Lecture 11: Eligibility Traces

Oliver Wallscheid
Recall \( n \)-step bootstrapping updates:

\[
\begin{align*}
g_{k:k+n} &= r_{k+1} + \gamma r_{k+2} + \cdots + \gamma^{n-1} r_{k+n} + \gamma^n \hat{v}_{k+n-1}(x_{k+n}), \\
g_{k:k+n} &= r_{k+1} + \gamma r_{k+2} + \cdots + \gamma^{n-1} r_{k+n} + \gamma^n \hat{q}_{k+n-1}(x_{k+n}, u_{k+n}).
\end{align*}
\]

Motivation: retrieve bootstrapped estimates between one-step updates and Monte Carlo.

- Use \( n \) as degree of freedom to find the learning optimum.
- However, there are two significant drawbacks of \( n \)-step bootstrapping:
  - Delay: we are looking \( n \)-steps into the future and, therefore, have to wait \( n \)-steps before we can perform the update.
  - Memory: we have to store \( n \) transitions until we can process them.

Goal of today's lecture

- Find an algorithm with the same flexibility as \( n \)-step bootstrapping.
- Avoid the \( n \)-step disadvantages (delay, memory demand).
Table of Contents

1. $\lambda$-Returns
2. TD($\lambda$)
3. Online $\lambda$-Return Updates
4. Sarsa($\lambda$)
General Averaging of $n$-Step Returns

- Averaging different $n$-step returns is possible without introducing a bias (if sum of weights is one).
- Example on the left:

$$g = \frac{1}{3}g_{k:k+1} + \frac{2}{3}g_{k:k+3}$$

- Horizontal line in backup diagram indicates the averaging.
- Enables additional degree of freedom to reduce prediction error.
- Such updates are called **compound updates**.

Fig. 11.1: Exemplary averaging of $n$-step returns
λ-Return (1)

Fig. 11.2: Backup diagram for λ-returns

- **λ-return**: is a compound update with exponentially decaying weights:

\[ g^\lambda_k = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{(n-1)} g_{k:k+n} \cdot \]

(11.1)

- Parameter is \( \lambda \in \{ \mathbb{R} | 0 \leq \lambda \leq 1 \} \).
- Geometric series of weights is one:

\[ (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{(n-1)} = 1 \]
Rewrite $\lambda$-return for episodic tasks with termination at $k = T$:

$$g_k^{\lambda} = (1 - \lambda) \sum_{n=1}^{T-k-1} \lambda^{(n-1)} g_{k+n} + \lambda^{T-k-1} g_k. \quad (11.2)$$

Return $g_k$ after termination is weighted with residual weight $\lambda^{T-k-1}$.

Above, (11.2) includes two special cases:
- If $\lambda = 0$: becomes TD(0) update.
- If $\lambda = 1$: becomes MC update.

Fig. 11.3: Weighting overview in $\lambda$-return series (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)
### Offline $\lambda$-Return Semi-Gradient Algorithm

- Applying semi-gradient updates with fct. approximation we receive:

$$w_{k+1} = w_k + \alpha \left[ g_k^\lambda - \hat{v}(x_k, w_k) \right] \nabla_w \hat{v}(x_k, w_k). \quad (11.3)$$

- Offline refers to that $w$ is not changed until the episode's end.

---

**Fig. 11.4:** Prediction accuracy comparison based on 19-state random walk example from Fig. 6.3 (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)
Truncated $\lambda$-Returns

- Using $\lambda$-returns as in (11.1) is not feasible for continuing tasks.
- One would have to wait infinitely long to receive the trajectory.
- Intuitive approximation: truncate $\lambda$-return after $h$ steps

$$g_{k:h}^\lambda = (1 - \lambda) \sum_{n=1}^{h-k-1} \lambda^{n-1} g_{k:k+n} + \lambda^{h-k-1} g_{k:h}.$$  \hspace{1cm} (11.4)

- Horizon $h$ divides continuing tasks in rolling episodes.
- The truncated $\lambda$-return (11.4) can be used analogously to $n$-step returns in semi-gradient TD updates (cf. Algo. 9.3).
Both, $n$-step and $\lambda$-return updates, are based on a forward view.

We have to wait for future states and rewards to arrive before we are able to perform an update.

Currently, $\lambda$-returns are only an alternative to $n$-step updates with different weighting without a particular advantage.

Fig. 11.5: The forward view: an update of the current state value is evaluated by future transitions (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)
Table of Contents

1. λ-Returns
2. TD(λ)
3. Online λ-Return Updates
4. Sarsa(λ)
Backward View of TD($\lambda$)

General idea:

- Use $\lambda$-weighted returns looking into the past.
- Implement this in a recursive fashion to save memory.
- Therefore, an eligibility trace $z_k$ denoting the importance of past events to the current state update is introduced.

$$\delta_t = R_{t+1} + \hat{v}(S_{t+1}, \omega_t) - \hat{v}(S_t, \omega_t)$$ (12.6)

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

$$\omega_{t+1} = \omega_t + \gamma \delta_t z_t.$$ (12.7)

Fig. 11.6: The backward view: an update of the current state value is evaluated based on a trace of past transitions (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)
Why is the exponential weighting particular suitable for TD($\lambda$)?
▶ Because we can easily implement this in a recursive fashion.

**Exponential smoothing filter**

Let $x_k$ be an arbitrary signal sampled at equally distributed time steps $k$. Then, the filtered signal $y_k$ with an exponential window function of past observations is

$$y_k = \beta x_k + (1 - \beta)y_{k-1}, \quad k > 0 \quad \text{and} \quad y_0 = x_0$$

(11.5)

with $\beta$ being the smoothing factor.

▶ Above is a simple recursive exponential smoothing filter which converges for $k >> 1$. 
Eligibility Trace (2)

With TD and function approximation

- the eligibility trace \( z_k \in \mathbb{R}^\zeta \) is a vector with same dimensions as \( w \):
  \[
  z_0 = 0,
  z_k = \gamma \lambda z_{k-1} + \nabla_w \hat{v}(x_k, w_k).
  \]
  \[\tag{11.6}\]

- It tracks which components of \( w \) have contributed to recent state valuations. Here, \( \lambda \) is also called the trace-decay parameter.

Further remarks:

- \( z_k \) can be interpreted a short-term memory while \( w_k \) is the long-term memory within the learning process of \( \hat{v} \).
- Comparing (11.6) with the filter (11.5) it becomes obvious that \( z \) is not an idealized exp. filtered form of \( \nabla_w \hat{v} \).
- Consider case \( k \to \infty \) and \( \nabla_w \hat{v}(x_k, w_k) = \nabla_w \hat{v} = \text{const.} \), then \( z \) is only a bias-free trace of \( \nabla_w \hat{v} \) in the TD sense if \( \lambda = 1 \):
  \[
  z = \frac{\nabla_w \hat{v}}{1 - \gamma \lambda}.
  \]
Consider linear function approximation with a single state being the feature vector 
\[ \hat{v}(x_k, w) = x_k w \rightarrow \nabla_w \hat{v}(x_k, w) = x_k. \]

For illustration purpose we assume that \( w \) is constant.

Below is an example for the eligibility trace with \( \lambda = 0.9, \gamma = 1 \).

Fig. 11.7: Illustration of gradient \( \nabla_w \hat{v} \) and corresponding trace \( z \) based on (11.6) for single state example with linear function approximation.
Together with (11.6) the semi-gradient TD(\(\lambda\)) update is:

\[ \delta_k = r_{k+1} + \gamma \hat{v}(x_{k+1}, w_k) - \hat{v}(x_k, w_k), \]

\[ w_{k+1} = w_k + \alpha \delta_k z_k. \]  

\[(11.7)\]

**input:** a policy \(\pi\) to be evaluated  
**input:** a differentiable function \(\hat{v} : \mathbb{R}^\kappa \times \mathbb{R}^\zeta \to \mathbb{R}\) with \(\hat{v}(x_T, \cdot) = 0\)  
**parameter:** step size \(\alpha \in \{\mathbb{R}\mid 0 < \alpha < 1\}\), decay rate \(\lambda \in \{\mathbb{R}\mid 0 \leq \lambda \leq 1\}\)  
**init:** value-function weights \(w \in \mathbb{R}^\zeta\) arbitrarily  
**for** \(j = 1, 2, \ldots \) **episodes** **do**  
  **initialize** \(x_0\) and set \(z = 0\);  
  **for** \(k = 0, 1, 2 \ldots \) **time steps** **do**  
    \(u_k \leftarrow \) apply action from \(\pi(x_k)\);  
    observe \(x_{k+1}\) and \(r_{k+1}\);  
    \(z \leftarrow \gamma \lambda z + \nabla_w \hat{v}(x_k, w)\);  
    \(\delta \leftarrow r_{k+1} + \gamma \hat{v}(x_{k+1}, w) - \hat{v}(x_k, w)\);  
    \(w \leftarrow w + \alpha \delta z\);  
    exit loop if \(x_{k+1}\) is terminal;  

**Algo. 11.1:** Semi-gradient TD(\(\lambda\)) (output: parameter vector \(w\) for \(\hat{v}_\pi\))
Exemplary Comparison

- TD($\lambda$) is not an exact representation of offline $\lambda$-returns (see below).
- For $\lambda = 0$: matches exactly semi-gradient one-step TD (’TD(0)’).
- For $\lambda = 1$: mimics long-term Monte Carlo updates.

Fig. 11.8: Prediction accuracy comparison based on 19-state random walk example from Fig. 6.3 (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)
Table of Contents

1 λ-Returns
2 TD(λ)
3 Online λ-Return Updates
4 Sarsa(λ)
Truncated $\lambda$-Returns for Online Updates

- Recall the truncated $\lambda$-return after $h$ steps

\[
g_{k:h}^\lambda = (1 - \lambda) \sum_{n=1}^{h-k-1} \lambda^{n-1} g_{k:k+n} + \lambda^{h-k-1} g_{k:h}.
\]

- If $h < T$ then we can perform online updates since $w$ is changed within an episode.

- Implementation is very similar to $n$-step bootstrapping, also forward view (e.g., still delayed with increased memory demand):

\[
\delta'_k = r_{k+1} + \gamma \hat{v}(x_{k+1}, w_k) - \hat{v}(x_k, w_k),
\]

\[
g_{k:k+n}^\lambda = \hat{v}(x_k, w_{k-1}) + \sum_{i=k}^{k+n-1} (\gamma \lambda)^{i-k} \delta'_i,
\]

\[
w_{k+n} = w_{k+n-1} + \alpha \left[ g_{k:k+n}^\lambda - \hat{v}(x_k, w_{k+n-1}) \right] \nabla_w \hat{v}(x_k, w_{k+n-1}).
\]
Increase Approximation Quality by Redoing Updates (1)

General trade-off regarding the truncated $\lambda$-returns (forward view):

- If $n$ is small: delay and memory demand are low.
- If $n$ is high: approximation of offline $\lambda$-returns is more accurate.

Idea to solve this compromise: redoing updates.

- If new data is available, we go back and redo all previous updates.
- Reuse data samples $\mathcal{D}_k \sim \langle x_k, r_k, x_{k+1} \rangle$.
- Update parameter vector $w_k^h$ at $k$-th time step up to horizon $h$.

Fig. 11.9: Simplified flowchart for redoing updates at time step $k$ and horizon $h$
Increase Approximation Quality by Redoing Updates (2)

The update sequence from example Fig. 11.9 using semi-gradients:

\[ h = 1 : \quad w_1^1 = w_0^1 + \alpha \left[ g_{0:1}^\lambda - \hat{v}(x_0, w_0^1) \right] \nabla_w \hat{v}(x_0, w_0^1) \]

\[ h = 2 : \quad w_1^2 = w_0^2 + \alpha \left[ g_{0:2}^\lambda - \hat{v}(x_0, w_0^2) \right] \nabla_w \hat{v}(x_0, w_0^2) \]
\[ w_2^2 = w_1^2 + \alpha \left[ g_{1:2}^\lambda - \hat{v}(x_1, w_1^2) \right] \nabla_w \hat{v}(x_1, w_1^2) \]

\[ h = 3 : \quad w_1^3 = w_0^3 + \alpha \left[ g_{0:3}^\lambda - \hat{v}(x_0, w_0^3) \right] \nabla_w \hat{v}(x_0, w_0^3) \]
\[ w_2^3 = w_1^3 + \alpha \left[ g_{1:3}^\lambda - \hat{v}(x_1, w_1^3) \right] \nabla_w \hat{v}(x_1, w_1^3) \]
\[ w_3^3 = w_2^3 + \alpha \left[ g_{2:3}^\lambda - \hat{v}(x_2, w_2^3) \right] \nabla_w \hat{v}(x_2, w_2^3) \]
Generalization of the example:

**Online \( \lambda \)-return**

Having samples \( \mathcal{D}_k \sim \langle x_k, r_k, x_{k+1} \rangle \) up to horizon \( h \) available, the general online \( \lambda \)-return update is

\[
\mathbf{w}_{k+1}^h = \mathbf{w}_k^h + \alpha \left[ g_{k:h}^{\lambda} - \hat{v}(x_k, \mathbf{w}_k^h) \right] \nabla \hat{v}(x_k, \mathbf{w}_k^h), \quad 0 \leq k \leq h \leq T
\]

(11.8)

with the final parameter vector \( \mathbf{w}_k = \mathbf{w}_k^T \) at the given time step.

- Online approach: a new \( \mathbf{w}_k \) is calculated at each step using only information available at step \( k \).
- Obviously, the approach is very computationally demanding.
  - Computations increase with every time step.
  - Is likely to become infeasible for long episodes.
Comparison Online Vs. Offline $\lambda$-Returns

Through the repeated updates, the online $\lambda$-return algorithm can even increase the prediction quality compared to its offline variant.

![Comparison Graph](image)

Fig. 11.10: Prediction accuracy comparison based on 19-state random walk example from Fig. 6.3 (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)
True Online TD(\(\lambda\)) for Linear Approximation

If linear function approximation \(\hat{v} = \tilde{x}^T w\) is applied the

- online \(\lambda\)-return updates (11.8) and
- the elegant implementation using eligibility traces (11.6)

can be nicely combined. By doing so we receive

- backward-viewing linear algorithm which
- is denoted as 'true' online TD(\(\lambda\)) because it is not an approximation
  of the offline \(\lambda\)-returns as the regular TD(\(\lambda\)) from Algo. 11.1.

The true online TD(\(\lambda\)) update equations are\(^1\):

\[
\begin{align*}
    z_k &= \gamma \lambda z_{k-1} + (1 - \alpha \gamma \lambda \tilde{x}_k^T z_{k-1}) \tilde{x}_k, \\
    \delta_k &= r_{k+1} + \gamma \tilde{x}_{k+1}^T w_k - \tilde{x}_k^T w_k, \\
    w_{k+1} &= w_k + \alpha \delta_k z_k + \alpha \left( \tilde{x}_k^T w_k - \tilde{x}_k^T w_{k-1} \right) (z_k - \tilde{x}_k).
\end{align*}
\]

Algorithmic Implementation: True Online TD(\(\lambda\))

\begin{itemize}
\item \textbf{input:} a policy \(\pi\) to be evaluated
\item \textbf{input:} a feature representation \(\tilde{x}\) with \(\tilde{x}_T = 0\) (i.e., \(\hat{v}(\tilde{x}_T, \cdot) = 0\))
\item \textbf{parameter:} step size \(\alpha \in \{\mathbb{R} | 0 < \alpha < 1\}\), decay rate \(\lambda \in \{\mathbb{R} | 0 \leq \lambda \leq 1\}\)
\item \textbf{init:} value-function weights \(\mathbf{w} \in \mathbb{R}^{\zeta}\) arbitrarily
\end{itemize}

\begin{algorithm}
\For {\(j = 1, 2, \ldots\) episodes} {
\For {\(k = 0, 1, 2 \ldots\) time steps} {
\begin{align*}
\mathbf{u}_k &\leftarrow \text{apply action from } \pi(\mathbf{x}_k); \\
\mathbf{x}_{k+1} &\text{ and } r_{k+1}; \\
\hat{v} &\leftarrow \tilde{x}_k^T \mathbf{w}; \\
\hat{v}' &\leftarrow \tilde{x}_{k+1}^T \mathbf{w}; \\
\delta &\leftarrow r_{k+1} + \gamma \hat{v}' - \hat{v}; \\
z &\leftarrow \gamma \lambda z + (1 - \alpha \gamma \lambda \tilde{x}_k^T z) \tilde{x}_k; \\
\mathbf{w} &\leftarrow \mathbf{w} + \alpha (\delta + \hat{v} - \hat{v}_{old}) z - \alpha (\hat{v} - \hat{v}_{old}) \tilde{x}_k; \\
\hat{v}_{old} &\leftarrow \hat{v}' ;
\end{align*}
exit loop if \(\mathbf{x}_{k+1}\) is terminal;
}
}
\end{algorithm}

\textbf{Algo. 11.2:} True Online TD(\(\lambda\)) (output: parameter vector \(\mathbf{w}\) for \(\hat{v}_\pi\))
Transfer for Action Values

First, transfer truncated $\lambda$-returns to action values (forward view):

$$g_{k:h}^\lambda = (1 - \lambda) \sum_{n=1}^{h-k-1} \lambda^{(n-1)} g_{k:k+n} + \lambda^{h-k-1} g_{k:h},$$

$$g_{k:k+n} = r_{k+1} + \gamma r_{k+2} + \cdots + \gamma^{n-1} r_{k+n} + \gamma^n \hat{q}(x_{k+n}, u_{k+n}, w_{k+n-1}).$$

(11.10)

From that, the forward view offline $\lambda$-return update for $\hat{q}$ is:

$$w_{k+1} = w_k + \alpha \left[ g_k^\lambda - \hat{q}(x_k, u_k, w_k) \right] \nabla_w \hat{q}(x_k, u_k, w_k).$$

(11.11)

The backward view approximation known as Sarsa($\lambda$) is then:

$$\delta_k = r_{k+1} + \gamma \hat{q}(x_{k+1}, u_{k+1}, w_k) - \hat{q}(x_k, u_k, w_k),$$

$$z_k = \gamma \lambda z_{k-1} + \nabla_w \hat{q}(x_k, u_k, w_k), \quad z_0 = 0,$$

$$w_{k+1} = w_k + \alpha \delta_k z_k.$$  

(11.12)
Algorithmic Implementation: Semi-Gradient Sarsa($\lambda$)

input: a differentiable function $\hat{q} : \mathbb{R}^\kappa \times \mathbb{R}^\zeta \rightarrow \mathbb{R}$
input: a policy $\pi$ (only if estimating $q_\pi$)
parameter: $\alpha \in \{\mathbb{R}|0 < \alpha < 1\}$, $\varepsilon \in \{\mathbb{R}|0 < \varepsilon << 1\}$, $\lambda \in \{\mathbb{R}|0 \leq \lambda \leq 1\}$
init: parameter vector $w \in \mathbb{R}^\zeta$ arbitrarily

for $j = 1, 2, \ldots$ episodes do
  initialize $x_0$ and set $z = 0$;
  $u_0 \leftarrow$ choose action from $\pi(x_0)$ or $\varepsilon$-greedy on $\hat{q}(x_0, \cdot, w)$;
  for $k = 0, 1, 2 \ldots$ time steps do
    apply action $u_k$, observe $x_{k+1}$ and $r_{k+1}$;
    if $x_{k+1}$ is terminal then $\delta \leftarrow r_{k+1} - \hat{q}(x_k, u_k, w)$;
    else
      $u_{k+1} \leftarrow \pi(x_{k+1})$ or $\varepsilon$-greedy on $\hat{q}(x_{k+1}, \cdot, w)$;
      $\delta \leftarrow r_{k+1} + \gamma \hat{q}(x_{k+1}, u_{k+1}, w) - \hat{q}(x_k, u_k, w)$;
      $z \leftarrow \gamma \lambda z + \nabla_w \hat{q}(x_k, u_k, w)$;
      $w \leftarrow w + \alpha \delta z$;
    exit loop if $x_{k+1}$ is terminal;

Algo. 11.3: Semi-gradient Sarsa($\lambda$) (output: parameter vector $w$ for $\hat{q}_\pi$ or $\hat{q}^*$)
Sarsa Learning Comparison in Gridworld Example

- $\lambda$ can be interpreted as the discounting factor acting on the eligibility trace (see right-most panel below).
- Intuitive interpretation: more recent transitions are more certain/relevant for the current update step.

\[
\begin{align*}
S & \quad T \\
St & \quad At \\
At+1 & \quad AT1 \\
St+1 & \quad Rt+1 \\
\ldots & \quad \ldots \\
St+2 & \quad \ldots \\
\end{align*}
\]

**Fig. 11.11:** Sarsa variants after an arbitrary episode within a gridworld environment – arrows indicate action-value change starting from initially zero estimates (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)
Sarsa(\(\lambda\)) vs \(n\)-Step Sarsa on Mountain Car Task

- Sarsa(\(\lambda\)) is able to learn significantly faster than any \(n\)-step variant.
- However, only intermediate performance is shown after 50 episodes.

Fig. 11.12: Sarsa-based control performance comparison based on previous example from Fig. 10.4. Both algorithms use linear function approximation and tile coding (source: R. Sutton and G. Barto, Reinforcement learning: an introduction, 2018, CC BY-NC-ND 2.0)
True Online Sarsa($\lambda$)

- Likewise TD($\lambda$) for state values, Sarsa($\lambda$) is not an exact backward view implementation of the offline $\lambda$-return algorithm (11.11).
- To improve the Sarsa($\lambda$) performance we again have two options:
  - Redoing updates: Online Sarsa($\lambda$) (e.g., with non-linear function approximation, direct transfer from (11.8)),
  - or re-use the true online implementation with linear approximators for action values.

- The true online Sarsa($\lambda$) updates are analog to TD($\lambda$), but the features additionally contain action information $\tilde{x} = f(x, u)$:

$$
\begin{align*}
    z_k &= \gamma \lambda z_{k-1} + (1 - \alpha \gamma \lambda \tilde{x}_k^T z_{k-1}) \tilde{x}_k, \\
    \delta_k &= r_{k+1} + \gamma \tilde{x}_{k+1}^T w_k - \tilde{x}_k^T w_k, \\
    w_{k+1} &= w_k + \alpha \delta_k z_k + \alpha \left( \tilde{x}_k^T w_k - \tilde{x}_k^T w_{k-1} \right) (z_k - \tilde{x}_k).
\end{align*}
$$

(11.13)
### Algorithmic Implementation: True Online Sarsa($\lambda$)

**input:** a policy $\pi$ to be evaluated  
**input:** a feature representation $\tilde{x}$ with $\tilde{x}_T = 0$ (i.e., $\hat{q}(\tilde{x}_T, \cdot, \cdot) = 0$)  
**parameter:** $\alpha \in \mathbb{R}|0 < \alpha < 1\}$, $\varepsilon \in \{\mathbb{R}|0 < \varepsilon << 1\}$, $\lambda \in \{\mathbb{R}|0 \leq \lambda \leq 1\}$  
**init:** value-function weights $w \in \mathbb{R}^\zeta$ arbitrarily

**for** $j = 1, 2, \ldots$ episodes **do**

- initialize $x_0$ and set $u_0 \leftarrow$ from $\pi(x_0)$ or $\varepsilon$-greedy on $\hat{q}(x_0, \cdot, w)$;  
- set $z = 0$ and $\hat{q}_{old} = 0$;  

**for** $k = 0, 1, 2 \ldots$ time steps **do**

- apply action $u_k$, observe $x_{k+1}$ and $r_{k+1}$;  
- $u_{k+1} \leftarrow$ choose action from $\pi(x_{k+1})$ or $\varepsilon$-greedy on $\hat{q}(x_{k+1}, \cdot, w)$;  
- $\hat{q} \leftarrow \tilde{x}_k^T w$ and $\hat{q}' \leftarrow \tilde{x}_{k+1}^T w$;  
- $\delta \leftarrow r_{k+1} + \gamma\hat{q}' - \hat{q}$;  
- $z \leftarrow \gamma \lambda z + (1 - \alpha \gamma \lambda \tilde{x}_k^T z) \tilde{x}_k$;  
- $w \leftarrow w + \alpha(\delta + \hat{q} - \hat{q}_{old}) z - \alpha(\hat{q} - \hat{q}_{old}) \tilde{x}_k$;  
- $\hat{q}_{old} \leftarrow \hat{q}'$;  
- exit loop if $x_{k+1}$ is terminal;

**Algorithm 11.4:** True Online Sarsa($\lambda$) (output: parameter vector $w$ for $\hat{q}_\pi$ or $\hat{q}^*$)
Multiple $n$-step return estimates can be weighted to form a compound update (adds more degrees of freedom).

$\lambda$-returns use this idea with exponentially-decaying weights.

However, like $n$-step bootstrapping also $\lambda$-returns are forward-viewing and, therefore suffer from increased memory demand and delay times.

Using eligibility traces we introduce backward-facing algorithms:

The trace is acting as a short-term memory.

How important was a parameter for the current value update?

$\text{TD}(\lambda)$ is using the traces for state-value prediction.

Applicable with general nonlinear function approximation.

However, not an exact representation of $\lambda$-returns.

Only if linear function approximation is used, true online $\text{TD}(\lambda)$ allows identically backward-facing updates as $\lambda$-returns.

Transfer to action values by Sarsa is straightforward.
The End for Today

Thanks for your attention and have a nice week!