Foundations of Reinforcement Learning with Applications in Finance

Ashwin Rao, Tikhon Jelvis

Part I.

Reinforcement Learning Algorithms

1. Monte-Carlo and Temporal-Difference for Prediction

1.1. Overview of the Reinforcement Learning approach

In Module I, we covered Dynamic Programming (DP) and Approximate Dynamic Programming (ADP) algorithms to solve the problems of Prediction and Control. DP and ADP algorithms assume that we have access to a *model* of the MDP environment (by *model*, we mean the transitions defined by $\mathcal{P}_R$ - notation from Chapter **??** - refering to probabilities of next state and reward, given current state and action). However, in real-world situations, we often do not have access to a model of the MDP environment and so, we'd need to access the actual (real) MDP environment directly. As an example, a robotics application might not have access to a model of a certain type of terrain to learn to walk on, and so we'd need to access the actual (physical) terrain. This means we'd need to *interact* with the real MDP environment. Note that the real MDP environment doesn't give us transition probabilities - it simple serves up a new state and reward when we take an action in a certain state. In other words, it gives us individual experiences of next state and reward, rather than the explicit probabilities of occurrence of next states and rewards. So, the natural question to ask is whether we can infer the Optimal Value Function/Optimal Policy without access to a model (in the case of Prediction - the question is whether we can infer the Value Function for a given policy). The answer to this question is *Yes* and the algorithms that achieve this are known as Reinforcement Learning algorithms.

But Reinforcement Learning is often a great option even in situations where we do have a model. In typical real-world problems, the state space is large and the transitions structure is complex, so transition probabilities are either hard to compute or impossible to store/compute (within practical storage/compute constraints). This means even if we could *theoretically* estimate a model from interactions with the real environment and then run a DP/ADP algorithm, it's typically intractable/infeasible in a typical real-world problem. Moreover, a typical real-world environment is not stationary (meaning the probabilities $\mathcal{P}_R$ change over time) and so, the $\mathcal{P}_R$ probabilities model would need to be re-estimated periodically (making it cumbersome to do DP/ADP). All of this points to the practical alternative of constructing a *sampling model* (a model that serves up samples of next state and reward) - this is typically much more feasible than estimating a *probabilities model* (i.e. a model of explicit transition probabilities). A sampling model can then be used by a Reinforcement Learning algorithm (as we shall explain shortly).

So what we are saying is that practically we are left with one of the following two options:

1. The AI Agent interacts with the real environment and doesn't bother with either a model of explicit transition probabilities (*probabilities model*) or a model of transition samples (*sampling model*).
2. We create a sampling model (by learning from interaction with the real environment) and treat this sampling model as a simulated environment (meaning, the AI agent interacts with this simulated environment).

From the perspective of the AI agent, either way there is an environment interface that will serve up (at each time step) a single experience of (next state, reward) pair when the agent performs a certain action in a given state. So essentially, either way, our access is simply to a stream of individual experiences of next state and reward rather than their explicit probabilities. So, then the question is - at a conceptual level, how does RL go about solving Prediction and Control problems with just this limited access (access to only experiences and not explicit probabilities)? This will become clearer and clearer as we make our way through Module III, but it would be a good idea now for us to briefly sketch an intuitive overview of the RL approach (before we dive into the actual RL algorithms).

To understand the core idea of how RL works, we take you back to the start of the book where we went over how a baby learns to walk. Specifically, we'd like you to develop intuition for how humans and other animals learn to perform requisite tasks or behave in appropriate ways, so as to get trained to make suitable decisions. Humans/animals don't build a model of explicit probabilities in their minds in a way that a DP/ADP algorithm would require. Rather, their learning is essentially a sort of "trial and error" method - they try an action, receive an experience (i.e., next state and reward) from their environment, then take a new action, receive another experience, and so on ... and then over a period of time, they figure out which actions might be leading to good outcomes (producing good rewards) and which actions might be leading to poor outcomes (poor rewards). This learning process involves raising the priority of actions perceived as good, and lowering the priority of actions perceived as bad. Humans/animals don't quite link their actions to the immediate reward - they link their actions to the cumulative rewards (*Return*s) obtained after performing an action. Linking actions to cumulative rewards is challenging because multiple actions have significantly overlapping rewards sequences, and often rewards show up in a delayed manner. Indeed, learning by attributing good versus bad outcomes to specific past actions is the powerful part of human/animal learning. Humans/animals are essentially estimating a Q-Value Function and are updating their Q-Value function each time they receive a new experience (of essentially a pair of next state and reward). Exactly how humans/animals manage to estimate Q-Value functions efficiently is unclear (a big area of ongoing research), but RL algorithms have specific techniques to estimate the Q-Value function in an incremental manner by updating the Q-Value function in subtle ways after each experience of next state and reward received from either the real environment or simulated environment.

We should also point out another important feature of human/animal learning - it is the fact that humans/animals are good at generalizing their inferences from experiences, i.e., they can interpolate and extrapolate the linkages between their actions and the outcomes received from their environment. Technically, this translates to a suitable function approximation of the Q-Value function. So before we embark on studying the details of various RL algorithms, it's important to recognize that RL overcomes complexity (specifically, the Curse of Dimensionality and Curse of Modeling, as we have alluded to in previous chapters) with a combination of:

1. Learning from individual experiences of next state and reward received after performing actions in specific states. 2. Good generalization ability of the Q-Value function with a suitable function approximation (indeed, recent progress in capabilities of deep neural networks have helped considerably).

This idea of solving the MDP Prediction and Control problems in this manner (learning from a stream of experiences data with appropriate generalization ability in the Q-Value function approximation) came from the Ph.D. thesis of Chris Watkins (Watkins 1989). As mentioned before, we consider the RL book by Sutton and Barto (Sutton and Barto 2018)

as the best source for a comprehensive study of RL algorithms as well as the best source for all references associated with RL (hence, we don't provide too many references in this book).

As mentioned in previous chapters, most RL algorithms are founded on the Bellman Equations and all RL Control algorithms are based on the fundamental idea of *Generalized Policy Iteration* that we have explained in Chapter **??**. But the exact ways in which the Bellman Equations and Generalized Policy Iteration idea are utilized in RL algorithms differ from one algorithm to another, and they differ significantly from how the Bellman Equations/Generalized Policy Iteration idea is utilized in DP algorithms.

As has been our practice, we start with the Prediction problem (this chapter) and then cover the Control problem (next chapter).

1.2. RL for Prediction

We re-use a lot of the notation we had developed in Module I. As a reminder, Prediction is the problem of estimating the Value Function of an MDP for a given policy π. We know from Chapter **??** that this is equivalent to estimating the Value Function of the π-implied MRP. So in this chapter, we assume that we are working with an MRP (rather than an MDP) and we assume that the MRP is available in the form of an interface that serves up an individual experience of (next state, reward) pair, given current state. The interface might be a real environment or a simulated environment. We refer to the agent's receipt of an individual experience of (next state, reward), given current state, as an *atomic experience*. Interacting with this interface in succession (starting from a state S_0) gives us a *trace experience* consisting of alternating states and rewards as follows:

$$S_0, R_1, S_1, R_2, S_2, \ldots$$

Given a stream of atomic experiences or a stream of trace experiences, the RL Prediction problem is to estimate the *Value Function* $V : \mathcal{N} \to \mathbb{R}$ of the MRP defined as:

$$V(s) = \mathbb{E}[G_t | S_t = s] \text{ for all } s \in \mathcal{N}, \text{ for all } t = 0, 1, 2, \ldots$$

where the *Return* G_t for each $t = 0, 1, 2, \ldots$ is defined as:

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

We use the above definition of *Return* even for a terminating trace experience (say terminating at $t = T$, i.e., $S_T \in \mathcal{T}$), by treating $R_i = 0$ for all $i > T$.

The RL prediction algorithms we will soon develop consume a stream of atomic experiences or a stream of trace experiences to learn the requisite Value Function. So we want the input to an RL Prediction algorithm to be either an `Iterable` of atomic experiences or an `Iterable` of trace experiences. Now let's talk about the representation (in code) of a single atomic experience and the representation of a single trace experience. We take you back to the code in Chapter **??** where we had set up a `@dataclass TransitionStep` that served as a building block in the method `simulate_reward` in the abstract class `MarkovRewardProcess`.

```
@dataclass(frozen=True)
class TransitionStep(Generic[S]):
    state: NonTerminal[S]
    next_state: State[S]
    reward: float
```

`TransitionStep[S]` represents a single atomic experience. `simulate_reward` produces an `Iterator[TransitionStep[S]]` (i.e., a stream of atomic experiences in the form of a sampling trace) but in general, we can represent a single trace experience as an `Iterable[TransitionStep[S]]` (i.e., a sequence *or* stream of atomic experiences). Therefore, we want the input to an RL prediction algorithm to be either:

- an `Iterable[TransitionStep[S]]` representing a stream of atomic experiences
- an `Iterable[Iterable[TransitionStep[S]]]` representing a stream of trace experiences

Let's add a method `reward_traces` to `MarkovRewardProcess` that produces an `Iterator` (stream) of the sampling traces produced by `simulate_reward`.[1] So then we'd be able to use the output of `reward_traces` as the `Iterable[Iterable[TransitionStep[S]]]` input to an RL Prediction algorithm. Note that the input `start_state_distribution` is the specification of the probability distribution of start states (state from which we start a sampling trace that can be used as a trace experience).

```
    def reward_traces(
            self,
            start_state_distribution: Distribution[NonTerminal[S]]
    ) -> Iterable[Iterable[TransitionStep[S]]]:
        while True:
            yield self.simulate_reward(start_state_distribution)
```

1.3. Monte-Carlo (MC) Prediction

Monte-Carlo (MC) Prediction is a very simple RL algorithm that performs supervised learning to predict the expected return from any state of an MRP (i.e., it estimates the Value Function of an MRP), given a stream of trace experiences. Note that we wrote the abstract class `FunctionApprox` in Chapter **??** for supervised learning that takes data in the form of (x, y) pairs where x is the predictor variable and $y \in \mathbb{R}$ is the response variable. For the Monte-Carlo prediction problem, the x-values are the encountered states across the stream of input trace experiences and the y-values are the associated returns on the trace experiences (starting from the corresponding encountered state). The following function (in the file rl/monte_carlo.py) `mc_prediction` takes as input an `Iterable` of trace experiences, with each trace experience represented as an `Iterable` of `TransitionSteps`. `mc_prediction` performs the requisite supervised learning in an incremental manner, by calling the method `iterate_updates` of `approx_0: ValueFunctionApprox[S]` on an `Iterator` of (state, return) pairs that are extracted from each trace experience. As a reminder, the method `iterate_updates` calls the method `update` of `FunctionApprox` iteratively (in this case, each call to `update` updates the `ValueFunctionApprox` for a single (state, return) data point). `mc_prediction` produces as output an `Iterator` of `ValueFunctionApprox[S]`, i.e., an updated function approximation of the Value Function at the end of each trace experience (note that function approximation updates can be done only at the end of trace experiences because the trace experience returns are available only at the end of trace experiences).

```
import MarkovRewardProcess as mp

def mc_prediction(
    traces: Iterable[Iterable[mp.TransitionStep[S]]],
    approx_0: ValueFunctionApprox[S],
```

[1] `reward_traces` is defined in the file rl/markov_process.py.

```
    gamma: float,
    episode_length_tolerance: float = 1e-6
) -> Iterator[ValueFunctionApprox[S]]:
    episodes: Iterator[Iterator[mp.ReturnStep[S]]] = \
        (returns(trace, gamma, episode_length_tolerance) for trace in traces)
    f = approx_0
    yield f

    for episode in episodes:
        f = last(f.iterate_updates(
            [(step.state, step.return_)] for step in episode
        ))
        yield f
```

The core of the `mc_prediction` function above is the call to the `returns` function (detailed below and available in the file rl/returns.py). `returns` takes as input: `trace` representing a trace experience (`Iterable` of `TransitionStep`), the discount factor `gamma`, and an `episodes_length_tolerance` that determines how many time steps to cover in each trace experience when $\gamma < 1$ (as many steps as until γ^{steps} falls below `episodes_length_tolerance` or until the trace experience ends in a terminal state, whichever happens first). If $\gamma = 1$, each trace experience needs to end in a terminal state (else the `returns` function will loop forever).

The `returns` function calculates the returns G_t (accumulated discounted rewards) starting from each state S_t in the trace experience.[2] The key is to walk backwards from the end of the trace experience to the start (so as to reuse the calculated returns while walking backwards: $G_t = R_{t+1} + \gamma \cdot G_{t+1}$). Note the use of `iterate.accumulate` to perform this backwards-walk calculation, which in turn uses the `add_return` method in `TransitionStep` to create an instance of `ReturnStep`. The `ReturnStep` (as seen in the code below) class is derived from the `TransitionStep` class and includes the additional attribute named `return_`.

We add a method called `add_return` in `TransitionStep` so we can augment the attributes `state`, `reward`, `next_state` with the additional attribute `return_` that is computed as reward plus gamma times the `return_` from the next state.[3]

```
@dataclass(frozen=True)
class TransitionStep(Generic[S]):
    state: NonTerminal[S]
    next_state: State[S]
    reward: float

    def add_return(self, gamma: float, return_: float) -> ReturnStep[S]:
        return ReturnStep(
            self.state,
            self.next_state,
            self.reward,
            return_=self.reward + gamma * return_
        )

@dataclass(frozen=True)
class ReturnStep(TransitionStep[S]):
    return_: float
```

```
import itertools

import rl.iterate as iterate
import rl.markov_process as mp

def returns(
        trace: Iterable[mp.TransitionStep[S]],
```

[2] `returns` is defined in the file rl/returns.py.

[3] `TransitionStep` and the `add_return` method are defined in the file rl/markov_process.py.

```
        gamma: float,
        tolerance: float
) -> Iterator[mp.ReturnStep[S]]:
    trace = iter(trace)
    max_steps = round(math.log(tolerance) / math.log(gamma)) if gamma < 1 \
        else None
    if max_steps is not None:
        trace = itertools.islice(trace, max_steps * 2)
    *transitions, last_transition = list(trace)
    return_steps = iterate.accumulate(
        reversed(transitions),
        func=lambda next, curr: curr.add_return(gamma, next.return_),
        initial=last_transition.add_return(gamma, 0)
    )
    return_steps = reversed(list(return_steps))
    if max_steps is not None:
        return_steps = itertools.islice(return_steps, max_steps)
    return return_steps
```

We say that the trace experiences are *episodic traces* if each trace experience ends in a terminal state to signify that each trace experience is an episode, after whose termination we move on to the next episode. Trace experiences that do not terminate are known as *continuing traces*. We say that an RL problem is *episodic* if the input trace experiences are all *episodic* (likewise, we say that an RL problem is *continuing* if some of the input trace experiences are *continuing*).

Assume that the probability distribution of returns conditional on a state is modeled by a function approximation as a (state-conditional) normal distribution, whose mean (Value Function) we denote as $V(s; \boldsymbol{w})$ where s denotes a state for which the function approximation is being evaluated and $\boldsymbol{w}$ denotes the set of parameters in the function approximation (eg: the weights in a neural network). Then, the loss function for supervised learning of the Value Function is the sum of squares of differences between observed returns and the Value Function estimate from the function approximation. For a state S_t visited at time t in a trace experience and it's associated return G_t on the trace experience, the contribution to the loss function is:

$$\mathcal{L}_{(S_t,G_t)}(\boldsymbol{w}) = \frac{1}{2} \cdot (V(S_t; \boldsymbol{w}) - G_t)^2 \tag{1.1}$$

It's gradient with respect to $\boldsymbol{w}$ is:

$$\nabla_{\boldsymbol{w}} \mathcal{L}_{(S_t,G_t)}(\boldsymbol{w}) = (V(S_t; \boldsymbol{w}) - G_t) \cdot \nabla_{\boldsymbol{w}} V(S_t; \boldsymbol{w})$$

We know that the change in the parameters (adjustment to the parameters) is equal to the negative of the gradient of the loss function, scaled by the learning rate (let's denote the learning rate as α). Then the change in parameters is:

$$\Delta \boldsymbol{w} = \alpha \cdot (G_t - V(S_t; \boldsymbol{w})) \cdot \nabla_{\boldsymbol{w}} V(S_t; \boldsymbol{w}) \tag{1.2}$$

This is a standard formula for change in parameters in response to incremental atomic data for supervised learning when the response variable has a conditional normal distribution. But it's useful to see this formula in an intuitive manner for this specialization of incremental supervised learning to Reinforcement Learning parameter updates. We should interpret the change in parameters $\Delta \boldsymbol{w}$ as the product of three conceptual entities:

- *Learning Rate* α
- *Return Residual* of the observed return G_t relative to the estimated conditional expected return $V(S_t; \boldsymbol{w})$
- *Estimate Gradient* of the conditional expected return $V(S_t; \boldsymbol{w})$ with respect to the parameters $\boldsymbol{w}$

This interpretation of the change in parameters as the product of these three conceptual entities: (Learning rate, Return Residual, Estimate Gradient) is important as this will be a repeated pattern in many of the RL algorithms we will cover.

Now we consider a simple case of Monte-Carlo Prediction where the MRP consists of a finite state space with the non-terminal states $\mathcal{N} = \{s_1, s_2, \ldots, s_m\}$. In this case, we represent the Value Function of the MRP in a data structure (dictionary) of (state, expected return) pairs. This is known as "Tabular" Monte-Carlo (more generally as Tabular RL to reflect the fact that we represent the calculated Value Function in a "table" , i.e., dictionary). Note that in this case, Monte-Carlo Prediction reduces to a very simple calculation wherein for each state, we simply maintain the average of the trace experience returns from that state onwards (averaged over state visitations across trace experiences), and the average is updated in an incremental manner. Recall from Section **??** of Chapter **??** that this is exactly what's done in the `Tabular` class (in file rl/func_approx.py). We also recall from Section **??** of Chapter **??** that `Tabular` implements the interface of the abstract class `FunctionApprox` and so, we can perform Tabular Monte-Carlo Prediction by passing a `Tabular` instance as the `approx0: FunctionApprox` argument to the `mc_prediction` function above. The implementation of the `update` method in Tabular is exactly as we desire: it performs an incremental averaging of the trace experience returns obtained from each state onwards (over a stream of trace experiences).

Let us denote $V_n(s_i)$ as the estimate of the Value Function for a state s_i after the n-th occurrence of the state s_i (when doing Tabular Monte-Carlo Prediction) and let $Y_i^{(1)}, Y_i^{(2)}, \ldots, Y_i^{(n)}$ be the trace experience returns associated with the n occurrences of state s_i. Let us denote the `count_to_weight_func` attribute of `Tabular` as f. Then, the `Tabular` update at the n-th occurrence of state s_i (with it's associated return $Y_i^{(n)}$) is as follows:

$$V_n(s_i) = (1 - f(n)) \cdot V_{n-1}(s_i) + f(n) \cdot Y_i^{(n)} = V_{n-1}(s_i) + f(n) \cdot (Y_i^{(n)} - V_{n-1}(s_i)) \quad (1.3)$$

Thus, we see that the update (change) to the Value Function for a state s_i is equal to $f(n)$ (weight for the latest trace experience return $Y_i^{(n)}$ from state s_i) times the difference between the latest trace experience return $Y_i^{(n)}$ and the current Value Function estimate $V_{n-1}(s_i)$. This is a good perspective as it tells us how to adjust the Value Function estimate in an intuitive manner. In the case of the default setting of `count_to_weight_func` as $f(n) = \frac{1}{n}$, we get:

$$V_n(s_i) = \frac{n-1}{n} \cdot V_{n-1}(s_i) + \frac{1}{n} \cdot Y_i^{(n)} = V_{n-1}(s_i) + \frac{1}{n} \cdot (Y_i^{(n)} - V_{n-1}(s_i)) \quad (1.4)$$

So if we have 9 occurrences of a state with an average trace experience return of 50 and if the 10th occurrence of the state gives a trace experience return of 60, then we consider $\frac{1}{10}$ of $60 - 50$ (equal to 1) and increase the Value Function estimate for the state from 50 to 50+1 = 51. This illustrates how we move the Value Function estimate in the direction from the current estimate to the latest trace experience return, by a magnitude of $\frac{1}{n}$ of their gap.

Expanding the incremental updates across values of n in Equation (1.3), we get:

$$\begin{aligned} V_n(s_i) =& f(n) \cdot Y_i^{(n)} + (1 - f(n)) \cdot f(n-1) \cdot Y_i^{(n-1)} + \ldots \\ &+ (1 - f(n)) \cdot (1 - f(n-1)) \cdots (1 - f(2)) \cdot f(1) \cdot Y_i^{(1)} \end{aligned} \tag{1.5}$$

In the case of the default setting of `count_to_weight_func` as $f(n) = \frac{1}{n}$, we get:

$$V_n(s_i) = \frac{1}{n} \cdot Y_i^{(n)} + \frac{n-1}{n} \cdot \frac{1}{n-1} \cdot Y_i^{(n-1)} + \ldots + \frac{n-1}{n} \cdot \frac{n-2}{n-1} \cdots \frac{1}{2} \cdot \frac{1}{1} \cdot Y_i^{(1)} = \frac{\sum_{k=1}^n Y_i^{(k)}}{n} \tag{1.6}$$

which is an equally-weighted average of the trace experience returns from the state. From the Law of Large Numbers, we know that the sample average converges to the expected value, which is the core idea behind the Monte-Carlo method.

Note that the `Tabular` class as an implementation of the abstract class `FunctionApprox` is not just a software design happenstance - there is a formal mathematical specialization here that is vital to recognize. This tabular representation is actually a special case of linear function approximation by setting a feature function $\phi_i(\cdot)$ for each x_i as: $\phi_i(x_i) = 1$ and $\phi(x) = 0$ for each $x \neq x_i$ (i.e., $\phi_i(\cdot)$ is the indicator function for x_i, and the $\mathbf{\Phi}$ matrix of Chapter **??** reduces to the identity matrix). So we can conceptualize Tabular Monte-Carlo Prediction as a linear function approximation with the feature functions equal to the indicator functions for each of the non-terminal states and the linear-approximation parameters w_i equal to the Value Function estimates for the corresponding non-terminal states.

With this perspective, more broadly, we can view Tabular RL as a special case of RL with Linear Function Approximation of the Value Function. Moreover, the `count_to_weight_func` attribute of `Tabular` plays the role of the learning rate (as a function of the number of iterations in stochastic gradient descent). This becomes clear if we write Equation (1.3) in terms of parameter updates: write $V_n(s_i)$ as parameter value $w_i^{(n)}$ to denote the n-th update to parameter w_i corresponding to state s_i, and write $f(n)$ as learning rate α_n for the n-th update to w_i.

$$w_i^{(n)} = w_i^{(n-1)} + \alpha_n \cdot (Y_i^{(n)} - w_i^{(n-1)})$$

So, the change in parameter w_i for state s_i is α_n times $Y_i^{(n)} - w_i^{(n-1)}$. We observe that $Y_i^{(n)} - w_i^{(n-1)}$ represents the gradient of the loss function for the data point $(s_i, Y_i^{(n)})$ in the case of linear function approximation with features as indicator variables (for each state). This is because the loss function for the data point $(s_i, Y_i^{(n)})$ is $\frac{1}{2} \cdot (Y_i^{(n)} - \sum_{j=1}^m \phi_j(s_i) \cdot w_j^{(n-1)})^2$ which reduces to $\frac{1}{2} \cdot (Y_i^{(n)} - w_i^{(n-1)})^2$, whose gradient in the direction of w_i is $Y_i^{(n)} - w_i^{(n-1)}$ and 0 in the other directions (for $j \neq i$). So we see that `Tabular` updates are basically a special case of `LinearFunctionApprox` updates if we set the features to be indicator functions for each of the states (with `count_to_weight_func` playing the role of the learning rate).

Now that you recognize that `count_to_weight_func` essentially plays the role of the learning rate and governs the importance given to the latest trace experience return relative to past trace experience returns, we want to point out that real-world situations are not stationary in the sense that the environment typically evolves over a period of time and so, RL algorithms have to appropriately adapt to the changing environment. The way to adapt effectively is to have an element of "forgetfulness" of the past because if one learns about

the distant past far too strongly in a changing environment, our predictions (and eventually control) would not be effective. So, how does an RL algorithm "forget?" Well, one can "forget" through an appropriate time-decay of the weights when averaging trace experience returns. If we set a constant learning rate α (in `Tabular`, this would correspond to `count_to_weight_func=lambda _: alpha`), we'd obtain "forgetfulness" with lower weights for old data points and higher weights for recent data points. This is because with a constant learning rate α, Equation (1.5) reduces to:

$$\begin{aligned} V_n(s_i) &= \alpha \cdot Y_i^{(n)} + (1-\alpha) \cdot \alpha \cdot Y_i^{(n-1)} + \ldots + (1-\alpha)^{n-1} \cdot \alpha \cdot Y_i^{(1)} \\ &= \sum_{j=1}^{n} \alpha \cdot (1-\alpha)^{n-j} \cdot Y_i^{(j)} \end{aligned}$$

which means we have exponentially-decaying weights in the weighted average of the trace experience returns for any given state.

Note that for $0 < \alpha \leq 1$, the weights sum up to 1 as n tends to infinity, i.e.,

$$\lim_{n \to \infty} \sum_{j=1}^{n} \alpha \cdot (1-\alpha)^{n-j} = \lim_{n \to \infty} 1 - (1-\alpha)^n = 1$$

It's worthwhile pointing out that the Monte-Carlo algorithm we've implemented above is known as Each-Visit Monte-Carlo to refer to the fact that we include each occurrence of a state in a trace experience. So if a particular state appears 10 times in a given trace experience, we have 10 (state, return) pairs that are used to make the update (for just that state) at the end of that trace experience. This is in contrast to First-Visit Monte-Carlo in which only the first occurrence of a state in a trace experience is included in the set of (state, return) pairs used to make an update at the end of the trace experience. So First-Visit Monte-Carlo needs to keep track of whether a state has already been visited in a trace experience (repeat occurrences of states in a trace experience are ignored). We won't implement First-Visit Monte-Carlo in this book, and leave it to you as an exercise.

Now let's write some code to test our implementation of Monte-Carlo Prediction. To do so, we go back to a simple finite MRP example from Chapter **??** - `SimpleInventoryMRPFinite`. The following code creates an instance of the MRP and computes it's exact Value Function based on Equation (**??**).

```
from rl.chapter2.simple_inventory_mrp import SimpleInventoryMRPFinite

user_capacity = 2
user_poisson_lambda = 1.0
user_holding_cost = 1.0
user_stockout_cost = 10.0
user_gamma = 0.9

si_mrp = SimpleInventoryMRPFinite(
    capacity=user_capacity,
    poisson_lambda=user_poisson_lambda,
    holding_cost=user_holding_cost,
    stockout_cost=user_stockout_cost
)
si_mrp.display_value_function(gamma=user_gamma)
```

This prints the following:

```
{NonTerminal(state=InventoryState(on_hand=0, on_order=0)): -35.511,
```

```
 NonTerminal(state=InventoryState(on_hand=0, on_order=1)): -27.932,
 NonTerminal(state=InventoryState(on_hand=0, on_order=2)): -28.345,
 NonTerminal(state=InventoryState(on_hand=1, on_order=0)): -28.932,
 NonTerminal(state=InventoryState(on_hand=1, on_order=1)): -29.345,
 NonTerminal(state=InventoryState(on_hand=2, on_order=0)): -30.345}
```

Next, we run Monte-Carlo Prediction by first generating a stream of trace experiences (in the form of sampling traces) from the MRP, and then calling `mc_prediction` using `Tabular` with equal-weights-learning-rate (i.e., default `count_to_weight_func` of `lambda n: 1.0 / n`).

```
from rl.chapter2.simple_inventory_mrp import InventoryState
from rl.function_approx import Tabular
from rl.approximate_dynamic_programming import ValueFunctionApprox
from rl.distribution import Choose
from rl.iterate import last
from rl.monte_carlo import mc_prediction
from itertools import islice
from pprint import pprint

traces: Iterable[Iterable[TransitionStep[S]]] = \
        mrp.reward_traces(Choose(si_mrp.non_terminal_states))
it: Iterator[ValueFunctionApprox[InventoryState]] = mc_prediction(
    traces=traces,
    approx_0=Tabular(),
    gamma=user_gamma,
    episode_length_tolerance=1e-6
)

num_traces = 60000

last_func: ValueFunctionApprox[InventoryState] = last(islice(it, num_traces))
pprint({s: round(last_func.evaluate([s])[0], 3)
        for s in si_mrp.non_terminal_states})
```

This prints the following:

```
{NonTerminal(state=InventoryState(on_hand=1, on_order=1)): -29.341,
 NonTerminal(state=InventoryState(on_hand=2, on_order=0)): -30.349,
 NonTerminal(state=InventoryState(on_hand=0, on_order=0)): -35.52,
 NonTerminal(state=InventoryState(on_hand=0, on_order=1)): -27.931,
 NonTerminal(state=InventoryState(on_hand=0, on_order=2)): -28.355,
 NonTerminal(state=InventoryState(on_hand=1, on_order=0)): -28.93}
```

We see that the Value Function computed by Tabular Monte-Carlo Prediction with 60000 trace experiences is within 0.01 of the exact Value Function, for each of the states.

This completes the coverage of our first RL Prediction algorithm: Monte-Carlo Prediction. This has the advantage of being a very simple, easy-to-understand algorithm with an unbiased estimate of the Value Function. But Monte-Carlo can be slow to converge to the correct Value Function and another disadvantage of Monte-Carlo is that it requires entire trace experiences (or long-enough trace experiences when $\gamma < 1$). The next RL Prediction algorithm we cover (Temporal-Difference) overcomes these weaknesses.

1.4. Temporal-Difference (TD) Prediction

To understand Temporal-Difference (TD) Prediction, we start with it's Tabular version as it is simple to understand (and then we can generalize to TD Prediction with Function

Approximation). To understand Tabular TD prediction, we begin by taking another look at the Value Function update in Tabular Monte-Carlo (MC) Prediction with constant learning rate.

$$V(S_t) \leftarrow V(S_t) + \alpha \cdot (G_t - V(S_t)) \tag{1.7}$$

where S_t is the state visited at time step t in the current trace experience, G_t is the trace experience return obtained from time step t onwards, and α denotes the learning rate (based on `count_to_weight_func` attribute in the `Tabular` class). The key in moving from MC to TD is to take advantage of the recursive structure of the Value Function as given by the MRP Bellman Equation (Equation (**??**)). Although we only have access to individual experiences of next state S_{t+1} and reward R_{t+1}, and not the transition probabilities of next state and reward, we can approximate G_t as experience reward R_{t+1} plus γ times $V(S_{t+1})$ (where S_{t+1} is the experience's next state). The idea is to build upon (the term we use is *bootstrap*) the Value Function that is currently estimated. Clearly, this is a biased estimate of the Value Function meaning the update to the Value Function for S_t will be biased. But the bias disadvantage is outweighed by the reduction in variance (which we will discuss more about later), by speedup in convergence (bootstrapping is our friend here), and by the fact that we don't actually need entire/long-enough trace experiences (again, bootstrapping is our friend here). So, the update for Tabular TD Prediction is:

$$V(S_t) \leftarrow V(S_t) + \alpha \cdot (R_{t+1} + \gamma \cdot V(S_{t+1}) - V(S_t)) \tag{1.8}$$

Note how we've simply replaced G_t in Equation (1.7) (Tabular MC Prediction update) with $R_{t+1} + \gamma \cdot V(S_{t+1})$.

To facilitate understanding, for the remainder of the book, we shall interpret $V(S_{t+1})$ as being equal to 0 if $S_{t+1} \in \mathcal{T}$ (note: technically, this notation is incorrect because $V(\cdot)$ is a function with domain $\mathcal{N}$). Likewise, we shall interpret the function approximation notation $V(S_{t+1}; \boldsymbol{w})$ as being equal to 0 if $S_{t+1} \in \mathcal{T}$.

We refer to $R_{t+1} + \gamma \cdot V(S_{t+1})$ as the TD target and we refer to $\delta_t = R_{t+1} + \gamma \cdot V(S_{t+1}) - V(S_t)$ as the TD Error. The TD Error is the crucial quantity since it represents the "sample Bellman Error" and hence, the TD Error can be used to move $V(S_t)$ appropriately (as shown in the above adjustment to $V(S_t)$), which in turn has the effect of bridging the TD error (on an expected basis).

An important practical advantage of TD is that (unlike MC) we can use it 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). The other appealing thing about TD is that it is learning (updating Value Function) after each atomic experience (we call it *continuous learning*) versus MC's learning at the end of trace experiences. This also means that TD can be run on *any* stream of atomic experiences, not just atomic experiences that are part of a trace experience. This is a major advantage as we can chop the available data and serve it any order, freeing us from the order in which the data arrives.

Now that we understand how TD Prediction works for the Tabular case, let's consider TD Prediction with Function Approximation. Here, each time we transition from a state S_t to state S_{t+1} with reward R_{t+1}, we make an update to the parameters of the function approximation. To understand how the parameters of the function approximation update, let's consider the loss function for TD. We start with the single-state loss function for MC (Equation (1.1)) and simply replace G_t with $R_{t+1} + \gamma \cdot V(S_{t+1}, \boldsymbol{w})$ as follows:

$$\mathcal{L}_{(S_t,S_{t+1},R_{t+1})}(\boldsymbol{w}) = \frac{1}{2} \cdot (V(S_t;\boldsymbol{w}) - (R_{t+1} + \gamma \cdot V(S_{t+1};\boldsymbol{w})))^2 \tag{1.9}$$

Unlike MC, in the case of TD, we don't take the gradient of this loss function. Instead we "cheat" in the gradient calculation by ignoring the dependency of $V(S_{t+1};\boldsymbol{w})$ on $\boldsymbol{w}$. This "gradient with cheating" calculation is known as *semi-gradient*. Specifically, we pretend that the only dependency of the loss function on $\boldsymbol{w}$ is through $V(S_t;\boldsymbol{w})$. Hence, the semi-gradient calculation results in the following formula for change in parameters $\boldsymbol{w}$:

$$\Delta\boldsymbol{w} = \alpha \cdot (R_{t+1} + \gamma \cdot V(S_{t+1};\boldsymbol{w}) - V(S_t;\boldsymbol{w})) \cdot \nabla_{\boldsymbol{w}} V(S_t;\boldsymbol{w}) \tag{1.10}$$

This looks similar to the formula for parameters update in the case of MC (with G_t replaced by $R_{t+1} + \gamma \cdot V(S_{t+1};\boldsymbol{w})$). Hence, this has the same structure as MC in terms of conceptualizing the change in parameters as the product of the following 3 entities:

- *Learning Rate* α
- *TD Error* $\delta_t = R_{t+1} + \gamma \cdot V(S_{t+1};\boldsymbol{w}) - V(S_t;\boldsymbol{w})$
- *Estimate Gradient* of the conditional expected return $V(S_t;\boldsymbol{w})$ with respect to the parameters $\boldsymbol{w}$

Now let's write some code to implement TD Prediction (with Function Approximation). Unlike MC which takes as input a stream of trace experiences, TD works with a more granular stream: a stream of *atomic experiences*. Note that a stream of trace experiences can be broken up into a stream of atomic experiences, but we could also obtain a stream of atomic experiences in other ways (not necessarily from a stream of trace experiences). Thus, the TD prediction algorithm we write below (`td_prediction`) takes as input an `Iterable[TransitionStep[S]]`. `td_prediction` produces an `Iterator` of `ValueFunctionApprox[S]`, i.e., an updated function approximation of the Value Function after each atomic experience in the input atomic experiences stream. Similar to our implementation of MC, our implementation of TD is based on supervised learning on a stream of (x, y) pairs, but there are two key differences:

1. The `update` of the `ValueFunctionApprox` is done after each atomic experience, versus MC where the `updates` are done at the end of each trace experience.
2. The y-value depends on the Value Function estimate, as seen from the update Equation (1.10) above. This means we cannot use the `iterate_updates` method of `FunctionApprox` that MC Prediction uses. Rather, we need to directly use the `rl.iterate.accumulate` function (a wrapped version of `itertools.accumulate`). As seen in the code below, the accumulation is performed on the input `transitions: Iterable[TransitionStep[S]]` and the function governing the accumulation is the `step` function in the code below that calls the `update` method of `ValueFunctionApprox`. Note that the y-values passed to `update` involve a call to the estimated Value Function `v` for the `next_state` of each `transition`. However, since the `next_state` could be `Terminal` or `NonTerminal`, and since `ValueFunctionApprox` is valid only for non-terminal states, we use the `extended_vf` function we had implemented in Chapter **??** to handle the cases of the next state being `Terminal` or `NonTerminal` (with terminal states evaluating to the default value of 0).

```
import rl.iterate as iterate
import rl.markov_process as mp
from rl.approximate_dynamic_programming import ValueFunctionApprox
```

```
from rl.approximate_dynamic_programming import extended_vf

def td_prediction(
        transitions: Iterable[mp.TransitionStep[S]],
        approx_0: ValueFunctionApprox[S],
        gamma: float
) -> Iterator[ValueFunctionApprox[S]]:
    def step(
            v: ValueFunctionApprox[S],
            transition: mp.TransitionStep[S]
    ) -> ValueFunctionApprox[S]:
        return v.update([(
            transition.state,
            transition.reward + gamma * extended_vf(v, transition.next_state)
        )])

    return iterate.accumulate(transitions, step, initial=approx_0)
```

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

Now let's write some code to test our implementation of TD Prediction. We test on the same `SimpleInventoryMRPFinite` that we had tested MC Prediction on. Let us see how close we can get to the true Value Function (that we had calculated above while testing MC Prediction). But first we need to write a function to construct a stream of atomic experiences (`Iterator[TransitionStep[S]]`) from a given `FiniteMarkovRewardProcess` (below code is in the file rl/chapter10/prediction_utils.py). Note the use of `itertools.chain.from.iterable` to chain together a stream of trace experiences (obtained by calling method `reward_traces`) into a stream of atomic experiences in the below function `unit_experiences_from_episodes`.

```
import itertools
from rl.distribution import Distribution, Choose
from rl.approximate_dynamic_programming import NTStateDistribution

def mrp_episodes_stream(
    mrp: MarkovRewardProcess[S],
    start_state_distribution: NTStateDistribution[S]
) -> Iterable[Iterable[TransitionStep[S]]]:
    return mrp.reward_traces(start_state_distribution)

def fmrp_episodes_stream(
    fmrp: FiniteMarkovRewardProcess[S]
) -> Iterable[Iterable[TransitionStep[S]]]:
    return mrp_episodes_stream(fmrp, Choose(fmrp.non_terminal_states))

def unit_experiences_from_episodes(
    episodes: Iterable[Iterable[TransitionStep[S]]],
    episode_length: int
) -> Iterable[TransitionStep[S]]:
    return itertools.chain.from_iterable(
        itertools.islice(episode, episode_length) for episode in episodes
    )
```

Effective use of Tabular TD Prediction requires us to create an appropriate learning rate schedule by suitably lowering the learning rate as a function of the number of occurrences of a state in the atomic experiences stream (learning rate schedule specified by `count_to_weight_func` attribute of `Tabular` class). We write below (code in the file rl/function_approx.py) the following learning rate schedule:

$$\alpha_n = \frac{\alpha}{1 + (\frac{n-1}{H})^\beta} \tag{1.11}$$

where α_n is the learning rate to be used at the n-th Value Function update for a given state, α is the initial learning rate (i.e. $\alpha = \alpha_1$), H (we call it "half life") is the number of

updates for the learning rate to decrease to half the initial learning rate (if β is 1), and β is the exponent controlling the curvature of the decrease in the learning rate. We shall often set $\beta = 0.5$.

```
def learning_rate_schedule(
    initial_learning_rate: float,
    half_life: float,
    exponent: float
) -> Callable[[int], float]:
    def lr_func(n: int) -> float:
        return initial_learning_rate * (1 + (n - 1) / half_life) ** -exponent
    return lr_func
```

With these functions available, we can now write code to test our implementation of TD Prediction. We use the same instance `si_mrp: SimpleInventoryMRPFinite` that we had created above when testing MC Prediction. We use the same number of episodes (60000) we had used when testing MC Prediction. We set initial learning rate $\alpha = 0.03$, half life $H = 1000$ and exponent $\beta = 0.5$. We set the episode length (number of atomic experiences in a single trace experience) to be 100 (about the same as with the settings we had for testing MC Prediction). We use the same discount factor $\gamma = 0.9$.

```
import rl.iterate as iterate
import rl.td as td
import itertools
from pprint import pprint
from rl.chapter10.prediction_utils import fmrp_episodes_stream
from rl.chapter10.prediction_utils import unit_experiences_from_episodes
from rl.function_approx import learning_rate_schedule

episode_length: int = 100
initial_learning_rate: float = 0.03
half_life: float = 1000.0
exponent: float = 0.5
gamma: float = 0.9

episodes: Iterable[Iterable[TransitionStep[S]]] = \
    fmrp_episodes_stream(si_mrp)
td_experiences: Iterable[TransitionStep[S]] = \
    unit_experiences_from_episodes(
        episodes,
        episode_length
    )
learning_rate_func: Callable[[int], float] = learning_rate_schedule(
    initial_learning_rate=initial_learning_rate,
    half_life=half_life,
    exponent=exponent
)
td_vfs: Iterator[ValueFunctionApprox[S]] = td.td_prediction(
    transitions=td_experiences,
    approx_0=Tabular(count_to_weight_func=learning_rate_func),
    gamma=gamma
)

num_episodes = 60000

final_td_vf: ValueFunctionApprox[S] = \
    iterate.last(itertools.islice(td_vfs, episode_length * num_episodes))
pprint({s: round(final_td_vf(s), 3) for s in si_mrp.non_terminal_states})
```

This prints the following:

```
{NonTerminal(state=InventoryState(on_hand=0, on_order=0)): -35.529,
 NonTerminal(state=InventoryState(on_hand=0, on_order=1)): -27.868,
```

```
NonTerminal(state=InventoryState(on_hand=0, on_order=2)): -28.344,
NonTerminal(state=InventoryState(on_hand=1, on_order=0)): -28.935,
NonTerminal(state=InventoryState(on_hand=1, on_order=1)): -29.386,
NonTerminal(state=InventoryState(on_hand=2, on_order=0)): -30.305}
```

Thus, we see that our implementation of TD prediction with the above settings fetches us an estimated Value Function within 0.065 of the true Value Function after 60,000 episodes.

As ever, we encourage you to play with various settings for MC Prediction and TD prediction to develop some intuition for how the results change as you change the settings. You can play with the code in the file rl/chapter10/simple_inventory_mrp.py.

1.5. TD versus MC

It is often claimed that TD is the most significant and innovative idea in the development of the field of Reinforcement Learning. The key to TD is that it blends the advantages of Dynamic Programming (DP) and Monte-Carlo (MC). Like DP, TD updates the Value Function estimate by bootstrapping from the Value Function estimate of the next state experienced (essentially, drawing from Bellman Equation). Like MC, TD learns from experiences without requiring access to transition probabilities (MC and TD updates are *experience updates* while DP updates are *transition-probabilities-averaged-updates*). So TD overcomes curse of dimensionality and curse of modeling (computational limitation of DP), and also has the advantage of not requiring entire trace experiences (practical limitation of MC).

The TD idea has it's origins in a seminal book by Harry Klopf (Klopf and Data Sciences Laboratory 1972) that greatly influenced Richard Sutton and Andrew Barto to pursue the TD idea further, after which they published several papers on TD, much of whose content is covered in their RL book (Sutton and Barto 2018).

1.5.1. TD learning akin to human learning

Perhaps the most attractive thing about TD (versus MC) is that it is akin to how humans learn. Let us illustrate this point with how a soccer player learns to improve her game in the process of playing many soccer games. Let's simplify the soccer game to a "golden-goal" soccer game, i.e., the game ends when a team scores a goal. The reward in such a soccer game is +1 for scoring (and winning), 0 if the opponent scores, and also 0 for the entire duration of the game before the goal is scored. The soccer player (who is learning) has her *State* comprising of her position/velocity/posture etc., the other players' positions/velocity etc., the soccer ball's position/velocity etc. The *Action*s of the soccer player are her physical movements, including the ways to dribble/kick the ball. If the soccer player learns in an MC style (a single episode is a single soccer game), then the soccer player analyzes (at the end of the game) all possible states and actions that occurred during the game and assesses how the actions in each state might have affected the final outcome of the game. You can see how laborious and difficult this actions-reward linkage would be, and you might even argue that it's impossible to disentangle the effects of various actions during the game on the goal that was eventually scored. In any case, you should recognize that this is absolutely not how a soccer player would analyze and learn. Rather, a soccer player learns *during the game* - she is continuously evaluating how her actions change the probability of scoring the goal (which is essentially the Value Function). If a pass to her teammate did not result in a goal but greatly increased the chances of scoring a goal, then the action of

passing the ball to one's teammate in that state is a good action, boosting the action's Q-value immediately, and she will likely try that action (or a similar action) again, meaning actions with better Q-values are prioritized, which drives towards better and quicker goal-scoring opportunities, and likely eventually results in a goal. Such goal-scoring (based on active learning during the game, cutting out poor actions and promoting good actions) would be hailed by commentators as "success from continuous and eager learning" on the part of the soccer player. This is essentially TD learning.

If you think about career decisions and relationship decisions in our lives, MC-style learning is quite infeasible because we simply don't have sufficient "episodes" (for certain decisions, our entire life might be a single episode), and waiting to analyze and adjust until the end of an episode might be far too late in our lives. Rather, we learn and adjust our evaluations of situations constantly in a TD-like manner. Think about various important decisions we make in our lives and you will see that we learn by perpetual adjustment of estimates and we are efficient in the use of limited experiences we obtain in our lives.

1.5.2. Bias, Variance and Convergence

Now let's talk about bias and variance of the MC and TD prediction estimates, and their convergence properties.

Say we are at state S_t at time step t on a trace experience, and G_t is the return from that state S_t onwards on this trace experience. G_t is an unbiased estimate of the true value function for state S_t, which is a big advantage for MC when it comes to convergence, even with function approximation of the Value Function. On the other hand, the TD Target $R_{t+1} + \gamma \cdot V(S_{t+1}; \boldsymbol{w})$ is a biased estimate of the true value function for state S_t. There is considerable literature on formal proofs of TD Prediction convergence and we won't cover it in detail here, but here's a qualitative summary: Tabular TD Prediction converges to the true value function in the mean for constant learning rate, and converges to the true value function if the following stochastic approximation conditions are satisfied for the learning rate schedule $\alpha_n, n = 1, 2, \ldots$, where the index n refers to the n-th occurrence of a particular state whose Value Function is being updated:

$$\sum_{n=1}^{\infty} \alpha_n = \infty \text{ and } \sum_{n=1}^{\infty} \alpha_n^2 < \infty$$

The stochastic approximation conditions above are known as the Robbins-Monro schedule and apply to a general class of iterative methods used for root-finding or optimization when data is noisy. The intuition here is that the steps should be large enough (first condition) to eventually overcome any unfavorable initial values or noisy data and yet the steps should eventually become small enough (second condition) to ensure convergence. Note that in Equation (1.11), exponent $\beta = 1$ satisfies the Robbins-Monro conditions. In particular, our default choice of `count_to_weight_func=lambda n: 1.0 / n` in `Tabular` satisfies the Robbins-Monro conditions, but our other common choice of constant learning rate does not satisfy the Robbins-Monro conditions. However, we want to emphasize that the Robbins-Monro conditions are typically not that useful in practice because it is not a statement of speed of convergence and it is not a statement on closeness to the true optima (in practice, the goal is typically simply to get fairly close to the true answer reasonably quickly).

The bad news with TD (due to the bias in it's update) is that TD Prediction with function approximation does not always converge to the true value function. Most TD Prediction

convergence proofs are for the Tabular case, however some proofs are for the case of linear function approximation of the Value Function.

The flip side of MC's bias advantage over TD is that the TD Target $R_{t+1} + \gamma \cdot V(S_{t+1}; \boldsymbol{w})$ 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}.

As for speed of convergence and efficiency in use of limited set of experiences data, we still don't have formal proofs on whether MC is better or TD is better. More importantly, 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 TD learns faster than MC with constant learning rates.

A popular simple problem in the literature (when comparing RL prediction algorithms) is a random walk MRP with states $\{0, 1, 2, \ldots, B\}$ with 0 and B as the terminal states (think of these as terminating barriers of a random walk) and the remaining states as the non-terminal states. From any non-terminal state i, we transition to state $i+1$ with probability p and to state $i-1$ with probability $1-p$. The reward is 0 upon each transition, except if we transition from state $B-1$ to terminal state B which results in a reward of 1. It's quite obvious that for $p = 0.5$ (symmetric random walk), the Value Function is given by: $V(i) = \frac{i}{B}$ for all $0 < i < B$. We'd like to analyze how MC and TD converge, if at all, to this Value Function, starting from a neutral initial Value Function of $V(i) = 0.5$ for all $0 < i < B$. The following code sets up this random walk MRP.

```
from rl.distribution import Categorical

class RandomWalkMRP(FiniteMarkovRewardProcess[int]):
    barrier: int
    p: float

    def __init__(
        self,
        barrier: int,
        p: float
    ):
        self.barrier = barrier
        self.p = p
        super().__init__(self.get_transition_map())

    def get_transition_map(self) -> \
            Mapping[int, Categorical[Tuple[int, float]]]:
        d: Dict[int, Categorical[Tuple[int, float]]] = {
            i: Categorical({
                (i + 1, 0. if i < self.barrier - 1 else 1.): self.p,
                (i - 1, 0.): 1 - self.p
            }) for i in range(1, self.barrier)
        }
        return d
```

The above code is in the file rl/chapter10/random_walk_mrp.py. Next, we generate a stream of trace experiences from the MRP, use the trace experiences stream to perform MC Prediction, split the trace experiences stream into a stream of atomic experiences so as to perform TD Prediction, run MC and TD Prediction with a variety of learning rate choices, and plot the root-mean-squared-errors (RMSE) of the Value Function averaged across the non-terminal states as a function of episode batches (i.e., visualize how the RMSE of the

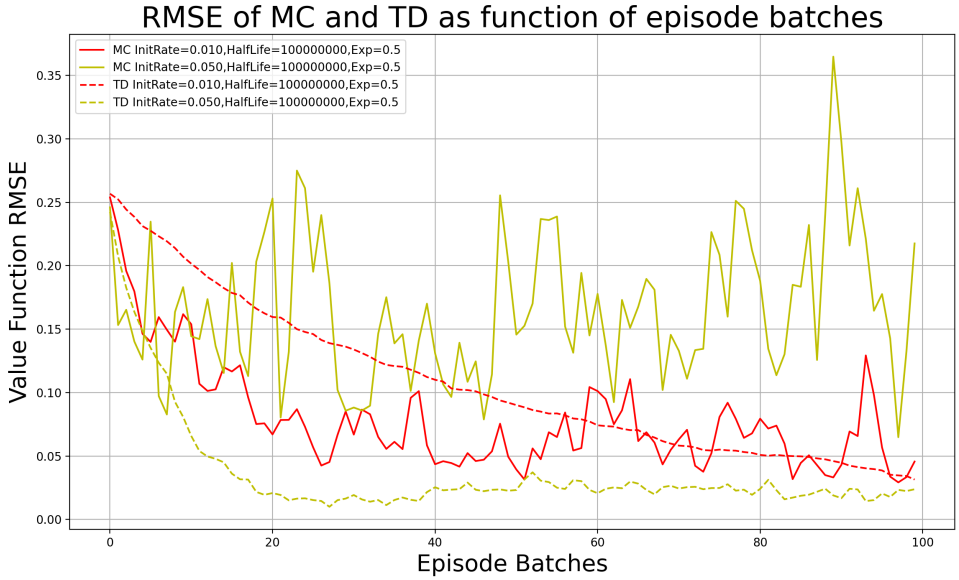

Figure 1.1.: MC and TD Convergence for Random Walk MRP

Value Function evolves as the MC/TD algorithm progresses). This is done by calling the function `compare_mc_and_td` which is in the file rl/chapter10/prediction_utils.py.

Figure 1.1 depicts the convergence for our implementations of MC and TD Prediction for constant learning rates of $\alpha = 0.01$ (darker curves) and $\alpha = 0.05$ (lighter curves). We produced this Figure by using data from 700 episodes generated from the random walk MRP with barrier $B = 10$, $p = 0.5$ and discount factor $\gamma = 1$ (a single episode refers to a single trace experience that terminates either at state 0 or at state B). We plotted the RMSE after each batch of 7 episodes, hence each of the 4 curves shown in the Figure have 100 RMSE data points plotted. Firstly, we clearly see that MC has significantly more variance as evidenced by the choppy MC RMSE progression curves. Secondly, we note that $\alpha = 0.01$ is a fairly small learning rate and so, the progression of RMSE is quite slow on the darker curves. On the other hand, notice the quick learning for $\alpha = 0.05$ (lighter curves). MC RMSE curve is not just choppy, it's evident that it progresses quite quickly in the first few episode batches (relative to the corresponding TD) but is slow after the first few episode batches (relative to the corresponding TD). This results in TD reaching fairly small RMSE quicker than the corresponding MC (this is especially stark for TD with $\alpha = 0.005$, i.e. the dashed lighter curve in the Figure). This behavior of TD outperforming the comparable MC (with constant learning rate) is typical for MRP problems.

Lastly, it's important to recognize that MC is not very sensitive to the initial Value Function while TD is more sensitive to the initial Value Function. We encourage you to play with the initial Value Function for this random walk example and evaluate how it affects MC and TD convergence speed.

More generally, we encourage you to play with the `compare_mc_and_td` function on other choices of MRP (ones we have created earlier in this book such as the inventory examples, or make up your own MRPs) so you can develop good intuition for how MC and TD Prediction algorithms converge for a variety of choices of learning rate schedules, initial Value Function choices, choices of discount factor etc.

1.5.3. Fixed-Data Experience Replay on TD versus MC

We have talked a lot about *how* TD learns versus *how* MC learns. In this subsection, we turn our focus to *what* TD learns and *what* MC learns, to shed light on a profound conceptual difference between TD and MC. We illuminate this difference with a special setting - we are given a fixed finite set of trace experiences (versus usual settings considered in this chapter so far where we had an "endless" stream of trace experiences). The agent is allowed to tap into this fixed finite set of traces experiences endlessly, i.e., the MC or TD Prediction RL agent can indeed consume an endless stream of experiences, but all of that stream of experiences must ultimately be sourced from the given fixed finite set of trace experiences. This means we'd end up tapping into trace experiences (or it's component atomic experiences) repeatedly. We call this technique of re-using experiences data encountered previously as *Experience Replay*. We will cover this Experience Replay technique in more detail in Chapter **??**, but for now, we shall uncover the key conceptual difference between *what* MC and TD learn by running the algorithms on an *Experience Replay* of a fixed finite set of trace experiences.

So let us start by setting up this experience replay with some code. Firstly, we represent the given input data of the fixed finite set of trace experiences as the data type:

`Sequence[Sequence[Tuple[S, float]]]`

The outer `Sequence` refers to the sequence of trace experiences, and the inner `Sequence` refers to the sequence of (state, reward) pairs in a trace experience (to represent the alternating sequence of states and rewards in a trace experience). The first function we write is to convert this data set into a:

`Sequence[Sequence[TransitionStep[S]]]`

which is consumable by MC and TD Prediction algorithms (since their interfaces work with the `TransitionStep[S]` data type). The following function does this job:

```
def get_fixed_episodes_from_sr_pairs_seq(
    sr_pairs_seq: Sequence[Sequence[Tuple[S, float]]],
    terminal_state: S
) -> Sequence[Sequence[TransitionStep[S]]]:
    return [[TransitionStep(
        state=NonTerminal(s),
        reward=r,
        next_state=NonTerminal(trace[i+1][0])
        if i < len(trace) - 1 else Terminal(terminal_state)
    ) for i, (s, r) in enumerate(trace)] for trace in sr_pairs_seq]
```

We'd like MC Prediction to run on an endless stream of `Sequence[TransitionStep[S]]` sourced from the fixed finite data set produced by `get_fixed_episodes_from_sr_pairs_seq`. So we write the following function to generate an endless stream by repeatedly randomly (uniformly) sampling from the fixed finite set of trace experiences:

```
import numpy as np

def get_episodes_stream(
    fixed_episodes: Sequence[Sequence[TransitionStep[S]]]
) -> Iterator[Sequence[TransitionStep[S]]]:
    num_episodes: int = len(fixed_episodes)
    while True:
        yield fixed_episodes[np.random.randint(num_episodes)]
```

As we know, TD works with atomic experiences rather than trace experiences. So we need the following function to split the fixed finite set of trace experiences into a fixed finite set of atomic experiences:

```
import itertools

def fixed_experiences_from_fixed_episodes(
    fixed_episodes: Sequence[Sequence[TransitionStep[S]]]
) -> Sequence[TransitionStep[S]]:
    return list(itertools.chain.from_iterable(fixed_episodes))
```

We'd like TD Prediction to run on an endless stream of `TransitionStep[S]` from the fixed finite set of atomic experiences produced by `fixed_experiences_from_fixed_episodes`. So we write the following function to generate an endless stream by repeatedly randomly (uniformly) sampling from the fixed finite set of atomic experiences:

```
def get_experiences_stream(
    fixed_experiences: Sequence[TransitionStep[S]]
) -> Iterator[TransitionStep[S]]:
    num_experiences: int = len(fixed_experiences)
    while True:
        yield fixed_experiences[np.random.randint(num_experiences)]
```

Ok - now we are ready to run MC and TD Prediction algorithms on an experience replay of the given input of a fixed finite set of trace experiences. It is quite obvious what the MC Prediction algorithm would learn. MC Prediction is simply supervised learning of a data set of states and their associated returns, and here we have a fixed finite set of states (across the trace experiences) and the corresponding trace experience returns associated with each of those states. Hence, MC Prediction should return a Value Function comprising of the average returns seen in the fixed finite data set for each of the states in the data set. So let us first write a function to explicitly calculate the average returns, and then we can confirm that MC Prediction will give the same answer.

```
from rl.returns import returns
from rl.markov_process import ReturnStep

def get_return_steps_from_fixed_episodes(
    fixed_episodes: Sequence[Sequence[TransitionStep[S]]],
    gamma: float
) -> Sequence[ReturnStep[S]]:
    return list(itertools.chain.from_iterable(returns(episode, gamma, 1e-8)
                                              for episode in fixed_episodes))

def get_mean_returns_from_return_steps(
    returns_seq: Sequence[ReturnStep[S]]
) -> Mapping[NonTerminal[S], float]:
    def by_state(ret: ReturnStep[S]) -> S:
        return ret.state.state

    sorted_returns_seq: Sequence[ReturnStep[S]] = sorted(
        returns_seq,
        key=by_state
    )
    return {NonTerminal(s): np.mean([r.return_ for r in l])
            for s, l in itertools.groupby(
                sorted_returns_seq,
                key=by_state
            )}
```

To facilitate comparisons, we will do all calculations on the following simple hand-entered input data set:

```
given_data: Sequence[Sequence[Tuple[str, float]]] = [
    [('A', 2.), ('A', 6.), ('B', 1.), ('B', 2.)],
    [('A', 3.), ('B', 2.), ('A', 4.), ('B', 2.), ('B', 0.)],
```

```
    [('B', 3.), ('B', 6.), ('A', 1.), ('B', 1.)],
    [('A', 0.), ('B', 2.), ('A', 4.), ('B', 4.), ('B', 2.), ('B', 3.)],
    [('B', 8.), ('B', 2.)]
]
```

The following code runs `get_mean_returns_from_return_steps` on this simple input data set.

```
from pprint import pprint
gamma: float = 0.9

fixed_episodes: Sequence[Sequence[TransitionStep[str]]] = \
    get_fixed_episodes_from_sr_pairs_seq(
        sr_pairs_seq=given_data,
        terminal_state='T'
    )

returns_seq: Sequence[ReturnStep[str]] = \
    get_return_steps_from_fixed_episodes(
        fixed_episodes=fixed_episodes,
        gamma=gamma
    )

mean_returns: Mapping[NonTerminal[str], float] = \
    get_mean_returns_from_return_steps(returns_seq)

pprint(mean_returns)
```

This prints:

```
{NonTerminal(state='B'): 5.190378571428572,
 NonTerminal(state='A'): 8.261809999999999}
```

Now let's run MC Prediction with experience-replayed 100,000 trace experiences with equal weighting for each of the (state, return) pairs, i.e., with `count_to_weights_func` attribute of `Tabular` set to the function `lambda n: 1.0 / n`:

```
import rl.monte_carlo as mc
import rl.iterate as iterate

def mc_prediction(
    episodes_stream: Iterator[Sequence[TransitionStep[S]]],
    gamma: float,
    num_episodes: int
) -> Mapping[NonTerminal[S], float]:
    return iterate.last(itertools.islice(
        mc.mc_prediction(
            traces=episodes_stream,
            approx_0=Tabular(),
            gamma=gamma,
            episode_length_tolerance=1e-10
        ),
        num_episodes
    )).values_map

num_mc_episodes: int = 100000

episodes: Iterator[Sequence[TransitionStep[str]]] = \
    get_episodes_stream(fixed_episodes)

mc_pred: Mapping[NonTerminal[str], float] = mc_prediction(
    episodes_stream=episodes,
    gamma=gamma,
    num_episodes=num_mc_episodes
)

pprint(mc_pred)
```

This prints:

```
{NonTerminal(state='A'): 8.262643843836214,
 NonTerminal(state='B'): 5.191276907315868}
```

So, as expected, it ties out within the standard error for 100,000 trace experiences. Now let's move on to TD Prediction. Let's run TD Prediction on experience-replayed 1,000,000 atomic experiences with a learning rate schedule having an initial learning rate of 0.01, decaying with a half life of 10000, and with an exponent of 0.5.

```
import rl.td as td
from rl.function_approx import learning_rate_schedule, Tabular

def td_prediction(
    experiences_stream: Iterator[TransitionStep[S]],
    gamma: float,
    num_experiences: int
) -> Mapping[NonTerminal[S], float]:
    return iterate.last(itertools.islice(
        td.td_prediction(
            transitions=experiences_stream,
            approx_0=Tabular(count_to_weight_func=learning_rate_schedule(
                initial_learning_rate=0.01,
                half_life=10000,
                exponent=0.5
            )),
            gamma=gamma
        ),
        num_experiences
    )).values_map

num_td_experiences: int = 1000000

fixed_experiences: Sequence[TransitionStep[str]] = \
    fixed_experiences_from_fixed_episodes(fixed_episodes)

experiences: Iterator[TransitionStep[str]] = \
    get_experiences_stream(fixed_experiences)

td_pred: Mapping[NonTerminal[str], float] = td_prediction(
    experiences_stream=experiences,
    gamma=gamma,
    num_experiences=num_td_experiences
)

pprint(td_pred)
```

This prints:

```
{NonTerminal(state='A'): 9.899838136517303,
 NonTerminal(state='B'): 7.444114569419306}
```

We note that this Value Function is vastly different from the Value Function produced by MC Prediction. Is there a bug in our code, or perhaps a more serious conceptual problem? Nope - there is neither a bug here nor a more serious problem. This is exactly what TD Prediction on Experience Replay on a fixed finite data set is meant to produce. So, what Value Function does this correspond to? It turns out that TD Prediction drives towards a Value Function of an MRP that is *implied* by the fixed finite set of given experiences. By the term *implied*, we mean the maximum likelihood estimate for the transition probabilities $\mathcal{P}_R$, estimated from the given fixed finite data, i.e.,

$$\mathcal{P}_R(s, r, s') = \frac{\sum_{i=1}^{N} \mathbb{I}_{S_i=s, R_{i+1}=r, S_{i+1}=s'}}{\sum_{i=1}^{N} \mathbb{I}_{S_i=s}} \tag{1.12}$$

where the fixed finite set of atomic experiences are $[(S_i, R_{i+1}, S_{i+1}) | 1 \leq i \leq N]$, and $\mathbb{I}$ denotes the indicator function.

So let's write some code to construct this MRP based on the above formula.

```
from rl.distribution import Categorical
from rl.markov_process import FiniteMarkovRewardProcess

def finite_mrp(
    fixed_experiences: Sequence[TransitionStep[S]]
) -> FiniteMarkovRewardProcess[S]:
    def by_state(tr: TransitionStep[S]) -> S:
        return tr.state.state

    d: Mapping[S, Sequence[Tuple[S, float]]] = \
        {s: [(t.next_state.state, t.reward) for t in l] for s, l in
         itertools.groupby(
             sorted(fixed_experiences, key=by_state),
             key=by_state
         )}
    mrp: Dict[S, Categorical[Tuple[S, float]]] = \
        {s: Categorical({x: y / len(l) for x, y in
                         collections.Counter(l).items()})
         for s, l in d.items()}
    return FiniteMarkovRewardProcess(mrp)
```

Now let's print it's Value Function.

```
fmrp: FiniteMarkovRewardProcess[str] = finite_mrp(fixed_experiences)
fmrp.display_value_function(gamma)
```

This prints:

```
{NonTerminal(state='A'): 9.958, NonTerminal(state='B'): 7.545}
```

So our TD Prediction algorithm doesn't exactly match the Value Function of the data-implied MRP, but it gets close. It turns out that a variation of our TD Prediction algorithm exactly matches the Value Function of the data-implied MRP. We won't implement this variation in this chapter, but will describe it briefly here. The variation is as follows:

- The Value Function is not updated after each atomic experience, rather the Value Function is updated at the end of each *batch of atomic experiences*.
- Each batch of atomic experiences consists of a single occurrence of each atomic experience in the given fixed finite data set.

- The updates to the Value Function to be performed at the end of each batch are accumulated in a buffer after each atomic experience and the buffer's contents are used to update the Value Function only at the end of the batch. Specifically, this means that the right-hand-side of Equation (1.10) is calculated at the end of each atomic experience and these calculated values are accumulated in the buffer until the end of the batch, at which point the buffer's contents are used to update the Value Function.

This variant of the TD Prediction algorithm is known as *Batch Updating* and more broadly, RL algorithms that update the Value Function at the end of a batch of experiences are refered to as *Batch Methods*. This contrasts with *Incremental Methods*, which are RL algorithms that update the Value Function after each atomic experience (in the case of TD) or at the end of each trace experience (in the case of MC). The MC and TD Prediction algorithms we implemented earlier in this chapter are Incremental Methods. We will cover Batch Methods in detail in Chapter **??**.

Although our TD Prediction algorithm is an Incremental Method, it did get fairly close to the Value Function of the data-implied MRP. So let us ignore the nuance that our TD Prediction algorithm didn't exactly match the Value Function of the data-implied MRP and instead focus on the fact that our MC Prediction algorithm and our TD Prediction algorithm drove towards two very different Value Functions. The MC Prediction algorithm learns a "fairly naive" Value Function - one that is based on the mean of the observed returns (for each state) in the given fixed finite data. The TD Prediction algorithm is learning something "deeper" - it is (implicitly) constructing an MRP based on the given fixed finite data (Equation (1.12)), and then (implicitly) calculating the Value Function of the constructed MRP. The mechanics of the TD Prediction algorithms don't actually construct the MRP and calculate the Value Function of the MRP - rather, the TD Prediction algorithm directly drives towards the Value Function of the data-implied MRP. However, the fact that it gets to this "more nuanced" Value Function means that it is (implictly) trying to infer a transitions structure from the given data, and hence, we say that it is learning something "deeper" than what MC is learning. This has practical implications. Firstly, this learning facet of TD means that it exploits any Markov property in the environment and so, TD algorithms are more efficient (learn faster than MC) in Markov environments. On the other hand, the naive nature of MC (not exploiting any Markov property in the environment) is advantageous (more effective than TD) in non-Markov environments.

We encourage you to try Experience Replay on larger input data sets, and to code up Batch Method variants of MC and TD prediction algorithms. As a starting point, the experience replay code for this chapter is in the file rl/chapter10/mc_td_experience_replay.py.

1.5.4. Bootstrapping and Experiencing

We summarize MC, TD and DP in terms of whether they bootstrap (or not) and in terms of whether they experience interactions with an real/simulated environment (or not).

- Bootstrapping: By "bootstrapping," we mean that an update to the Value Function utilizes a current or prior estimate of the Value Function. MC *does not bootstrap* since it's Value Function updates use actual trace experience returns and not any current or prior estimates of the Value Function. On the other hand, TD and DP *do bootstrap*.
- Experiencing: By "experiencing," we mean that the algorithm uses experiences obtained by interacting with a real or simulated environment, rather than performing expectation calculations with a model of transition probabilities (the latter doesn't

require interactions with an environment and hence, doesn't "experience"). MC and TD *do experience*, while DP *does not experience*.

We illustrate this perspective of bootstrapping (or not) and experiencing (or not) with some very popular diagrams that we are borrowing from lecture slides from David Silver's RL course and from teaching content prepared by Richard Sutton.

The first diagram is Figure 1.2, known as the MC *backup* diagram for an MDP (although we are covering Prediction in this chapter, these concepts also apply to MDP Control). The root of the tree is the state whose Value Function we want to update. The remaining nodes of the tree are the future states that might be visited and future actions that might be taken. The branching on the tree is due to the probabilistic transitions of the MDP and the multiple choices of actions that might be taken at each time step. The nodes marked as "T" are the terminal states. The highlighted path on the tree from the root node (current state) to a terminal state indicates a particular trace experience used by the MC algorithm. The highlighted path is the set of future states/actions used in updating the Value Function of the current state (root node). We say that the Value Function is "backed up" along this highlighted path (to mean that the Value Function update calculation propagates from the bottom of the highlighted path to the top, since the trace experience return is calculated as accumulated rewards from the bottom to the top, i.e., from the end of the trace experience to the beginning of the trace experience). This is why we refer to such diagrams as *backup* diagrams. Since MC "experiences" , it only considers a single child node from any node (rather than all the child nodes, which would be the case if we considered all probabilistic transitions or considered all action choices). So the backup is narrow (doesn't go wide across the tree). Since MC does not "bootstrap," it doesn't use the Value Function estimate from it's child/grandchild node (next time step's state/action) - instead, it utilizes the rewards at all future states/actions along the entire trace experience. So the backup works deep into the tree (is not shallow as would be the case in "bootstrapping"). In summary, the MC backup is narrow and deep.

The next diagram is Figure 1.3, known as the TD *backup* diagram for an MDP. Again, the highlighting applies to the future states/actions used in updating the Value Function of the current state (root node). The Value Function is "backed up" along this highlighted portion of the tree. Since TD "experiences" , it only considers a single child node from any node (rather than all the child nodes, which would be the case if we considered all probabilistic transitions or considered all actions choices). So the backup is narrow (doesn't go wide across the tree). Since TD "bootstraps" , it uses the Value Function estimate from it's child/grandchild node (next time step's state/action) and doesn't utilize rewards at states/actions beyond the next time step's state/action. So the backup is shallow (doesn't work deep into the tree). In summary, the TD backup is narrow and shallow.

The next diagram is Figure 1.4, known as the DP *backup* diagram for an MDP. Again, the highlighting applies to the future states/actions used in updating the Value Function of the current state (root node). The Value Function is "backed up" along this highlighted portion of the tree. Since DP does not "experience" and utilizes the knowledge of probabilities of all next states and considers all choices of actions (in the case of Control), it considers all child nodes (all choices of actions) and all grandchild nodes (all probabilistic transitions to next states) from the root node (current state). So the backup goes wide across the tree. Since DP "bootstraps" , it uses the Value Function estimate from it's children/grandchildren nodes (next time step's states/actions) and doesn't utilize rewards at states/actions beyond the next time step's states/actions. So the backup is shallow (doesn't work deep into the tree). In summary, the DP backup is wide and shallow.

Monte Carlo (Supervised Learning) (MC)

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

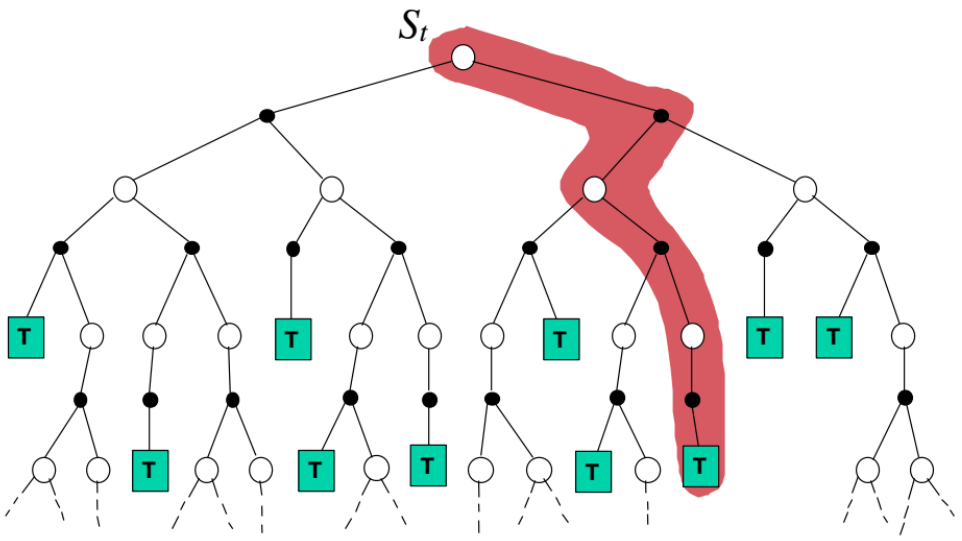

Figure 1.2.: MC Backup Diagram (Image Credit: David Silver's RL Course)

This perspective of shallow versus deep (for "bootstrapping" or not) and of narrow versus wide (for "experiencing" or not) is a great way to visualize and internalize the core ideas within MC, TD and DP, and it helps us compare and contrast these methods in a simple and intuitive manner. We must thank Rich Sutton for this excellent pedagogical contribution. This brings us to the next diagram (Figure 1.5) which provides a unified view of RL in a single picture. The top of this Figure shows methods that "bootstrap" (including TD and DP) and the bottom of this Figure shows methods that do not "bootstrap" (including MC and methods known as "Exhaustive Search" that go both deep into the tree and wide across the tree - we shall cover some of these methods in a later chapter). Therefore the vertical dimension of this Figure refers to the depth of the backup. The left of this Figure shows methods that "experience" (including TD and MC) and the right of this Figure shows methods than do not "experience" (including DP and "Exhaustive Search"). Therefore, the horizontal dimension of this Figure refers to the width of the backup.

1.6. TD(λ) Prediction

Now that we've seen the contrasting natures of TD and MC (and their respective pros and cons), it's natural to wonder if we could design an RL Prediction algorithm that combines the features of TD and MC and perhaps fetch us a blend of their respective benefits. It turns out this is indeed possible, and is the subject of this section - an innovative approach to RL Prediction known as TD(λ). λ is a continuous-valued parameter in the range $[0, 1]$ such that $\lambda = 0$ corresponds to the TD approach and $\lambda = 1$ corresponds to the MC approach. Tuning λ between 0 and 1 allows us to span the spectrum from the TD approach to the

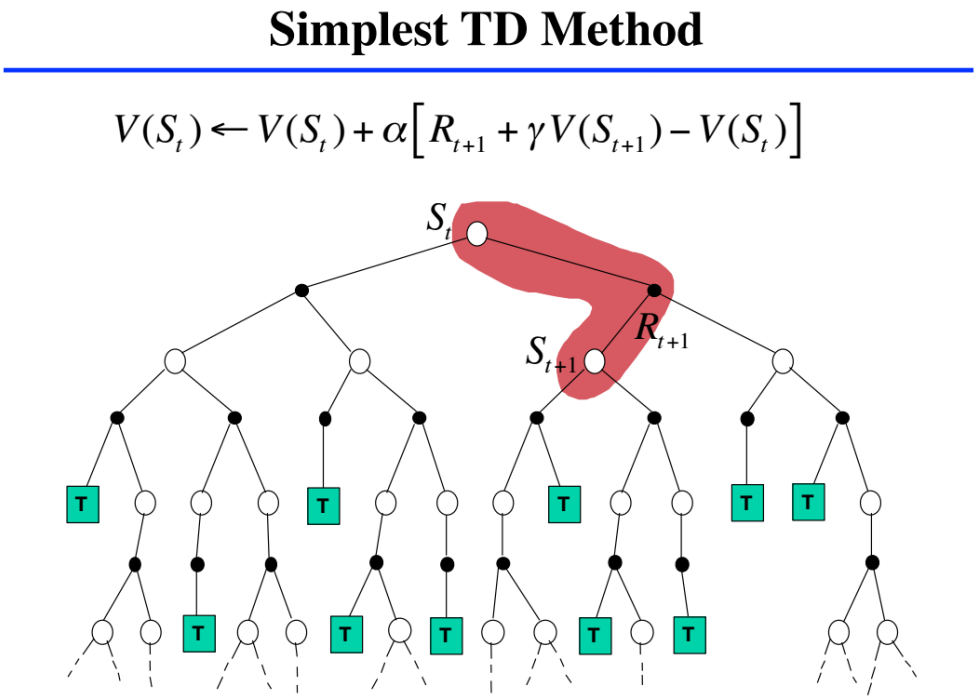

Figure 1.3.: TD Backup Diagram (Image Credit: David Silver's RL Course)

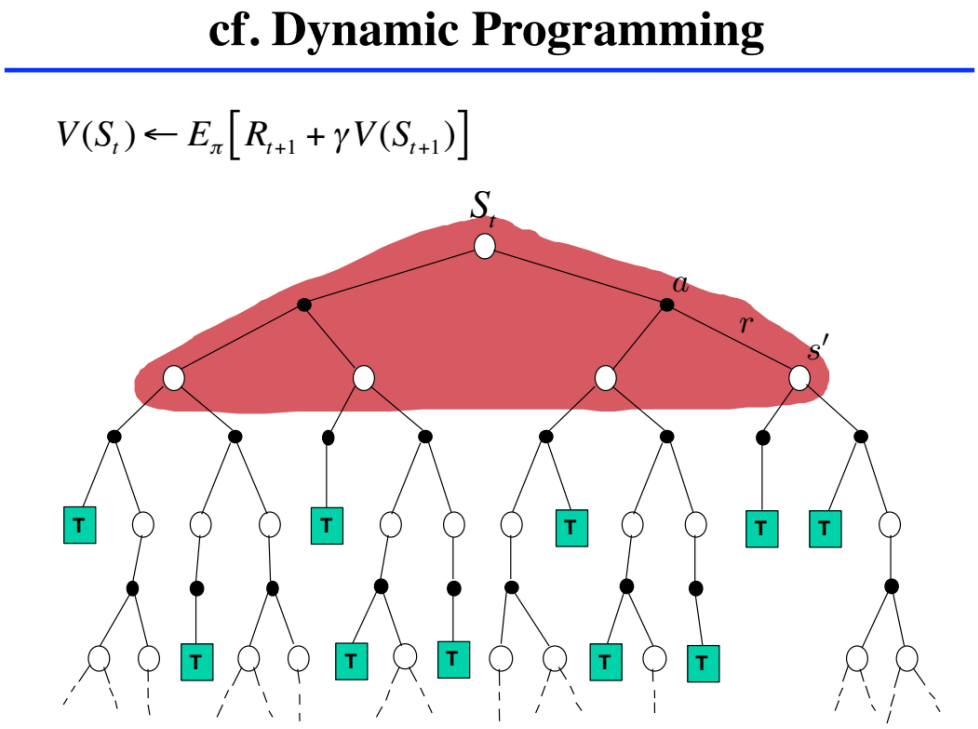

Figure 1.4.: DP Backup Diagram (Image Credit: David Silver's RL Course)

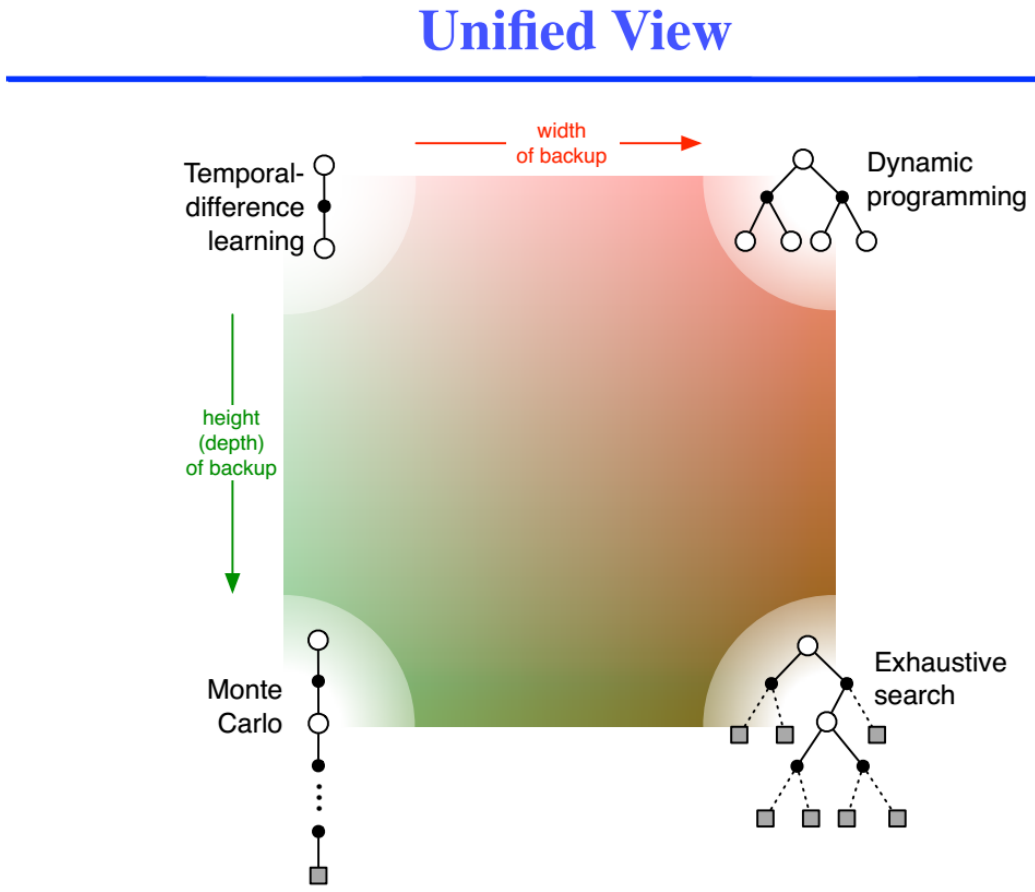

Figure 1.5.: Unified View of RL (Image Credit: Sutton-Barto's RL Book)

MC approach, essentially a blended approach known as the TD(λ) approach. The TD(λ) approach for RL Prediction gives us the TD(λ) Prediction algorithm. To get to the TD(λ) Prediction algorithm (in this section), we start with the TD Prediction algorithm we wrote earlier, generalize it to a multi-time-step bootstrapping prediction algorithm, extend that further to an algorithm known as the λ-Return Prediction algorithm, after which we shall be ready to present the TD(λ) Prediction algorithm.

1.6.1. n-Step Bootstrapping Prediction Algorithm

In this subsection, we generalize the bootstrapping approach of TD to multi-time-step bootstrapping (which we refer to as n-step bootstrapping). We start with Tabular Prediction as it is very easy to explain and understand. To understand n-step bootstrapping, let us take another look at the TD update equation for the Tabular case (Equation (1.8)):

$$V(S_t) \leftarrow V(S_t) + \alpha \cdot (R_{t+1} + \gamma \cdot V(S_{t+1}) - V(S_t))$$

The basic idea was that we replaced G_t (in the case of the MC update) with the estimate $R_{t+1} + \gamma \cdot V(S_{t+1})$, by using the current estimate of the Value Function of the state that is 1 time step ahead on the trace experience. It's then natural to extend this idea to instead use the current estimate of the Value Function for the state that is 2 time steps ahead on the trace experience, which would yield the following update:

$$V(S_t) \leftarrow V(S_t) + \alpha \cdot (R_{t+1} + \gamma \cdot R_{t+2} + \gamma^2 \cdot V(S_{t+2}) - V(S_t))$$

We can generalize this to an update that uses the current estimate of the Value Function for the state that is $n \geq 1$ time steps ahead on the trace experience, as follows:

$$V(S_t) \leftarrow V(S_t) + \alpha \cdot (G_{t,n} - V(S_t)) \tag{1.13}$$

where $G_{t,n}$ (known as n-step bootstrapped return) is defined as:

$$\begin{aligned} G_{t,n} &= \sum_{i=t+1}^{t+n} \gamma^{i-t-1} \cdot R_i + \gamma^n \cdot V(S_{t+n}) \\ &= R_{t+1} + \gamma \cdot R_{t+2} + \gamma^2 \cdot R_{t+3} + \ldots + \gamma^{n-1} \cdot R_{t+n} + \gamma^n \cdot V(S_{t+n}) \end{aligned}$$

If the trace experience terminates at $t = T$, i.e., $S_T \in \mathcal{T}$, the above equation applies only for t, n such that $t + n < T$. Essentially, each n-step bootstrapped return $G_{t,n}$ is an approximation of the full return G_t, by truncating G_t at n steps and adjusting for the remainder with the Value Function estimate $V(S_{t+n})$ for the state S_{t+n}. If $t + n \geq T$, then there is no need for a truncation and the n-step bootstrapped return $G_{t,n}$ is equal to the full return G_t.

It is easy to generalize this n-step bootstrapping Prediction algorithm to the case of Function Approximation for the Value Function. The update Equation (1.13) generalizes to:

$$\Delta \boldsymbol{w} = \alpha \cdot (G_{t,n} - V(S_t; \boldsymbol{w})) \cdot \nabla_{\boldsymbol{w}} V(S_t; \boldsymbol{w}) \tag{1.14}$$

where the n-step bootstrapped return $G_{t,n}$ is now defined in terms of the function approximation for the Value Function (rather than the tabular Value Function), as follows:

$$G_{t,n} = \sum_{i=t+1}^{t+n} \gamma^{i-t-1} \cdot R_i + \gamma^n \cdot V(S_{t+n}; \boldsymbol{w})$$
$$= R_{t+1} + \gamma \cdot R_{t+2} + \gamma^2 \cdot R_{t+3} + \ldots + \gamma^{n-1} \cdot R_{t+n} + \gamma^n \cdot V(S_{t+n}; \boldsymbol{w})$$

The nuances we outlined above for when the trace experience terminates naturally apply here as well.

Equation (1.14) looks similar to the parameters update equations for the MC and TD Prediction algorithms we covered earlier, in terms of conceptualizing the change in parameters as the product of the following 3 entities:

- *Learning Rate* α
- n*-step Bootstrapped Error* $G_{t,n} - V(S_t; \boldsymbol{w})$
- *Estimate Gradient* of the conditional expected return $V(S_t; \boldsymbol{w})$ with respect to the parameters $\boldsymbol{w}$

n serves as a parameter taking us across the spectrum from TD to MC. $n = 1$ is the case of TD while sufficiently large n is the case of MC. If a trace experience is of length T (i.e., $S_T \in \mathcal{T}$), then $n \geq T$ will not have any bootstrapping (since the bootstrapping target goes beyond the length of the trace experience) and hence, this makes it identical to MC.

We note that for large n, the update to the Value Function for state S_t visited at time t happens in a delayed manner (after n steps, at time $t + n$), which is unlike the TD algorithm we had developed earlier where the update happens at the very next time step. We won't be implementing this n-step bootstrapping Prediction algorithm and leave it as an exercise for you to implement (re-using some of the functions/classes we have developed so far in this book). A key point to note for your implementation: The input won't be an `Iterable` of atomic experiences (like in the case of the TD Prediction algorithm we implemented), rather it will be an `Iterable` of trace experiences (i.e., the input will be the same as for our MC Prediction algorithm: `Iterable[Iterable[TransitionStep[S]]]`) since we need multiple future rewards in the trace to perform an update to the current state.

1.6.2. λ-Return Prediction Algorithm

Now we extend the n-step bootstrapping Prediction Algorithm to the λ-Return Prediction Algorithm. The idea behind this extension is really simple: Since the target for each n (in n-step bootstrapping) is $G_{t,n}$, a valid target can also be a weighted-average target:

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

Note that 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 the weights u_n and u, and applies to episodic problems (i.e., where every trace experience terminates). For a given state S_t with the episode terminating at time T (i.e., $S_T \in \mathcal{T}$), the weights for the λ-Return target are 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:

$$G_t^{(\lambda)} = (1-\lambda) \cdot \sum_{n=1}^{T-t-1} \lambda^{n-1} \cdot G_{t,n} + \lambda^{T-t-1} \cdot G_t \tag{1.15}$$

Thus, the update Equation is:

$$\Delta \boldsymbol{w} = \alpha \cdot (G_t^{(\lambda)} - V(S_t; \boldsymbol{w})) \cdot \nabla_{\boldsymbol{w}} V(S_t; \boldsymbol{w}) \tag{1.16}$$

We note that for $\lambda = 0$, the λ-Return target reduces to the TD (1-step bootstrapping) target and for $\lambda = 1$, the λ-Return target reduces to the MC target G_t. The λ parameter gives us a smooth way of tuning from TD ($\lambda = 0$) to MC ($\lambda = 1$).

Note that for $\lambda > 0$, Equation (1.16) tells us that the parameters $\boldsymbol{w}$ of the function approximation can be updated only at the end of an episode (the term *episode* refers to a terminating trace experience). Updating $\boldsymbol{w}$ according to Equation (1.16) for all states $S_t, t = 0, \ldots, T-1$, *at the end of each episode* gives us the *Offline* λ-Return Prediction algorithm. The term *Offline* refers to the fact that we have to wait till the end of an episode to make an update to the parameters $\boldsymbol{w}$ of the function approximation (rather than making parameter updates after each time step in the episode, which we refer to as an *Online* algorithm). Online algorithms are appealing because the Value Function update for an atomic experience could be utilized immediately by the updates for the next few atomic experiences, and so it facilitates continuous/fast learning. So the natural question to ask here is if we can turn the Offline λ-return Prediction algorithm outlined above to an Online version. An online version is indeed possible (it's known as the TD(λ) Prediction algorithm) and is the topic of the remaining subsections of this section. But before we begin the coverage of the (Online) TD(λ) Prediction algorithm, let's wrap up this subsection with an implementation of this Offline version (i.e., the λ-Return Prediction algorithm).

```
import rl.markov_process as mp
import numpy as np
from rl.approximate_dynamic_programming import ValueFunctionApprox

def lambda_return_prediction(
        traces: Iterable[Iterable[mp.TransitionStep[S]]],
        approx_0: ValueFunctionApprox[S],
        gamma: float,
        lambd: float
) -> Iterator[ValueFunctionApprox[S]]:
    func_approx: ValueFunctionApprox[S] = approx_0
    yield func_approx

    for trace in traces:
        gp: List[float] = [1.]
        lp: List[float] = [1.]
        predictors: List[NonTerminal[S]] = []
        partials: List[List[float]] = []
        weights: List[List[float]] = []
        trace_seq: Sequence[mp.TransitionStep[S]] = list(trace)
        for t, tr in enumerate(trace_seq):
            for i, partial in enumerate(partials):
                partial.append(
                    partial[-1] +
                    gp[t - i] * (tr.reward - func_approx(tr.state)) +
                    (gp[t - i] * gamma * extended_vf(func_approx, tr.next_state)
                     if t < len(trace_seq) - 1 else 0.)
                )
                weights[i].append(
                    weights[i][-1] * lambd if t < len(trace_seq)
                    else lp[t - i]
```

```
                )
            predictors.append(tr.state)
            partials.append([tr.reward +
                             (gamma * extended_vf(func_approx, tr.next_state)
                              if t < len(trace_seq) - 1 else 0.)])
            weights.append([1. - (lambd if t < len(trace_seq) else 0.)])
            gp.append(gp[-1] * gamma)
            lp.append(lp[-1] * lambd)
        responses: Sequence[float] = [np.dot(p, w) for p, w in
                                      zip(partials, weights)]
        for p, r in zip(predictors, responses):
            func_approx = func_approx.update([(p, r)])
        yield func_approx
```

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

Note that this λ-Return Prediction algorithm is not just Offline, it is also a highly inefficient algorithm because of the two loops within each trace experience. However, it serves as a pedagogical benefit before moving on to the (efficient) Online TD(λ) Prediction algorithm.

1.6.3. Eligibility Traces

Now we are ready to start developing the TD(λ) Prediction algorithm. The TD(λ) Prediction algorithm is founded on the concept of *Eligibility Traces*. So we start by introducing the concept of Eligibility traces (first for the Tabular case, then generalize to Function Approximations), then go over the TD(λ) Prediction algorithm (based on Eligibility traces), and finally explain why the TD(λ) Prediction algorithm is essentially the *Online* version of the *Offline* λ-Return Prediction algorithm we've implemented above.

We begin the story of Eligibility Traces with the concept of a (for lack of a better term) *Memory* function. Assume that we have an event happening at specific points in time, say at times $t_1, t_2, \ldots, t_n \in \mathbb{R}_{\geq 0}$ with $t_1 < t_2 < \ldots < t_n$, and we'd like to construct a *Memory* function $M : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ such that the *Memory* function (at any point in time t) remembers the number of times the event has occurred up to time t, but also has an element of "forgetfulness" in the sense that recent occurrences of the event are remembered better than older occurrences of the event. So the function M needs to have an element of memory-decay in remembering the count of the occurrences of the events. In other words, we want the function M to produce a time-decayed count of the event occurrences. We do this by constructing the function M as follows (for some decay-parameter $\theta \in [0, 1]$):

$$M(t) = \begin{cases} \mathbb{I}_{t=t_1} & \text{if } t \leq t_1, \text{ else} \\ M(t_i) \cdot \theta^{t-t_i} + \mathbb{I}_{t=t_{i+1}} & \text{if } t_i < t \leq t_{i+1} \text{ for any } 1 \leq i < n, \text{ else} \\ M(t_n) \cdot \theta^{t-t_n} & \text{otherwise (i.e., if } t > t_n) \end{cases} \tag{1.17}$$

where $\mathbb{I}$ denotes the indicator function.

This means the memory function has an uptick of 1 each time the event occurs (at time t_i, for each $i = 1, 2, \ldots n$), but then decays by a factor of $\theta^{\Delta t}$ over any interval Δt where the event doesn't occur. Thus, the memory function captures the notion of frequency of the events as well as the recency of the events.

Let's write some code to plot this function in order to visualize it and gain some intuition.

```
def plot_memory_function(theta: float, event_times: List[float]) -> None:
    step: float = 0.01
```

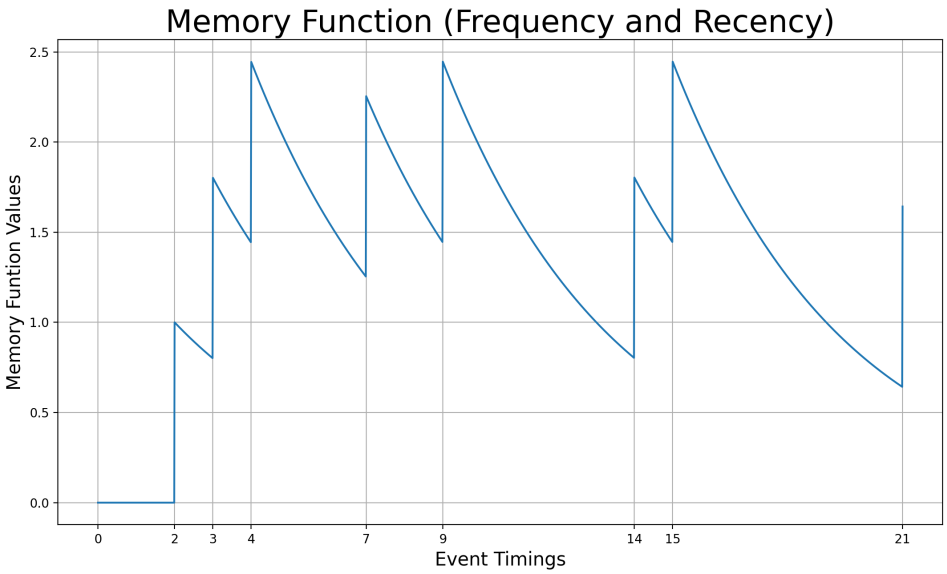

Figure 1.6.: Memory Function (Frequency and Recency)

```
    x_vals: List[float] = [0.0]
    y_vals: List[float] = [0.0]
    for t in event_times:
        rng: Sequence[int] = range(1, int(math.floor((t - x_vals[-1]) / step)))
        x_vals += [x_vals[-1] + i * step for i in rng]
        y_vals += [y_vals[-1] * theta ** (i * step) for i in rng]
        x_vals.append(t)
        y_vals.append(y_vals[-1] * theta ** (t - x_vals[-1]) + 1.0)
    plt.plot(x_vals, y_vals)
    plt.grid()
    plt.xticks([0.0] + event_times)
    plt.xlabel("Event Timings", fontsize=15)
    plt.ylabel("Memory Funtion Values", fontsize=15)
    plt.title("Memory Function (Frequency and Recency)", fontsize=25)
    plt.show()
```

Let's run this for $\theta = 0.8$ and an arbitrary sequence of event times:

```
theta = 0.8
event_times = [2.0, 3.0, 4.0, 7.0, 9.0, 14.0, 15.0, 21.0]
plot_memory_function(theta, event_times)
```

This produces the graph in Figure 1.6.

The above code is in the file rl/chapter10/memory_function.py.

This memory function is actually quite useful as a model for a variety of modeling situations in the broader world of Applied Mathematics where we want to combine the notions of frequency and recency. However, here we want to use this memory function as a way to model *Eligibility Traces* for the tabular case, which in turn will give us the tabular TD(λ) Prediction algorithm (online version of the offline tabular λ-Return Prediction algorithm we covered earlier).

Now we are ready to define Eligibility Traces for the Tabular case. We assume a finite state space with the set of non-terminal states $\mathcal{N} = \{s_1, s_2, \ldots, s_m\}$. Eligibility Trace for each state $s \in \mathcal{N}$ is defined as the Memory function $M(\cdot)$ with $\theta = \gamma \cdot \lambda$ (i.e., the product

of the discount factor and the TD-λ parameter) and the event timings are the time steps at which the state s occurs in a trace experience. Thus, we define Eligibility Traces for a given trace experience at any time step t (of the trace experience) as a function $E_t : \mathcal{N} \to \mathbb{R}_{\geq 0}$ as follows:

$$E_0(s) = \mathbb{I}_{S_0=s}, \text{ for all } s \in \mathcal{N}$$

$$E_t(s) = \gamma \cdot \lambda \cdot E_{t-1}(s) + \mathbb{I}_{S_t=s}, \text{ for all } s \in \mathcal{N}, \text{ for all } t = 1, 2, \ldots$$

where $\mathbb{I}$ denotes the indicator function.

Then, the Tabular TD(λ) Prediction algorithm performs the following updates to the Value Function at each time step t in each trace experience:

$$V(s) \leftarrow V(s) + \alpha \cdot (R_{t+1} + \gamma \cdot V(S_{t+1}) - V(S_t)) \cdot E_t(s), \textit{ for all } s \in \mathcal{N}$$

Note the similarities and differences relative to the TD update we have seen earlier. Firstly, this is an online algorithm since we make an update at each time step in a trace experience. Secondly, we update the Value Function *for all* states at each time step (unlike TD Prediction which updates the Value Function only for the particular state that is visited at that time step). Thirdly, the change in the Value Function for each state $s \in \mathcal{N}$ is proportional to the TD-Error $\delta_t = R_{t+1} + \gamma \cdot V(S_{t+1}) - V(S_t)$, much like in the case of the TD update. However, here the TD-Error is multiplied by the eligibility trace $E_t(s)$ for each state s at each time step t. So, we can compactly write the update as:

$$V(s) \leftarrow V(s) + \alpha \cdot \delta_t \cdot E_t(s), \textit{ for all } s \in \mathcal{N} \tag{1.18}$$

where α is the learning rate.

This is it - this is the Tabular TD(λ) Prediction algorithm! Now the question is - how is this linked to the Tabular λ-Return Prediction algorithm? It turns out that if we made all the updates of Equation (1.18) in an offline manner (at the end of each trace experience), then the sum of the changes in the Value Function for any specific state $s \in \mathcal{N}$ over the course of the entire trace experience is equal to the change in the Value Function for s in the Tabular λ-Return Prediction algorithm as a result of it's offline update for state s. Concretely,

Theorem 1.6.1.

$$\sum_{t=0}^{T-1} \alpha \cdot \delta_t \cdot E_t(s) = \sum_{t=0}^{T-1} \alpha \cdot (G_t^{(\lambda)} - V(S_t)) \cdot \mathbb{I}_{S_t=s}, \textit{ for all } s \in \mathcal{N}$$

where $\mathbb{I}$ denotes the indicator function.

Proof. We begin the proof with the following important identity:

$$\begin{aligned}
G_t^{(\lambda)} - V(S_t) = -V(S_t) \quad & +(1-\lambda)\cdot\lambda^0\cdot(R_{t+1}+\gamma\cdot V(S_{t+1})) \\
& +(1-\lambda)\cdot\lambda^1\cdot(R_{t+1}+\gamma\cdot R_{t+2}+\gamma^2\cdot V(S_{t+2})) \\
& +(1-\lambda)\cdot\lambda^2\cdot(R_{t+1}+\gamma\cdot R_{t+2}+\gamma^2\cdot R_{t+3}+\gamma^3\cdot V(S_{t+3})) \\
& +\ldots \\
= -V(S_t) \quad & +(\gamma\lambda)^0\cdot(R_{t+1}+\gamma\cdot V(S_{t+1})-\gamma\lambda\cdot V(S_{t+1})) \\
& +(\gamma\lambda)^1\cdot(R_{t+2}+\gamma\cdot V(S_{t+2})-\gamma\lambda\cdot V(S_{t+2})) \\
& +(\gamma\lambda)^2\cdot(R_{t+3}+\gamma\cdot V(S_{t+3})-\gamma\lambda\cdot V(S_{t+3})) \\
& +\ldots \\
= \quad & (\gamma\lambda)^0\cdot(R_{t+1}+\gamma\cdot V(S_{t+1})-V(S_t)) \\
& +(\gamma\lambda)^1\cdot(R_{t+2}+\gamma\cdot V(S_{t+2})-V(S_{t+1})) \\
& +(\gamma\lambda)^2\cdot(R_{t+3}+\gamma\cdot V(S_{t+3})-V(S_{t+2})) \\
& +\ldots \\
= \delta_t+\gamma\lambda\cdot\delta_{t+1}+(\gamma\lambda)^2\cdot\delta_{t+2}+\ldots &
\end{aligned} \tag{1.19}$$

Now assume that a specific non-terminal state s appears at time steps $t_1, t_2, \ldots, t_n$. Then,

$$\begin{aligned}
\sum_{t=0}^{T-1}\alpha\cdot(G_t^{(\lambda)}-V(S_t))\cdot\mathbb{I}_{S_t=s} &= \sum_{i=1}^{n}\alpha\cdot(G_{t_i}^{(\lambda)}-V(S_{t_i})) \\
&= \sum_{i=1}^{n}\alpha\cdot(\delta_{t_i}+\gamma\lambda\cdot\delta_{t_i+1}+(\gamma\lambda)^2\cdot\delta_{t_i+2}+\ldots) \\
&= \sum_{t=0}^{T-1}\alpha\cdot\delta_t\cdot E_t(s)
\end{aligned}$$

□

If we set $\lambda = 0$ in this Tabular TD(λ) Prediction algorithm, we note that $E_t(s)$ reduces to $\mathbb{I}_{S_t=s}$ and so, the Tabular TD(λ) prediction algorithm's update for $\lambda = 0$ at each time step t reduces to:

$$V(S_t) \leftarrow V(S_t) + \alpha\cdot\delta_t$$

which is exactly the update of the Tabular TD Prediction algorithm. Therefore, TD algorithms are often refered to as TD(0).

If we set $\lambda = 1$ in this Tabular TD(λ) Prediction algorithm with episodic traces (i.e., all trace experiences terminating), Theorem 1.6.1 tells us that the sum of all changes in the Value Function for any specific state $s \in \mathcal{N}$ over the course of the entire trace experience ($= \sum_{t=0}^{T-1}\alpha\cdot\delta_t\cdot E_t(s)$) is equal to the change in the Value Function for s in the Every-Visit MC Prediction algorithm as a result of it's offline update for state s ($= \sum_{t=0}^{T-1}\alpha\cdot(G_t - V(S_t)\cdot\mathbb{I}_{S_t=s})$). Hence, TD(1) is considered to be "equivalent" to Every-Visit MC.

To clarify, TD(λ) Prediction is an online algorithm and hence, not exactly equivalent to the offline λ-Return Prediction algorithm. However, if we modified the TD(λ) Prediction algorithm to be offline, then they are equivalent. The offline version of TD(λ) Prediction

would not make the updates to the Value Function at each time step - rather, it would accumulate the changes to the Value Function (as prescribed by the TD(λ) update formula) in a buffer, and then at the end of the trace experience, it would update the Value Function with the contents of the buffer.

However, as explained earlier, online update are desirable because the changes to the Value Function at each time step can be immediately usable for the next time steps' updates and so, it promotes rapid learning without having to wait for a trace experience to end. Moreover, online algorithms can be used in situations where we don't have a complete episode.

With an understanding of Tabular TD(λ) Prediction in place, we can generalize TD(λ) Prediction to the case of function approximation in a straightforward manner. In the case of function approximation, the data type of eligibility traces will be the same data type as that of the parameters $\boldsymbol{w}$ in the function approximation (so here we denote eligibility traces at time t of a trace experience as simply $\boldsymbol{E}_t$ rather than as a function of states as we had done for the Tabular case above). We initialize $\boldsymbol{E}_0$ at the start of each trace experience to $\nabla_{\boldsymbol{w}} V(S_0; \boldsymbol{w})$. Then, for each time step $t > 0$, $\boldsymbol{E}_t$ is calculated recursively in terms of the previous time step's value $\boldsymbol{E}_{t-1}$, which is then used to update the parameters of the Value Function approximation, as follows:

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

$$\Delta \boldsymbol{w} = \alpha \cdot (R_{t+1} + \gamma \cdot V(S_{t+1}; \boldsymbol{w}) - V(S_t; \boldsymbol{w})) \cdot \boldsymbol{E}_t$$

The update to the parameters $\boldsymbol{w}$ can be expressed more succinctly as:

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

where δ_t now denotes the TD Error based on the function approximation for the Value Function.

The idea of Eligibility Traces has it's origins in a seminal book by Harry Klopf (Klopf and Data Sciences Laboratory 1972) that greatly influenced Richard Sutton and Andrew Barto to pursue the idea of Eligibility Traces further, after which they published several papers on Eligibility Traces, much of whose content is covered in their RL book (Sutton and Barto 2018).

1.6.4. Implementation of the TD(λ) Prediction algorithm

You'd have observed that the TD(λ) update is not as simple as the MC and TD updates, where we were able to use the `FunctionApprox` interface in a straightforward manner. For TD(λ), it might appear that we can't quite use the `FunctionApprox` interface and would need to write custom-code for it's implementation. However, by noting that the `FunctionApprox` method `objective_gradient` is quite generic and that `FunctionApprox` and `Gradient` support methods `__add__` and `__mul__` (vector space operations), we can actually implement the TD(λ) in terms of the `FunctionApprox` interface.

The function `td_lambda_prediction` below takes as input an `Iterable` of trace experiences (`traces`), an initial `FunctionApprox` (`approx_0`), and the γ and λ parameters. At the start of each trace experience, we need to initialize the eligibility traces to 0. The data type of the eligibility traces is the `Gradient` type and so we invoke the `zero` method for `Gradient(func_approx)` in order to initialize the eligibility traces to 0. Then, at every time step in every trace experience, we first set the predictor variable x_t to be the state and the

response variable y_t to be the TD target. Then we need to update the eligibility traces `el_tr` and update the function approximation `func_approx` using the updated `el_tr`.

Thankfully, the `__mul__` method of `Gradient` class enables us to conveniently multiply `el_tr` with $\gamma \cdot \lambda$ and then, it also enables us to multiply the updated `el_tr` with the prediction error $\mathbb{E}_M[y|x_t] - y_t = V(S_t; \boldsymbol{w}) - (R_{t+1} + \gamma \cdot V(S_{t+1}; \boldsymbol{w}))$ (in the code as `func_approx(x) - y`), which is then used (as a `Gradient` type) to update the internal parameters of the `func_approx`. The `__add__` method of `Gradient` enables us to add $\nabla_{\boldsymbol{w}} V(S_t; \boldsymbol{w})$ (as a `Gradient` type) to `el_tr * gamma * lambd`. The only seemingly difficult part is calculating $\nabla_{\boldsymbol{w}} V(S_t; \boldsymbol{w})$. The `FunctionApprox` interface provides us with a method `objective_gradient` to calculate the gradient of any specified objective (call it $Obj(x, y)$). But here we have to calculate the gradient of the prediction of the function approximation. Thankfully, the interface of `objective_gradient` is fairly generic and we actually have a choice of constructing $Obj(x, y)$ to be whatever function we want (not necessarily a minimizing Objective Function). We specify $Obj(x, y)$ in terms of the `obj_deriv_out_func` argument, which as a reminder, represents $\frac{\partial Obj(x,y)}{\partial Out(x)}$. Note that we have assumed a gaussian distribution for the returns conditioned on the state. So we can set $Out(x)$ to be the function approximation's prediction $V(S_t; \boldsymbol{w})$ and we can set $Obj(x, y) = Out(x)$, meaning `obj_deriv_out_func` ($\frac{\partial Obj(x,y)}{\partial Out(x)}$) is a function returning the constant value of 1 (as seen in the code below).

```
import rl.markov_process as mp
import numpy as np
from rl.function_approx import Gradient
from rl.approximate_dynamic_programming import ValueFunctionApprox

def td_lambda_prediction(
        traces: Iterable[Iterable[mp.TransitionStep[S]]],
        approx_0: ValueFunctionApprox[S],
        gamma: float,
        lambd: float
) -> Iterator[ValueFunctionApprox[S]]:
    func_approx: ValueFunctionApprox[S] = approx_0
    yield func_approx

    for trace in traces:
        el_tr: Gradient[ValueFunctionApprox[S]] = Gradient(func_approx).zero()
        for step in trace:
            x: NonTerminal[S] = step.state
            y: float = step.reward + gamma * \
                extended_vf(func_approx, step.next_state)
            el_tr = el_tr * (gamma * lambd) + func_approx.objective_gradient(
                xy_vals_seq=[(x, y)],
                obj_deriv_out_fun=lambda x1, y1: np.ones(len(x1))
            )
            func_approx = func_approx.update_with_gradient(
                el_tr * (func_approx(x) - y)
            )
            yield func_approx
```

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

Let's use the same instance `si_mrp: SimpleInventoryMRPFinite` that we had created above when testing MC and TD Prediction. We use the same number of episodes (60000) we had used when testing MC Prediction. Just like in the case of testing TD prediction, we set initial learning rate $\alpha = 0.03$, half life $H = 1000$ and exponent $\beta = 0.5$. We set the episode length (number of atomic experiences in a single trace experience) to be 100 (same as with the settings we had for testing TD Prediction and consistent with MC Prediction as well). We use the same discount factor $\gamma = 0.9$. Let's set $\lambda = 0.3$.

```
import rl.iterate as iterate
import rl.td_lambda as td_lambda
import itertools
from pprint import pprint
from rl.chapter10.prediction_utils import fmrp_episodes_stream
from rl.function_approx import learning_rate_schedule

gamma: float = 0.9
episode_length: int = 100
initial_learning_rate: float = 0.03
half_life: float = 1000.0
exponent: float = 0.5
lambda_param = 0.3

episodes: Iterable[Iterable[TransitionStep[S]]] = \
    fmrp_episodes_stream(si_mrp)
curtailed_episodes: Iterable[Iterable[TransitionStep[S]]] = \
        (itertools.islice(episode, episode_length) for episode in episodes)
learning_rate_func: Callable[[int], float] = learning_rate_schedule(
    initial_learning_rate=initial_learning_rate,
    half_life=half_life,
    exponent=exponent
)
td_lambda_vfs: Iterator[ValueFunctionApprox[S]] = td_lambda.td_lambda_prediction(
    traces=curtailed_episodes,
    approx_0=Tabular(count_to_weight_func=learning_rate_func),
    gamma=gamma,
    lambd=lambda_param
)

num_episodes = 60000

final_td_lambda_vf: ValueFunctionApprox[S] = \
    iterate.last(itertools.islice(td_lambda_vfs, episode_length * num_episodes))
pprint({s: round(final_td_lambda_vf(s), 3) for s in si_mrp.non_terminal_states})
```

This prints the following:

```
{NonTerminal(state=InventoryState(on_hand=0, on_order=0)): -35.545,
 NonTerminal(state=InventoryState(on_hand=0, on_order=1)): -27.97,
 NonTerminal(state=InventoryState(on_hand=0, on_order=2)): -28.396,
 NonTerminal(state=InventoryState(on_hand=1, on_order=0)): -28.943,
 NonTerminal(state=InventoryState(on_hand=1, on_order=1)): -29.506,
 NonTerminal(state=InventoryState(on_hand=2, on_order=0)): -30.339}
```

Thus, we see that our implementation of TD(λ) Prediction with the above settings fetches us an estimated Value Function fairly close to the true Value Function. As ever, we encourage you to play with various settings for TD(λ) Prediction to develop an intuition for how the results change as you change the settings, and particularly as you change the λ parameter. You can play with the code in the file rl/chapter10/simple_inventory_mrp.py.

1.7. Key Takeaways from this Chapter

- Bias-Variance tradeoff of TD versus MC.
- MC learns the statistical mean of the observed returns while TD learns something "deeper" - it implicitly estimates an MRP from the observed data and produces the Value Function of the implicitly-estimated MRP.
- Understanding TD versus MC versus DP from the perspectives of "bootstrapping" and "experiencing" (Figure 1.5 provides a great view).

- "Equivalence" of λ-Return Prediction and TD(λ) Prediction, hence TD is equivalent to TD(0) and MC is "equivalent" to TD(1).

Bibliography

Klopf, A. H., and Air Force Cambridge Research Laboratories (U.S.). Data Sciences Laboratory. 1972. *Brain Function and Adaptive Systems–a Heterostatic Theory*. Special Reports. Data Sciences Laboratory, Air Force Cambridge Research Laboratories, Air Force Systems Command, United States Air Force. https://books.google.com/books?id=C2hztwEACAAJ.

Sutton, Richard S., and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction*. Second. The MIT Press. http://incompleteideas.net/book/the-book-2nd.html.

Watkins, C. J. C. H. 1989. "Learning from Delayed Rewards." PhD thesis, King's College, Oxford.