_life=1e8,
                exponent=1.0,
                gamma=1.0,
                epsilon=epsilon,
                max_episode_length=int(1e8)
            )
```

```
        final_qvf: QValueFunctionApprox[Cell, Move] = \
            iterate.last(itertools.islice(qvfs, num_updates))
        return get_vf_and_policy_from_qvf(
            mdp=self.get_finite_mdp(),
            qvf=final_qvf
        )
```

The above code is in the file rl/chapter11/windy_grid.py. Note that this file also contains some helpful printing functions that pretty-prints the grid, along with the calculated Optimal Value Functions and Optimal Policies. The method `print_wind_and_bumps` prints the column wind probabilities and the cost of bumping into a block/boundary. The method `print_vf_and_policy` prints a given Value Function and a given Deterministic Policy - this method can be used to print the Optimal Value Function and Optimal Policy produced by Value Iteration, by SARSA and by Q-Learning. In the printing of a deterministic policy, "X" represents a block, "T" represents a terminal cell, and the characters "L," "R," "D," "U" represent "Left," "Right," "Down," "Up" moves respectively.

Now let's run our code on a small instance of a Windy Grid.

```
wg = WindyGrid(
    rows=5,
    columns=5,
    blocks={(0, 1), (0, 2), (0, 4), (2, 3), (3, 0), (4, 0)},
    terminals={(3, 4)},
    wind=[(0., 0.9), (0.0, 0.8), (0.7, 0.0), (0.8, 0.0), (0.9, 0.0)],
    bump_cost=4.0
)
wg.print_wind_and_bumps()
vi_vf_dict, vi_policy = wg.get_vi_vf_and_policy()
print("Value Iteration\n")
wg.print_vf_and_policy(
    vf_dict=vi_vf_dict,
    policy=vi_policy
)
epsilon_as_func_of_episodes: Callable[[int], float] = lambda k: 1. / k
learning_rate: float = 0.03
num_updates: int = 100000
sarsa_vf_dict, sarsa_policy = wg.get_glie_sarsa_vf_and_policy(
    epsilon_as_func_of_episodes=epsilon_as_func_of_episodes,
    learning_rate=learning_rate,
    num_updates=num_updates
)
print("SARSA\n")
wg.print_vf_and_policy(
    vf_dict=sarsa_vf_dict,
    policy=sarsa_policy
)
epsilon: float = 0.2
ql_vf_dict, ql_policy = wg.get_q_learning_vf_and_policy(
    epsilon=epsilon,
    learning_rate=learning_rate,
    num_updates=num_updates
)
print("Q-Learning\n")
wg.print_vf_and_policy(
    vf_dict=ql_vf_dict,
    policy=ql_policy
)
```

This prints the following:

```
Column 0: Down Prob = 0.00, Up Prob = 0.90
```

```
Column 1: Down Prob = 0.00, Up Prob = 0.80
Column 2: Down Prob = 0.70, Up Prob = 0.00
Column 3: Down Prob = 0.80, Up Prob = 0.00
Column 4: Down Prob = 0.90, Up Prob = 0.00
Bump Cost = 4.00

Value Iteration

       0     1     2     3     4
 4 XXXXX  5.25  2.02  1.10  1.00
 3 XXXXX  8.53  5.20  1.00  0.00
 2  9.21  6.90  8.53 XXXXX  1.00
 1  8.36  9.21  8.36 12.16 11.00
 0 10.12 XXXXX XXXXX 17.16 XXXXX

    0  1  2  3  4
 4  X  R  R  R  D
 3  X  R  R  R  T
 2  R  U  U  X  U
 1  R  U  L  L  U
 0  U  X  X  U  X

SARSA

       0     1     2     3     4
 4 XXXXX  5.47  2.02  1.08  1.00
 3 XXXXX  8.78  5.37  1.00  0.00
 2  9.14  7.03  8.29 XXXXX  1.00
 1  8.51  9.16  8.27 11.92 12.58
 0 10.05 XXXXX XXXXX 16.48 XXXXX

    0  1  2  3  4
 4  X  R  R  R  D
 3  X  R  R  R  T
 2  R  U  U  X  U
 1  R  U  L  L  U
 0  U  X  X  U  X

Q-Learning

       0     1     2     3     4
 4 XXXXX  5.45  2.02  1.09  1.00
 3 XXXXX  8.09  5.12  1.00  0.00
 2  8.78  6.76  7.92 XXXXX  1.00
 1  8.31  8.85  8.09 11.52 10.93
 0  9.85 XXXXX XXXXX 16.16 XXXXX

    0  1  2  3  4
 4  X  R  R  R  D
```

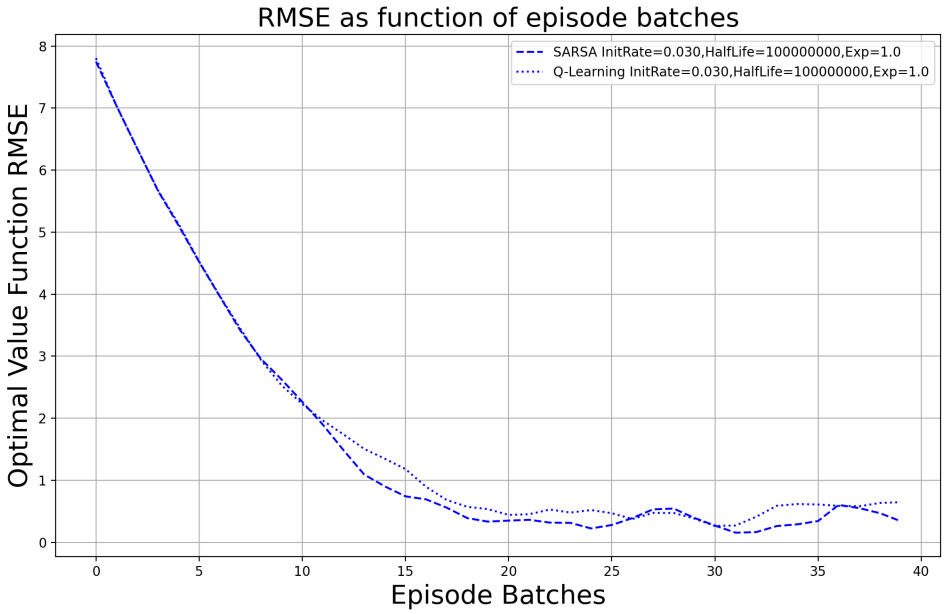

Figure 1.5: GLIE SARSA and Q-Learning Convergence for Windy Grid (Bump Cost = 4)

```
3  X  R  R  R  T
2  R  U  U  X  U
1  R  U  L  L  U
0  U  X  X  U  X
```

Value Iteration should be considered as the benchmark since it calculates the Optimal Value Function within the default tolerance of 1e-5. We see that both SARSA and Q-Learning get fairly close to the Optimal Value Function after only 100,000 updates (i.e., 100,000 moves across various episodes). We also see that both SARSA and Q-Learning obtain the true Optimal Policy, consistent with Value Iteration.

Now let's explore SARSA and Q-Learning's speed of convergence to the Optimal Value Function.

We first run GLIE SARSA and Q-Learning for the above settings of bump cost = 4.0, GLIE SARSA $\epsilon(k) = \frac{1}{k}$, Q-Learning $\epsilon = 0.2$. Figure 1.5 depicts the trajectory of Root-Mean-Squared-Error (RMSE) of the Q-Values relative to the Q-Values obtained by Value Iteration. The RMSE is plotted as a function of progressive batches of 10 episodes. We can see that GLIE SARSA and Q-Learning have roughly the same convergence trajectory.

Now let us set the bump cost to a very high value of 100,000. Figure 1.6 depicts the convergence trajectory for bump cost of 100,000. We see that Q-Learning converges much faster than GLIE SARSA (we kept GLIE SARSA $\epsilon(k) = \frac{1}{k}$ and Q-Learning $\epsilon = 0.2$). So why does Q-Learning do better? Q-Learning has two advantages over GLIE SARSA here: Firstly, it's behavior policy is exploring at the constant amount of 20% whereas GLIE SARSA's exploration declines to 10% after just the 10th episode. This means Q-Learning gets sufficient data quicker than GLIE SARSA for the entire set of (state, action) pairs. Secondly, Q-Learning's target policy is greedy, versus GLIE SARSA's declining-ϵ-greedy. This means GLIE SARSA's Optimal Q-Value estimation is compromised due to the exploration of actions in it's target policy (rather than a pure exploitation with max over actions,

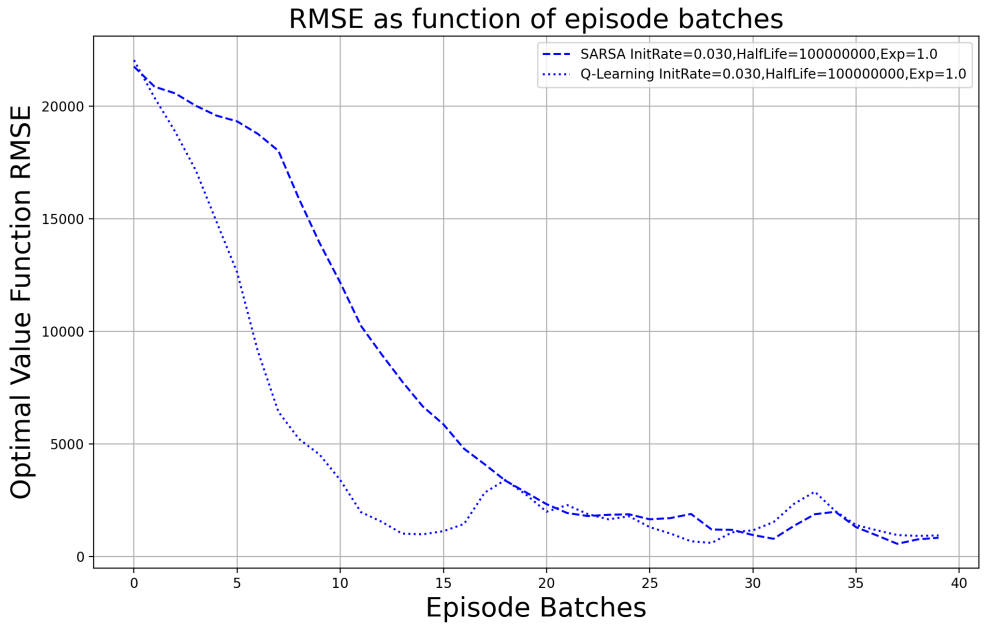

Figure 1.6: GLIE SARSA and Q-Learning Convergence for Windy Grid (Bump Cost = 100,000)

as is the case with Q-Learning). Thus, the separation between behavior policy and target policy in Q-Learning fetches it the best of both worlds and enables it to perform better than GLIE SARSA in this example.

SARSA is a more "conservative" algorithm in the sense that if there is a risk of a large negative reward close to the optimal path, SARSA will tend to avoid that dangerous optimal path and only slowly learn to use that optimal path when ϵ (exploration) reduces sufficiently. Q-Learning, on the other hand, will tend to take that risk while exploring and learns fast through "big failures." This provides us with a guide on when to use SARSA and when to use Q-Learning. Roughly speaking, use SARSA if you are training your AI agent with interaction with the real environment where you care about time and money consumed while doing the training with real environment-interaction (eg: you don't want to risk damaging a robot by walking it towards an optimal path in the proximity of physical danger). On the other hand, use Q-Learning if you are training your AI agent with a simulated environment where large negative rewards don't cause actual time/money losses, but these large negative rewards help the AI agent learn quickly. In a financial trading example, if you are training your RL agent in a real trading environment, you'd want to use SARSA as Q-Learning can potentially incur big losses while SARSA (although slower in learning) will avoid real trading losses during the process of learning. On the other hand, if you are training your RL agent in a simulated trading environment, Q-Learning is the way to go as it will learn fast by incuring "paper trading" losses as part of the process of executing risky trades.

Note that Q-Learning (and Off-policy Learning in general) has higher per-sample variance than SARSA, which could lead to problems in convergence, especially when we employ function approximation for the Q-Value Function. Q-Learning has been shown to be particularly problematic in converging when using neural networks for it's Q-Value

function approximation.

The SARSA algorithm was introduced in a paper by Rummery and Niranjan (Rummery and Niranjan 1994). The Q-Learning algorithm was introduced in the Ph.D. thesis of Chris Watkins (Watkins 1989).

1.6.3 Importance Sampling

Now that we've got a good grip of Off-Policy Learning through the Q-Learning algorithm, we show a very different (arguably simpler) method of doing Off-Policy Learning. This method is known as Importance Sampling, a fairly general technique (beyond RL) for estimating properties of a particular probability distribution, while only having access to samples of a different probability distribution. Specializing this technique to Off-Policy Control, we estimate the Value Function for the target policy (probability distribution of interest) while having access to samples generated from the probability distribution of the behavior policy. Specifically, Importance Sampling enables us to calculate $\mathbb{E}_{X\sim P}[f(X)]$ (where P is the probability distribution of interest), given samples from probability distribution Q, as follows:

$$\begin{aligned}\mathbb{E}_{X\sim P}[f(X)] &= \sum P(X)\cdot f(X)\\ &= \sum Q(X)\cdot \frac{P(X)}{Q(X)}\cdot f(X)\\ &= \mathbb{E}_{X\sim Q}[\frac{P(X)}{Q(X)}\cdot f(X)]\end{aligned}$$

So basically, the function $f(X)$ of samples X are scaled by the ratio of the probabilities $P(X)$ and $Q(X)$.

Let's employ this Importance Sampling method for Off-Policy Monte Carlo Prediction, where we need to estimate the Value Function for policy π while only having access to trace experience returns generated using policy μ. The idea is straightforward - we simply weight the returns G_t according to the similarity between policies π and μ, by multiplying importance sampling corrections along whole episodes. Let us define ρ_t as the product of the ratio of action probabilities (on the two policies π and μ) from time t to time $T-1$ (assume episode ends at time T). Specifically,

$$\rho_t = \frac{\pi(S_t,A_t)}{\mu(S_t,A_t)}\cdot\frac{\pi(S_{t+1},A_{t+1})}{\mu(S_{t+1},A_{t+1})}\cdots\frac{\pi(S_{T-1},A_{T-1})}{\mu(S_{T-1},A_{T-1})}$$

We've learnt in Chapter **??** that the learning rate α (treated as an update step-size) serves as a weight to the update target (in the case of MC, the update target is the return G_t). So all we have to do is to scale the step-size α for the update for time t by ρ_t. Hence, the MC Prediction update is tweaked to be the following when doing Off-Policy with Importance Sampling:

$$\Delta\boldsymbol{w} = \alpha\cdot\rho_t\cdot(G_t - V(S_t;\boldsymbol{w}))\cdot\nabla_{\boldsymbol{w}}V(S_t;\boldsymbol{w})$$

For MC Control, we make the analogous tweak to the update for the Q-Value Function, as follows:

$$\Delta\boldsymbol{w} = \alpha\cdot\rho_t\cdot(G_t - Q(S_t,A_t;\boldsymbol{w}))\cdot\nabla_{\boldsymbol{w}}Q(S_t,A_t;\boldsymbol{w})$$

Note that we cannot use this method if μ is zero when π is non-zero (since μ is in the denominator).

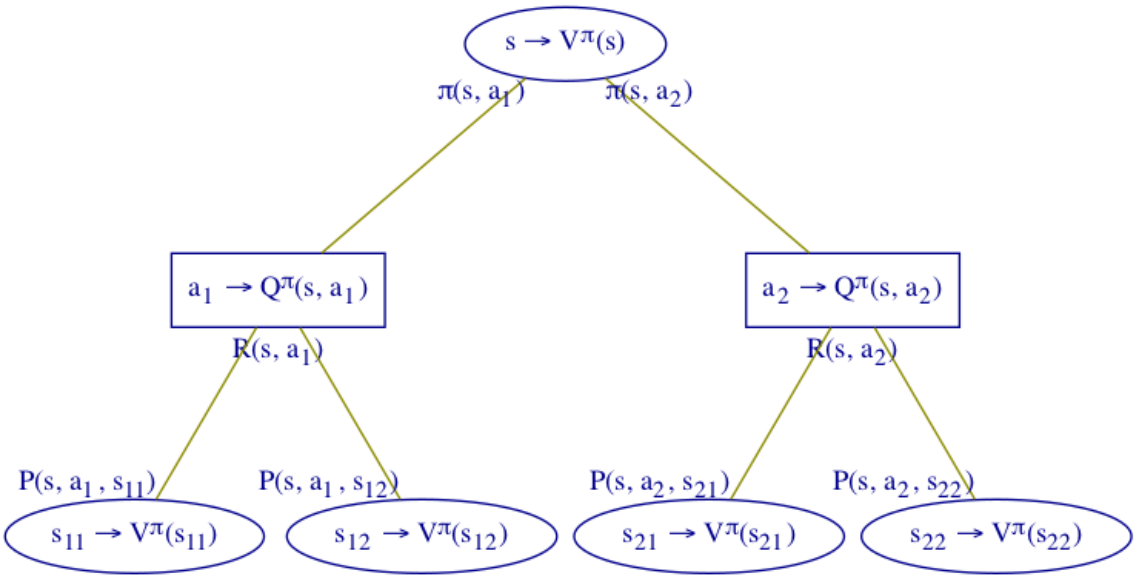

Figure 1.7: Policy Evaluation (DP Algorithm with Full Backup)

A key disadvantage of Off-Policy MC with Importance Sampling is that it dramatically increases the variance of the Value Function estimate. To contain the variance, we can use TD targets (instead of trace experience returns) generated from μ to evaluate the Value Function for π. For Off-Policy TD Prediction, we essentially weight TD target $R + \gamma \cdot V(S'; \boldsymbol{w})$ with importance sampling. Here we only need a single importance sampling correction, as follows:

$$\Delta \boldsymbol{w} = \alpha \cdot \frac{\pi(S_t, A_t)}{\mu(S_t, A_t)} \cdot (R_{t+1} + \gamma \cdot V(S_{t+1}; \boldsymbol{w}) - V(S_t; \boldsymbol{w})) \cdot \nabla_{\boldsymbol{w}} V(S_t; \boldsymbol{w})$$

For TD Control, we do the analogous update for the Q-Value Function:

$$\Delta \boldsymbol{w} = \alpha \cdot \frac{\pi(S_t, A_t)}{\mu(S_t, A_t)} \cdot (R_{t+1} + \gamma \cdot Q(S_{t+1}, A_{t+1}; \boldsymbol{w}) - Q(S_t, A_t; \boldsymbol{w})) \cdot \nabla_{\boldsymbol{w}} Q(S_t, A_t; \boldsymbol{w})$$

This has much lower variance than MC importance sampling. A key advantage of TD importance sampling is that policies only need to be similar over a single time step.

Since the modifications from On-Policy algorithms to Off-Policy algorithms based on Importance Sampling are just a small tweak of scaling the update by importance sampling corrections, we won't implement the Off-Policy Importance Sampling algorithms in Python code. However, we encourage you to implement the Prediction and Control MC and TD Off-Policy algorithms (based on Importance Sampling) described above.

1.7 Conceptual Linkage between DP and TD algorithms

It's worthwhile placing RL algorithms in terms of their conceptual relationship to DP algorithms. Let's start with the Prediction problem, whose solution is based on the Bellman Expectation Equation. Figure 1.8 depicts TD Prediction, which is the sample backup version of Policy Evaluation, depicted in Figure 1.7 as a full backup DP algorithm.

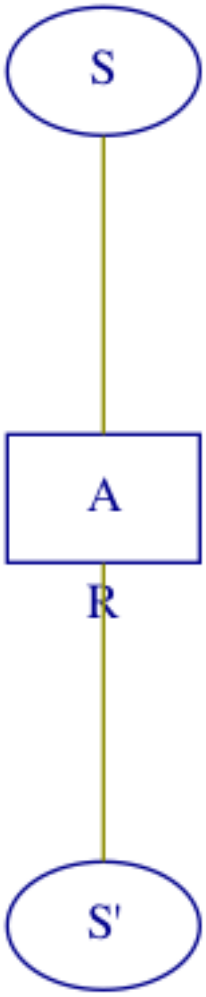

Figure 1.8: TD Prediction (RL Algorithm with Sample Backup)

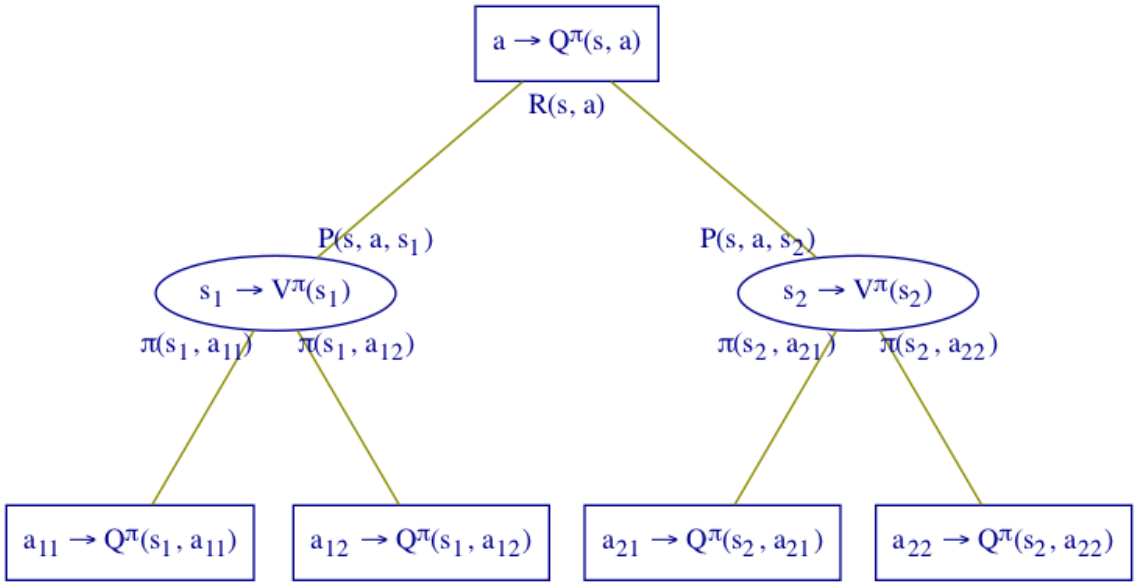

Figure 1.9: Q-Policy Iteration (DP Algorithm with Full Backup)

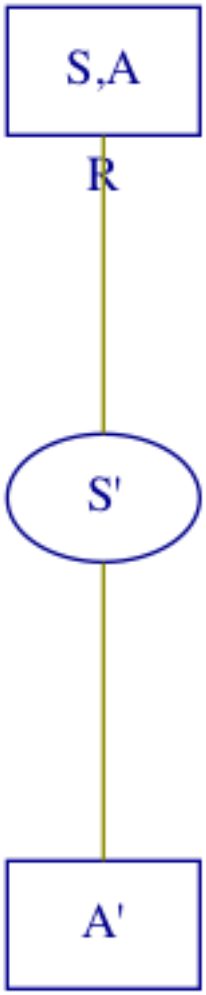

Figure 1.10: SARSA (RL Algorithm with Sample Backup)

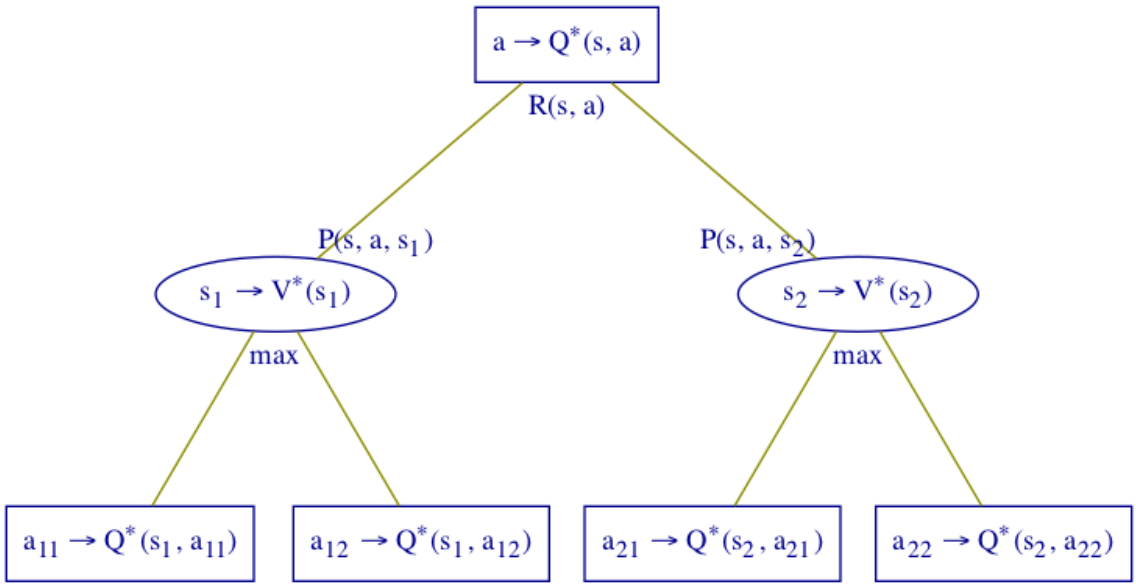

Figure 1.11: Q-Value Iteration (DP Algorithm with Full Backup)

Likewise, Figure 1.10 depicts SARSA, which is the sample backup version of Q-Policy Iteration (Policy Iteration on Q-Value), depicted in Figure 1.9 as a full backup DP algorithm.

Finally, Figure 1.12 depicts Q-Learning, which is the sample backup version of Q-Value Iteration (Value Iteration on Q-Value), depicted in Figure 1.11 as a full backup DP algorithm.

The table in Figure 1.13 summarizes these RL algorithms, along with their corresponding DP algorithms, showing the expectation targets of the DP algorithms' updates along with the corresponding sample targets of the RL algorithms' updates.

1.8 Convergence of RL Algorithms

Now we provide an overview of convergence of RL Algorithms. Let us start with RL Prediction. Figure 1.14 provides the overview of RL Prediction. As you can see, Monte-Carlo Prediction has convergence guarantees, whether On-Policy or Off-Policy, whether Tabular or with Function Approximation (even with non-linear Function Approximation). However, Temporal-Difference Prediction can have convergence issues - the core reason for this is that the TD update is not a true gradient update (as we've explained in Chapter **??**, it is a *semi-gradient* update). As you can see, although we have convergence guarantees for On-Policy TD Prediction with linear function approximation, there is no convergence guarantee for On-Policy TD Prediction with non-linear function approximation. The situation is even worse for Off-Policy TD Prediction - there is no convergence guarantee even for linear function approximation.

We want to highlight a confluence pattern in RL Algorithms where convergence problems arise. As a rule of thumb, if we do all of the following three, then we run into convergence problems.

- Bootstrapping, i.e., updating with a target that involves the current Value Function estimate (as is the case with Temporal-Difference)

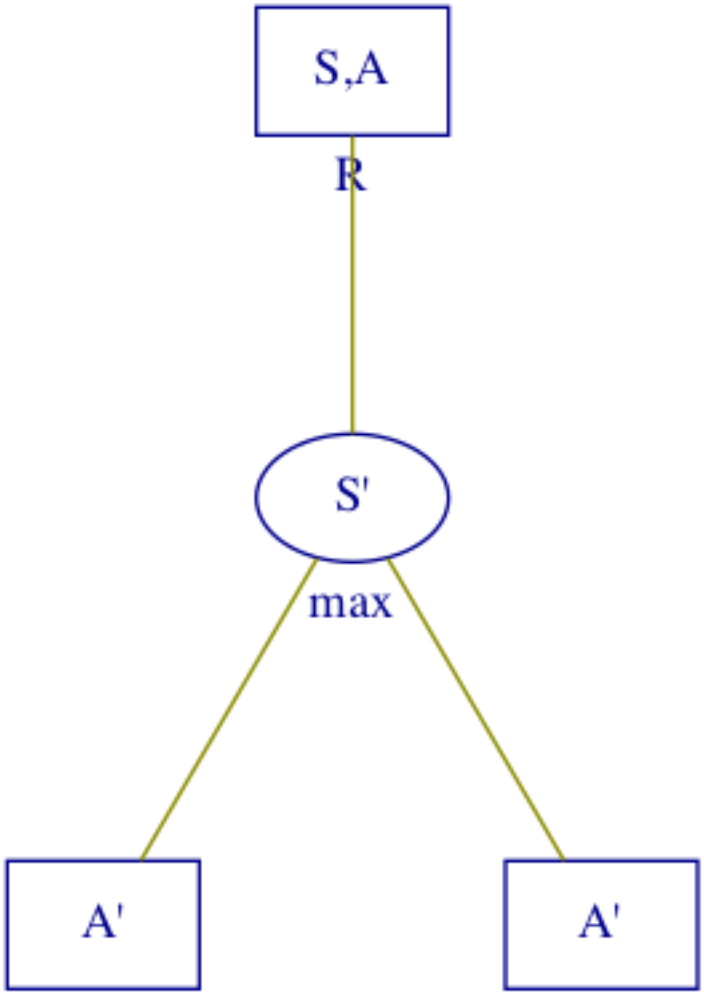

Figure 1.12: Q-Learning (RL Algorithm with Sample Backup)

Full Backup (DP)	**Sample Backup (TD)**
Policy Evaluation's $V(S)$ update: $\mathbb{E}[R + \gamma V(S')\|S]$	TD Learning's $V(S)$ update: sample $R + \gamma V(S')$
Q-Policy Iteration's $Q(S, A)$ update: $\mathbb{E}[R + \gamma Q(S', A')\|S, A]$	SARSA's $Q(S, A)$ update: sample $R + \gamma Q(S', A')$
Q-Value Iteration's $Q(S, A)$ update: $\mathbb{E}[R + \gamma \max_{a'} Q(S', a')\|S, A]$	Q-Learning's $Q(S, A)$ update: sample $R + \gamma \max_{a'} Q(S', a')$

Figure 1.13: Relationship between DP and RL algorithms

On/Off Policy	Algorithm	Tabular	Linear	Non-Linear
On-Policy	MC	✓	✓	✓
	TD(0)	✓	✓	✗
	TD(λ)	✓	✓	✗
Off-Policy	MC	✓	✓	✓
	TD(0)	✓	✗	✗
	TD(λ)	✓	✗	✗

Figure 1.14: Convergence of RL Prediction Algorithms

On/Off Policy	Algorithm	Tabular	Linear	Non-Linear
On-Policy	MC	✓	✓	✓
	TD	✓	✓	✗
	Gradient TD	✓	✓	✓
Off-Policy	MC	✓	✓	✓
	TD	✓	✗	✗
	Gradient TD	✓	✓	✓

Figure 1.15: Convergence of RL Prediction Algorithms, including Gradient TD

- Off-Policy
- Function Approximation of the Value Function

Hence, [Bootstrapping, Off-Policy, Function Approximation] is known as the *Deadly Triad*, a term emphasized and popularized by Richard Sutton in a number of publications and lectures. We should highlight that the *Deadly Triad* phenomenon is not a theorem - rather, it should be viewed as a rough pattern and as a rule of thumb. So to achieve convergence, we avoid at least one of the above three. We have seen that each of [Bootstrapping, Off-Policy, Function Approximation] provides benefits, but when all three come together, we run into convergence problems. The fundamental problem is that semi-gradient bootstrapping does not follow the gradient of *any* objective function and this causes TD to diverge when running off-policy and when using function approximations.

Function Approximation is typically unavoidable in real-world problems because of the size of real-world problems. So we are looking at avoiding semi-gradient bootstrapping or avoiding off-policy. Note that semi-gradient bootstrapping can be mitigated by tuning the TD λ parameter to a high-enough value. However, if we want to get around this problem in a fundamental manner, we can avoid the core issue of semi-gradient bootstrapping by instead doing a *true gradient* with a method known as *Gradient Temporal-Difference* (or *Gradient TD*, for short). We will cover Gradient TD in detail in Chapter **??**, but for now, we want to simply share that Gradient TD updates the value function approximation's parameters with the actual gradient (not semi-gradient) of an appropriate loss function and the gradient formula involves bootstrapping. Thus, it avails of the advantages of bootstrapping without the disadvantages of semi-gradient (which we cheekily refered to as "cheating" in Chapter **??**). Figure 1.15 expands upon Figure 1.14 by incorporating convergence properties of Gradient TD.

Now let's move on to convergence of Control Algorithms. Figure 1.16 provides the picture. (✓) means it doesn't quite hit the Optimal Value Function, but bounces around near

Algorithm	Tabular	Linear	Non-Linear
MC Control	✓	(✓)	✗
SARSA	✓	(✓)	✗
Q-Learning	✓	✗	✗
Gradient Q-Learning	✓	✓	✗

Figure 1.16: Convergence of RL Control Algorithms

the Optimal Value Function. Gradient Q-Learning is the adaptation of Q-Learning with Gradient TD. So this method is Off-Policy, is bootstrapped, but avoids semi-gradient. This enables it to converge for linear function approximations. However, it diverges when used with non-linear function approximations. So, for Control, even with Gradient TD, the deadly triad still exists for a combination of [Bootstrapping, Off-Policy, Non-Linear Function Approximation]. In Chapter **??**, we shall cover the DQN algorithm which is an innovative and practically effective method for getting around the deadly triad for RL Control.

1.9 Key Takeaways from this Chapter

- RL Control is based on the idea of Generalized Policy Iteration (GPI):
  - Policy Evaluation with Q-Value Function (instead of State-Value Function V)
  - Improved Policy needs to be exploratory, eg: ϵ-greedy
- On-Policy versus Off-Policy (eg: SARSA versus Q-Learning)
- Deadly Triad := [Bootstrapping, Off-Policy, Function Approximation]

Bibliography

Rummery, G. A., and M. Niranjan. 1994. "On-Line Q-Learning Using Connectionist Systems." CUED/F-INFENG/TR-166. Engineering Department, Cambridge University.
Watkins, C. J. C. H. 1989. "Learning from Delayed Rewards." PhD thesis, King's College, Oxford.