

# Eligibility Traces



DEPARTMENT OF COMPUTER SCIENCE  
ST. FRANCIS XAVIER UNIVERSITY

CSCI-531 - Reinforcement Learning

Fall 2025

# The Unifying Mechanism

## Two Methods We've Seen

- ▶ **Monte Carlo Methods:** Wait for complete episodes

# The Unifying Mechanism

## Two Methods We've Seen

- ▶ **Monte Carlo Methods**: Wait for complete episodes
- ▶ **Temporal-Difference Methods**: Learn at each step

# The Unifying Mechanism

## Two Methods We've Seen

- ▶ **Monte Carlo Methods**: Wait for complete episodes
- ▶ **Temporal-Difference Methods**: Learn at each step

## Activity

What is the main difference between Monte Carlo and TD methods?

# The Unifying Mechanism

## Two Methods We've Seen

- ▶ **Monte Carlo Methods**: Wait for complete episodes
- ▶ **Temporal-Difference Methods**: Learn at each step

## Activity

What is the main difference between Monte Carlo and TD methods?

## Answer

- ▶ **Monte Carlo**: Uses actual returns, no bootstrapping

# The Unifying Mechanism

## Two Methods We've Seen

- ▶ **Monte Carlo Methods**: Wait for complete episodes
- ▶ **Temporal-Difference Methods**: Learn at each step

## Activity

What is the main difference between Monte Carlo and TD methods?

## Answer

- ▶ **Monte Carlo**: Uses actual returns, no bootstrapping
- ▶ **TD**: Uses estimated returns, bootstrapping from current estimates

# The Unifying Mechanism

## Two Methods We've Seen

- ▶ **Monte Carlo Methods**: Wait for complete episodes
- ▶ **Temporal-Difference Methods**: Learn at each step

## Activity

What is the main difference between Monte Carlo and TD methods?

## Answer

- ▶ **Monte Carlo**: Uses actual returns, no bootstrapping
- ▶ **TD**: Uses estimated returns, bootstrapping from current estimates
- ▶ **Trade-off**: Bias vs. variance, speed vs. accuracy

# The Unifying Mechanism

## Two Methods We've Seen

- ▶ **Monte Carlo Methods**: Wait for complete episodes
- ▶ **Temporal-Difference Methods**: Learn at each step

## Activity

What is the main difference between Monte Carlo and TD methods?

## Answer

- ▶ **Monte Carlo**: Uses actual returns, no bootstrapping
- ▶ **TD**: Uses estimated returns, bootstrapping from current estimates
- ▶ **Trade-off**: Bias vs. variance, speed vs. accuracy

## Enter Eligibility Traces

**Eligibility traces** provide a mechanism that **unifies** both methods!

# What Are Eligibility Traces?

## Key Concept

**Eligibility traces** are a short-term memory mechanism that bridges MC and TD methods.

# What Are Eligibility Traces?

## Key Concept

**Eligibility traces** are a short-term memory mechanism that bridges MC and TD methods.

## The Mechanism

- ▶ **Eligibility trace vector**:  $\mathbf{z}_t \in \mathbb{R}^d$

# What Are Eligibility Traces?

## Key Concept

**Eligibility traces** are a short-term memory mechanism that bridges MC and TD methods.

## The Mechanism

- ▶ **Eligibility trace vector**:  $\mathbf{z}_t \in \mathbb{R}^d$
- ▶ Parallel to the weight vector  $\mathbf{w}_t$

# What Are Eligibility Traces?

## Key Concept

**Eligibility traces** are a short-term memory mechanism that bridges MC and TD methods.

## The Mechanism

- ▶ **Eligibility trace vector**:  $\mathbf{z}_t \in \mathbb{R}^d$
- ▶ Parallel to the weight vector  $\mathbf{w}_t$
- ▶ **Trace-decay parameter**:  $\lambda \in [0, 1]$

# What Are Eligibility Traces?

## Key Concept

**Eligibility traces** are a short-term memory mechanism that bridges MC and TD methods.

## The Mechanism

- ▶ **Eligibility trace vector**:  $\mathbf{z}_t \in \mathbb{R}^d$
- ▶ Parallel to the weight vector  $\mathbf{w}_t$
- ▶ **Trace-decay parameter**:  $\lambda \in [0, 1]$

## How It Works

1. When a component of  $\mathbf{w}_t$  is used to estimate a value

# What Are Eligibility Traces?

## Key Concept

**Eligibility traces** are a short-term memory mechanism that bridges MC and TD methods.

## The Mechanism

- ▶ **Eligibility trace vector**:  $z_t \in \mathbb{R}^d$
- ▶ Parallel to the weight vector  $w_t$
- ▶ **Trace-decay parameter**:  $\lambda \in [0, 1]$

## How It Works

1. When a component of  $w_t$  is used to estimate a value
2. The corresponding component of  $z_t$  is **increased**

# What Are Eligibility Traces?

## Key Concept

**Eligibility traces** are a short-term memory mechanism that bridges MC and TD methods.

## The Mechanism

- ▶ **Eligibility trace vector**:  $z_t \in \mathbb{R}^d$
- ▶ Parallel to the weight vector  $w_t$
- ▶ **Trace-decay parameter**:  $\lambda \in [0, 1]$

## How It Works

1. When a component of  $w_t$  is used to estimate a value
2. The corresponding component of  $z_t$  is **increased**
3. Then it starts to **fade away**

# What Are Eligibility Traces?

## Key Concept

**Eligibility traces** are a short-term memory mechanism that bridges MC and TD methods.

## The Mechanism

- ▶ **Eligibility trace vector**:  $\mathbf{z}_t \in \mathbb{R}^d$
- ▶ Parallel to the weight vector  $\mathbf{w}_t$
- ▶ **Trace-decay parameter**:  $\lambda \in [0, 1]$

## How It Works

1. When a component of  $\mathbf{w}_t$  is used to estimate a value
2. The corresponding component of  $\mathbf{z}_t$  is **increased**
3. Then it starts to **fade away**

# Advantages of Eligibility Traces

## Benefits Over Previous Methods

- ▶ **Single error signal:** Only one TD error needed for updates

# Advantages of Eligibility Traces

## Benefits Over Previous Methods

- ▶ **Single error signal**: Only one TD error needed for updates
- ▶ **Online learning**: Updates happen at each step

# Advantages of Eligibility Traces

## Benefits Over Previous Methods

- ▶ **Single error signal**: Only one TD error needed for updates
- ▶ **Online learning**: Updates happen at each step
- ▶ **Backward view**: Past states get immediate credit

# Advantages of Eligibility Traces

## Benefits Over Previous Methods

- ▶ **Single error signal**: Only one TD error needed for updates
- ▶ **Online learning**: Updates happen at each step
- ▶ **Backward view**: Past states get immediate credit
- ▶ **Computational efficiency**: No need to store entire trajectories

# Advantages of Eligibility Traces

## Benefits Over Previous Methods

- ▶ **Single error signal**: Only one TD error needed for updates
- ▶ **Online learning**: Updates happen at each step
- ▶ **Backward view**: Past states get immediate credit
- ▶ **Computational efficiency**: No need to store entire trajectories

## The Credit Assignment Problem

- ▶ **Problem**: Which past actions led to current reward?

# Advantages of Eligibility Traces

## Benefits Over Previous Methods

- ▶ **Single error signal**: Only one TD error needed for updates
- ▶ **Online learning**: Updates happen at each step
- ▶ **Backward view**: Past states get immediate credit
- ▶ **Computational efficiency**: No need to store entire trajectories

## The Credit Assignment Problem

- ▶ **Problem**: Which past actions led to current reward?
- ▶ **Solution**: Traces remember recent state visits

# Advantages of Eligibility Traces

## Benefits Over Previous Methods

- ▶ **Single error signal**: Only one TD error needed for updates
- ▶ **Online learning**: Updates happen at each step
- ▶ **Backward view**: Past states get immediate credit
- ▶ **Computational efficiency**: No need to store entire trajectories

## The Credit Assignment Problem

- ▶ **Problem**: Which past actions led to current reward?
- ▶ **Solution**: Traces remember recent state visits
- ▶ More recent visits get **more credit**

# Advantages of Eligibility Traces

## Benefits Over Previous Methods

- ▶ **Single error signal**: Only one TD error needed for updates
- ▶ **Online learning**: Updates happen at each step
- ▶ **Backward view**: Past states get immediate credit
- ▶ **Computational efficiency**: No need to store entire trajectories

## The Credit Assignment Problem

- ▶ **Problem**: Which past actions led to current reward?
- ▶ **Solution**: Traces remember recent state visits
- ▶ More recent visits get **more credit**

## Key Insight

**Eligibility traces allow us to learn from every step while considering the full**

# Building Up to $\lambda$ -Returns

## Recall: N-Step Returns

$$G_{t:t+n} = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n \hat{v}(s_{t+n}, \mathbf{w}_{t+n-1}) \quad (1)$$

where  $0 \leq t \leq T - n$

# Building Up to $\lambda$ -Returns

## Recall: N-Step Returns

$$G_{t:t+n} = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n \hat{v}(s_{t+n}, \mathbf{w}_{t+n-1}) \quad (1)$$

where  $0 \leq t \leq T - n$

## Different Time Horizons

- **1-step:**  $G_{t:t+1} = r_{t+1} + \gamma \hat{v}(s_{t+1}, \mathbf{w}_t)$  (TD)

# Building Up to $\lambda$ -Returns

## Recall: N-Step Returns

$$G_{t:t+n} = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n \hat{v}(s_{t+n}, \mathbf{w}_{t+n-1}) \quad (1)$$

where  $0 \leq t \leq T - n$

## Different Time Horizons

- ▶ **1-step:**  $G_{t:t+1} = r_{t+1} + \gamma \hat{v}(s_{t+1}, \mathbf{w}_t)$  (TD)
- ▶ **2-step:**  $G_{t:t+2} = r_{t+1} + \gamma r_{t+2} + \gamma^2 \hat{v}(s_{t+2}, \mathbf{w}_{t+1})$

# Building Up to $\lambda$ -Returns

## Recall: N-Step Returns

$$G_{t:t+n} = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n \hat{v}(s_{t+n}, w_{t+n-1}) \quad (1)$$

where  $0 \leq t \leq T - n$

## Different Time Horizons

- ▶ **1-step:**  $G_{t:t+1} = r_{t+1} + \gamma \hat{v}(s_{t+1}, w_t)$  (TD)
- ▶ **2-step:**  $G_{t:t+2} = r_{t+1} + \gamma r_{t+2} + \gamma^2 \hat{v}(s_{t+2}, w_{t+1})$
- ▶  **$\infty$ -step:**  $G_t = r_{t+1} + \gamma r_{t+2} + \cdots$  (Monte Carlo)

# Building Up to $\lambda$ -Returns

## Recall: N-Step Returns

$$G_{t:t+n} = r_{t+1} + \gamma r_{t+2} + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n \hat{v}(s_{t+n}, w_{t+n-1}) \quad (1)$$

where  $0 \leq t \leq T - n$

## Different Time Horizons

- ▶ **1-step:**  $G_{t:t+1} = r_{t+1} + \gamma \hat{v}(s_{t+1}, w_t)$  (TD)
- ▶ **2-step:**  $G_{t:t+2} = r_{t+1} + \gamma r_{t+2} + \gamma^2 \hat{v}(s_{t+2}, w_{t+1})$
- ▶  **$\infty$ -step:**  $G_t = r_{t+1} + \gamma r_{t+2} + \cdots$  (Monte Carlo)

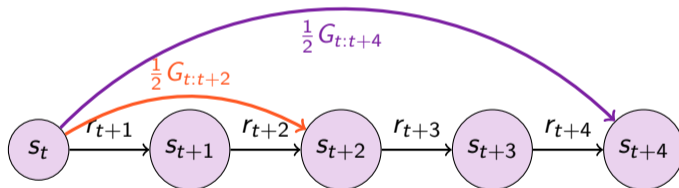
## Question

Can we combine different n-step returns intelligently?

## Compound Updates

### Example: Averaging Returns

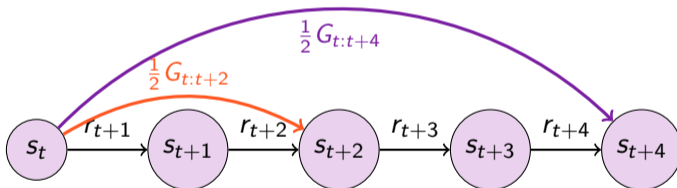
Update toward a target that combines multiple n-step returns:  $\frac{1}{2} G_{t:t+2} + \frac{1}{2} G_{t:t+4}$



# Compound Updates

## Example: Averaging Returns

Update toward a target that combines multiple n-step returns:  $\frac{1}{2} G_{t:t+2} + \frac{1}{2} G_{t:t+4}$



## Limitation

Compound updates can only be computed when the **longest component** is complete.

# The $\lambda$ -Return Definition

## TD( $\lambda$ ) Weighting Scheme

Weight each n-step return proportionally to  $\lambda^{n-1}$ , then normalize:

# The $\lambda$ -Return Definition

## TD( $\lambda$ ) Weighting Scheme

Weight each n-step return proportionally to  $\lambda^{n-1}$ , then normalize:

## $\lambda$ -Return Formula

$$G_t^\lambda = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_{t:t+n} \quad (2)$$

# The $\lambda$ -Return Definition

## TD( $\lambda$ ) Weighting Scheme

Weight each  $n$ -step return proportionally to  $\lambda^{n-1}$ , then normalize:

## $\lambda$ -Return Formula

$$G_t^\lambda = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_{t:t+n} \quad (2)$$

## Key Properties

- **Exponential decay:** Weight fades by  $\lambda$  with each additional step

# The $\lambda$ -Return Definition

## TD( $\lambda$ ) Weighting Scheme

Weight each  $n$ -step return proportionally to  $\lambda^{n-1}$ , then normalize:

## $\lambda$ -Return Formula

$$G_t^\lambda = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_{t:t+n} \quad (2)$$

## Key Properties

- ▶ **Exponential decay**: Weight fades by  $\lambda$  with each additional step
- ▶ **Normalization**: Factor  $(1 - \lambda)$  ensures weights sum to 1

# The $\lambda$ -Return Definition

## TD( $\lambda$ ) Weighting Scheme

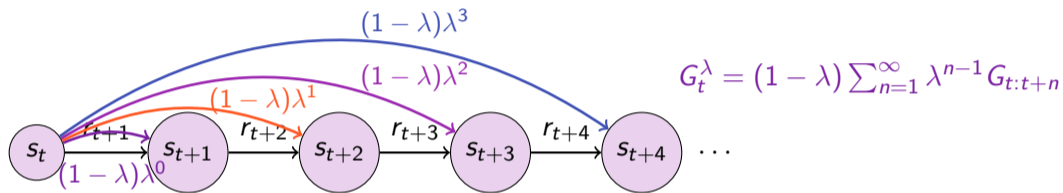
Weight each n-step return proportionally to  $\lambda^{n-1}$ , then normalize:

## $\lambda$ -Return Formula

$$G_t^\lambda = (1 - \lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_{t:t+n} \quad (2)$$

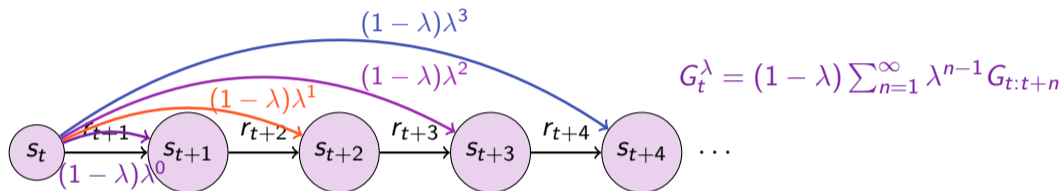
## Key Properties

- ▶ **Exponential decay**: Weight fades by  $\lambda$  with each additional step
- ▶ **Normalization**: Factor  $(1 - \lambda)$  ensures weights sum to 1
- ▶ **Special cases**:
  - ▶  $\lambda = 0$ :  $G_t^\lambda = G_{t:t+1}$  (Pure TD)
  - ▶  $\lambda = 1$ :  $G_t^\lambda = G_t$  (Pure MC)

Visualizing  $\lambda$ -Returns

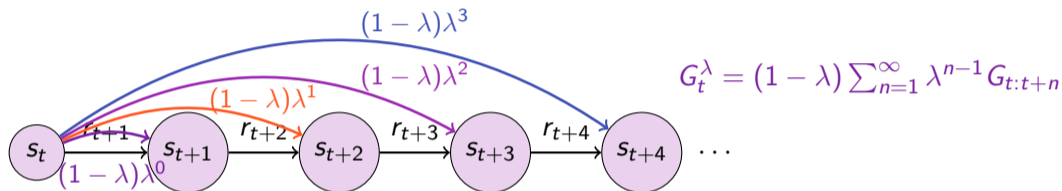
## Understanding the Diagram

- Each arrow represents an n-step return

Visualizing  $\lambda$ -Returns

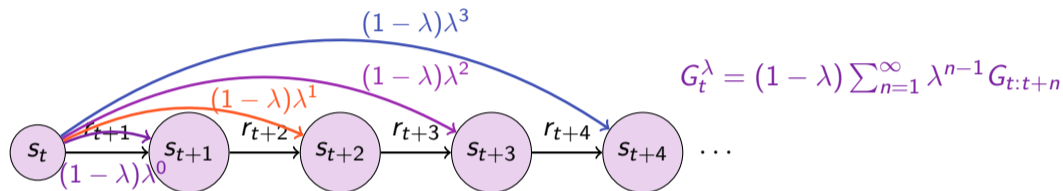
## Understanding the Diagram

- Each arrow represents an n-step return

Visualizing  $\lambda$ -Returns

## Understanding the Diagram

- ▶ Each arrow represents an  $n$ -step return
- ▶ Longer returns have exponentially smaller weights

Visualizing  $\lambda$ -Returns

## Understanding the Diagram

- Each arrow represents an n-step return
- Longer returns have exponentially smaller weights

## Activity

Why do we normalize by  $(1 - \lambda)$ ? What happens after terminal states?

## Activity Solution

Why Normalize by  $(1 - \lambda)$ ?

Need weights to sum to 1 for a proper average:

$$\sum_{n=1}^{\infty} \lambda^{n-1} = \frac{1}{1 - \lambda} \quad (3)$$

$$\text{So } (1 - \lambda) \times \frac{1}{1 - \lambda} = 1$$

## Activity Solution

### Why Normalize by $(1 - \lambda)$ ?

Need weights to sum to 1 for a proper average:

$$\sum_{n=1}^{\infty} \lambda^{n-1} = \frac{1}{1 - \lambda} \quad (3)$$

So  $(1 - \lambda) \times \frac{1}{1 - \lambda} = 1$

### After Terminal States

- ▶ After terminal state, all n-step returns equal the conventional return  $G_t$

## Activity Solution

### Why Normalize by $(1 - \lambda)$ ?

Need weights to sum to 1 for a proper average:

$$\sum_{n=1}^{\infty} \lambda^{n-1} = \frac{1}{1 - \lambda} \quad (3)$$

So  $(1 - \lambda) \times \frac{1}{1 - \lambda} = 1$

### After Terminal States

- ▶ After terminal state, all  $n$ -step returns equal the conventional return  $G_t$
- ▶  $G_{t:t+n} = G_t$  for all  $n \geq T - t$

# Activity Solution

## Why Normalize by $(1 - \lambda)$ ?

Need weights to sum to 1 for a proper average:

$$\sum_{n=1}^{\infty} \lambda^{n-1} = \frac{1}{1 - \lambda} \quad (3)$$

So  $(1 - \lambda) \times \frac{1}{1 - \lambda} = 1$

## After Terminal States

- ▶ After terminal state, all  $n$ -step returns equal the conventional return  $G_t$
- ▶  $G_{t:t+n} = G_t$  for all  $n \geq T - t$
- ▶ Therefore:  $G_t^\lambda = G_t$  (becomes pure Monte Carlo)

# Activity Solution

## Why Normalize by $(1 - \lambda)$ ?

Need weights to sum to 1 for a proper average:

$$\sum_{n=1}^{\infty} \lambda^{n-1} = \frac{1}{1 - \lambda} \quad (3)$$

So  $(1 - \lambda) \times \frac{1}{1 - \lambda} = 1$

## After Terminal States

- ▶ After terminal state, all  $n$ -step returns equal the conventional return  $G_t$
- ▶  $G_{t:t+n} = G_t$  for all  $n \geq T - t$
- ▶ Therefore:  $G_t^\lambda = G_t$  (becomes pure Monte Carlo)

## Key Insight

# TD( $\lambda$ ): The Classic Algorithm

## Historical Significance

**TD( $\lambda$ )** is one of the oldest and most widely used algorithms in reinforcement learning.

# TD( $\lambda$ ): The Classic Algorithm

## Historical Significance

**TD( $\lambda$ )** is one of the oldest and most widely used algorithms in reinforcement learning.

## The Eligibility Trace Vector

- ▶  $z_t \in \mathbb{R}^d$  has same dimensions as weight vector  $w_t$

# TD( $\lambda$ ): The Classic Algorithm

## Historical Significance

**TD( $\lambda$ )** is one of the oldest and most widely used algorithms in reinforcement learning.

## The Eligibility Trace Vector

- ▶  $z_t \in \mathbb{R}^d$  has same dimensions as weight vector  $w_t$
- ▶ Initialized to zero at episode start:  $z_0 = 0$

# TD( $\lambda$ ): The Classic Algorithm

## Historical Significance

**TD( $\lambda$ )** is one of the oldest and most widely used algorithms in reinforcement learning.

## The Eligibility Trace Vector

- ▶  $z_t \in \mathbb{R}^d$  has same dimensions as weight vector  $w_t$
- ▶ Initialized to zero at episode start:  $z_0 = 0$
- ▶ Updated at each time step with two components:

# TD( $\lambda$ ): The Classic Algorithm

## Historical Significance

**TD( $\lambda$ )** is one of the oldest and most widely used algorithms in reinforcement learning.

## The Eligibility Trace Vector

- ▶  $z_t \in \mathbb{R}^d$  has same dimensions as weight vector  $w_t$
- ▶ Initialized to zero at episode start:  $z_0 = 0$
- ▶ Updated at each time step with two components:
  - ▶ **Decay**: Previous traces fade by  $\gamma\lambda$

# TD( $\lambda$ ): The Classic Algorithm

## Historical Significance

**TD( $\lambda$ )** is one of the oldest and most widely used algorithms in reinforcement learning.

## The Eligibility Trace Vector

- ▶  $z_t \in \mathbb{R}^d$  has same dimensions as weight vector  $w_t$
- ▶ Initialized to zero at episode start:  $z_0 = 0$
- ▶ Updated at each time step with two components:
  - ▶ **Decay**: Previous traces fade by  $\gamma\lambda$
  - ▶ **Increment**: Add current state's feature gradient

# TD( $\lambda$ ): The Classic Algorithm

## Historical Significance

**TD( $\lambda$ )** is one of the oldest and most widely used algorithms in reinforcement learning.

## The Eligibility Trace Vector

- ▶  $z_t \in \mathbb{R}^d$  has same dimensions as weight vector  $w_t$
- ▶ Initialized to zero at episode start:  $z_0 = 0$
- ▶ Updated at each time step with two components:
  - ▶ **Decay**: Previous traces fade by  $\gamma\lambda$
  - ▶ **Increment**: Add current state's feature gradient

## Trace Update Equation

$$z_0 = 0$$

(4)

# Understanding Eligibility Traces

## What Do Traces Track?

Eligibility traces keep track of which components of  $w$  have contributed to **recent** state valuations.

# Understanding Eligibility Traces

## What Do Traces Track?

Eligibility traces keep track of which components of  $w$  have contributed to **recent** state valuations.

## Definition of "Recent"

- ▶ "Recency" is defined by the product  $\gamma\lambda$

# Understanding Eligibility Traces

## What Do Traces Track?

Eligibility traces keep track of which components of  $w$  have contributed to **recent** state valuations.

## Definition of "Recent"

- ▶ "Recency" is defined by the product  $\gamma\lambda$
- ▶  $\lambda = 0$ : Only current state eligible

# Understanding Eligibility Traces

## What Do Traces Track?

Eligibility traces keep track of which components of  $w$  have contributed to **recent** state valuations.

## Definition of "Recent"

- ▶ "Recency" is defined by the product  $\gamma\lambda$
- ▶  $\lambda = 0$ : Only current state eligible
- ▶  $\lambda = 1$ : All visited states remain eligible

# Understanding Eligibility Traces

## What Do Traces Track?

Eligibility traces keep track of which components of  $w$  have contributed to **recent** state valuations.

## Definition of "Recent"

- ▶ "Recency" is defined by the product  $\gamma\lambda$
- ▶  $\lambda = 0$ : Only current state eligible
- ▶  $\lambda = 1$ : All visited states remain eligible
- ▶  $0 < \lambda < 1$ : Exponential decay of eligibility

# Understanding Eligibility Traces

## What Do Traces Track?

Eligibility traces keep track of which components of  $w$  have contributed to **recent** state valuations.

## Definition of "Recent"

- ▶ "Recency" is defined by the product  $\gamma\lambda$
- ▶  $\lambda = 0$ : Only current state eligible
- ▶  $\lambda = 1$ : All visited states remain eligible
- ▶  $0 < \lambda < 1$ : Exponential decay of eligibility

## Eligibility for Learning

- ▶ Traces indicate **eligibility** of each weight component

# Understanding Eligibility Traces

## What Do Traces Track?

Eligibility traces keep track of which components of  $w$  have contributed to **recent** state valuations.

## Definition of "Recent"

- ▶ "Recency" is defined by the product  $\gamma\lambda$
- ▶  $\lambda = 0$ : Only current state eligible
- ▶  $\lambda = 1$ : All visited states remain eligible
- ▶  $0 < \lambda < 1$ : Exponential decay of eligibility

## Eligibility for Learning

- ▶ Traces indicate **eligibility** of each weight component
- ▶ Higher trace  $\Rightarrow$  more eligible for learning changes

# Understanding Eligibility Traces

## What Do Traces Track?

Eligibility traces keep track of which components of  $w$  have contributed to **recent** state valuations.

## Definition of "Recent"

- ▶ "Recency" is defined by the product  $\gamma\lambda$
- ▶  $\lambda = 0$ : Only current state eligible
- ▶  $\lambda = 1$ : All visited states remain eligible
- ▶  $0 < \lambda < 1$ : Exponential decay of eligibility

## Eligibility for Learning

- ▶ Traces indicate **eligibility** of each weight component
- ▶ Higher trace  $\Rightarrow$  more eligible for learning changes
- ▶ Based on current TD error  $\delta_t$

# TD( $\lambda$ ) Update Rules

TD Error (Same as TD(0))

$$\delta_t = r_{t+1} + \gamma \hat{v}(s_{t+1}, \mathbf{w}_t) - \hat{v}(s_t, \mathbf{w}_t) \quad (6)$$

# TD( $\lambda$ ) Update Rules

TD Error (Same as TD(0))

$$\delta_t = r_{t+1} + \gamma \hat{v}(s_{t+1}, \mathbf{w}_t) - \hat{v}(s_t, \mathbf{w}_t) \quad (6)$$

Weight Update (New!)

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \delta_t \mathbf{z}_t \quad (7)$$

# TD( $\lambda$ ) Update Rules

TD Error (Same as TD(0))

$$\delta_t = r_{t+1} + \gamma \hat{v}(s_{t+1}, \mathbf{w}_t) - \hat{v}(s_t, \mathbf{w}_t) \quad (6)$$

Weight Update (New!)

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \delta_t \mathbf{z}_t \quad (7)$$

Key Insight

- ▶ **Single TD error**  $\delta_t$  affects multiple weight components

# TD( $\lambda$ ) Update Rules

TD Error (Same as TD(0))

$$\delta_t = r_{t+1} + \gamma \hat{v}(s_{t+1}, w_t) - \hat{v}(s_t, w_t) \quad (6)$$

Weight Update (New!)

$$w_{t+1} = w_t + \alpha \delta_t z_t \quad (7)$$

Key Insight

- ▶ **Single TD error**  $\delta_t$  affects multiple weight components
- ▶ Each component updated proportional to its **eligibility**

# TD( $\lambda$ ) Update Rules

TD Error (Same as TD(0))

$$\delta_t = r_{t+1} + \gamma \hat{v}(s_{t+1}, w_t) - \hat{v}(s_t, w_t) \quad (6)$$

Weight Update (New!)

$$w_{t+1} = w_t + \alpha \delta_t z_t \quad (7)$$

Key Insight

- ▶ **Single TD error**  $\delta_t$  affects multiple weight components
- ▶ Each component updated proportional to its **eligibility**
- ▶ Recently visited states get larger updates

# TD( $\lambda$ ) Update Rules

TD Error (Same as TD(0))

$$\delta_t = r_{t+1} + \gamma \hat{v}(s_{t+1}, w_t) - \hat{v}(s_t, w_t) \quad (6)$$

Weight Update (New!)

$$w_{t+1} = w_t + \alpha \delta_t z_t \quad (7)$$

Key Insight

- ▶ **Single TD error**  $\delta_t$  affects multiple weight components
- ▶ Each component updated proportional to its **eligibility**
- ▶ Recently visited states get larger updates

# TD( $\lambda$ ) Update Rules

TD Error (Same as TD(0))

$$\delta_t = r_{t+1} + \gamma \hat{v}(s_{t+1}, w_t) - \hat{v}(s_t, w_t) \quad (6)$$

Weight Update (New!)

$$w_{t+1} = w_t + \alpha \delta_t z_t \quad (7)$$

Key Insight

- ▶ **Single TD error**  $\delta_t$  affects multiple weight components
- ▶ Each component updated proportional to its **eligibility**
- ▶ Recently visited states get larger updates

# TD( $\lambda$ ) Algorithm

Input: the policy  $\pi$  to be evaluated

Input: a differentiable function  $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^d \rightarrow \mathbb{R}$  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  $R, S'$

$\mathbf{z} \leftarrow \gamma \lambda \mathbf{z} + \nabla \hat{v}(S, \mathbf{w})$

$\delta \leftarrow R + \gamma \hat{v}(S', \mathbf{w}) - \hat{v}(S, \mathbf{w})$

$\mathbf{w} \leftarrow \mathbf{w} + \alpha \delta \mathbf{z}$

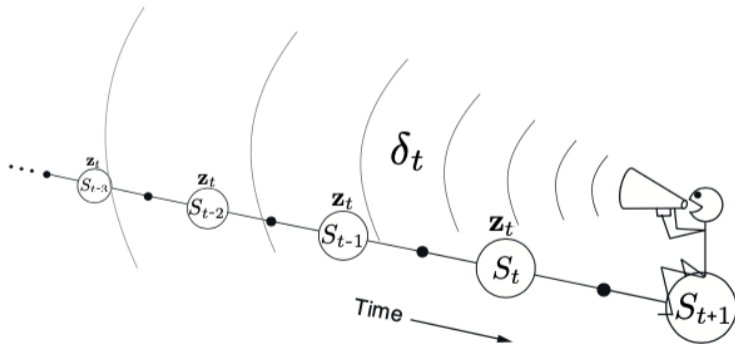
$S \leftarrow S'$

  until  $S'$  is terminal

# The Backward View Mechanism

## Backward Perspective

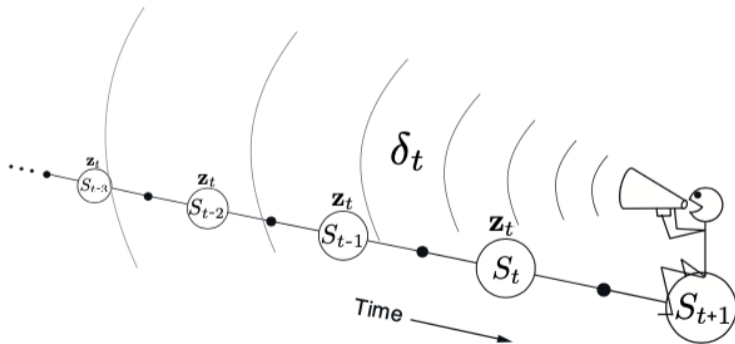
**Backward view:** Each update depends on current TD error combined with eligibility traces of past events.



# The Backward View Mechanism

## Backward Perspective

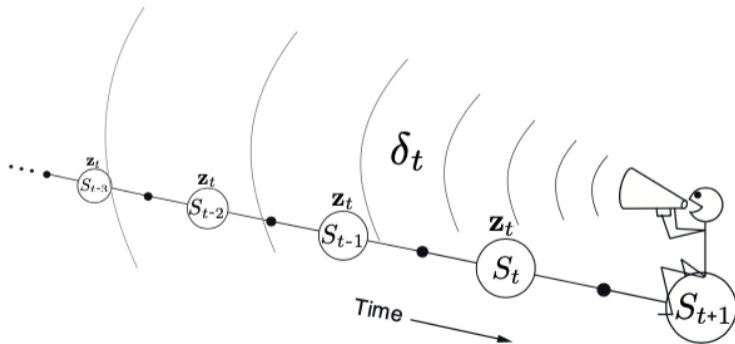
**Backward view:** Each update depends on current TD error combined with eligibility traces of past events.



# The Backward View Mechanism

## Backward Perspective

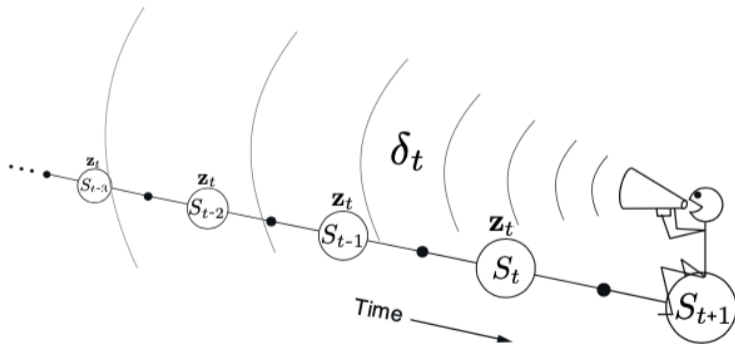
**Backward view:** Each update depends on current TD error combined with eligibility traces of past events.



# The Backward View Mechanism

## Backward Perspective

**Backward view:** Each update depends on current TD error combined with eligibility traces of past events.



# From State Values to Action Values

## Moving to Control

We want to learn approximate action values  $\hat{q}(s, a, w)$  instead of state values.

## From State Values to Action Values

### Moving to Control

We want to learn approximate action values  $\hat{q}(s, a, w)$  instead of state values.

### Modified Components

**Weight Update (Same Pattern):**

$$w_{t+1} = w_t + \alpha \delta_t z_t \quad (8)$$

**TD Error (For Action Values):**

$$\delta_t = r_{t+1} + \gamma \hat{q}(s_{t+1}, a_{t+1}, w_t) - \hat{q}(s_t, a_t, w_t) \quad (9)$$

**Eligibility Trace (For Action Values):**

$$z_0 = 0 \quad (10)$$

$$z_t = \gamma \lambda z_{t-1} + \nabla \hat{q}(s_t, a_t, w_t) \quad (11)$$

# SARSA( $\lambda$ ) Algorithm

```

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

Loop for each episode:
  Initialize  $S$ 
  Choose  $A \sim \pi(\cdot|S)$  or  $\varepsilon$ -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  $\varepsilon$ -greedy according to  $\hat{q}(S', \cdot, \mathbf{w})$ 
    Loop for  $i$  in  $\mathcal{F}(S', A')$ :  $\delta \leftarrow \delta + \gamma w_i$ 
     $\mathbf{w} \leftarrow \mathbf{w} + \alpha \delta \mathbf{z}$ 
     $\mathbf{z} \leftarrow \gamma \lambda \mathbf{z}$ 
     $S \leftarrow S'; A \leftarrow A'$ 

```

# Understanding SARSA( $\lambda$ )

## Key Differences from SARSA(0)

- ▶ **SARSA(0)**: Only current  $(s, a)$  pair updated

# Understanding SARSA( $\lambda$ )

## Key Differences from SARSA(0)

- ▶ **SARSA(0)**: Only current  $(s, a)$  pair updated
- ▶ **SARSA( $\lambda$ )**: All recently visited  $(s, a)$  pairs updated

# Understanding SARSA( $\lambda$ )

## Key Differences from SARSA(0)

- ▶ **SARSA(0)**: Only current  $(s, a)$  pair updated
- ▶ **SARSA( $\lambda$ )**: All recently visited  $(s, a)$  pairs updated
- ▶ **Eligibility traces** remember which actions were taken recently

# Understanding SARSA( $\lambda$ )

## Key Differences from SARSA(0)

- ▶ **SARSA(0)**: Only current  $(s, a)$  pair updated
- ▶ **SARSA( $\lambda$ )**: All recently visited  $(s, a)$  pairs updated
- ▶ **Eligibility traces** remember which actions were taken recently

## Benefits

- ▶ **Faster learning**: Information propagates to multiple state-action pairs

# Understanding SARSA( $\lambda$ )

## Key Differences from SARSA(0)

- ▶ **SARSA(0)**: Only current  $(s, a)$  pair updated
- ▶ **SARSA( $\lambda$ )**: All recently visited  $(s, a)$  pairs updated
- ▶ **Eligibility traces** remember which actions were taken recently

## Benefits

- ▶ **Faster learning**: Information propagates to multiple state-action pairs
- ▶ **Better credit assignment**: Past actions get immediate feedback

# Understanding SARSA( $\lambda$ )

## Key Differences from SARSA(0)

- ▶ **SARSA(0)**: Only current  $(s, a)$  pair updated
- ▶ **SARSA( $\lambda$ )**: All recently visited  $(s, a)$  pairs updated
- ▶ **Eligibility traces** remember which actions were taken recently

## Benefits

- ▶ **Faster learning**: Information propagates to multiple state-action pairs
- ▶ **Better credit assignment**: Past actions get immediate feedback
- ▶ **Sample efficiency**: Each experience updates many values

# Understanding SARSA( $\lambda$ )

## Key Differences from SARSA(0)

- ▶ **SARSA(0)**: Only current  $(s, a)$  pair updated
- ▶ **SARSA( $\lambda$ )**: All recently visited  $(s, a)$  pairs updated
- ▶ **Eligibility traces** remember which actions were taken recently

## Benefits

- ▶ **Faster learning**: Information propagates to multiple state-action pairs
- ▶ **Better credit assignment**: Past actions get immediate feedback
- ▶ **Sample efficiency**: Each experience updates many values

## Activity

Read through the SARSA( $\lambda$ ) pseudocode and identify the key steps.

# Types of Eligibility Traces

## Accumulating Traces (What We've Seen)

$$\mathbf{z}_t = \gamma \lambda \mathbf{z}_{t-1} + \nabla \hat{q}(s_t, a_t, \mathbf{w}_t) \quad (12)$$

Traces **accumulate** each time a state-action pair is revisited.

# Types of Eligibility Traces

## Accumulating Traces (What We've Seen)

$$z_t = \gamma \lambda z_{t-1} + \nabla \hat{q}(s_t, a_t, w_t) \quad (12)$$

Traces **accumulate** each time a state-action pair is revisited.

## Replacing Traces

$$z_t(s, a) = \begin{cases} \gamma \lambda z_{t-1}(s, a) & \text{if } (s, a) \neq (s_t, a_t) \\ 1 & \text{if } (s, a) = (s_t, a_t) \end{cases} \quad (13)$$

Traces are **reset to 1** when a state-action pair is revisited.

# Types of Eligibility Traces

## Accumulating Traces (What We've Seen)

$$\mathbf{z}_t = \gamma \lambda \mathbf{z}_{t-1} + \nabla \hat{q}(s_t, a_t, \mathbf{w}_t) \quad (12)$$

Traces **accumulate** each time a state-action pair is revisited.

## Replacing Traces

$$z_t(s, a) = \begin{cases} \gamma \lambda z_{t-1}(s, a) & \text{if } (s, a) \neq (s_t, a_t) \\ 1 & \text{if } (s, a) = (s_t, a_t) \end{cases} \quad (13)$$

Traces are **reset to 1** when a state-action pair is revisited.

## When to Use Each

- **Accumulating:** Function approximation, continuous spaces

# Types of Eligibility Traces

## Accumulating Traces (What We've Seen)

$$\mathbf{z}_t = \gamma \lambda \mathbf{z}_{t-1} + \nabla \hat{q}(s_t, a_t, \mathbf{w}_t) \quad (12)$$

Traces **accumulate** each time a state-action pair is revisited.

## Replacing Traces

$$z_t(s, a) = \begin{cases} \gamma \lambda z_{t-1}(s, a) & \text{if } (s, a) \neq (s_t, a_t) \\ 1 & \text{if } (s, a) = (s_t, a_t) \end{cases} \quad (13)$$

Traces are **reset to 1** when a state-action pair is revisited.

## When to Use Each

- ▶ **Accumulating:** Function approximation, continuous spaces
- ▶ **Replacing:** Tabular methods, discrete spaces

# Types of Eligibility Traces

## Accumulating Traces (What We've Seen)

$$\mathbf{z}_t = \gamma \lambda \mathbf{z}_{t-1} + \nabla \hat{q}(s_t, a_t, \mathbf{w}_t) \quad (12)$$

Traces **accumulate** each time a state-action pair is revisited.

## Replacing Traces

$$z_t(s, a) = \begin{cases} \gamma \lambda z_{t-1}(s, a) & \text{if } (s, a) \neq (s_t, a_t) \\ 1 & \text{if } (s, a) = (s_t, a_t) \end{cases} \quad (13)$$

Traces are **reset to 1** when a state-action pair is revisited.

## When to Use Each

- ▶ **Accumulating:** Function approximation, continuous spaces
- ▶ **Replacing:** Tabular methods, discrete spaces

# Implementation Considerations

## Computational Efficiency

- ▶ **Sparse updates:** Only update non-zero traces

# Implementation Considerations

## Computational Efficiency

- ▶ **Sparse updates**: Only update non-zero traces
- ▶ **Trace cutoff**: Set traces to zero when below threshold

# Implementation Considerations

## Computational Efficiency

- ▶ **Sparse updates**: Only update non-zero traces
- ▶ **Trace cutoff**: Set traces to zero when below threshold
- ▶ **Memory management**: Traces can grow large in long episodes

# Implementation Considerations

## Computational Efficiency

- ▶ **Sparse updates**: Only update non-zero traces
- ▶ **Trace cutoff**: Set traces to zero when below threshold
- ▶ **Memory management**: Traces can grow large in long episodes

## Parameter Selection

- ▶  $\lambda$ : Higher values  $\Rightarrow$  more like Monte Carlo

# Implementation Considerations

## Computational Efficiency

- ▶ **Sparse updates**: Only update non-zero traces
- ▶ **Trace cutoff**: Set traces to zero when below threshold
- ▶ **Memory management**: Traces can grow large in long episodes

## Parameter Selection

- ▶  $\lambda$ : Higher values  $\Rightarrow$  more like Monte Carlo
- ▶  $\alpha$ : Learning rate may need adjustment with  $\lambda$

# Implementation Considerations

## Computational Efficiency

- ▶ **Sparse updates**: Only update non-zero traces
- ▶ **Trace cutoff**: Set traces to zero when below threshold
- ▶ **Memory management**: Traces can grow large in long episodes

## Parameter Selection

- ▶  $\lambda$ : Higher values  $\Rightarrow$  more like Monte Carlo
- ▶  $\alpha$ : Learning rate may need adjustment with  $\lambda$
- ▶  $\gamma$ : Affects both discounting and trace decay

# Implementation Considerations

## Computational Efficiency

- ▶ **Sparse updates**: Only update non-zero traces
- ▶ **Trace cutoff**: Set traces to zero when below threshold
- ▶ **Memory management**: Traces can grow large in long episodes

## Parameter Selection

- ▶  $\lambda$ : Higher values  $\Rightarrow$  more like Monte Carlo
- ▶  $\alpha$ : Learning rate may need adjustment with  $\lambda$
- ▶  $\gamma$ : Affects both discounting and trace decay

## Practical Tips

- ▶ Start with  $\lambda = 0.9$  as a default

# Implementation Considerations

## Computational Efficiency

- ▶ **Sparse updates**: Only update non-zero traces
- ▶ **Trace cutoff**: Set traces to zero when below threshold
- ▶ **Memory management**: Traces can grow large in long episodes

## Parameter Selection

- ▶  $\lambda$ : Higher values  $\Rightarrow$  more like Monte Carlo
- ▶  $\alpha$ : Learning rate may need adjustment with  $\lambda$
- ▶  $\gamma$ : Affects both discounting and trace decay

## Practical Tips

- ▶ Start with  $\lambda = 0.9$  as a default
- ▶ Use replacing traces for tabular methods

# Implementation Considerations

## Computational Efficiency

- ▶ **Sparse updates**: Only update non-zero traces
- ▶ **Trace cutoff**: Set traces to zero when below threshold
- ▶ **Memory management**: Traces can grow large in long episodes

## Parameter Selection

- ▶  $\lambda$ : Higher values  $\Rightarrow$  more like Monte Carlo
- ▶  $\alpha$ : Learning rate may need adjustment with  $\lambda$
- ▶  $\gamma$ : Affects both discounting and trace decay

## Practical Tips

- ▶ Start with  $\lambda = 0.9$  as a default
- ▶ Use replacing traces for tabular methods

# Beyond Basic Eligibility Traces

## Limitation of Semi-Gradient Methods

The  $TD(\lambda)$  we've seen is a **semi-gradient** method:

- Updates don't account for changing targets

# Beyond Basic Eligibility Traces

## Limitation of Semi-Gradient Methods

The  $TD(\lambda)$  we've seen is a **semi-gradient** method:

- ▶ Updates don't account for changing targets
- ▶ Not equivalent to gradient descent on any objective function

# Beyond Basic Eligibility Traces

## Limitation of Semi-Gradient Methods

The  $TD(\lambda)$  we've seen is a **semi-gradient** method:

- ▶ Updates don't account for changing targets
- ▶ Not equivalent to gradient descent on any objective function
- ▶ Can be unstable with function approximation

# Beyond Basic Eligibility Traces

## Limitation of Semi-Gradient Methods

The  $TD(\lambda)$  we've seen is a **semi-gradient** method:

- ▶ Updates don't account for changing targets
- ▶ Not equivalent to gradient descent on any objective function
- ▶ Can be unstable with function approximation

## True Online $TD(\lambda)$

- ▶ **True gradient** method for  $\lambda$ -return objective

## Beyond Basic Eligibility Traces

### Limitation of Semi-Gradient Methods

The  $TD(\lambda)$  we've seen is a **semi-gradient** method:

- ▶ Updates don't account for changing targets
- ▶ Not equivalent to gradient descent on any objective function
- ▶ Can be unstable with function approximation

### True Online $TD(\lambda)$

- ▶ **True gradient** method for  $\lambda$ -return objective
- ▶ More complex update rules but better theoretical properties

## Beyond Basic Eligibility Traces

### Limitation of Semi-Gradient Methods

The  $TD(\lambda)$  we've seen is a **semi-gradient** method:

- ▶ Updates don't account for changing targets
- ▶ Not equivalent to gradient descent on any objective function
- ▶ Can be unstable with function approximation

### True Online $TD(\lambda)$

- ▶ **True gradient** method for  $\lambda$ -return objective
- ▶ More complex update rules but better theoretical properties
- ▶ Often performs better in practice with function approximation

# Beyond Basic Eligibility Traces

## Limitation of Semi-Gradient Methods

The  $TD(\lambda)$  we've seen is a **semi-gradient** method:

- ▶ Updates don't account for changing targets
- ▶ Not equivalent to gradient descent on any objective function
- ▶ Can be unstable with function approximation

## True Online $TD(\lambda)$

- ▶ **True gradient** method for  $\lambda$ -return objective
- ▶ More complex update rules but better theoretical properties
- ▶ Often performs better in practice with function approximation

## When It Matters

- ▶ **Function approximation:** Neural networks, linear functions

# Beyond Basic Eligibility Traces

## Limitation of Semi-Gradient Methods

The  $TD(\lambda)$  we've seen is a **semi-gradient** method:

- ▶ Updates don't account for changing targets
- ▶ Not equivalent to gradient descent on any objective function
- ▶ Can be unstable with function approximation

## True Online $TD(\lambda)$

- ▶ **True gradient** method for  $\lambda$ -return objective
- ▶ More complex update rules but better theoretical properties
- ▶ Often performs better in practice with function approximation

## When It Matters

- ▶ **Function approximation:** Neural networks, linear functions

# Beyond Basic Eligibility Traces

## Limitation of Semi-Gradient Methods

The  $TD(\lambda)$  we've seen is a **semi-gradient** method:

- ▶ Updates don't account for changing targets
- ▶ Not equivalent to gradient descent on any objective function
- ▶ Can be unstable with function approximation

## True Online $TD(\lambda)$

- ▶ **True gradient** method for  $\lambda$ -return objective
- ▶ More complex update rules but better theoretical properties
- ▶ Often performs better in practice with function approximation

## When It Matters

- ▶ **Function approximation:** Neural networks, linear functions

# Real-World Applications

## Game Playing

- ▶ **TD-Gammon**: Used  $TD(\lambda)$  to achieve superhuman backgammon play

# Real-World Applications

## Game Playing

- ▶ **TD-Gammon**: Used  $TD(\lambda)$  to achieve superhuman backgammon play
- ▶ **Chess engines**: Temporal difference learning with eligibility traces

# Real-World Applications

## Game Playing

- ▶ **TD-Gammon**: Used  $TD(\lambda)$  to achieve superhuman backgammon play
- ▶ **Chess engines**: Temporal difference learning with eligibility traces
- ▶ **Go programs**: Combined with Monte Carlo Tree Search

# Real-World Applications

## Game Playing

- ▶ **TD-Gammon**: Used  $TD(\lambda)$  to achieve superhuman backgammon play
- ▶ **Chess engines**: Temporal difference learning with eligibility traces
- ▶ **Go programs**: Combined with Monte Carlo Tree Search

## Robotics

- ▶ **Robot navigation**: Learning from delayed rewards

# Real-World Applications

## Game Playing

- ▶ **TD-Gammon**: Used  $TD(\lambda)$  to achieve superhuman backgammon play
- ▶ **Chess engines**: Temporal difference learning with eligibility traces
- ▶ **Go programs**: Combined with Monte Carlo Tree Search

## Robotics

- ▶ **Robot navigation**: Learning from delayed rewards
- ▶ **Manipulation**: Credit assignment for complex motor skills

# Real-World Applications

## Game Playing

- ▶ **TD-Gammon**: Used  $TD(\lambda)$  to achieve superhuman backgammon play
- ▶ **Chess engines**: Temporal difference learning with eligibility traces
- ▶ **Go programs**: Combined with Monte Carlo Tree Search

## Robotics

- ▶ **Robot navigation**: Learning from delayed rewards
- ▶ **Manipulation**: Credit assignment for complex motor skills
- ▶ **Multi-agent systems**: Coordinated learning with traces

# Real-World Applications

## Game Playing

- ▶ **TD-Gammon**: Used  $TD(\lambda)$  to achieve superhuman backgammon play
- ▶ **Chess engines**: Temporal difference learning with eligibility traces
- ▶ **Go programs**: Combined with Monte Carlo Tree Search

## Robotics

- ▶ **Robot navigation**: Learning from delayed rewards
- ▶ **Manipulation**: Credit assignment for complex motor skills
- ▶ **Multi-agent systems**: Coordinated learning with traces

## Finance and Control

- ▶ **Portfolio optimization**: Learning from market feedback

# Real-World Applications

## Game Playing

- ▶ **TD-Gammon**: Used  $TD(\lambda)$  to achieve superhuman backgammon play
- ▶ **Chess engines**: Temporal difference learning with eligibility traces
- ▶ **Go programs**: Combined with Monte Carlo Tree Search

## Robotics

- ▶ **Robot navigation**: Learning from delayed rewards
- ▶ **Manipulation**: Credit assignment for complex motor skills
- ▶ **Multi-agent systems**: Coordinated learning with traces

## Finance and Control

- ▶ **Portfolio optimization**: Learning from market feedback
- ▶ **Process control**: Industrial control systems

# Real-World Applications

## Game Playing

- ▶ **TD-Gammon**: Used  $TD(\lambda)$  to achieve superhuman backgammon play
- ▶ **Chess engines**: Temporal difference learning with eligibility traces
- ▶ **Go programs**: Combined with Monte Carlo Tree Search

## Robotics

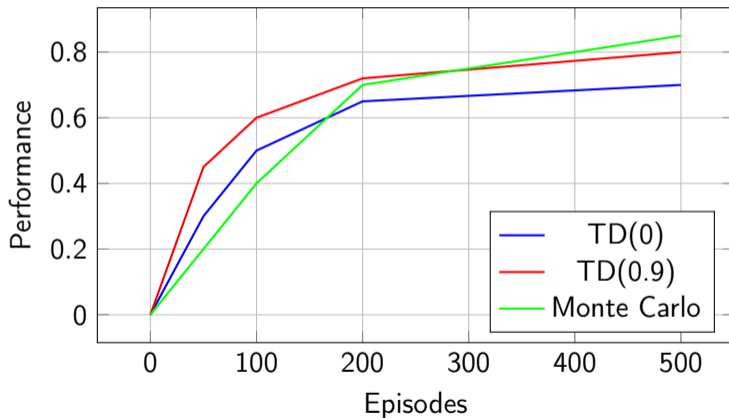
- ▶ **Robot navigation**: Learning from delayed rewards
- ▶ **Manipulation**: Credit assignment for complex motor skills
- ▶ **Multi-agent systems**: Coordinated learning with traces

## Finance and Control

- ▶ **Portfolio optimization**: Learning from market feedback
- ▶ **Process control**: Industrial control systems

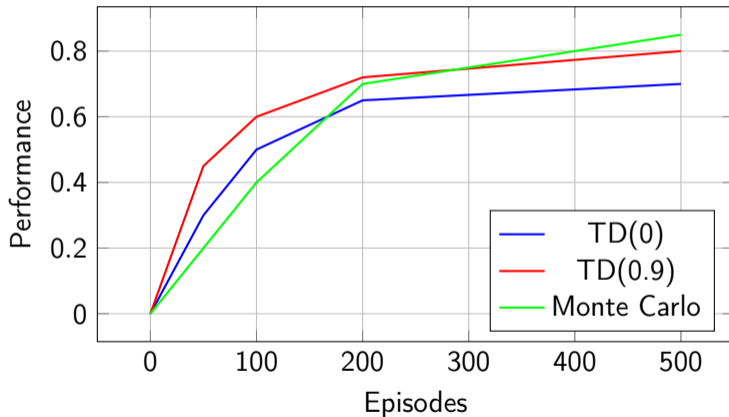
# Performance Characteristics

## Learning Speed



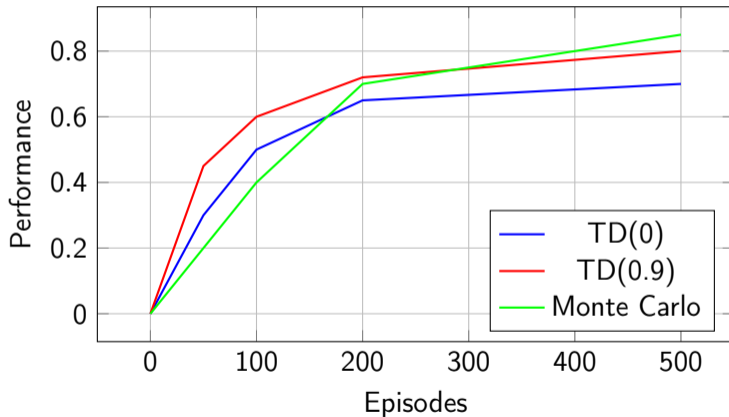
# Performance Characteristics

## Learning Speed



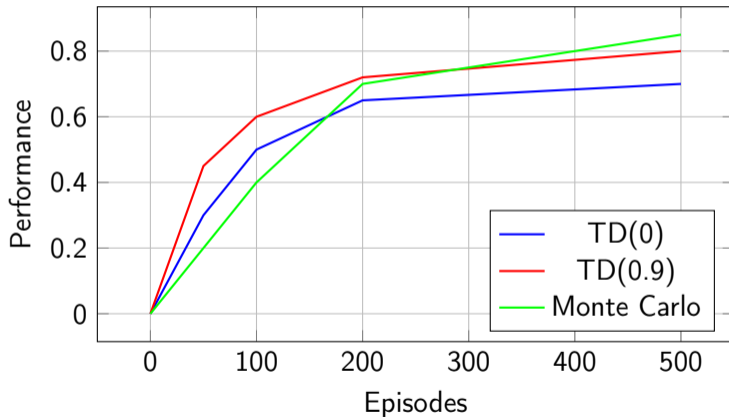
# Performance Characteristics

## Learning Speed



# Performance Characteristics

## Learning Speed



# Key Takeaways

## What We've Learned

- ▶ **Eligibility traces** unify Monte Carlo and TD methods

# Key Takeaways

## What We've Learned

- ▶ **Eligibility traces** unify Monte Carlo and TD methods
- ▶  **$\lambda$ -parameter** controls the trade-off between bias and variance

# Key Takeaways

## What We've Learned

- ▶ **Eligibility traces** unify Monte Carlo and TD methods
- ▶  **$\lambda$ -parameter** controls the trade-off between bias and variance
- ▶ **Backward view**: Efficient credit assignment mechanism

# Key Takeaways

## What We've Learned

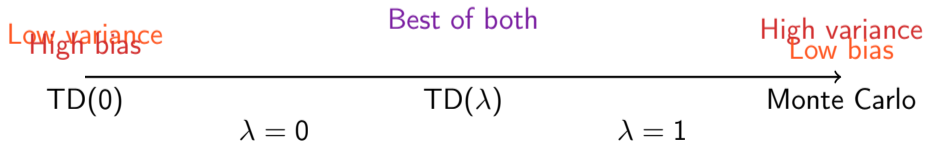
- ▶ **Eligibility traces** unify Monte Carlo and TD methods
- ▶  **$\lambda$ -parameter** controls the trade-off between bias and variance
- ▶ **Backward view**: Efficient credit assignment mechanism
- ▶ **Computational efficiency**: Single error updates multiple states

# Key Takeaways

## What We've Learned

- ▶ **Eligibility traces** unify Monte Carlo and TD methods
- ▶  **$\lambda$ -parameter** controls the trade-off between bias and variance
- ▶ **Backward view**: Efficient credit assignment mechanism
- ▶ **Computational efficiency**: Single error updates multiple states

## Algorithm Spectrum



# Extensions and Modern Developments

## Function Approximation

- ▶ **Neural networks:** Deep RL with eligibility traces

# Extensions and Modern Developments

## Function Approximation

- ▶ **Neural networks**: Deep RL with eligibility traces
- ▶ **Linear function approximation**: Convergence guarantees

# Extensions and Modern Developments

## Function Approximation

- ▶ **Neural networks**: Deep RL with eligibility traces
- ▶ **Linear function approximation**: Convergence guarantees
- ▶ **Feature selection**: Sparse eligibility traces

# Extensions and Modern Developments

## Function Approximation

- ▶ **Neural networks**: Deep RL with eligibility traces
- ▶ **Linear function approximation**: Convergence guarantees
- ▶ **Feature selection**: Sparse eligibility traces

## Policy Methods

- ▶ **Actor-Critic( $\lambda$ )**: Traces for both value and policy

# Extensions and Modern Developments

## Function Approximation

- ▶ **Neural networks**: Deep RL with eligibility traces
- ▶ **Linear function approximation**: Convergence guarantees
- ▶ **Feature selection**: Sparse eligibility traces

## Policy Methods

- ▶ **Actor-Critic( $\lambda$ )**: Traces for both value and policy
- ▶ **Policy gradient traces**: Eligibility traces for policy parameters

# Extensions and Modern Developments

## Function Approximation

- ▶ **Neural networks**: Deep RL with eligibility traces
- ▶ **Linear function approximation**: Convergence guarantees
- ▶ **Feature selection**: Sparse eligibility traces

## Policy Methods

- ▶ **Actor-Critic( $\lambda$ )**: Traces for both value and policy
- ▶ **Policy gradient traces**: Eligibility traces for policy parameters
- ▶ **Natural gradients**: Combined with eligibility traces

# Extensions and Modern Developments

## Function Approximation

- ▶ **Neural networks**: Deep RL with eligibility traces
- ▶ **Linear function approximation**: Convergence guarantees
- ▶ **Feature selection**: Sparse eligibility traces

## Policy Methods

- ▶ **Actor-Critic( $\lambda$ )**: Traces for both value and policy
- ▶ **Policy gradient traces**: Eligibility traces for policy parameters
- ▶ **Natural gradients**: Combined with eligibility traces

## Modern Algorithms

- ▶ **Retrace( $\lambda$ )**: Off-policy traces with importance sampling

# Extensions and Modern Developments

## Function Approximation

- ▶ **Neural networks**: Deep RL with eligibility traces
- ▶ **Linear function approximation**: Convergence guarantees
- ▶ **Feature selection**: Sparse eligibility traces

## Policy Methods

- ▶ **Actor-Critic( $\lambda$ )**: Traces for both value and policy
- ▶ **Policy gradient traces**: Eligibility traces for policy parameters
- ▶ **Natural gradients**: Combined with eligibility traces

## Modern Algorithms

- ▶ **Retrace( $\lambda$ )**: Off-policy traces with importance sampling
- ▶ **Impala**: Distributed RL with traces

# Extensions and Modern Developments

## Function Approximation

- ▶ **Neural networks**: Deep RL with eligibility traces
- ▶ **Linear function approximation**: Convergence guarantees
- ▶ **Feature selection**: Sparse eligibility traces

## Policy Methods

- ▶ **Actor-Critic( $\lambda$ )**: Traces for both value and policy
- ▶ **Policy gradient traces**: Eligibility traces for policy parameters
- ▶ **Natural gradients**: Combined with eligibility traces

## Modern Algorithms

- ▶ **Retrace( $\lambda$ )**: Off-policy traces with importance sampling
- ▶ **Impala**: Distributed RL with traces

# Looking Forward

## Next Steps in Your RL Journey

- ▶ **Function approximation**: Linear and non-linear methods

# Looking Forward

## Next Steps in Your RL Journey

- ▶ **Function approximation:** Linear and non-linear methods
- ▶ **Policy gradient methods:** Direct policy optimization

# Looking Forward

## Next Steps in Your RL Journey

- ▶ **Function approximation**: Linear and non-linear methods
- ▶ **Policy gradient methods**: Direct policy optimization
- ▶ **Deep reinforcement learning**: Neural network function approximation

# Looking Forward

## Next Steps in Your RL Journey

- ▶ **Function approximation**: Linear and non-linear methods
- ▶ **Policy gradient methods**: Direct policy optimization
- ▶ **Deep reinforcement learning**: Neural network function approximation
- ▶ **Advanced topics**: Multi-agent RL, hierarchical RL, meta-learning

# Looking Forward

## Next Steps in Your RL Journey

- ▶ **Function approximation**: Linear and non-linear methods
- ▶ **Policy gradient methods**: Direct policy optimization
- ▶ **Deep reinforcement learning**: Neural network function approximation
- ▶ **Advanced topics**: Multi-agent RL, hierarchical RL, meta-learning

## Key Insight

**Eligibility traces** provide the foundation for understanding how credit assignment works in reinforcement learning - a crucial concept that appears in many modern algorithms!

# Looking Forward

## Next Steps in Your RL Journey

- ▶ **Function approximation**: Linear and non-linear methods
- ▶ **Policy gradient methods**: Direct policy optimization
- ▶ **Deep reinforcement learning**: Neural network function approximation
- ▶ **Advanced topics**: Multi-agent RL, hierarchical RL, meta-learning

## Key Insight

**Eligibility traces** provide the foundation for understanding how credit assignment works in reinforcement learning - a crucial concept that appears in many modern algorithms!

## Final Thought

*"The best way to understand eligibility traces is to implement and experiment with them yourself."*