

A Guided Tour of Chapter 9: Reinforcement Learning for Prediction

Ashwin Rao

ICME, Stanford University

RL does not have access to a probability model

- DP/ADP assume access to probability model (knowledge of \mathcal{P}_R)
- Often in real-world, we do not have access to these probabilities
- Which means we'd need to *interact* with the *actual environment*
- Actual Environment serves up individual experiences, not probabilities
- Even if MDP model is available, model updates can be challenging
- Often real-world models end up being too large or too complex
- Sometimes estimating a *sampling model* is much more feasible
- So RL interacts with either *actual* or *simulated* environment
- Either way, we receive *individual experiences* of next state and reward
- RL learns Value Functions from a stream of individual experiences
- How does RL solve Prediction and Control with such limited access?

The RL Approach

- Like humans/animals, RL doesn't aim to estimate probability model
- Rather, RL is a “trial-and-error” approach to linking actions to returns
- This is hard because actions have overlapping reward sequences
- Also, sometimes actions result in *delayed rewards*
- The key is incrementally updating Q -Value Function from experiences
- Appropriate Approximation of Q -Value Function is also key to success
- Most RL algorithms are founded on the *Bellman Equations*
- Moreover, RL Control is based on *Generalized Policy Iteration*
- This lecture/chapter focuses on RL for Prediction

- Prediction: Problem of estimating MDP Value Function for a policy π
- Equivalently, problem of estimating π -implied MRP's Value Function
- Assume interface serves an *atomic experience* of (next state, reward)
- Interacting with this interface repeatedly provides a *trace experience*

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

- Value Function $V : \mathcal{N} \rightarrow \mathbb{R}$ of an MRP is defined as:

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

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

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

Code interface for RL Prediction

An *atomic experience* is represented as a `TransitionStep [S]`

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

Input to RL prediction can be either of:

- Atomic Experiences as `Iterable [TransitionStep [S]]`, or
- Trace Experiences as `Iterable [Iterable [TransitionStep [S]]]`

Note that `Iterable` can be either a `Sequence` or an `Iterator` (i.e., *stream*)

Monte-Carlo (MC) Prediction

- Supervised learning with states and returns from trace experiences
- Incremental estimation with update method of FunctionApprox
- x-values are states S_t , y-values are returns G_t
- Note that updates can be done only at the end of a trace experience
- Returns calculated with a backward walk: $G_t = R_{t+1} + \gamma \cdot G_{t+1}$

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

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

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

MC Prediction code

```
def mc_prediction(  
    trs: Iterable[Iterable[TransitionStep[S]]],  
    approx_0: FunctionApprox[S],  
    gamma: float,  
    tol: float = 1e-6  
) -> Iterator[FunctionApprox[S]]:  
  
    episodes: Iterator[Iterator[ReturnStep[S]]] = \  
        (returns(tr, gamma, tol) for tr in trs)  
  
    return approx_0.iterate_updates(  
        ((st.state, st.return_) for st in episode)  
        for episode in episodes  
    )
```

Structure of the parameters update formula

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

The update $\Delta \mathbf{w}$ to parameters \mathbf{w} should be seen as product of:

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

This structure (as product of above 3 entities) will be a repeated pattern.

Tabular MC Prediction

- Finite state space, let's say non-terminal states $\mathcal{N} = \{s_1, s_2, \dots, s_m\}$
- Denote $V_n(s_i)$ as estimate of VF after the n -th occurrence of s_i
- Denote $Y_i^{(1)}, Y_i^{(2)}, \dots, Y_i^{(n)}$ as returns for first n occurrences of s_i
- Denote `count_to_weight_func` attribute of Tabular as $f(\cdot)$
- Then the Tabular update at the n -th occurrence of s_i is:

$$\begin{aligned} V_n(s_i) &= (1 - f(n)) \cdot V_{n-1}(s_i) + f(n) \cdot Y_i^{(n)} \\ &= V_{n-1}(s_i) + f(n) \cdot (Y_i^{(n)} - V_{n-1}(s_i)) \end{aligned}$$

- So update to VF for s_i is *Latest Weight* times *Return Residual*
- For default setting of `count_to_weight_func` as $f(n) = \frac{1}{n}$:

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

Tabular MC Prediction

- Expanding the incremental updates across values of n , we get:

$$V_n(s_i) = f(n) \cdot Y_i^{(n)} + (1 - f(n)) \cdot f(n-1) \cdot Y_i^{(n-1)} + \dots \\ \dots + (1 - f(n)) \cdot (1 - f(n-1)) \dots (1 - f(2)) \cdot f(1) \cdot Y_i^{(1)}$$

- For default setting of `count_to_weight_func` as $f(n) = \frac{1}{n}$:

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

- Tabular MC is simply incremental calculation of averages of returns
- Exactly the calculation in the update method of Tabular class
- View Tabular MC as an application of Law of Large Numbers

Tabular MC as a special case of Linear Func Approximation

- Features functions are indicator functions for states
- Linear-approx parameters are Value Function estimates for states
- `count_to_weight_func` plays the role of learning rate
- So tabular Value Function update can be written as:

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

- $Y_i^{(n)} - w_i^{(n-1)}$ represents the gradient of the loss function
- For non-stationary problems, algorithm needs to “forget” distant past
- With constant learning rate α , time-decaying weights:

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

- Weights sum to 1 asymptotically: $\lim_{n \rightarrow \infty} \sum_{j=1}^n \alpha \cdot (1 - \alpha)^{n-j} = 1$

Each-Visit MC and First-Visit MC

- The MC algorithm we covered is known as *Each-Visit Monte-Carlo*
- Because we include each occurrence of a state in a trace experience
- Alternatively, we can do *First-Visit Monte-Carlo*
- Only the first occurrence of a state in a trace experience is considered
- Keep track of whether a state has been visited in a trace experience
- MC Prediction algorithms are easy to understand and implement
- MC produces unbiased estimates but can be slow to converge
- Key disadvantage: MC requires complete trace experiences

Testing RL Algorithms

- Start with `MarkovRewardProcess` or `MarkovDecisionProcess`
- Solve it with a DP/ADP Algorithm
- But RL does not (cannot) have access to transition probabilities
- Use method `reward_traces` or `simulate_actions` to generate episodes
- Use episodes or transitions `Iterable` as input to RL algorithm
- Compare solution of RL algorithm to that of DP/ADP Algorithm

Temporal-Difference (TD) Prediction

- To understand TD, we start with Tabular TD Prediction
- Key: Exploit recursive structure of VF in MRP Bellman Equation
- Replace G_t with $R_{t+1} + \gamma \cdot V(S_{t+1})$ using atomic experience data
- So we are *bootstrapping* the VF (“estimate from estimate”)
- The tabular MC Prediction update (for constant α) is modified from:

$$V(S_t) \leftarrow V(S_t) + \alpha \cdot (G_t - V(S_t))$$

to:

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

- $R_{t+1} + \gamma \cdot V(S_{t+1})$ known as *TD target*
- $\delta_t = R_{t+1} + \gamma \cdot V(S_{t+1}) - V(S_t)$ known as *TD Error*
- TD Error is the crucial quantity - it represents “sample Bellman Error”
- VF is adjusted so as to bridge TD error (on an expected basis)

TD updates after each atomic experience

- Unlike MC, we can use TD when we have incomplete traces
- Often in real-world situations, experiments gets curtailed/disrupted
- Also, we can use TD in non-episodic (known as *continuing*) traces
- TD updates VF after each atomic experience (“continuous learning”)
- So TD can be run on *any* stream of atomic experiences
- This means we can chop up the input stream and serve in any order

TD Prediction with Function Approximation

- Each atomic experience leads to a parameters update
- To understand how parameters update work, consider:

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

- Above formula replaces G_t (of MC) with $R_{t+1} + \gamma \cdot V(S_{t+1}, \mathbf{w})$
- Unlike MC, in TD, we don't take the gradient of this loss function
- "Cheat" in gradient calc by ignoring dependency of $V(S_{t+1}; \mathbf{w})$ on \mathbf{w}
- This "gradient with cheating" calculation is known as *semi-gradient*
- So we pretend the only dependency on \mathbf{w} is through $V(S_t; \mathbf{w})$

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

TD Prediction code

```
def td_prediction(  
    trans: Iterable[TransitionStep[S]],  
    approx_0: FunctionApprox[S],  
    gamma: float,  
) -> Iterator[FunctionApprox[S]:  
  
    def step(  
        v: FunctionApprox[S],  
        tr: TransitionStep[S]  
    ) -> FunctionApprox[S]:  
        return v.update([(  
            tr.state,  
            tr.reward + gamma * v(tr.next_state)  
        )])  
  
    return iterate.accumulate(trans, step, approx_0)
```

Structure of the parameters update formula

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

The update $\Delta \mathbf{w}$ to parameters \mathbf{w} should be seen as product of:

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

So parameters update formula has same product-structure as MC

TD's many benefits

- “TD is the most significant and innovative idea in RL” - Rich Sutton
- Blends the advantages of DP and MC
- Like DP, TD learns by bootstrapping (drawing from Bellman Eqn)
- Like MC, TD learns from experiences without access to probabilities
- So TD overcomes curse of dimensionality and curse of modeling
- TD also has the advantage of not requiring entire trace experiences
- Most significantly, TD is akin to human (continuous) learning

Bias, Variance and Convergence of TD versus MC

- MC uses G_t is an unbiased estimate of the Value Function
- This helps MC with convergence even with function approximation
- TD uses $R_{t+1} + \gamma \cdot V(S_{t+1}; \mathbf{w})$ as a biased estimate of the VF
- Tabular TD prediction converges to true VF in the mean for const α
- And converges to true VF under Robbins-Monro learning rate schedule

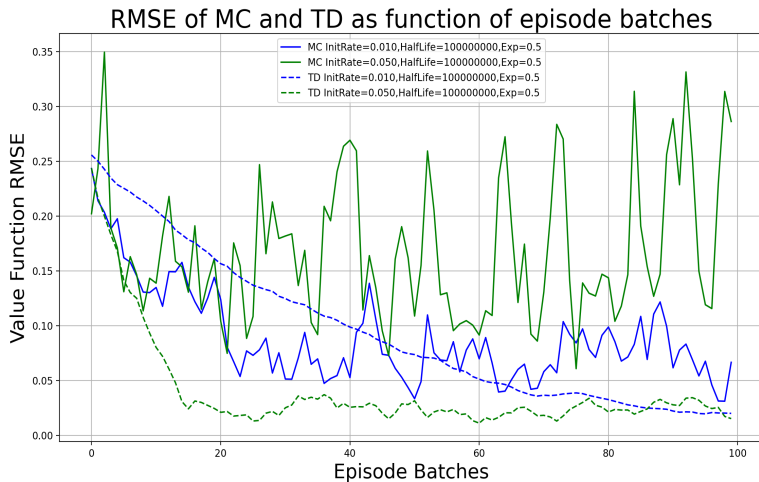
$$\sum_{n=1}^{\infty} \alpha_n = \infty \text{ and } \sum_{n=1}^{\infty} \alpha_n^2 < \infty$$

- However, Robbins-Monro schedule is not so useful in practice
- TD Prediction with func-approx does not always converge to true VF
- Most convergence proofs are for Tabular, some for linear func-approx
- TD Target $R_{t+1} + \gamma \cdot V(S_{t+1}; \mathbf{w})$ has much lower variance than G_t
- G_t depends on many random rewards whose variances accumulate
- TD Target depends on only the next reward, so lower variance

Speed of Convergence of TD versus MC

- We typically compare algorithms based on:
 - Speed of convergence
 - Efficiency in use of limited set of experiences data
- There are no formal proofs for MC v/s TD on above criterion
- MC and TD have significant differences in their:
 - Usage of data
 - Nature of updates
 - Frequency of updates
- So unclear exactly how to compare them apples to apples
- Typically, MC and TD are compared with constant α
- Practically/empirically, TD does better than MC with constant α
- Also, MC is not very sensitive to initial Value Function, but TD is

Convergence of MC versus TD with constant α



RMSE of MC versus TD as function of episodes

- Symmetric random walk with barrier $B = 10$ and no discounting
- Graph depicts RMSE after every 7th episode (700 episodes in all)
- Blue curves for constant $\alpha = 0.01$, green for constant $\alpha = 0.05$
- Notice how MC has significantly more variance
- RMSE progression is quite slow on blue curves (small learning rate)
- MC progresses quite fast initially but then barely progresses
- TD gets to fairly small RMSE quicker than corresponding MC
- This performance of TD versus MC is typical for constant α

Fixed-Data Experience Replay on TD versus MC

- So far, we've understood *how* TD learns versus *how* MC learns
- Now we want to understand *what* TD learns versus *what* MC learns
- To illustrate, we consider a finite set of trace experiences
- The agent can tap into this finite set of traces experiences endlessly
- But everything is ultimately sourced from this finite data set
- So we'd end up tapping into these experiences repeatedly
- We call this technique *Experience Replay*

```
data: Sequence[Sequence[Tuple[str, float]]] = [  
    [( 'A', 2.), ( 'A', 6.), ( 'B', 1.), ( 'B', 2.)],  
    [( 'A', 3.), ( 'B', 2.), ( 'A', 4.), ( 'B', 2.), ( 'B',  
    [( 'B', 3.), ( 'B', 6.), ( 'A', 1.), ( 'B', 1.)],  
    [( 'A', 0.), ( 'B', 2.), ( 'A', 4.), ( 'B', 4.), ( 'B',  
    [( 'B', 8.), ( 'B', 2.)]  
]
```


MC and TD learn different Value Functions

- It is quite obvious what MC Prediction algorithm would learn
- MC Prediction is simply supervised learning with (state, return) pairs
- But here those pairs ultimately come from the given finite pairs
- So, MC estimates Value Function as average returns in the finite data
- Running MC Prediction algo matches explicit average returns calc
- But running TD Prediction algo gives significantly different answer
- So what is TD Prediction algorithm learning?
- TD drives towards VF of MRP *implied* by the finite experiences
- Specifically, learns MLE for \mathcal{P}_R from the given finite data

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

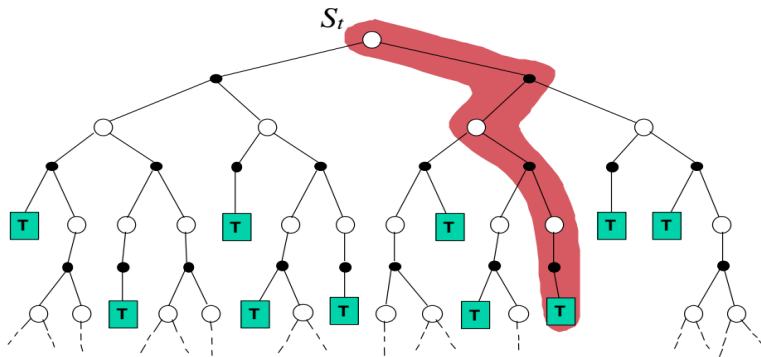
- TD is advantageous in Markov environments, MC in non-Markov

Bootstrapping and Experiencing

- We summarize MC, TD and DP in terms of whether they:
 - Bootstrap: Update to VF utilizes a current or prior estimate of the VF
 - Experience: Interaction with actual or simulated environment
- TD and DP *do bootstrap* (updates use current/prior estimate of VF)
- MC *does not bootstrap* (updates use trace experience returns)
- MC and TD *do experience* (actual/simulated environment interaction)
- DP *does not experience* (updates use transition probabilities)
- Bootstrapping means backups are *shallow* (MC backups are *deep*)
- Experiencing means backups are *narrow* (DP backups are *wide*)

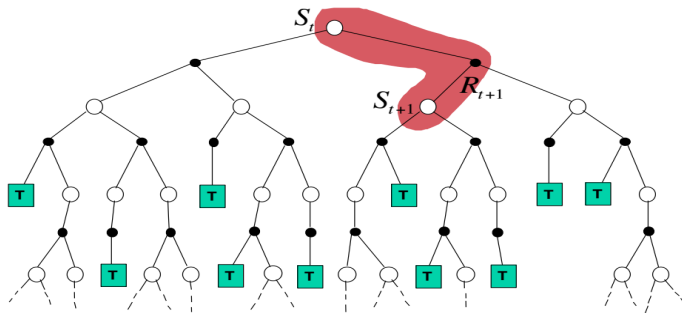
Monte Carlo (Supervised Learning) (MC)

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



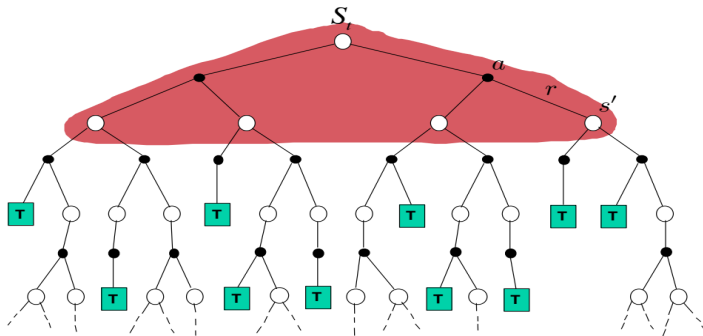
Simplest TD Method

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

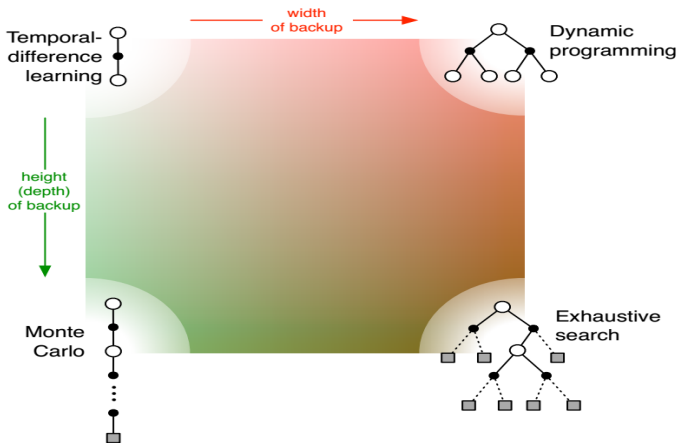


cf. Dynamic Programming

$$V(S_t) \leftarrow E_{\pi} [R_{t+1} + \gamma V(S_{t+1})]$$



Unified View



Tabular n -step Bootstrapping

- Tabular TD Prediction bootstraps the Value Function with update:

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

- So it's natural to extend this to bootstrapping with 2 steps ahead:

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

- Generalize to bootstrapping with $n \geq 1$ time steps ahead:

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

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

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

n -step Bootstrapping with Function Approximation

- Generalizing this to the case of Function Approximation, we get:

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

- This looks similar to formula for parameters update for MC and TD
- In terms of conceptualizing the change in parameters as product of:
 - *Learning Rate* α
 - *n -step Bootstrapped Error* $G_{t,n} - V(S_t; \mathbf{w})$
 - *Estimate Gradient* of the conditional expected return $V(S_t; \mathbf{w})$ with respect to the parameters \mathbf{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

λ -Return Prediction Algorithm

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

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

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

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

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

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

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

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

Online versus Offline

- Note that for $\lambda = 0$, the λ -Return target reduces to the TD target
- Note that for $\lambda = 1$, the λ -Return target reduces to the MC target G_t
- λ parameter enables us to finely tune from TD ($\lambda = 0$) to MC ($\lambda = 1$)
- Note that for $\lambda > 0$, updates are made only at the end of an episode
- Algorithms updating at end of episodes known as *Offline Algorithms*
- Online algorithms (updates after each time step) are appealing:
 - Updated VF can be utilized immediately for next time step's update
 - This facilitates continuous/fast learning
- Can we have a similar λ -tunable online algorithm for Prediction?
- Yes - this is known as the TD(λ) Prediction algorithm

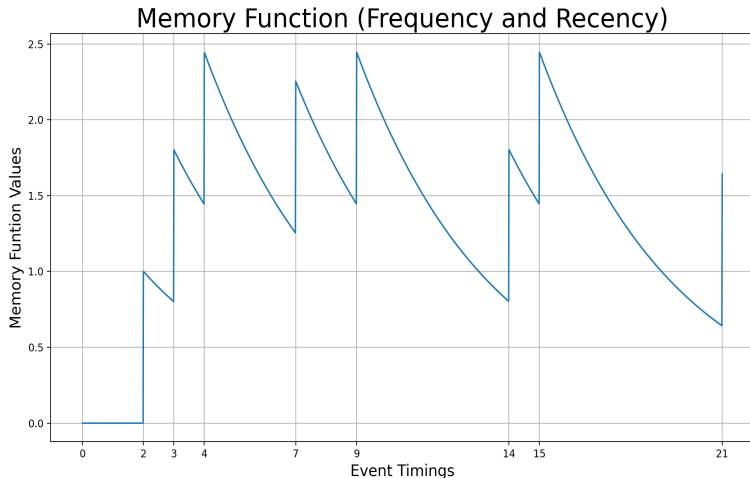
Memory Function

- TD(λ) algorithm is based on the concept of *Eligibility Traces*
- We introduce the concept by defining a *Memory Function* $M(t)$
- Assume an event occurs at times $t_1 < t_2 < \dots < t_n \in \mathbb{R}_{\geq 0}$
- We want $M(t)$ to remember the $\#$ of times the event has occurred
- But we also want it to have an element of “forgetfulness”
- Recent event-occurrences remembered better than older occurrences
- We want $M(\cdot)$ to give us a time-decayed count of event-occurrences

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

- There's an uptick of 1 each time the event occurs, but it decays by a factor of $\theta^{\Delta t}$ over any interval Δt where the event doesn't occur
- Thus, $M(\cdot)$ captures the notion of frequency as well as recency

Memory Function with $\theta = 0.8$



Eligibility Traces and Tabular TD(λ) Prediction

- Assume a finite state space with non-terminals $\mathcal{N} = \{s_1, s_2, \dots, s_m\}$
- Eligibility Trace for each state $s \in S$ is defined as the Memory function $M(\cdot)$ with $\theta = \gamma \cdot \lambda$, and the event timings are the time steps at which the state s occurs in a trace experience
- Eligibility trace for a given trace experience at time t is a function

$$E_t : \mathcal{N} \rightarrow \mathbb{R}_{\geq 0}$$

$$E_0(s) = 0, \text{ for all } s \in \mathcal{N}$$

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

- Tabular TD(λ) Prediction algorithm performs following update at each time step t in each trace experience:

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

“Equivalence” of TD(λ) and λ -Return

- TD(λ) is an online algorithm, similar to TD
- But unlike TD, we update the VF *for all* states at each time step
- VF update for each state is proportional to TD-Error δ_t (like TD)
- But here, δ_t is scaled by $E_t(s)$ for each state s at each t

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

- But how is TD(λ) Prediction linked to the λ -Return Prediction?
- It turns out that if we made all updates in an offline manner, then sum of updates for a fixed state $s \in \mathcal{N}$ over entire trace experience equals (offline) update for s in the λ -Return prediction algorithm

Theorem

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

$$\begin{aligned} G_t^{(\lambda)} &= (1 - \lambda) \cdot \lambda^0 \cdot (R_{t+1} + \gamma \cdot V(S_{t+1})) \\ &\quad + (1 - \lambda) \cdot \lambda^1 \cdot (R_{t+1} + \gamma \cdot R_{t+2} + \gamma^2 \cdot V(S_{t+2})) \\ &\quad + (1 - \lambda) \cdot \lambda^2 \cdot (R_{t+1} + \gamma \cdot R_{t+2} + \gamma^2 \cdot R_{t+3} + \gamma^3 \cdot V(S_{t+2})) \\ &\quad + \dots \\ &= (\gamma\lambda)^0 \cdot (R_{t+1} + \gamma \cdot V(S_{t+1}) - \gamma\lambda \cdot V(S_{t+1})) \\ &\quad + (\gamma\lambda)^1 \cdot (R_{t+2} + \gamma \cdot V(S_{t+2}) - \gamma\lambda \cdot V(S_{t+2})) \\ &\quad + (\gamma\lambda)^2 \cdot (R_{t+3} + \gamma \cdot V(S_{t+3}) - \gamma\lambda \cdot V(S_{t+3})) \\ &\quad + \dots \end{aligned}$$

$$\begin{aligned}
 G_t^{(\lambda)} = & (\gamma\lambda)^0 \cdot (R_{t+1} + \gamma \cdot V(S_{t+1}) - \gamma\lambda \cdot V(S_{t+1})) \\
 & + (\gamma\lambda)^1 \cdot (R_{t+2} + \gamma \cdot V(S_{t+2}) - \gamma\lambda \cdot V(S_{t+2})) \\
 & + (\gamma\lambda)^2 \cdot (R_{t+3} + \gamma \cdot V(S_{t+3}) - \gamma\lambda \cdot V(S_{t+3})) \\
 & + \dots
 \end{aligned}$$

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

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

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



TD(0) and TD(1) with Offline Updates

- To be clear, TD(λ) Prediction is an online algorithm
- So *not the same* as *offline* λ -Return Prediction
- If we modified TD(λ) to be offline, they'd be equivalent
- Offline version of TD(λ) would not update VF at each step
- Accumulate changes in buffer, update VF offline with buffer contents
- If we set $\lambda = 0$, $E_t(s) = \mathbb{I}_{S_t=s}$ and so, the update reduces to:

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

- This is exactly the TD update. So, TD is often referred to as TD(0)
- If we set $\lambda = 1$ with episodic traces, sum of all VF updates for a state over a trace experience is equal to its VF update in Every-Visit MC
- Hence, Offline TD(1) is equivalent to Every-Visit MC

TD(λ) Prediction with Function Approximation

- Generalize TD(λ) to the case of function approximation
- Data-Type of eligibility traces same as func-approx parameters \mathbf{w}
- So here we denote eligibility traces at time t as simply \mathbf{E}_t
- Initialize \mathbf{E}_0 to 0 for each component in it's data type
- For each time step $t > 0$, \mathbf{E}_t is calculated recursively:

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

- VF approximation update at each time step t is as follows:

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

- Expressed more succinctly in terms of function-approx TD-Error δ_t :

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

Key Takeaways from this Chapter

- Bias-Variance tradeoff of TD versus MC
- MC learns the mean of the observed returns while TD learns something "deeper" - it implicitly estimates an MRP from given data and produces the Value Function of the implicitly-estimated MRP
- Understanding TD versus MC versus DP from the perspectives of:
 - "Bootstrapping"
 - "Experiencing"
- "Equivalence" of λ -Return Prediction and $TD(\lambda)$ Prediction
- TD is equivalent to $TD(0)$ and MC is "equivalent" to $TD(1)$