Approximation - On-Policy Prediction

DEPARTMENT OF COMPUTER SCIENCE
ST. FRANCIS XAVIER UNIVERSITY

CSCI-531 - Reinforcement Learning

Fall 2025

The Tabular Method Approach

What are Tabular Methods?

- Store **exact value** $q_\pi(s, a)$ for **each** state-action pair

Q-Table

(s_1, a_1)	$q(s_1, a_1)$
(s_1, a_2)	$q(s_1, a_2)$
$\vdots$	$\vdots$
(s_n, a_m)	$q(s_n, a_m)$

The Tabular Method Approach

What are Tabular Methods?

- Store **exact value** $q_\pi(s, a)$ for each state-action pair
- Create a **lookup table** with all possible combinations

Q-Table

(s_1, a_1)	$q(s_1, a_1)$
(s_1, a_2)	$q(s_1, a_2)$
$\vdots$	$\vdots$
(s_n, a_m)	$q(s_n, a_m)$

The Tabular Method Approach

What are Tabular Methods?

- Store **exact value** $q_\pi(s, a)$ for **each** state-action pair
- Create a **lookup table** with all possible combinations
- Perfect representation when feasible

Q-Table

(s_1, a_1)	$q(s_1, a_1)$
(s_1, a_2)	$q(s_1, a_2)$
$\vdots$	$\vdots$
(s_n, a_m)	$q(s_n, a_m)$

The Tabular Method Approach

What are Tabular Methods?

- Store **exact value** $q_\pi(s, a)$ for **each** state-action pair
- Create a **lookup table** with all possible combinations
- Perfect representation when feasible

Q-Table

(s_1, a_1)	$q(s_1, a_1)$
(s_1, a_2)	$q(s_1, a_2)$
$\vdots$	$\vdots$
(s_n, a_m)	$q(s_n, a_m)$

Activity

What issues can arise with this approach for large state spaces?

Problems with Tabular Methods

Scalability Issues

- **Memory explosion**: Space complexity $O(|S| \times |A|)$

Problems with Tabular Methods

Scalability Issues

- **Memory explosion**: Space complexity $O(|S| \times |A|)$
- **Learning time**: Need to visit every state-action pair

Problems with Tabular Methods

Scalability Issues

- **Memory explosion**: Space complexity $O(|S| \times |A|)$
- **Learning time**: Need to visit every state-action pair
- **Generalization**: No knowledge transfer between similar states

Problems with Tabular Methods

Scalability Issues

- **Memory explosion**: Space complexity $O(|S| \times |A|)$
- **Learning time**: Need to visit every state-action pair
- **Generalization**: No knowledge transfer between similar states

Real-World Examples

- **Chess**: $\approx 10^{47}$ possible positions

Problems with Tabular Methods

Scalability Issues

- **Memory explosion**: Space complexity $O(|S| \times |A|)$
- **Learning time**: Need to visit every state-action pair
- **Generalization**: No knowledge transfer between similar states

Real-World Examples

- **Chess**: $\approx 10^{47}$ possible positions
- **Atari games**: $256^{84 \times 84} \approx 10^{170,000}$ pixel combinations

Problems with Tabular Methods

Scalability Issues

- **Memory explosion**: Space complexity $O(|S| \times |A|)$
- **Learning time**: Need to visit every state-action pair
- **Generalization**: No knowledge transfer between similar states

Real-World Examples

- **Chess**: $\approx 10^{47}$ possible positions
- **Atari games**: $256^{84\times84} \approx 10^{170,000}$ pixel combinations
- **Autonomous driving**: Continuous sensor readings

Problems with Tabular Methods

Scalability Issues

- **Memory explosion**: Space complexity $O(|S| \times |A|)$
- **Learning time**: Need to visit every state-action pair
- **Generalization**: No knowledge transfer between similar states

Real-World Examples

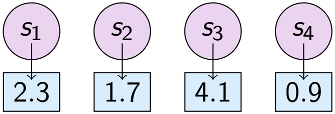
- **Chess**: $\approx 10^{47}$ possible positions
- **Atari games**: $256^{84\times 84} \approx 10^{170,000}$ pixel combinations
- **Autonomous driving**: Continuous sensor readings

Solution: Function Approximation

The Fundamental Shift: Tables vs Functions

Tabular Method (What We've Done Before)

- **One number per state**: Store $v(s_1), v(s_2), \ldots, v(s_n)$

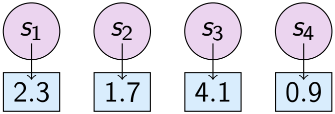


Tabular: Each state has its own value

The Fundamental Shift: Tables vs Functions

Tabular Method (What We've Done Before)

- **One number per state**: Store $v(s_1), v(s_2), \ldots, v(s_n)$
- **Independent values**: Updating $v(s_1)$ doesn't affect $v(s_2)$

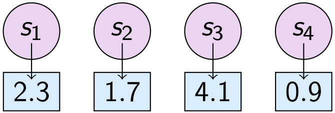


Tabular: Each state has its own value

The Fundamental Shift: Tables vs Functions

Tabular Method (What We've Done Before)

- **One number per state**: Store $v(s_1), v(s_2), \ldots, v(s_n)$
- **Independent values**: Updating $v(s_1)$ doesn't affect $v(s_2)$
- **Perfect but expensive**: Exact values but need $|S|$ memory slots

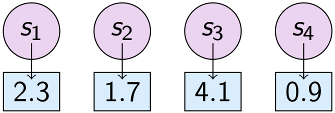


Tabular: Each state has its own value

The Fundamental Shift: Tables vs Functions

Tabular Method (What We've Done Before)

- **One number per state**: Store $v(s_1), v(s_2), \ldots, v(s_n)$
- **Independent values**: Updating $v(s_1)$ doesn't affect $v(s_2)$
- **Perfect but expensive**: Exact values but need $|S|$ memory slots



Tabular: Each state has its own value

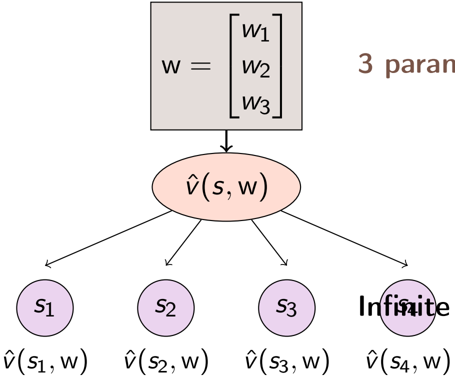
The Problem

What if we have 1 million states? We need 1 million separate values!

Function Approximation: The Core Idea

Key Innovation

Instead of storing individual values, use a **function** with **parameters**



3 parameters

Infinite states possible!

Why Function Approximation Works

The Magic of Shared Parameters

- **Shared parameters**: Same w computes values for ALL states

Why Function Approximation Works

The Magic of Shared Parameters

- **Shared parameters**: Same w computes values for ALL states
- **Generalization**: Similar states get similar values automatically

Why Function Approximation Works

The Magic of Shared Parameters

- **Shared parameters**: Same w computes values for ALL states
- **Generalization**: Similar states get similar values automatically
- **Efficiency**: Store only d parameters instead of $|S|$ values

Why Function Approximation Works

The Magic of Shared Parameters

- **Shared parameters**: Same w computes values for ALL states
- **Generalization**: Similar states get similar values automatically
- **Efficiency**: Store only d parameters instead of $|S|$ values

Key Benefits

- **Memory savings**: $d \ll |S|$ means huge space reduction

Why Function Approximation Works

The Magic of Shared Parameters

- **Shared parameters**: Same w computes values for ALL states
- **Generalization**: Similar states get similar values automatically
- **Efficiency**: Store only d parameters instead of $|S|$ values

Key Benefits

- **Memory savings**: $d \ll |S|$ means huge space reduction
- **Learning transfer**: Knowledge about one state helps with similar states

Why Function Approximation Works

The Magic of Shared Parameters

- **Shared parameters**: Same w computes values for ALL states
- **Generalization**: Similar states get similar values automatically
- **Efficiency**: Store only d parameters instead of $|S|$ values

Key Benefits

- **Memory savings**: $d \ll |S|$ means huge space reduction
- **Learning transfer**: Knowledge about one state helps with similar states
- **Handles continuous spaces**: Can work with infinite state spaces

Why Function Approximation Works

The Magic of Shared Parameters

- **Shared parameters**: Same w computes values for ALL states
- **Generalization**: Similar states get similar values automatically
- **Efficiency**: Store only d parameters instead of $|S|$ values

Key Benefits

- **Memory savings**: $d \ll |S|$ means huge space reduction
- **Learning transfer**: Knowledge about one state helps with similar states
- **Handles continuous spaces**: Can work with infinite state spaces

Trade-off

We gain efficiency and generalization, but lose the guarantee of exact values

Concrete Example: GridWorld

Scenario: 4×4 GridWorld (16 states)

Tabular: Need 16 separate values

Function Approximation: Maybe just 3 parameters!

(0,3)	(1,3)	(2,3)	(3,3)
(0,2)	(1,2)	(2,2)	(3,2)
(0,1)	(1,1)	(2,1)	(3,1)
(0,0)	(1,0)	(2,0)	(3,0)

GridWorld States

Function:

$$\hat{v}(x, y, \mathrm{w}) = w_1 \cdot x + w_2 \cdot y + w_3$$

$$\boxed{\mathrm{w} = \begin{bmatrix} 0.5 \\ -0.3 \\ 2.0 \end{bmatrix}}$$

$$\hat{v}(0,0) = 0.5(0) + (-0.3)(0) + 2.0 = 2.0$$
$$\hat{v}(1,0) = 0.5(1) + (-0.3)(0) + 2.0 = 2.5$$
$$\hat{v}(3,3) = 0.5(3) + (-0.3)(3) + 2.0 = 2.6$$

Concrete Example: GridWorld

(0,3)	(1,3)	(2,3)	(3,3)
(0,2)	(1,2)	(2,2)	(3,2)
(0,1)	(1,1)	(2,1)	(3,1)
(0,0)	(1,0)	(2,0)	(3,0)

GridWorld States

Function:

$$\hat{v}(x, y, \mathrm{w}) = w_1 \cdot x + w_2 \cdot y + w_3$$

$$\boxed{\mathrm{w} = \begin{bmatrix} 0.5 \\ -0.3 \\ 2.0 \end{bmatrix}}$$

$$\hat{v}(0, 0) = 0.5(0) + (-0.3)(0) + 2.0 = 2.0$$
$$\hat{v}(1, 0) = 0.5(1) + (-0.3)(0) + 2.0 = 2.5$$
$$\hat{v}(3, 3) = 0.5(3) + (-0.3)(3) + 2.0 = 2.6$$

Concrete Example: GridWorld

(0,3)	(1,3)	(2,3)	(3,3)
(0,2)	(1,2)	(2,2)	(3,2)
(0,1)	(1,1)	(2,1)	(3,1)
(0,0)	(1,0)	(2,0)	(3,0)

GridWorld States

Function:

$$\hat{v}(x, y, \mathrm{w}) = w_1 \cdot x + w_2 \cdot y + w_3$$

$$\mathrm{w} = \begin{bmatrix} 0.5 \\ -0.3 \\ 2.0 \end{bmatrix}$$

$$\hat{v}(0,0) = 0.5(0) + (-0.3)(0) + 2.0 = 2.0$$
$$\hat{v}(1,0) = 0.5(1) + (-0.3)(0) + 2.0 = 2.5$$
$$\hat{v}(3,3) = 0.5(3) + (-0.3)(3) + 2.0 = 2.6$$

Key Insight

Changing w_1 affects the value of **all** states. This is both powerful and challenging.

Update Targets in Approximation

General Update Form

$$\textit{NewEstimate} \leftarrow \textit{OldEstimate} + \textit{StepSize}[\textit{Target} - \textit{OldEstimate}]$$

Update Targets in Approximation

General Update Form

$$\textit{NewEstimate} \leftarrow \textit{OldEstimate} + \textit{StepSize}[\textit{Target} - \textit{OldEstimate}]$$

Update Notation: $s \mapsto u$

- s: The state being updated

Update Targets in Approximation

General Update Form

$$NewEstimate \leftarrow OldEstimate + StepSize[Target - OldEstimate]$$

Update Notation: $s \mapsto u$

- s: The state being updated
- u: The **target value** for that state

Update Targets in Approximation

General Update Form

$$NewEstimate \leftarrow OldEstimate + StepSize[Target - OldEstimate]$$

Update Notation: $s \mapsto u$

- s: The state being updated
- u: The **target value** for that state

Method-Specific Targets

- **Monte Carlo**: $s_t \mapsto G_t$ (return)

Update Targets in Approximation

General Update Form

$$NewEstimate \leftarrow OldEstimate + StepSize[Target - OldEstimate]$$

Update Notation: $s \mapsto u$

- s: The state being updated
- u: The **target value** for that state

Method-Specific Targets

- **Monte Carlo**: $s_t \mapsto G_t$ (return)
- **TD(0)**: $s_t \mapsto r_{t+1} + \gamma \hat{v}(s_{t+1}, \mathrm{w})$

Update Targets in Approximation

General Update Form

$$NewEstimate \leftarrow OldEstimate + StepSize[Target - OldEstimate]$$

Update Notation: $s \mapsto u$

- s: The state being updated
- u: The target value for that state

Method-Specific Targets

- **Monte Carlo**: $s_t \mapsto G_t$ (return)
- **TD(0)**: $s_t \mapsto r_{t+1} + \gamma \hat{v}(s_{t+1}, \mathrm{w})$
- **n-step TD**: $s_t \mapsto G_{t:t+n}$

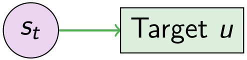
Update Targets in Approximation

General Update Form

$$NewEstimate \leftarrow OldEstimate + StepSize[Target - OldEstimate]$$

Update Notation: $s \mapsto u$

- s: The state being updated
- u: The **target value** for that state



Update direction

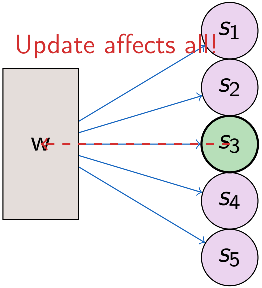
Method-Specific Targets

- **Monte Carlo**: $s_t \mapsto G_t$ (return)
- **TD(0)**: $s_t \mapsto r_{t+1} + \gamma \hat{v}(s_{t+1}, \mathrm{w})$
- **n-step TD**: $s_t \mapsto G_{t:t+n}$

The Generalization Challenge

Key Problem

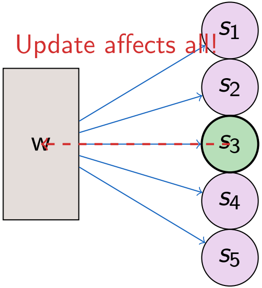
Updating one state affects the approximation for **all other states**!



The Generalization Challenge

Key Problem

Updating one state affects the approximation for **all other states**!



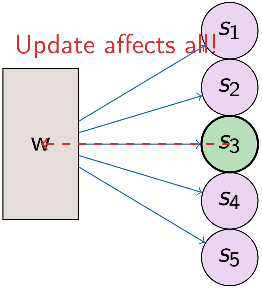
Consequence

- Making one state's estimate **more accurate**

The Generalization Challenge

Key Problem

Updating one state affects the approximation for **all other states**!



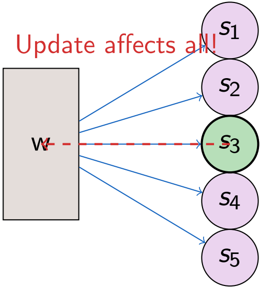
Consequence

- Making one state's estimate **more accurate**
- Invariably makes others' **less accurate**

The Generalization Challenge

Key Problem

Updating one state affects the approximation for **all other states**!



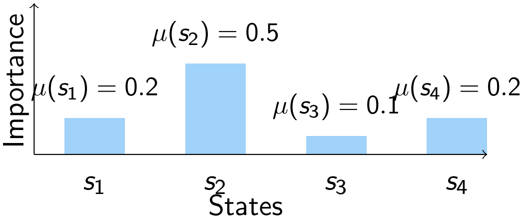
Consequence

- Making one state's estimate **more accurate**
- Invariably makes others' **less accurate**
- Need to prioritize which states matter most

Defining State Importance

State Distribution $\mu(s)$

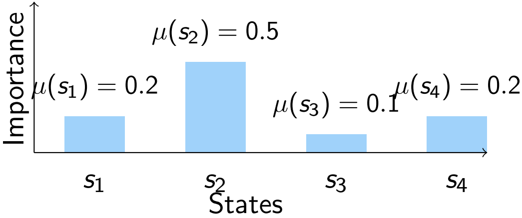
- $\mu(s) \geq 0$ for all states s



Defining State Importance

State Distribution $\mu(s)$

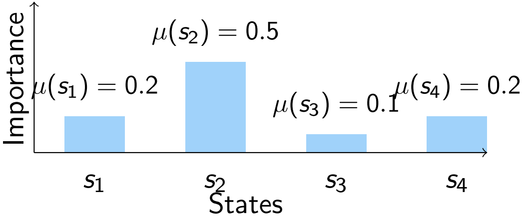
- $\mu(s) \geq 0$ for all states s
- $\sum_s \mu(s) = 1$ (probability distribution)



Defining State Importance

State Distribution $\mu(s)$

- $\mu(s) \geq 0$ for all states s
- $\sum_s \mu(s) = 1$ (probability distribution)
- **Higher** $\mu(s)$ = **more important** state



Defining State Importance

State Distribution $\mu(s)$

- $\mu(s) \geq 0$ for all states s
- $\sum_s \mu(s) = 1$ (probability distribution)
- Higher $\mu(s)$ = more important state

Defining State Importance

State Distribution $\mu(s)$

- $\mu(s) \geq 0$ for all states s
- $\sum_s \mu(s) = 1$ (probability distribution)
- **Higher** $\mu(s)$ = **more important** state

Common Choice: On-Policy Distribution

- $\mu(s)$ = fraction of time spent in state s under policy π

Defining State Importance

State Distribution $\mu(s)$

- $\mu(s) \geq 0$ for all states s
- $\sum_s \mu(s) = 1$ (probability distribution)
- **Higher** $\mu(s)$ = **more important** state

Common Choice: On-Policy Distribution

- $\mu(s)$ = fraction of time spent in state s under policy π
- Focus learning on **frequently visited** states

The Objective Function

Mean Squared Value Error (MSVE)

$$\overline{\mathrm{VE}}(\mathbf{w}) = \sum_{s \in S} \mu(s) \left[v_\pi(s) - \hat{v}(s, \mathbf{w}) \right]^2 \tag{1}$$

The Objective Function

Mean Squared Value Error (MSVE)

$$\overline{\text{VE}}(\mathbf{w}) = \sum_{s \in \mathcal{S}} \mu(s) \left[v_\pi(s) - \hat{v}(s, \mathbf{w}) \right]^2 \tag{1}$$

Components

- $v_\pi(s)$: **True value** under policy π

The Objective Function

Mean Squared Value Error (MSVE)

$$\overline{\text{VE}}(\mathbf{w}) = \sum_{s \in S} \mu(s) \left[v_\pi(s) - \hat{v}(s, \mathbf{w}) \right]^2 \tag{1}$$

Components

- $v_\pi(s)$: **True value** under policy π
- $\hat{v}(s, \mathbf{w})$: **Approximate value** with weights $\mathbf{w}$

The Objective Function

Mean Squared Value Error (MSVE)

$$\overline{\text{VE}}(\mathbf{w}) = \sum_{s \in S} \mu(s) \left[v_\pi(s) - \hat{v}(s, \mathbf{w}) \right]^2 \tag{1}$$

Components

- $v_\pi(s)$: **True value** under policy π
- $\hat{v}(s, \mathbf{w})$: **Approximate value** with weights $\mathbf{w}$
- $\mu(s)$: **State importance** weighting

The Objective Function

Mean Squared Value Error (MSVE)

$$\overline{\mathrm{VE}}(\mathrm{w}) = \sum_{s \in S} \mu(s) \left[v_\pi(s) - \hat{v}(s, \mathrm{w}) \right]^2 \tag{2}$$

The Objective Function

Mean Squared Value Error (MSVE)

$$\overline{\mathrm{VE}}(\mathrm{w}) = \sum_{s \in S} \mu(s) \left[v_\pi(s) - \hat{v}(s, \mathrm{w}) \right]^2 \tag{2}$$

Important Limitation

The Objective Function

Mean Squared Value Error (MSVE)

$$\overline{\mathrm{VE}}(\mathbf{w}) = \sum_{s \in S} \mu(s) \left[v_\pi(s) - \hat{v}(s, \mathbf{w}) \right]^2 \tag{2}$$

Important Limitation

- No guarantee of global optimum

The Objective Function

Mean Squared Value Error (MSVE)

$$\overline{\mathrm{VE}}(\mathrm{w}) = \sum_{s \in S} \mu(s) \left[v_\pi(s) - \hat{v}(s, \mathrm{w}) \right]^2 \tag{2}$$

Important Limitation

- No guarantee of global optimum
- Usually converges to local optimum

The Objective Function

Mean Squared Value Error (MSVE)

$$\overline{\text{VE}}(\mathbf{w}) = \sum_{s \in \mathcal{S}} \mu(s) \left[v_\pi(s) - \hat{v}(s, \mathbf{w}) \right]^2 \tag{2}$$

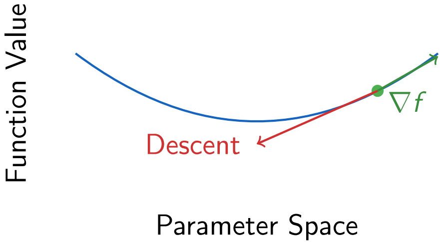
Important Limitation

- No guarantee of global optimum
- Usually converges to local optimum
- Quality depends on function approximation choice

Gradient Descent Intuition

What is a Gradient?

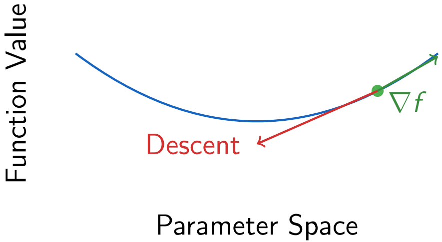
- **Slope** of a function at a specific point



Gradient Descent Intuition

What is a Gradient?

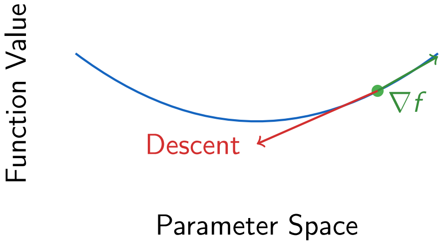
- **Slope** of a function at a specific point
- Points in direction of **steepest increase**



Gradient Descent Intuition

What is a Gradient?

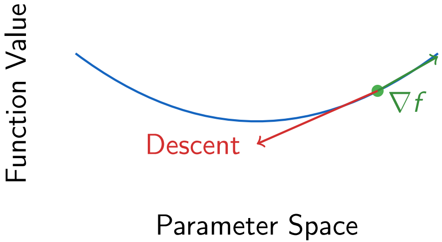
- **Slope** of a function at a specific point
- Points in direction of **steepest increase**
- For function $f(\mathrm{w})$: $\nabla f(\mathrm{w}) = \left(\frac{\partial f}{\partial w_1}, \ldots, \frac{\partial f}{\partial w_d}\right)^T$



Gradient Descent Intuition

What is a Gradient?

- **Slope** of a function at a specific point
- Points in direction of **steepest increase**
- For function $f(\mathbf{w})$: $\nabla f(\mathbf{w}) = \left(\frac{\partial f}{\partial w_1}, \ldots, \frac{\partial f}{\partial w_d}\right)^T$



Gradient Descent Update

Gradient Descent Intuition

What is a Gradient?

- **Slope** of a function at a specific point
- Points in direction of **steepest increase**
- For function $f(\mathrm{w})$: $\nabla f(\mathrm{w}) = \left(\frac{\partial f}{\partial w_1}, \ldots, \frac{\partial f}{\partial w_d}\right)^T$

Gradient Descent Intuition

What is a Gradient?

- **Slope** of a function at a specific point
- Points in direction of **steepest increase**
- For function $f(\mathbf{w})$: $\nabla f(\mathbf{w}) = \left(\frac{\partial f}{\partial w_1}, \ldots, \frac{\partial f}{\partial w_d}\right)^T$

Gradient Descent Update

$$\mathbf{w}_{t+1} = \mathbf{w}_t - \alpha \nabla f(\mathbf{w}_t)$$

where α is the **learning rate**

Example: Gradient Calculation - Setup

Function: $f(x, y) = 0.5x^2 + y^2$

Find the gradient at point $(x = 10, y = 10)$

Step 1: Partial Derivatives

$$\frac{\partial f}{\partial x} = x$$
$$\frac{\partial f}{\partial y} = 2y$$

Example: Gradient Calculation - Vector Form

Step 2: Gradient Vector

$$\nabla f(x, y) = \begin{bmatrix} x \\ 2y \end{bmatrix}$$

Step 3: Evaluate at $(10, 10)$

$$\nabla f(10, 10) = \begin{bmatrix} 10 \\ 20 \end{bmatrix}$$

Activity: 2D Gradient Descent Practice

Your Turn

Execute two steps of gradient descent starting from $(x_0, y_0) = (2, 3)$ with $\alpha = 0.1$ for:

$$f(x, y) = x^2 + 2y^2 - 4x - 6y + 10$$

Steps to follow:

1. Find the gradient $\nabla f(x, y)$
2. Calculate (x_1, y_1) after first step
3. Calculate (x_2, y_2) after second step

Solution: Step 1 - Gradient Calculation

Gradient of $f(x, y) = x^2 + 2y^2 - 4x - 6y + 10$

Taking partial derivatives:

$$\frac{\partial f}{\partial x} = 2x - 4 \tag{4}$$

$$\frac{\partial f}{\partial y} = 4y - 6 \tag{5}$$

Gradient Vector

$$\nabla f(x, y) = \begin{bmatrix} 2x - 4 \\ 4y - 6 \end{bmatrix}$$

Solution: Step 2 - First Iteration

Starting Point: $(x_0, y_0) = (2, 3)$

Evaluate gradient at $(2, 3)$:

$$\nabla f(2,3) = \begin{bmatrix} 2(2) - 4 \\ 4(3) - 6 \end{bmatrix} = \begin{bmatrix} 0 \\ 6 \end{bmatrix}$$

Update Rule: $\theta_{new} = \theta_{old} - \alpha \nabla f(\theta_{old})$

$$\begin{bmatrix} x_1 \\ y_1 \end{bmatrix} = \begin{bmatrix} 2 \\ 3 \end{bmatrix} - 0.1 \begin{bmatrix} 0 \\ 6 \end{bmatrix} = \begin{bmatrix} 2.0 \\ 2.4 \end{bmatrix}$$

Solution: Step 3 - Second Iteration

Starting Point: $(x_1, y_1) = (2.0, 2.4)$

Evaluate gradient at $(2.0, 2.4)$:

$$\nabla f(2.0, 2.4) = \begin{bmatrix} 2(2.0) - 4 \\ 4(2.4) - 6 \end{bmatrix} = \begin{bmatrix} 0 \\ 3.6 \end{bmatrix}$$

Second Update

$$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} 2.0 \\ 2.4 \end{bmatrix} - 0.1 \begin{bmatrix} 0 \\ 3.6 \end{bmatrix} = \begin{bmatrix} 2.0 \\ 2.04 \end{bmatrix}$$

Why SGD in Reinforcement Learning?

The Challenge

In RL, we want to learn value functions $v_\pi(s)$ or $q_\pi(s, a)$ but:

- We can't store a table for every state
- We need **function approximation**

Why SGD in Reinforcement Learning?

The Challenge

In RL, we want to learn value functions $v_\pi(s)$ or $q_\pi(s, a)$ but:

- We can't store a table for every state
- We need **function approximation**

Function Approximation

Represent value function with parameters: $\hat{v}(s, \mathrm{w})$

- w = weight vector (parameters)

Why SGD in Reinforcement Learning?

The Challenge

In RL, we want to learn value functions $v_\pi(s)$ or $q_\pi(s, a)$ but:

- We can't store a table for every state
- We need **function approximation**

Function Approximation

Represent value function with parameters: $\hat{v}(s, \mathrm{w})$

- w = weight vector (parameters)
- Could be linear: $\hat{v}(s, \mathrm{w}) = \mathrm{w}^T \mathrm{x}(s)$

Why SGD in Reinforcement Learning?

The Challenge

In RL, we want to learn value functions $v_\pi(s)$ or $q_\pi(s, a)$ but:

- We can't store a table for every state
- We need **function approximation**

Function Approximation

Represent value function with parameters: $\hat{v}(s, \mathrm{w})$

- w = weight vector (parameters)
- Could be linear: $\hat{v}(s, \mathrm{w}) = \mathrm{w}^T \mathrm{x}(s)$
- Or non-linear: neural networks, etc.

Why SGD in Reinforcement Learning?

The Challenge

In RL, we want to learn value functions $v_\pi(s)$ or $q_\pi(s, a)$ but:

- We can't store a table for every state
- We need **function approximation**

Function Approximation

Represent value function with parameters: $\hat{v}(s, \mathrm{w})$

- w = weight vector (parameters)
- Could be linear: $\hat{v}(s, \mathrm{w}) = \mathrm{w}^T \mathrm{x}(s)$
- Or non-linear: neural networks, etc.

Goal

Find weights w that make $\hat{v}(s, \mathrm{w}) \approx v_\pi(s)$ for all states

From Supervised Learning to RL

Supervised Learning Setting

- Have training data: $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$
- Know true labels y_i
- Minimize loss: $\sum_i L(f(x_i), y_i)$

From Supervised Learning to RL

Supervised Learning Setting

- Have training data: $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$
- Know true labels y_i
- Minimize loss: $\sum_i L(f(x_i), y_i)$

RL Setting (if we had true values)

- Training data: $(s_1, v_\pi(s_1)), (s_2, v_\pi(s_2)), \ldots$

From Supervised Learning to RL

Supervised Learning Setting

- Have training data: $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$
- Know true labels y_i
- Minimize loss: $\sum_i L(f(x_i), y_i)$

RL Setting (if we had true values)

- Training data: $(s_1, v_\pi(s_1)), (s_2, v_\pi(s_2)), \ldots$
- Minimize: $\sum_i [v_\pi(s_i) - \hat{v}(s_i, \mathrm{w})]^2$

From Supervised Learning to RL

Supervised Learning Setting

- Have training data: $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$
- Know true labels y_i
- Minimize loss: $\sum_i L(f(x_i), y_i)$

RL Setting (if we had true values)

- Training data: $(s_1, v_\pi(s_1)), (s_2, v_\pi(s_2)), \ldots$
- Minimize: $\sum_i [v_\pi(s_i) - \hat{v}(s_i, \mathrm{w})]^2$
- Use gradient descent to find optimal w

From Supervised Learning to RL

Supervised Learning Setting

- Have training data: $(x_1, y_1), (x_2, y_2), \ldots, (x_n, y_n)$
- Know true labels y_i
- Minimize loss: $\sum_i L(f(x_i), y_i)$

RL Setting (if we had true values)

- Training data: $(s_1, v_\pi(s_1)), (s_2, v_\pi(s_2)), \ldots$
- Minimize: $\sum_i [v_\pi(s_i) - \hat{v}(s_i, \mathbf{w})]^2$
- Use gradient descent to find optimal $\mathbf{w}$

The Problem

We don't know the true values $v_\pi(s_i)$! That's what we're trying to learn!

SGD for Value Function Approximation

Assumptions

- Approximate value function $\hat{v}(s, \mathrm{w})$ is differentiable

SGD for Value Function Approximation

Assumptions

- Approximate value function $\hat{v}(s, \mathrm{w})$ is **differentiable**
- Observe samples $s_t \mapsto v_\pi(s_t)$ (state and true value)

Gradient of Squared Error

$$\nabla \left[v_\pi(s_t) - \hat{v}(s_t, \mathrm{w})\right]^2 = -2\left[v_\pi(s_t) - \hat{v}(s_t, \mathrm{w})\right] \nabla \hat{v}(s_t, \mathrm{w})$$

SGD for Value Function Approximation

Assumptions

- Approximate value function $\hat{v}(s, \mathbf{w})$ is **differentiable**
- Observe samples $s_t \mapsto v_\pi(s_t)$ (state and true value)
- States appear according to distribution μ

Gradient of Squared Error

$$\nabla \left[v_\pi(s_t) - \hat{v}(s_t, \mathbf{w})\right]^2 = -2\left[v_\pi(s_t) - \hat{v}(s_t, \mathbf{w})\right] \nabla \hat{v}(s_t, \mathbf{w})$$

SGD Update Rule

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[v_\pi(s_t) - \hat{v}(s_t, \mathbf{w}_t)\right] \nabla \hat{v}(s_t, \mathbf{w}_t)$$

SGD for Value Function Approximation

Gradient of Squared Error

$$\nabla \left[v_\pi(s_t) - \hat{v}(s_t, \mathbf{w})\right]^2 = -2\left[v_\pi(s_t) - \hat{v}(s_t, \mathbf{w})\right] \nabla \hat{v}(s_t, \mathbf{w})$$

SGD Update Rule

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[v_\pi(s_t) - \hat{v}(s_t, \mathbf{w}_t)\right] \nabla \hat{v}(s_t, \mathbf{w}_t)$$

SGD for Value Function Approximation

Gradient of Squared Error

$$\nabla \left[v_\pi(s_t) - \hat{v}(s_t, \mathrm{w}) \right]^2 = -2 \left[v_\pi(s_t) - \hat{v}(s_t, \mathrm{w}) \right] \nabla \hat{v}(s_t, \mathrm{w})$$

SGD Update Rule

$$\mathrm{w}_{t+1} = \mathrm{w}_t + \alpha \left[v_\pi(s_t) - \hat{v}(s_t, \mathrm{w}_t) \right] \nabla \hat{v}(s_t, \mathrm{w}_t)$$

Problem: We don't know $v_\pi(s_t)$!

The SGD Update Rule Explained

Standard SGD Update

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[v_\pi(s_t) - \hat{v}(s_t, \mathbf{w}_t) \right] \nabla \hat{v}(s_t, \mathbf{w}_t)$$

The SGD Update Rule Explained

Standard SGD Update

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[v_\pi(s_t) - \hat{v}(s_t, \mathbf{w}_t) \right] \nabla \hat{v}(s_t, \mathbf{w}_t)$$

Breaking it Down

- $v_\pi(s_t) - \hat{v}(s_t, \mathbf{w}_t) =$ **prediction error**

The SGD Update Rule Explained

Standard SGD Update

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[v_\pi(s_t) - \hat{v}(s_t, \mathbf{w}_t) \right] \nabla \hat{v}(s_t, \mathbf{w}_t)$$

Breaking it Down

- $v_\pi(s_t) - \hat{v}(s_t, \mathbf{w}_t) =$ **prediction error**
- $\nabla \hat{v}(s_t, \mathbf{w}_t) =$ **gradient** (direction of steepest increase)

The SGD Update Rule Explained

Standard SGD Update

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[v_\pi(s_t) - \hat{v}(s_t, \mathbf{w}_t) \right] \nabla \hat{v}(s_t, \mathbf{w}_t)$$

Breaking it Down

- $v_\pi(s_t) - \hat{v}(s_t, \mathbf{w}_t) =$ **prediction error**
- $\nabla \hat{v}(s_t, \mathbf{w}_t) =$ **gradient** (direction of steepest increase)
- $\alpha =$ **learning rate** (step size)

The SGD Update Rule Explained

Standard SGD Update

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[v_\pi(s_t) - \hat{v}(s_t, \mathbf{w}_t) \right] \nabla \hat{v}(s_t, \mathbf{w}_t)$$

Breaking it Down

- $v_\pi(s_t) - \hat{v}(s_t, \mathbf{w}_t) =$ **prediction error**
- $\nabla \hat{v}(s_t, \mathbf{w}_t) =$ **gradient** (direction of steepest increase)
- $\alpha =$ **learning rate** (step size)

Intuition

- If $\hat{v}(s_t, \mathbf{w}_t) < v_\pi(s_t)$: error is positive $\Rightarrow$ increase $\mathbf{w}$ in gradient direction

The SGD Update Rule Explained

Standard SGD Update

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[v_\pi(s_t) - \hat{v}(s_t, \mathbf{w}_t) \right] \nabla \hat{v}(s_t, \mathbf{w}_t)$$

Breaking it Down

- $v_\pi(s_t) - \hat{v}(s_t, \mathbf{w}_t) =$ **prediction error**
- $\nabla \hat{v}(s_t, \mathbf{w}_t) =$ **gradient** (direction of steepest increase)
- $\alpha =$ **learning rate** (step size)

Intuition

- If $\hat{v}(s_t, \mathbf{w}_t) < v_\pi(s_t)$: error is positive $\Rightarrow$ increase $\mathbf{w}$ in gradient direction
- If $\hat{v}(s_t, \mathbf{w}_t) > v_\pi(s_t)$: error is negative $\Rightarrow$ decrease $\mathbf{w}$ in gradient direction

Using Target Approximations

Solution: Unbiased Target Estimates

Replace true value $v_\pi(s_t)$ with unbiased estimate U_t

Using Target Approximations

Solution: Unbiased Target Estimates

Replace true value $v_\pi(s_t)$ with unbiased estimate U_t

General SGD Update

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[U_t - \hat{v}(s_t, \mathbf{w}_t) \right] \nabla \hat{v}(s_t, \mathbf{w}_t)$$

Using Target Approximations

Solution: Unbiased Target Estimates

Replace true value $v_\pi(s_t)$ with unbiased estimate U_t

General SGD Update

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[U_t - \hat{v}(s_t, \mathbf{w}_t) \right] \nabla \hat{v}(s_t, \mathbf{w}_t)$$

Target Choices

- **Monte Carlo**: $U_t = G_t$ (sample return)

Using Target Approximations

Solution: Unbiased Target Estimates

Replace true value $v_\pi(s_t)$ with unbiased estimate U_t

General SGD Update

$$\mathrm{w}_{t+1} = \mathrm{w}_t + \alpha \left[U_t - \hat{v}(s_t, \mathrm{w}_t) \right] \nabla \hat{v}(s_t, \mathrm{w}_t)$$

Target Choices

- **Monte Carlo**: $U_t = G_t$ (sample return)
- **TD(0)**: $U_t = r_{t+1} + \gamma \hat{v}(s_{t+1}, \mathrm{w}_t)$

Using Target Approximations

Solution: Unbiased Target Estimates

Replace true value $v_\pi(s_t)$ with unbiased estimate U_t

General SGD Update

$$\mathrm{w}_{t+1} = \mathrm{w}_t + \alpha \left[U_t - \hat{v}(s_t, \mathrm{w}_t) \right] \nabla \hat{v}(s_t, \mathrm{w}_t)$$

Target Choices

- **Monte Carlo**: $U_t = G_t$ (sample return)
- **TD(0)**: $U_t = r_{t+1} + \gamma \hat{v}(s_{t+1}, \mathrm{w}_t)$

Key Insight

The target U_t must be an unbiased estimate of $v_\pi(s_t)$ for convergence guarantees

Target Approximations: Detailed Explanation

Why Different Targets?

Each target U_t represents a different way to estimate $v_\pi(s_t)$:

Target Approximations: Detailed Explanation

Why Different Targets?

Each target U_t represents a different way to estimate $v_\pi(s_t)$:

Monte Carlo Target: $U_t = G_t$

- $G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots$ (actual return)

Target Approximations: Detailed Explanation

Why Different Targets?

Each target U_t represents a different way to estimate $v_\pi(s_t)$:

Monte Carlo Target: $U_t = G_t$

- $G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots$ (actual return)
- **Pro**: Unbiased estimate of $v_\pi(s_t)$

Target Approximations: Detailed Explanation

Why Different Targets?

Each target U_t represents a different way to estimate $v_\pi(s_t)$:

Monte Carlo Target: $U_t = G_t$

- $G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots$ (actual return)
- **Pro**: Unbiased estimate of $v_\pi(s_t)$
- **Con**: High variance, need complete episodes

Target Approximations: Detailed Explanation

Why Different Targets?

Each target U_t represents a different way to estimate $v_\pi(s_t)$:

Monte Carlo Target: $U_t = G_t$

- $G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots$ (actual return)
- **Pro**: Unbiased estimate of $v_\pi(s_t)$
- **Con**: High variance, need complete episodes

TD(0) Target: $U_t = r_{t+1} + \gamma \hat{v}(s_{t+1}, \mathbf{w}_t)$

- Bootstrap from current estimate of next state

Target Approximations: Detailed Explanation

Why Different Targets?

Each target U_t represents a different way to estimate $v_\pi(s_t)$:

Monte Carlo Target: $U_t = G_t$

- $G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots$ (actual return)
- **Pro**: Unbiased estimate of $v_\pi(s_t)$
- **Con**: High variance, need complete episodes

TD(0) Target: $U_t = r_{t+1} + \gamma \hat{v}(s_{t+1}, \mathrm{w}_t)$

- Bootstrap from current estimate of next state
- **Pro**: Lower variance, online updates

Target Approximations: Detailed Explanation

Why Different Targets?

Each target U_t represents a different way to estimate $v_\pi(s_t)$:

Monte Carlo Target: $U_t = G_t$

- $G_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \dots$ (actual return)
- **Pro**: Unbiased estimate of $v_\pi(s_t)$
- **Con**: High variance, need complete episodes

TD(0) Target: $U_t = r_{t+1} + \gamma \hat{v}(s_{t+1}, \mathbf{w}_t)$

- Bootstrap from current estimate of next state
- **Pro**: Lower variance, online updates
- **Con**: Biased (depends on current approximation)

Convergence Properties

When Does SGD Converge?

For convergence, we need:

1. Target U_t is an **unbiased estimate** of $v_\pi(s_t)$

Convergence Properties

When Does SGD Converge?

For convergence, we need:

1. Target U_t is an **unbiased estimate** of $v_\pi(s_t)$
2. Learning rate conditions: $\sum_t \alpha_t = \infty$ and $\sum_t \alpha_t^2 < \infty$

Convergence Properties

When Does SGD Converge?

For convergence, we need:

1. Target U_t is an **unbiased estimate** of $v_\pi(s_t)$
2. Learning rate conditions: $\sum_t \alpha_t = \infty$ and $\sum_t \alpha_t^2 < \infty$
3. Function approximator is **differentiable**

Convergence Properties

When Does SGD Converge?

For convergence, we need:

1. Target U_t is an **unbiased estimate** of $v_\pi(s_t)$
2. Learning rate conditions: $\sum_t \alpha_t = \infty$ and $\sum_t \alpha_t^2 < \infty$
3. Function approximator is **differentiable**

Convergence Guarantees

- **Monte Carlo**: Converges to global optimum (unbiased target)

Convergence Properties

When Does SGD Converge?

For convergence, we need:

1. Target U_t is an **unbiased estimate** of $v_\pi(s_t)$
2. Learning rate conditions: $\sum_t \alpha_t = \infty$ and $\sum_t \alpha_t^2 < \infty$
3. Function approximator is **differentiable**

Convergence Guarantees

- **Monte Carlo**: Converges to global optimum (unbiased target)
- **TD methods**: May not converge (biased target)

Convergence Properties

When Does SGD Converge?

For convergence, we need:

1. Target U_t is an **unbiased estimate** of $v_\pi(s_t)$
2. Learning rate conditions: $\sum_t \alpha_t = \infty$ and $\sum_t \alpha_t^2 < \infty$
3. Function approximator is **differentiable**

Convergence Guarantees

- **Monte Carlo**: Converges to global optimum (unbiased target)
- **TD methods**: May not converge (biased target)
- **Practice**: TD methods often work well despite theory

Convergence Properties

When Does SGD Converge?

For convergence, we need:

1. Target U_t is an **unbiased estimate** of $v_\pi(s_t)$
2. Learning rate conditions: $\sum_t \alpha_t = \infty$ and $\sum_t \alpha_t^2 < \infty$
3. Function approximator is **differentiable**

Convergence Guarantees

- **Monte Carlo**: Converges to global optimum (unbiased target)
- **TD methods**: May not converge (biased target)
- **Practice**: TD methods often work well despite theory

Key Takeaway

There's a bias-variance tradeoff in choosing targets for SGD

Gradient Monte Carlo Method

Algorithm Gradient Monte Carlo for Value Function Approximation

Input: Policy π, number of episodes N, step size $\alpha \in (0, 1]$
Initialize: $\mathrm{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathrm{w} = 0$)
for N episodes **do**
    Generate episode using π: $S_0, A_0, R_1, S_1, \ldots, S_{T-1}, A_{T-1}, R_T$
    for each step $t = 0, 1, \ldots, T-1$ **do**
        $\mathrm{w} \leftarrow \mathrm{w} + \alpha[G_t - \hat{v}(S_t, \mathrm{w})]\nabla \hat{v}(S_t, \mathrm{w})$
    end for
end for

Gradient Monte Carlo Method

Algorithm Gradient Monte Carlo for Value Function Approximation

```
Input: Policy π, number of episodes N, step size α ∈ (0, 1]
Initialize: w ∈ ℝ^d arbitrarily (e.g., w = 0)
for N episodes do
    Generate episode using π: S_0, A_0, R_1, S_1, ..., S_{T−1}, A_{T−1}, R_T
    for each step t = 0, 1, ..., T − 1 do
        w ← w + α[G_t − v̂(S_t, w)]∇v̂(S_t, w)
    end for
end for
```

Key Features

- Uses actual returns G_t as targets

Gradient Monte Carlo Method

Algorithm Gradient Monte Carlo for Value Function Approximation

Input: Policy π, number of episodes N, step size $\alpha \in (0, 1]$
Initialize: $\mathrm{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathrm{w} = 0$)
for N episodes **do**
    Generate episode using π: $S_0, A_0, R_1, S_1, \ldots, S_{T-1}, A_{T-1}, R_T$
    for each step $t = 0, 1, \ldots, T-1$ **do**
        $\mathrm{w} \leftarrow \mathrm{w} + \alpha[G_t - \hat{v}(S_t, \mathrm{w})]\nabla \hat{v}(S_t, \mathrm{w})$
    end for
end for

Key Features

- Uses actual returns G_t as targets
- Unbiased - guaranteed convergence to local optimum

Gradient Monte Carlo Method

Algorithm Gradient Monte Carlo for Value Function Approximation

Input: Policy π, number of episodes N, step size $\alpha \in (0, 1]$
Initialize: $\mathrm{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathrm{w} = 0$)
for N episodes **do**
  Generate episode using π: $S_0, A_0, R_1, S_1, \ldots, S_{T-1}, A_{T-1}, R_T$
  for each step $t = 0, 1, \ldots, T-1$ **do**
    $\mathrm{w} \leftarrow \mathrm{w} + \alpha[G_t - \hat{v}(S_t, \mathrm{w})]\nabla \hat{v}(S_t, \mathrm{w})$
  end for
end for

Key Features

- Uses actual returns G_t as targets
- Unbiased - guaranteed convergence to local optimum

Semi-Gradient TD(0)

Algorithm Semi-Gradient TD(0)

Input: Policy π, step size $\alpha \in (0, 1]$
Initialize: $\mathrm{w} \in \mathbb{R}^d$ arbitrarily
repeat
    Initialize S
    repeat
        $A \leftarrow \pi(\cdot|S)$
        Take action A, observe R, S'
        $\mathrm{w} \leftarrow \mathrm{w} + \alpha[R + \gamma \hat{v}(S', \mathrm{w}) - \hat{v}(S, \mathrm{w})]\nabla \hat{v}(S, \mathrm{w})$
        $S \leftarrow S'$
    until S is terminal
until convergence

Semi-Gradient TD(0)

Why "Semi-Gradient"?

Semi-Gradient TD(0)

Why "Semi-Gradient"?

- Target includes $\hat{v}(S', \mathbf{w})$ which depends on $\mathbf{w}$

Semi-Gradient TD(0)

Why "Semi-Gradient"?

- Target includes $\hat{v}(S', \mathbf{w})$ which depends on $\mathbf{w}$
- We ignore gradient w.r.t. target: **not true gradient**

Semi-Gradient TD(0)

Why "Semi-Gradient"?

- Target includes $\hat{v}(S', \mathbf{w})$ which depends on $\mathbf{w}$
- We ignore gradient w.r.t. target: **not true gradient**
- Still converges but to different solution than true gradient

From State Values to Action Values

Extending to Control

- Need action-value function: $\hat{q}(s, a, \mathbf{w})$ instead of $\hat{v}(s, \mathbf{w})$

From State Values to Action Values

Extending to Control

- Need action-value function: $\hat{q}(s, a, \mathrm{w})$ instead of $\hat{v}(s, \mathrm{w})$
- Same principles apply with weight vector w

General Update for Action Values

$$\mathrm{w}_{t+1} = \mathrm{w}_t + \alpha \left[U_t - \hat{q}(S_t, A_t, \mathrm{w}_t) \right] \nabla \hat{q}(S_t, A_t, \mathrm{w}_t)$$

From State Values to Action Values

Extending to Control

- Need action-value function: $\hat{q}(s, a, \mathrm{w})$ instead of $\hat{v}(s, \mathrm{w})$
- Same principles apply with weight vector w

General Update for Action Values

$$\mathrm{w}_{t+1} = \mathrm{w}_t + \alpha \left[U_t - \hat{q}(S_t, A_t, \mathrm{w}_t) \right] \nabla \hat{q}(S_t, A_t, \mathrm{w}_t)$$

SARSA Target

For one-step SARSA: $U_t = R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathrm{w}_t)$

From State Values to Action Values

Extending to Control

- Need action-value function: $\hat{q}(s, a, \mathbf{w})$ instead of $\hat{v}(s, \mathbf{w})$
- Same principles apply with weight vector $\mathbf{w}$

General Update for Action Values

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[U_t - \hat{q}(S_t, A_t, \mathbf{w}_t) \right] \nabla \hat{q}(S_t, A_t, \mathbf{w}_t)$$

SARSA Target

For one-step SARSA: $U_t = R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t)$

Policy Improvement

- Compute $\hat{q}(s, a, \mathbf{w})$ for all actions a

From State Values to Action Values

Extending to Control

- Need action-value function: $\hat{q}(s, a, \mathbf{w})$ instead of $\hat{v}(s, \mathbf{w})$
- Same principles apply with weight vector $\mathbf{w}$

General Update for Action Values

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[U_t - \hat{q}(S_t, A_t, \mathbf{w}_t) \right] \nabla \hat{q}(S_t, A_t, \mathbf{w}_t)$$

SARSA Target

For one-step SARSA: $U_t = R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \mathbf{w}_t)$

Policy Improvement

- Compute $\hat{q}(s, a, \mathbf{w})$ for all actions a
- Use ϵ-greedy: $a^* = \arg\max_a \hat{q}(s, a, \mathbf{w})$

Semi-Gradient SARSA Algorithm

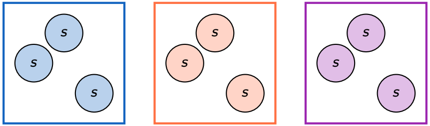
Input: $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^d \rightarrow \mathbb{R}$, step size α, $\epsilon > 0$
Initialize: $\mathrm{w} \in \mathbb{R}^d$ arbitrarily
repeat
    Initialize S
    $A \leftarrow \epsilon$-greedy$(\hat{q}(S, \cdot, \mathrm{w}))$
    repeat
        Take action A, observe R, S'
        if S' is terminal **then**
            $\mathrm{w} \leftarrow \mathrm{w} + \alpha[R - \hat{q}(S, A, \mathrm{w})]\nabla\hat{q}(S, A, \mathrm{w})$
            break
        end if
        $A' \leftarrow \epsilon$-greedy$(\hat{q}(S', \cdot, \mathrm{w}))$
        $\mathrm{w} \leftarrow \mathrm{w} + \alpha[R + \gamma\hat{q}(S', A', \mathrm{w}) - \hat{q}(S, A, \mathrm{w})]\nabla\hat{q}(S, A, \mathrm{w})$
        $S \leftarrow S'$, $A \leftarrow A'$

State Aggregation

Basic Idea

- Group **similar states** together

State Groups



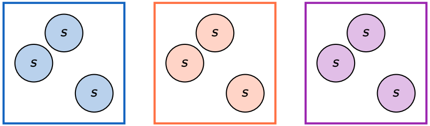
$\hat{v} = w_1$ $\hat{v} = w_2$ $\hat{v} = w_3$

State Aggregation

Basic Idea

- Group **similar states** together
- Assign **same value** to states in same group

State Groups



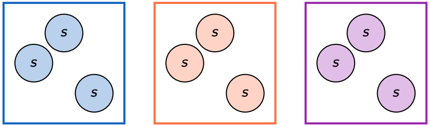
$\hat{v} = w_1$ $\hat{v} = w_2$ $\hat{v} = w_3$

State Aggregation

Basic Idea

- Group **similar states** together
- Assign **same value** to states in same group
- Simplest form of function approximation

State Groups



$\hat{v} = w_1$ $\hat{v} = w_2$ $\hat{v} = w_3$

State Aggregation

Basic Idea

- Group **similar states** together
- Assign **same value** to states in same group
- Simplest form of function approximation

Limitations

- Works only for **simple problems**

State Aggregation

Basic Idea

- Group **similar states** together
- Assign **same value** to states in same group
- Simplest form of function approximation

Limitations

- Works only for **simple problems**
- Hard to define good groupings

State Aggregation

Basic Idea

- Group **similar states** together
- Assign **same value** to states in same group
- Simplest form of function approximation

Limitations

- Works only for **simple problems**
- Hard to define good groupings
- No generalization within groups

Linear Methods

Linear Approximation

$$\hat{v}(s, \mathbf{w}) = \mathbf{w}^T \mathbf{x}(s) = \sum_{i=1}^{d} w_i x_i(s)$$

where $\mathbf{x}(s) = (x_1(s), x_2(s), \ldots, x_d(s))^T$ is the feature vector

Linear Methods

Linear Approximation

$$\hat{v}(s, \mathbf{w}) = \mathbf{w}^T \mathbf{x}(s) = \sum_{i=1}^{d} w_i x_i(s)$$

where $\mathbf{x}(s) = (x_1(s), x_2(s), \ldots, x_d(s))^T$ is the feature vector

Key Advantage: Simple Gradient

$$\nabla \hat{v}(s, \mathbf{w}) = \mathbf{x}(s)$$

Linear Methods

Linear Approximation

$$\hat{v}(s, \mathbf{w}) = \mathbf{w}^T \mathbf{x}(s) = \sum_{i=1}^{d} w_i x_i(s)$$

where $\mathbf{x}(s) = (x_1(s), x_2(s), \ldots, x_d(s))^T$ is the **feature vector**

Key Advantage: Simple Gradient

$$\nabla \hat{v}(s, \mathbf{w}) = \mathbf{x}(s)$$

Linear SGD Update

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[U_t - \hat{v}(S_t, \mathbf{w}_t) \right] \mathbf{x}(S_t)$$

Linear Methods

Linear Approximation

$$\hat{v}(s, \mathbf{w}) = \mathbf{w}^T \mathbf{x}(s) = \sum_{i=1}^{d} w_i x_i(s)$$

where $\mathbf{x}(s) = (x_1(s), x_2(s), \ldots, x_d(s))^T$ is the feature vector

Key Advantage: Simple Gradient

$$\nabla \hat{v}(s, \mathbf{w}) = \mathbf{x}(s)$$

Linear SGD Update

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[U_t - \hat{v}(S_t, \mathbf{w}_t) \right] \mathbf{x}(S_t)$$

Feature Construction Example

Problem Setup

State has two dimensions: $s = (s_1, s_2)$ where $s_1, s_2 \in \mathbb{R}$

Feature Construction Example

Problem Setup

State has two dimensions: $s = (s_1, s_2)$ where $s_1, s_2 \in \mathbb{R}$

Simple Linear Features

$\mathrm{x}(s) = [s_1, s_2]^T$

Feature Construction Example

Problem Setup

State has two dimensions: $s = (s_1, s_2)$ where $s_1, s_2 \in \mathbb{R}$

Simple Linear Features

$\mathbf{x}(s) = [s_1, s_2]^T$

- **Problem**: Cannot represent interactions between s_1 and s_2

Feature Construction Example

Problem Setup

State has two dimensions: $s = (s_1, s_2)$ where $s_1, s_2 \in \mathbb{R}$

Simple Linear Features

$\mathbf{x}(s) = [s_1, s_2]^T$

- **Problem**: Cannot represent interactions between s_1 and s_2
- **Problem**: If $s_1 = s_2 = 0$, then $\hat{v}(s, \mathbf{w}) = 0$ always

Feature Construction Example

Problem Setup

State has two dimensions: $s = (s_1, s_2)$ where $s_1, s_2 \in \mathbb{R}$

Simple Linear Features

$\mathbf{x}(s) = [s_1, s_2]^T$

- **Problem**: Cannot represent interactions between s_1 and s_2
- **Problem**: If $s_1 = s_2 = 0$, then $\hat{v}(s, \mathbf{w}) = 0$ always

Polynomial Features

$\mathbf{x}(s) = [1, s_1, s_2, s_1 s_2]^T$

Feature Construction Example

Problem Setup

State has two dimensions: $s = (s_1, s_2)$ where $s_1, s_2 \in \mathbb{R}$

Simple Linear Features

$\mathrm{x}(s) = [s_1, s_2]^T$

- **Problem**: Cannot represent interactions between s_1 and s_2
- **Problem**: If $s_1 = s_2 = 0$, then $\hat{v}(s, \mathrm{w}) = 0$ always

Polynomial Features

$\mathrm{x}(s) = [1, s_1, s_2, s_1 s_2]^T$

- **Benefit**: Can represent interactions

Feature Construction Example

Problem Setup

State has two dimensions: $s = (s_1, s_2)$ where $s_1, s_2 \in \mathbb{R}$

Simple Linear Features

$\mathrm{x}(s) = [s_1, s_2]^T$

- **Problem**: Cannot represent interactions between s_1 and s_2
- **Problem**: If $s_1 = s_2 = 0$, then $\hat{v}(s, \mathrm{w}) = 0$ always

Polynomial Features

$\mathrm{x}(s) = [1, s_1, s_2, s_1 s_2]^T$

- **Benefit**: Can represent interactions

Feature Construction Example

Problem Setup

State has two dimensions: $s = (s_1, s_2)$ where $s_1, s_2 \in \mathbb{R}$

Simple Linear Features

$\mathbf{x}(s) = [s_1, s_2]^T$

- **Problem**: Cannot represent interactions between s_1 and s_2
- **Problem**: If $s_1 = s_2 = 0$, then $\hat{v}(s, \mathbf{w}) = 0$ always

Polynomial Features

$\mathbf{x}(s) = [1, s_1, s_2, s_1 s_2]^T$

- **Benefit**: Can represent interactions

Higher-Order Polynomial

$\mathbf{x}(s) = [1, s_1, s_2, s_1 s_2, s_1^2, s_2^2, s_1^2 s_2, s_1 s_2^2, s_1^2 s_2^2]^T$