12.0: Intro

- ET = fundamental mechanism that bridges the gap btwn TD and MC. For TD, $\lambda = 0$. For MC, $\lambda = 1$.
- We had seen one way to bridge TD and MC, which is TD learning. What ET offers is significant computational advantage: only need a single trace vector, rather than a store of the last n vectors.
- ET provides a way to use MC online (↑ during the episode, rather than at the end), or in continuing problems.

The core idea:

The method maintains a short-term memory vector called the eligibility trace ($z_t \in \mathbb{R}^d$) that has the same dimension as w_t. What z_t does:

1) Tracks recent participation: when a state/action contributes to producing a value estimate, its corresponding trace value gets "bumped up".
2) Fades over time: these trace values then gradually decay/fade away.
3) Enables learning: If a TD error occurs before the trace completely fades, learning happens with intensity proportional to the trace value.

Key parameter is λ (trace-decay parameter)

Trace-decay parameter: $\lambda \in [0, 1]$.

- $\lambda = 0$: Pure TD method (only immediate predecessor gets credit.
- $\lambda = 1$: Pure MC method (all visited states get equal credit).
- $0 < \lambda < 1$: hybrid approach, where recent states get more credit.

Why this matters:

Advantages over n-step methods:

- Memory efficient: store a single trace z rather than n future steps of history.
- backward view: updates immediately when TD error occurs rather than waiting n steps.
- simpler implementation.

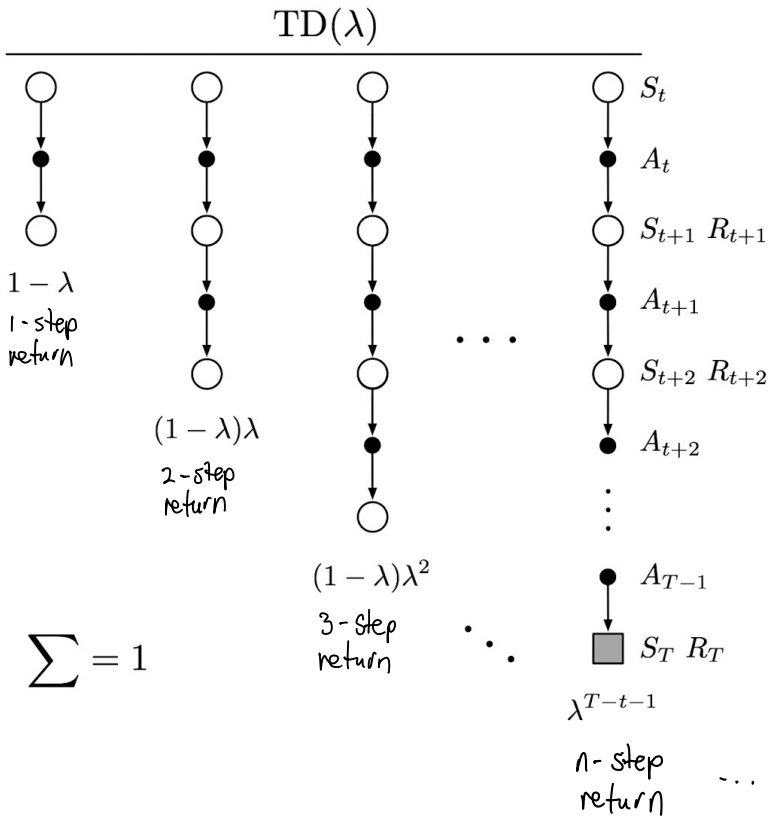
12.1: The λ-Return

The n-step return is the first n rewards + the estimated value of the state reached in n-steps, each appropriately discounted:

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \gamma^{n-1} R_{t+n} + \gamma^n \hat{V}_{t+n-1}(S_{t+n}, w_{t+n-1})$$

- A valid update can be done towards any n-step return, but also towards any average of n-step returns. For example, target could be 50% the two step return + 50% the 4 steps return: $\frac{1}{2} G_{t:t+2} + \frac{1}{2} G_{t:t+4}$,
- Any set can be averaged as long as the weights are positive and sum to 1.
- <u>compound update</u>: update that averages simpler component.

<u>TD(λ) algorithm</u>: a way of averaging n-step returns, each weighted by λ^{n-1} and normalized by $(1-\lambda)$ to ensure the weights sum to 1.



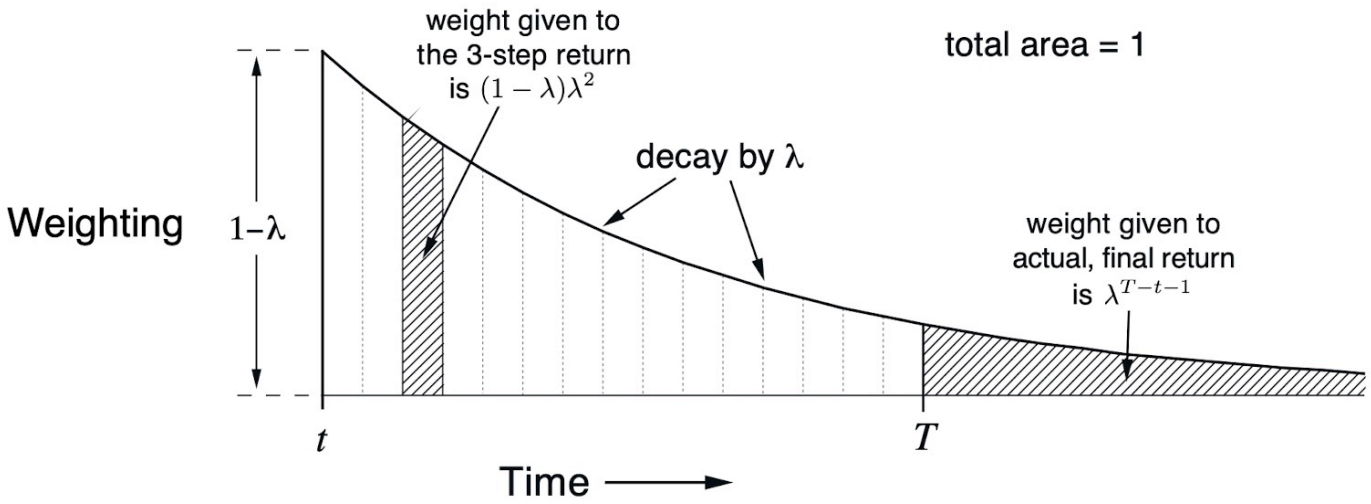
This results in what is called **lambda return**:

$$G_t^\lambda = \left[(1-\lambda) \sum_{n=1}^{T-t-1} \lambda^{n-1} G_{t:t+n} \right] + \lambda^{T-t-1} G_t$$

G_t^λ = sum of $G_{t:t+1}$, $G_{t:t+2}$, ..., to $G_{t:t+n}$, and G_t, with the more recent returns getting more weight. That weight is adjusted by λ.

- When $\lambda = 1$, the first term is 0, so the return is G_t = MC
- When $\lambda = 0$, then $G_t^\lambda = G_{t:t+1}$ = 1-step TD.

Weighting Illustration:



Offline λ-return algorithm:

- offline: waits till the end of the episode to make updates.
- semi-gradient update using the λ-return as a target:

$$w_{t+1} = w_t + \alpha\left[G_t^\lambda - \hat{v}(S_t, w_t)\right]\nabla\hat{v}(S_t, w_t), \quad t = 0, \ldots, T-1$$

- This is a theoretical, forward view learning algorithm. A way of moving smoothly btwn one-step TD and MC.

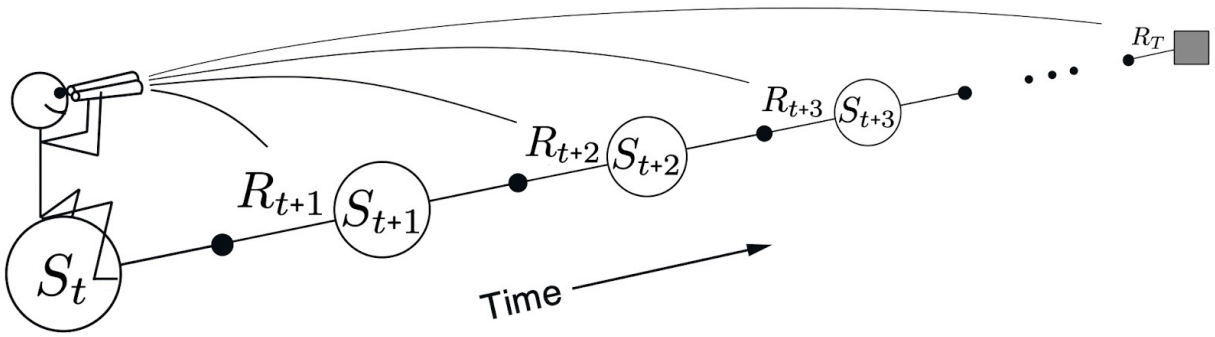
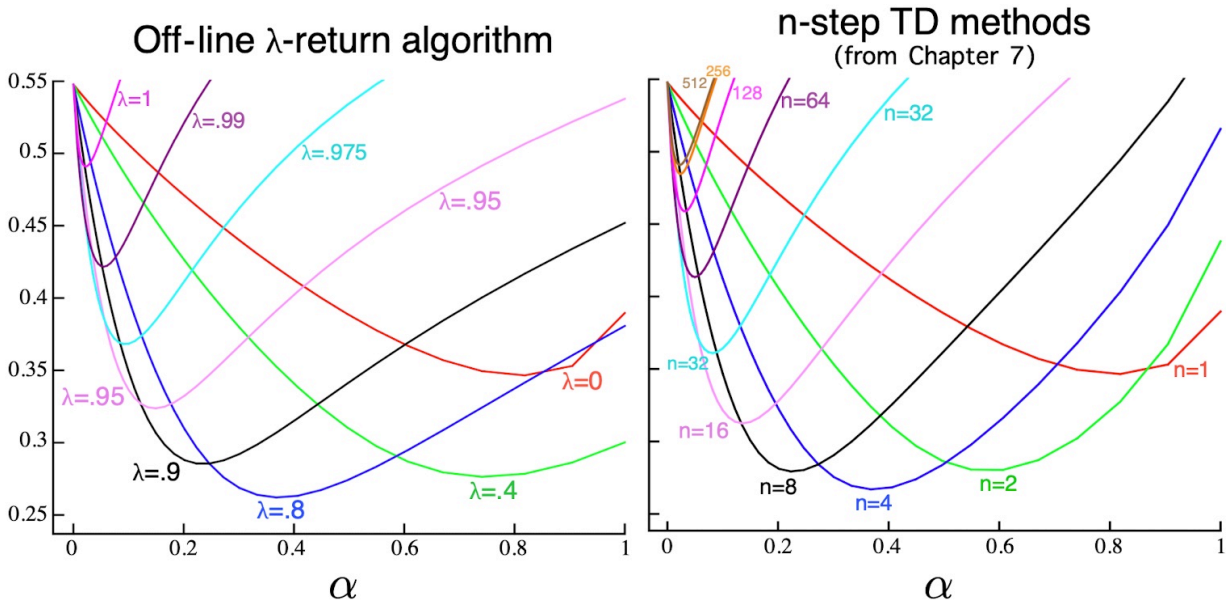


Figure 12.4: The forward view. We decide how to update each state by looking forward to future rewards and states.

Results:

RMS error at the end of the episode over the first 10 episodes



// Offline λ-return ≈ n-step TD!

12.2: TD(λ)

The 2-views problem: the λ return from 12.1 had a problem: it's a forward view: need to look into the future to compute. This means:

- You can't update as you go. Have to wait n-steps before having enough info to update the "past".
- It doesn't work for continuing problems.

TD(λ) solves this with a backward view using ET. It maintains the trace vector z so that update can be made at the same time.

TD(λ) approximates the off-line λ-return algorithm of section 12.1, improving it in 3 ways:

- update weight vector at every step and not just at the end of the episode.
- computation is now distributed across time (online), instead of done at the end (offline)
- can now use it in continuous problems.

Think of it as a "memory" that tracks.

Semi-gradient version of TD(λ) with function approximation:

- Eligibility trace z (same dimension as w), initialized to 0.

$$z_t = \gamma \lambda z_{t-1} + \nabla \hat{v}(S_t, w_t), \quad 0 \le t \le T,$$

ET vector; discount rate; trace-decay parameter; Boosted by gradient when that feature is active!

At each time step: You visit a state, and:

- z_t is decayed by $\gamma\lambda$ (fading memory)
- z_t is added by the value gradient $\nabla \hat{v}(S_t, w_t)$, if features are active.
- TD error occurs (see below).

Note: In linear approximation, the gradient is just feature vector x_t, in which case, ET vector is a sum of the past fading input vector.

The z components represent the "eligibility" of each component to learning when a reinforcing event occurs. These reinforcing events are moment-by-moment one-step TD errors. The TD error for state-value prediction is:

$$\delta_t = R_{t+1} + \gamma \hat{v}(S_{t+1}, w_t) - \hat{v}(S_t, w_t)$$

In TD(λ), the weight vector is updated on each step proportional to the scalar TD error and the vector eligibility trace:

$$w_{t+1} = w_t + \alpha \delta_t z_t$$

Algo: Prediction.

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

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$
Algorithm parameters: step size $\alpha > 0$, trace decay rate $\lambda \in [0, 1]$
Initialize value-function weights $\mathbf{w}$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  Initialize S
  $\mathbf{z} \leftarrow \mathbf{0}$ (a d-dimensional vector)
  Loop for each step of episode:
  | Choose $A \sim \pi(\cdot|S)$
  | Take action A, observe R, S'
  | $\mathbf{z} \leftarrow \gamma\lambda\mathbf{z} + \nabla\hat{v}(S,\mathbf{w})$
  | $\delta \leftarrow R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})$
  | $\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta\mathbf{z}$
  | $S \leftarrow S'$
  until S' is terminal

Why this works: looking in the algorithm box.

- $z \leftarrow \gamma\lambda z + \nabla\hat{V}(S, w)$ // update eligibility trace
- $\delta \leftarrow R + \gamma\hat{V}(S', w) - \hat{V}(S, w)$ // Compute TD error
- $w \leftarrow w + \alpha\delta z$ // update weights using both TD error and ET.

Scenarios:

$\lambda = 0$: Trace decays immediately → only current state is updated → TD(0).

$\lambda = 1$: Trace decays slowly → all visited states get credit → like MC.

$0 < \lambda < 1$: Recent states get more credit than distant ones.

Key Advantages

1) online
2) continuous problems
3) Distributed computation
4) Backward view.

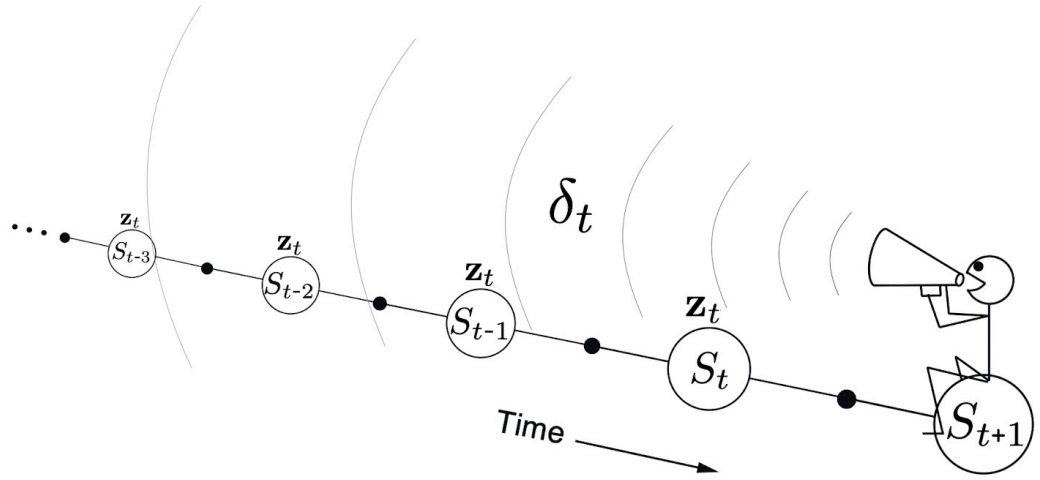


Figure 12.5: The backward or mechanistic view of TD(λ). Each update depends on the current TD error combined with the current eligibility traces of past events.

12.3: n-step truncated λ-return methods

The problem w/ the original λ-return:

1) Episodic limitation: have to wait till the end of episode to compute it,
2) Continuing problems: In non-episodic settings, there might be no "end" to wait for.
3) Dependency on an arbitrarily large # of future steps.

The key insight:

The weight given to each n-step return decays exponentially, by $\gamma\lambda$ for each additional step:

- 1-step return gets weight $(1-\lambda)$
- 2-step return " " $(1-\lambda)\lambda$
- 20-step return " " $(1-\lambda)\lambda^{19}$.

Since $0 < \lambda < 1$, these weights decrease rapidly.

How?

we can just truncate when the weights become negligible.

Introduce a horizon h, which has the same role as time of termination T.

Truncated λ-Return Formula:

$$G^{\lambda}_{t:h} \doteq \underbrace{\left((1-\lambda)\sum_{n=1}^{h-t-1}\lambda^{n-1}G_{t:t+n}\right)}_{\text{weighted sum:}} + \underbrace{\lambda^{h-t-1}G_{t:h}}_{\text{remainder sum.}}, \quad 0 \le t \le h \le T$$

Update Rule:

$$w_{t+n} = w_{t+n-1} + \alpha\left[G^{\lambda}_{t:t+n} - \hat{v}(S_t, w_{t+n-1})\right]\nabla\hat{v}(S_t, w_{t+n-1}), \quad 0 \le t < T.$$

The truncated λ-return gives rise to a family to n-step λ-return algorithms, similar to n-step TD methods.

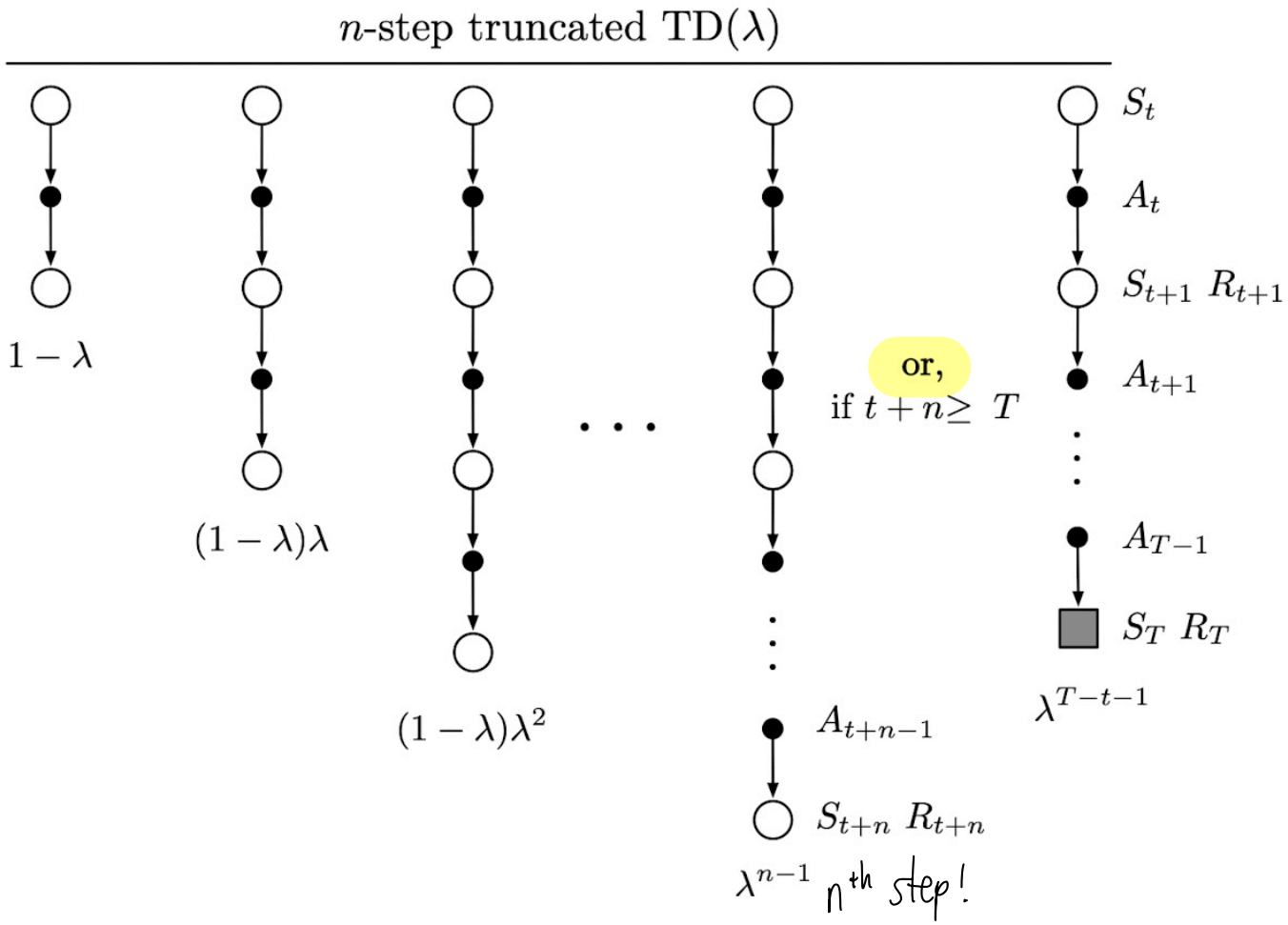


Figure 12.7: The backup diagram for Truncated TD(λ).

This can be implemented to start at $n-1$. Efficient implementation relies on the fact that k-step λ-return can be written as the sum of TD errors:

$$G^{\lambda}_{t:t+k} = \hat{v}(S_t, w_{t-1}) + \sum_{i=t}^{t+k-1} (\gamma\lambda)^{i-t}\delta'_i, \quad \text{where}$$

$$\delta'_t \doteq R_{t+1} + \gamma\hat{v}(S_{t+1}, w_t) - \hat{v}(S_t, w_{t-1})$$

12.4: Redoing Updates: The online λ-return algorithm

The big insight: As you gather more data, you can compute better λ-returns for past states. So can go back and redo all your updates w/ this improved information.

The trade off is larger horizon gives more accurate λ-returns but longer delays before update behavior. This algo tries to get both (quick update and accurate update) at the cost of more computation.

The procedure: At each time step, go back to the beginning of the episode and redo ALL updates using the current horizon of data.

Convention:

w_t^h = weight at time t in the sequence up to horizon h.

- w_0^h = first weight vector in each sequence that is inherited from the previous sequence.
- w_h^h = the last weight vector in the sequence. It defines the ultimate weight-vector sequence of the algorithm.

General form:

$$w_{t+1}^h = w_t^h + \alpha\left[G_{t:h}^{\lambda} - \hat{v}(S_t, w_t^h)\right]\nabla\hat{v}(S_t, w_t^h), \quad 0 \le t \le h \le T.$$

This update, along $w_t = w_t^t$ defines the online λ-return algorithm.

This means: update the weight for time t using the best λ-return estimate available (from current state to horizon h).

Example:

$$h = 1: \quad \mathbf{w}_1^1 \doteq \mathbf{w}_0^1 + \alpha \left[G_{0:1}^{\lambda} - \hat{v}(S_0, \mathbf{w}_0^1) \right] \nabla \hat{v}(S_0, \mathbf{w}_0^1),$$

$$\begin{aligned} h = 2: \quad \mathbf{w}_1^2 &\doteq \mathbf{w}_0^2 + \alpha \left[G_{0:2}^{\lambda} - \hat{v}(S_0, \mathbf{w}_0^2) \right] \nabla \hat{v}(S_0, \mathbf{w}_0^2), \\ \mathbf{w}_2^2 &\doteq \mathbf{w}_1^2 + \alpha \left[G_{1:2}^{\lambda} - \hat{v}(S_1, \mathbf{w}_1^2) \right] \nabla \hat{v}(S_1, \mathbf{w}_1^2), \end{aligned}$$

$$\begin{aligned} h = 3: \quad \mathbf{w}_1^3 &\doteq \mathbf{w}_0^3 + \alpha \left[G_{0:3}^{\lambda} - \hat{v}(S_0, \mathbf{w}_0^3) \right] \nabla \hat{v}(S_0, \mathbf{w}_0^3), \\ \mathbf{w}_2^3 &\doteq \mathbf{w}_1^3 + \alpha \left[G_{1:3}^{\lambda} - \hat{v}(S_1, \mathbf{w}_1^3) \right] \nabla \hat{v}(S_1, \mathbf{w}_1^3), \\ \mathbf{w}_3^3 &\doteq \mathbf{w}_2^3 + \alpha \left[G_{2:3}^{\lambda} - \hat{v}(S_2, \mathbf{w}_2^3) \right] \nabla \hat{v}(S_2, \mathbf{w}_2^3). \end{aligned}$$

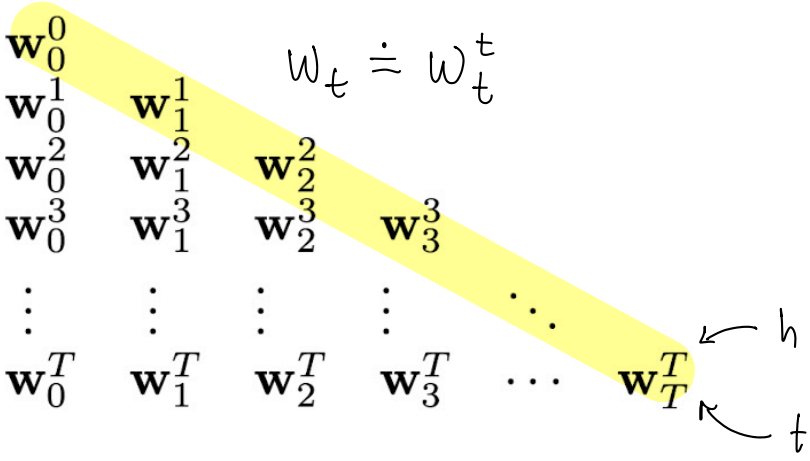
12.5: True online TD(λ)

The online λ-return algorithm from 12.4 was the gold standard, but was very computationally expensive.

True online TD(λ) achieves the exact same results but much more efficiently!

The triangular array insight:

The online λ-return algorithm produces a triangular pattern of weight vectors:



- Rows: produced by a time step
- Columns: produced by a horizon.
- Diagonal: w_t^t is what we need. These are the "final" weights at each time step: $w_t \doteq w_t^t$

Key observation: If we can compute each diagonal element directly from the previous one, we avoid recomputing the entire triangle. The formula is (for linear case):

$$w_{t+1} \doteq w_t + \alpha \delta_t z_t + \alpha (w_t^\top x_t - w_{t-1}^\top x_t)(z_t - x_t)$$

$w_t \doteq w_t^t$

$\alpha \delta_t z_t$: regular TD(λ) update.

$\alpha (w_t^\top x_t - w_{t-1}^\top x_t)(z_t - x_t)$: correction term that accounts for the difference btwn what was computed and what the online λ-return would have computed.

z_t is the dutch trace (true online TD(λ)), defined by:

$$z_t \doteq \underbrace{\gamma\lambda z_{t-1}}_{\text{same as before}} + \underbrace{\left(1 - \alpha\gamma\lambda z_{t-1}^\top x_t\right) x_t}_{\text{correction factor: helps it better approximate the online } \lambda\text{-return behavior.}} \quad \text{// linear approximation}$$

This is different from the accumulating trace. Proved to produce the same sequence of weights than online λ-return algorithm. Memory requirement is the same as TD(λ). 50% more computation bc there is one more inner-product in the ET update.

Algo: true online TD(λ) for linear

True online TD(λ) for estimating $\mathbf{w}^\top \mathbf{x} \approx v_\pi$

Input: the policy π to be evaluated
Input: a feature function $\mathbf{x} : \mathcal{S}^+ \to \mathbb{R}^d$ such that $\mathbf{x}(terminal, \cdot) = \mathbf{0}$
Algorithm parameters: step size $\alpha > 0$, trace decay rate $\lambda \in [0, 1]$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  Initialize state and obtain initial feature vector $\mathbf{x}$
  $\mathbf{z} \leftarrow \mathbf{0}$ (a d-dimensional vector)
  $V_{old} \leftarrow 0$ (a temporary scalar variable)
  Loop for each step of episode:
  | Choose $A \sim \pi$
  | Take action A, observe R, $\mathbf{x}'$ (feature vector of the next state)
  | $V \leftarrow \mathbf{w}^\top \mathbf{x}$
  | $V' \leftarrow \mathbf{w}^\top \mathbf{x}'$
  | $\delta \leftarrow R + \gamma V' - V$
  | $\mathbf{z} \leftarrow \gamma\lambda \mathbf{z} + \left(1 - \alpha\gamma\lambda \mathbf{z}^\top \mathbf{x}\right) \mathbf{x}$
  | $\mathbf{w} \leftarrow \mathbf{w} + \alpha(\delta + V - V_{old})\mathbf{z} - \alpha(V - V_{old})\mathbf{x}$
  | $V_{old} \leftarrow V'$
  | $\mathbf{x} \leftarrow \mathbf{x}'$
  until $\mathbf{x}' = \mathbf{0}$ (signaling arrival at a terminal state)

// uses the modified trace update and modified weight update.

12.6: * Dutch Traces in Monte Carlo Learning

12.7: Sarsa(λ)

This section shows how to extend eligibility traces from state-value learning (prediction) to action-value learning (control).

Action-value n-step Return:

Recall from ch 10 (semi-gradient n-step SARSA) that to learn action values we need to use the action-value form of the n-step return:

$$G_{t:t+n} \doteq R_{t+1} + \dots + \gamma^{n-1} R_{t+n} + \gamma^n \underbrace{\hat{q}(S_{t+n}, A_{t+n}, w_{t+n-1})}_{\text{uses } \hat{q} \text{ instead of } \hat{v}}, \quad t+n < T$$

OR

$$G_{t:t+n} \doteq G_t \quad \text{if} \quad t+n \geq T.$$

Action-value λ-return (not n-step return) update rule:

The action-value form of the offline λ-return algorithm uses $\hat{q}$ rather than $\hat{v}$, and G_t^λ instead of $G_{t:t+n}$:

$$w_{t+1} = w_t + \alpha \left[G_t^\lambda - \hat{q}(S_t, A_t, w_t) \right] \nabla \hat{q}(S_t, A_t, w_t)$$

$$G_t^\lambda \doteq G_{t:\infty}^\lambda \quad \text{// have to wait till episode is over.}$$

This is the forward view. It is theoretical but impratical bc you have to wait till the full return (aka till the episode is over).

The back-up diagram is similar to TD(λ):

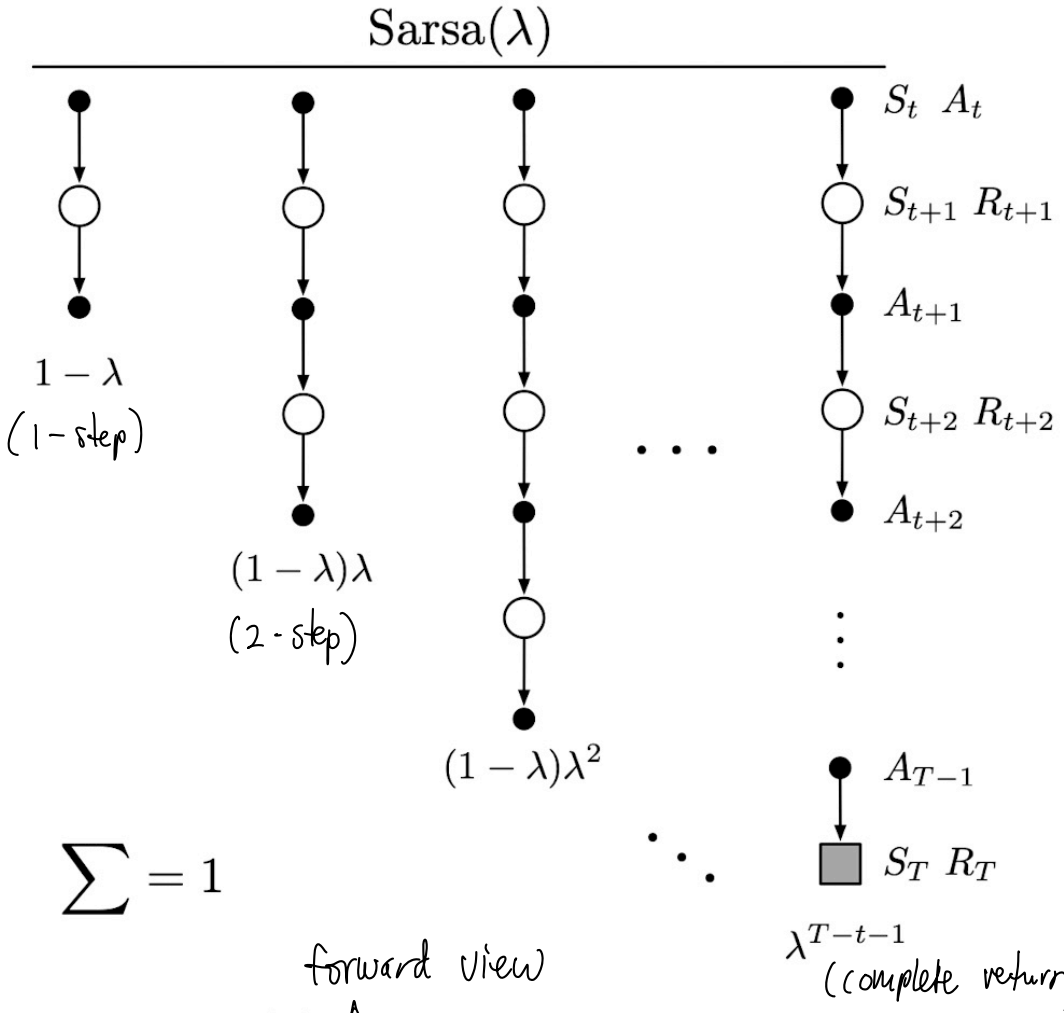


Figure 12.9: Sarsa(λ)'s backup diagram. Compare with Figure 12.1.

Backward View: SARSA(λ) with Eligibility Traces

A practical implementation using ET:

Action-value ET:

$Z_{-1} \doteq 0$, // initialize at 0

$Z_t \doteq \underbrace{\gamma\lambda Z_{t-1}}_{\text{decay for past state-action pairs}} + \underbrace{\nabla \hat{q}(S_t, A_t, w_t)}_{\text{current state-action pair is strengthened}}$ // vector keeps tracks of which weights w were involved in recent predictions.

Action-value TD error : // a measure of "surprise"

$$\delta_t \doteq R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, w_t) - \hat{q}(S_t, A_t, w_t)$$

Weight update :

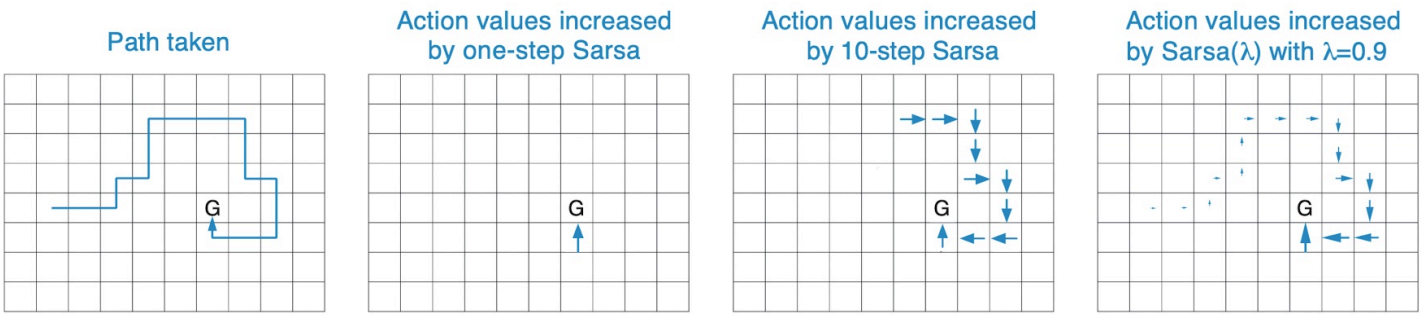
$$w_{t+1} \doteq w_t + \alpha \delta_t z_t$$ // when there is an error, update all recent weights

Key insight: the ET gets boosted by the gradient of the current state-action pair's value function.

Sarsa(λ) with binary features and linear function approximation for estimating $\mathbf{w}^\top \mathbf{x} \approx q_\pi$ or q_*

Input: a function $\mathcal{F}(s,a)$ returning the set of (indices of) active features for s, a
Input: a policy π (if estimating q_π)
Algorithm parameters: step size $\alpha > 0$, trace decay rate $\lambda \in [0,1]$
Initialize: $\mathbf{w} = (w_1, \ldots, w_d)^\top \in \mathbb{R}^d$ (e.g., $\mathbf{w} = \mathbf{0}$), $\mathbf{z} = (z_1, \ldots, z_d)^\top \in \mathbb{R}^d$

Loop for each episode:
  Initialize S
  Choose $A \sim \pi(\cdot|S)$ or ε-greedy according to $\hat{q}(S, \cdot, \mathbf{w})$
  $\mathbf{z} \leftarrow \mathbf{0}$
  Loop for each step of episode:
    Take action A, observe R, S'
    $\delta \leftarrow R$
    Loop for i in $\mathcal{F}(S, A)$:
      $\delta \leftarrow \delta - w_i$
      $z_i \leftarrow z_i + 1$ (accumulating traces)
      or $z_i \leftarrow 1$ (replacing traces)
    If S' is terminal then:
      $\mathbf{w} \leftarrow \mathbf{w} + \alpha \delta \mathbf{z}$
      Go to next episode
    Choose $A' \sim \pi(\cdot|S')$ or near greedily $\sim \hat{q}(S', \cdot, \mathbf{w})$
    Loop for i in $\mathcal{F}(S', A')$: $\delta \leftarrow \delta + \gamma w_i$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha \delta \mathbf{z}$
    $\mathbf{z} \leftarrow \gamma \lambda \mathbf{z}$
    $S \leftarrow S'; A \leftarrow A'$



1-step SARSA: only final action gets updated,

10-step SARSA: the last 10 actions get updated,

SARSA(λ): All visited action get updated, but with exponentially fading intensity.

True online Sarsa(λ) for estimating $\mathbf{w}^\top \mathbf{x} \approx q_\pi$ or q_*

Input: a feature function $\mathbf{x} : \mathcal{S}^+ \times \mathcal{A} \to \mathbb{R}^d$ such that $\mathbf{x}(terminal, \cdot) = \mathbf{0}$
Input: a policy π (if estimating q_π)
Algorithm parameters: step size $\alpha > 0$, trace decay rate $\lambda \in [0, 1]$
Initialize: $\mathbf{w} \in \mathbb{R}^d$ (e.g., $\mathbf{w} = \mathbf{0}$)

Loop for each episode:
  Initialize S
  Choose $A \sim \pi(\cdot|S)$ or near greedily from S using $\mathbf{w}$
  $\mathbf{x} \leftarrow \mathbf{x}(S, A)$
  $\mathbf{z} \leftarrow \mathbf{0}$
  $Q_{old} \leftarrow 0$
  Loop for each step of episode:
    Take action A, observe R, S'
    Choose $A' \sim \pi(\cdot|S')$ or near greedily from S' using $\mathbf{w}$
    $\mathbf{x}' \leftarrow \mathbf{x}(S', A')$
    $Q \leftarrow \mathbf{w}^\top \mathbf{x}$
    $Q' \leftarrow \mathbf{w}^\top \mathbf{x}'$
    $\delta \leftarrow R + \gamma Q' - Q$
    $\mathbf{z} \leftarrow \gamma\lambda\mathbf{z} + \left(1 - \alpha\gamma\lambda\mathbf{z}^\top \mathbf{x}\right) \mathbf{x}$
    $\mathbf{w} \leftarrow \mathbf{w} + \alpha(\delta + Q - Q_{old})\mathbf{z} - \alpha(Q - Q_{old})\mathbf{x}$
    $Q_{old} \leftarrow Q'$
    $\mathbf{x} \leftarrow \mathbf{x}'$
    $A \leftarrow A'$
  until S' is terminal

12.8: Variable λ and γ

This section introduces a generalization: making λ and γ adaptive rather than constant. Allows for more flexible learning.

Adaptive parameters:

$$\lambda_t = \lambda(S_t, A_t)$$

$$\gamma_t = \gamma(S_t)$$

Gamma:

γ becomes a termination function that determines "how much does the future matter from this state?".

$$\begin{aligned} G_t &\doteq R_{t+1} + \gamma_{t+1} G_{t+1} \\ &= R_{t+1} + \gamma_{t+1} R_{t+2} + \gamma_{t+1}\gamma_{t+2} R_{t+3} + \dots \\ &= \sum_{k=t}^{\infty} \left(\prod_{i=t+1}^{k} \gamma_i \right) R_{k+1} \end{aligned}$$

- to ensure the sums are finite, we require that $\prod_{k=t}^{\infty} \gamma_k = 0$ with probability 1 for all t.
- terminal state just becomes a state with $\gamma = 0$, and which transitions to a start state.

What this means:

$\gamma = 1$: No discounting, future matters fully.

$\gamma = 0.9$: Future matters, but discounted.

$\gamma = 0$: Future does not matter; acts like terminal state.

Variable λ:

Generalization to variable bootstrapping gives a new state-based λ-return:

$$G_t^{\lambda s} \doteq \underbrace{R_{t+1}}_{\text{immediate reward}} + \underbrace{\gamma_{t+1}\left[(1-\lambda_{t+1})\hat{v}(S_{t+1}, w_t) + \lambda_{t+1} G_{t+1}^{\lambda s}\right]}_{\text{depends on } \lambda_{t+1} \text{ and } \gamma_{t+1}. \text{ If}}$$

$\lambda_{t+1} = 0$: Bootstrap from $\hat{v}(S_{t+1})$ - pure TD behavior.

$\lambda_{t+1} = 1$: continue w/ $G_{t+1}^{\lambda s}$ - MC behavior.

$0 < \lambda_{t+1} < 1$: in between.

Notation:

- s- superscript: bootstrap from state-values
- a- superscript: bootstrap from action-values.

Action-Value Versions:

SARSA form:

$$G_t^{\lambda a} = R_{t+1} + \gamma_{t+1}\left[(1-\lambda_{t+1})\hat{q}(S_{t+1}, A_{t+1}, w_t) + \lambda_{t+1} G_{t+1}^{\lambda a}\right]$$

Expected SARSA: (expected value under current policy)

$$G_t^{\lambda a} = R_{t+1} + \gamma_{t+1}\left[(1-\lambda_{t+1})\bar{V}_t(S_{t+1}) + \lambda_{t+1} G_{t+1}^{\lambda a}\right]$$

where V is generalized to function approximation:

$$\bar{V}(s) = \sum_a \pi(a|s)\hat{q}(s, a, w_t)$$

12.9: Off-policy Eligibility Traces with Control Variates

The goal is to develop off-policy algorithms that can use eligibility trace. Off-policy learning is tricky bc we're learning target policy π while following a different behavioral policy b. This requires importance sampling to correct for the difference btwn policies.

Key Components:

- Importance sampling ratio $\rho_t = \frac{\pi(A_t \mid S_t)}{b(A_t \mid S_t)}$ weighs how much more likely the target policy would have taken action A_t compared to the behavioral policy. This corrects for the policy mismatch.

- control variates: aims to reduce the variance of the initial estimator (which includes ρ), by subtracting a baseline that has 0 expectation, $(1-\rho_t)\hat{v}(S_t, w_t)$.

The Mathematical Development:

1) Express return with control variates:

$$G_t^{\lambda s} = \rho_t \Big\{ R_{t+1} + \gamma_{t+1} \Big[(1-\lambda_{t+1}) \hat{v}(S_{t+1}, w_t) + \lambda_{t+1} G_{t+1}^{\lambda s} \Big] \Big\} + (1-\rho_t)\hat{v}(S_t, w_t)$$

This prevents the return from being 0 when there is no chance the target policy would take the observed actions.

2) Approximation of the truncated return:

use the truncated version of the return, which can be approximated as the sum of state-based TD errors δ_t^s:

$$\delta_t^s \doteq R_{t+1} + \gamma_{t+1}\hat{v}(S_{t+1}, w_t) - \hat{v}(S_t, w_t)$$

$$G_t^{\lambda s} \approx \hat{v}(S_t, w_t) + \rho \sum_{k=t}^{\infty} \delta_k^s \prod_{i=t+1}^{k} \gamma_i \lambda_i \rho_i$$

3) write the forward view update:

we can use the truncated return in the forward view update (i.e. w/o ET):

$$w_{t+1} = w_t + \alpha \left[G_t^{\lambda s} - \hat{v}(S_t, w_t) \right] \nabla \hat{v}(S_t, w_t)$$

$$\approx w_t + \alpha \rho_t \left(\sum_{k=t}^{\infty} \delta_k^t \prod_{i=t+1}^{k} \gamma_i \lambda_i \rho_i \right) \nabla \hat{v}(S_t, w_t)$$

4) ET update:

$$\sum_{t=0}^{\infty} (w_{t+1} - w_t) \approx \sum_{t=1}^{\infty} \sum_{k=t}^{\infty} \alpha \rho_t \delta_k^s \hat{\nabla}(S_t, w_t) \prod_{i=t+1}^{k} \gamma_i \lambda_i \rho_i$$

$$= \sum_{k=1}^{\infty} \sum_{t=1}^{k} \alpha \rho_t \nabla \hat{v}(S_t, w_t) \delta_k^s \prod_{i=t+1}^{k} \gamma_i \lambda_i \rho_i \quad // \sum_{t=x}^{y} \sum_{k=t}^{y} = \sum_{k=x}^{y} \sum_{t=x}^{k}$$

$$= \sum_{k=1}^{\infty} \alpha \delta_k^s \sum_{t=1}^{k} \rho_t \nabla \hat{v}(S_t, w_t) \prod_{i=t+1}^{k} \gamma_i \lambda_i \rho_i$$

The derivation shows that summing forward-view updates over time is equivalent to backward views updates.

In the end we get the general accumulating trace for state values:

$$z_t = \rho_t \left[\gamma_t \lambda_t z_{t-1} + \nabla \hat{v}(S_t, w_t) \right].$$

This is the off-policy version of ET where:

- When $\rho_t = 1$ (on-policy): it reduces to the standard ET.
- When $\rho_t = 0$ (target policy would never take this action): the trace gets cut off.
- The trace accumulates evidence weighted by how likely the target policy is to take observed actions.

5) <u>combine everything to get the semi-gradient update</u>:

This ET can be combined with the usual semi-gradient update rule for TD(λ) to form a general TD(λ) algorithm that can be off-policy.

12.10: Watkin's Q(λ) to Tree-Backup (λ)

This section covers two important approaches for extending Q-learning to use ET.

Watkins's Q(λ):

Basic principle: decay the ET the usual way as long as a greedy action is taken. Cut the traces to 0 when a non-greedy action is taken.

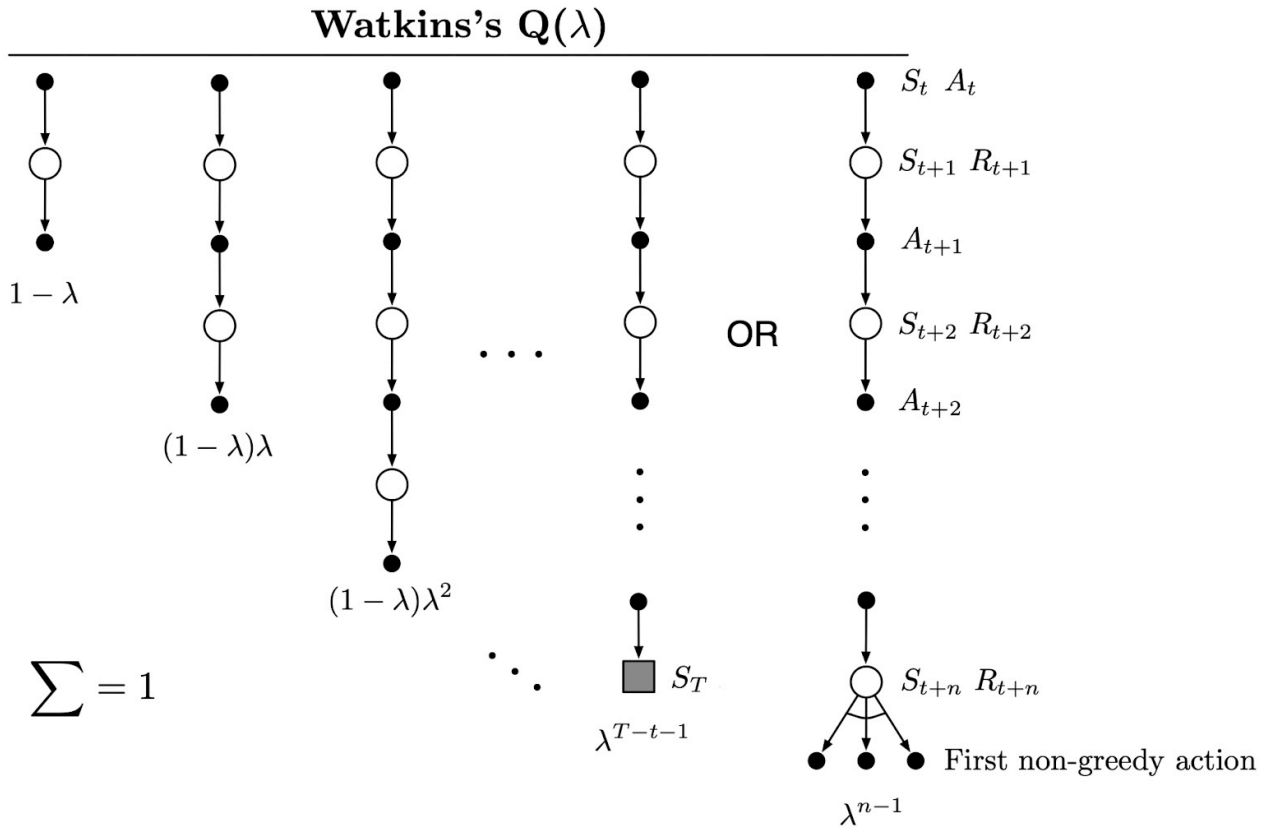


Figure 12.12: The backup diagram for Watkins's Q(λ). The series of component updates ends either with the end of the episode or with the first nongreedy action, whichever comes first.

Problem: approach is too harsh. The moment you explore (take a non-greedy action), you lose all accumulated ET information. This makes learning inefficient.

Tree-Backup (λ):

TB(λ) = true successor to Q-learning bc it does not require importance sampling.

TB(λ) considers all possible actions at each state, weighted by their target policy probabilities, instead of following a single path and cutting traces.

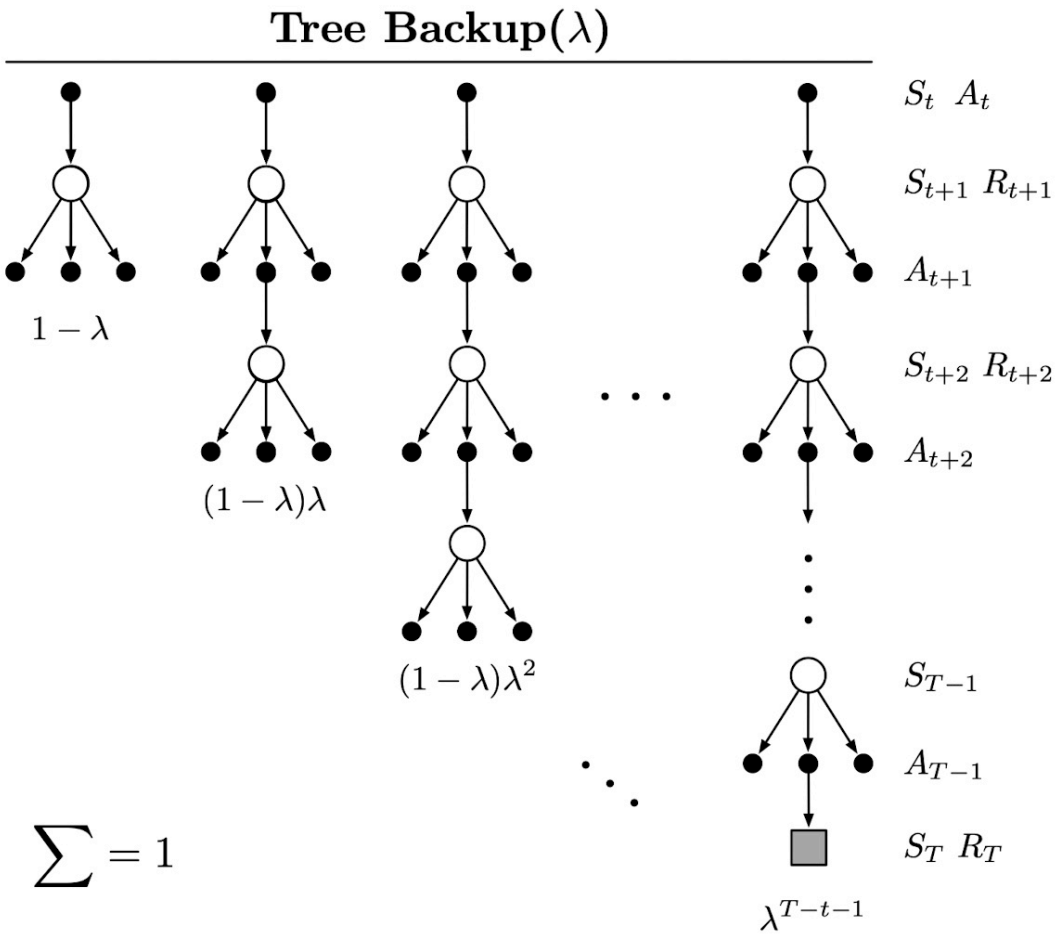


Figure 12.13: The backup diagram for the λ version of the Tree Backup algorithm.

At each state, it branches to consider all actions, but weights them by $\pi(a|s)$.

For the return we start with the recursive form of the λ-return using action-values, and then expand the bootstrapping case of the target:

$$G_t^{\lambda a} \doteq R_{t+1} + \gamma_{t+1}\Big\{(1-\lambda_{t+1})\,\bar{V}_t(S_{t+1}) + \lambda_{t+1}\Big[\underbrace{\sum_{a\neq A_{t+1}} \pi(a|S_{t+1})\,\hat{q}(S_{t+1},a,w_t)}_{\text{weighted sum of all actions } a\neq A_{t+1},\text{ where each action is weighted by } \pi(a|S_{t+1})} + \pi(A_{t+1}|S_{t+1})\,G_{t+1}^{\lambda a}\Big]\Big\}$$

(the underbrace: weighted sum of all actions $a \neq A_{t+1}$, where each action is weighted by $\pi(a|S_{t+1})$ — expected value of all the actions not taken.)

$$= R_{t+1} + \gamma_{t+1}\Big\{\bar{V}_t(S_{t+1}) + \lambda_{t+1}\,\pi(A_{t+1}|S_{t+1})\Big[G_{t+1}^{\lambda a} - \hat{q}(S_{t+1},A_{t+1},w_t)\Big]\Big\}$$

As usual this can be written as a sum of TD errors:

$$G_t^{\lambda a} \approx \hat{q}(S_t,A_t,w_t) + \sum_{k=t}^{\infty} \delta_k^a \prod_{i=t+1}^{k} \gamma_i \lambda_i \pi(A_i|S_i)$$

using the expectation form of the action-based TD error.

Trace update rule:

Trace update involving target-policy probabilities of the selected actions:

$$z_{t+1} = \gamma_t \lambda_t \pi(A_t | S_t) z_{t-1} + \nabla \hat{q}(S_t, A_t, w_t)$$

weight update rule:

We can combine this to the usual parameter update rule to get the TB(λ) algorithm:

$$w_{t+1} = w_t + \alpha \delta_t z_t$$

12.11: Stable off-policy methods with traces

This section presents 4 stable off-policy methods that successfully combines eligibility traces with off-policy learning. All are based on gradient-TD or emphatic-TD and use linear function approximation.

GTD(λ) - Gradient Temporal Differences with Traces

Purpose: learn parameter w_t s.t. $\hat{v}(s,w) = w_t^T x(s) \approx v_\pi(s)$ from off-policy data using eligibility traces. This is the ET version of TDC (TD w/ gradient correction).

update for w:

$$w_{t+1} \doteq w_t + \alpha \delta_t^s z_t - \underbrace{\alpha \gamma_{t+1}(1-\lambda_{t+1})(z_t^T v_t) x_{t+1}}_{\text{correction term helps remove bias that would otherwise occur in off-policy learning.}} \quad \text{with}$$

$$v_{t+1} \doteq v_t + \beta \delta_t^s z_t - \beta (v_t^T x_t) x_t .$$

- v is a vector with same dimensions as w.
- β is a second step-size parameter.

GQ(λ) - Gradient Q-learning with Traces

Purpose: learn action-values $\hat{q}(s, a, w)$ from off-policy data.
The ET version of gradient TD for action-values.

update:

$$w_{t+1} \doteq w_t + \alpha \delta_t^q z_t - \alpha \gamma_{t+1}(1-\lambda_{t+1})(z_t^\top v_t)\bar{x}_{t+1}$$

where $\bar{x}_t$ is the average feature vector for s_t under target policy:

$$\bar{x}_t \doteq \sum_a \pi(a|s_t)\, x(s_t, a)$$

δ_t^q is the expectation form TD error:

$$\delta_t^q = R_{t+1} + \gamma_{t+1} w_t^\top \bar{x}_{t+1} - w_t^\top x_t$$

The rest is the same as GTD including the update for v.

HTD(λ) - Hybrid Temporal Difference:

Hybrid state-value algorithm combining GTD(λ) and TD(λ).
It is a strict generalization of TD(λ) to off-policy;
meaning that it reduces to TD(λ) if $\rho_t = 1$ (on-policy).

It uses two sets of ET:

- z_t: the standard ET:

$$z_t \doteq \rho_t (\gamma_t \lambda_t z_{t-1} + x_t) \quad \text{with} \quad z_{-1} \doteq 0$$

- z_b: "Behavioral policy" traces (does not include ρ_t).

$$z_b \doteq \gamma_t \lambda_t z^b_{t-1} + x_t \quad \text{with} \quad z^b_{-1} \doteq 0.$$

update:

$$w_{t+1} \doteq w_t + \alpha \delta^s_t z_t + \alpha \left[(z_t - z^b_t)^T v_t \right] (x_t - \gamma_{t+1} x_{t+1})$$

$$v_{t+1} \doteq v_t + \beta \delta^s_t z_t + \beta \left(z^{bT}_t v_t \right) (x_t - \gamma_{t+1} x_{t+1}) \quad \text{w/} \quad v_0 \doteq 0.$$

Why hybrid: It combines aspects of both TD(λ) and gradient-based methods. If $\rho_t = 1$, $z^b_t \to z_t$, and you get TD(λ).

Emphatic TD(λ):

Purpose: extends emphatic TD to use ET while maintaining convergence guarantees.

High variance + potentially slow convergence.

Key components:

- M_t: emphasis. Determines how much to weight each update.
- F_t: followon trace. Tracks the discounted "followon" from past states.
- I_t: Interest. How much we care about accuracy at each state.

Math:

$$w_{t+1} \doteq w_t + \alpha \delta_t z_t$$

$$\delta_t \doteq R_{t+1} + \gamma_{t+1} w_t^T x_{t+1} - w_t^T x_t$$

$$z_t \doteq \rho_t (\gamma_t \lambda_t z_{t-1} + M_t x_t), \quad \text{w/} \quad z_{-1} \doteq 0$$

$$M_t \doteq \lambda_t I_t + (1 - \lambda_t) F_t$$

$$F_t \doteq \rho_{t-1} \gamma_t F_{t-1} + I_t, \quad \text{w/} \quad F_0 \doteq i(S_0).$$

12.13 : Conclusion

- ET w/ TD errors provide an efficient incremental way of shifting btwn TD and MC methods.
- ET are more general than n-step methods.
- MC may have advantages in non-Markov bc they do not bootstrap, use ET on TD methods make them closer to MC and thus more suited to non-Markov cases.
- Beware: most of the time we are in a deadly triad scenario, and do not have convergence guarantee!