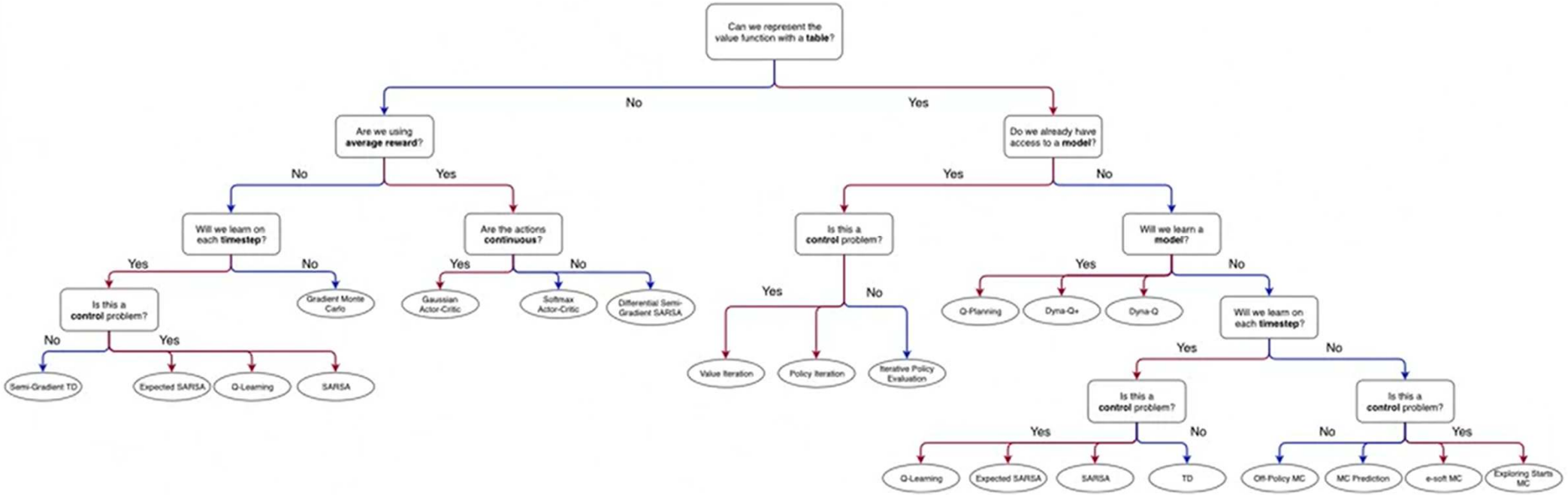
On-policy Prediction with Approximation

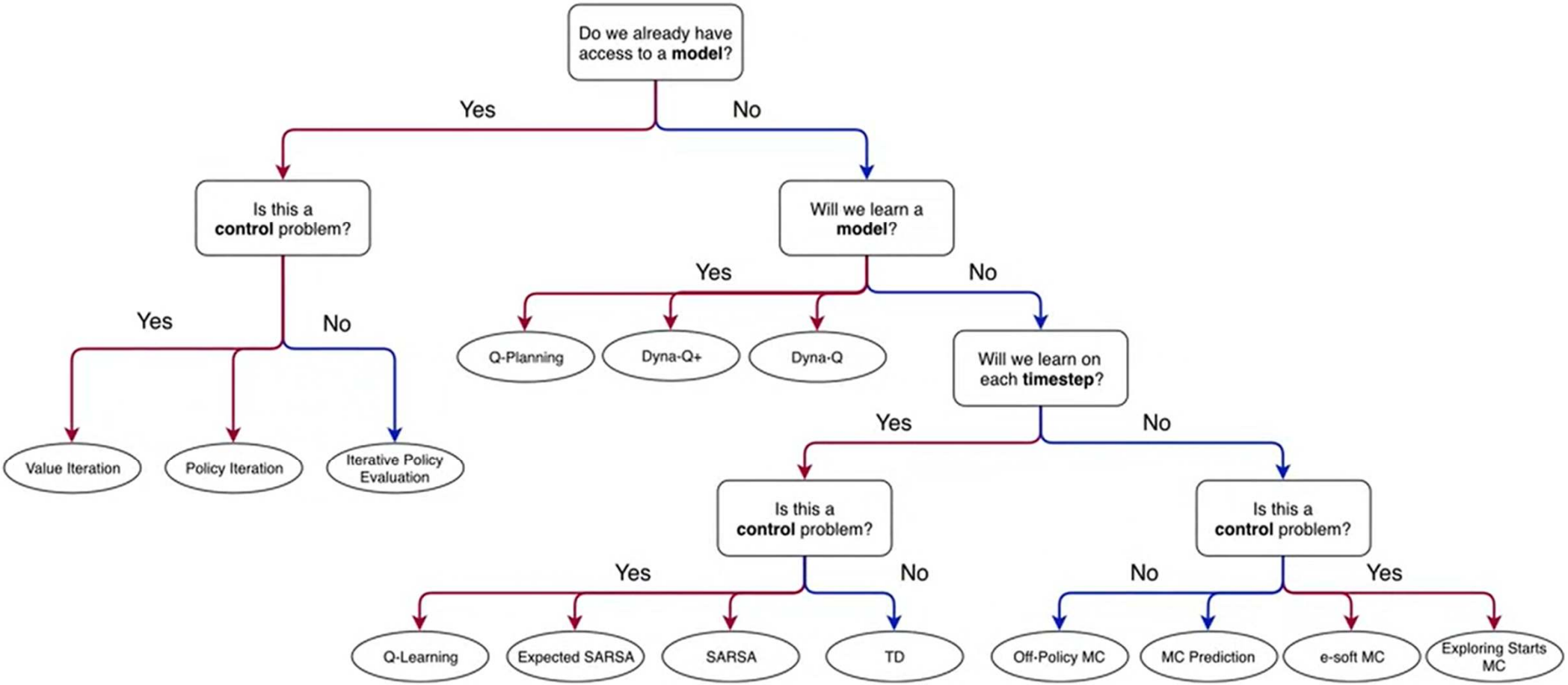
By Rohit Pardasani

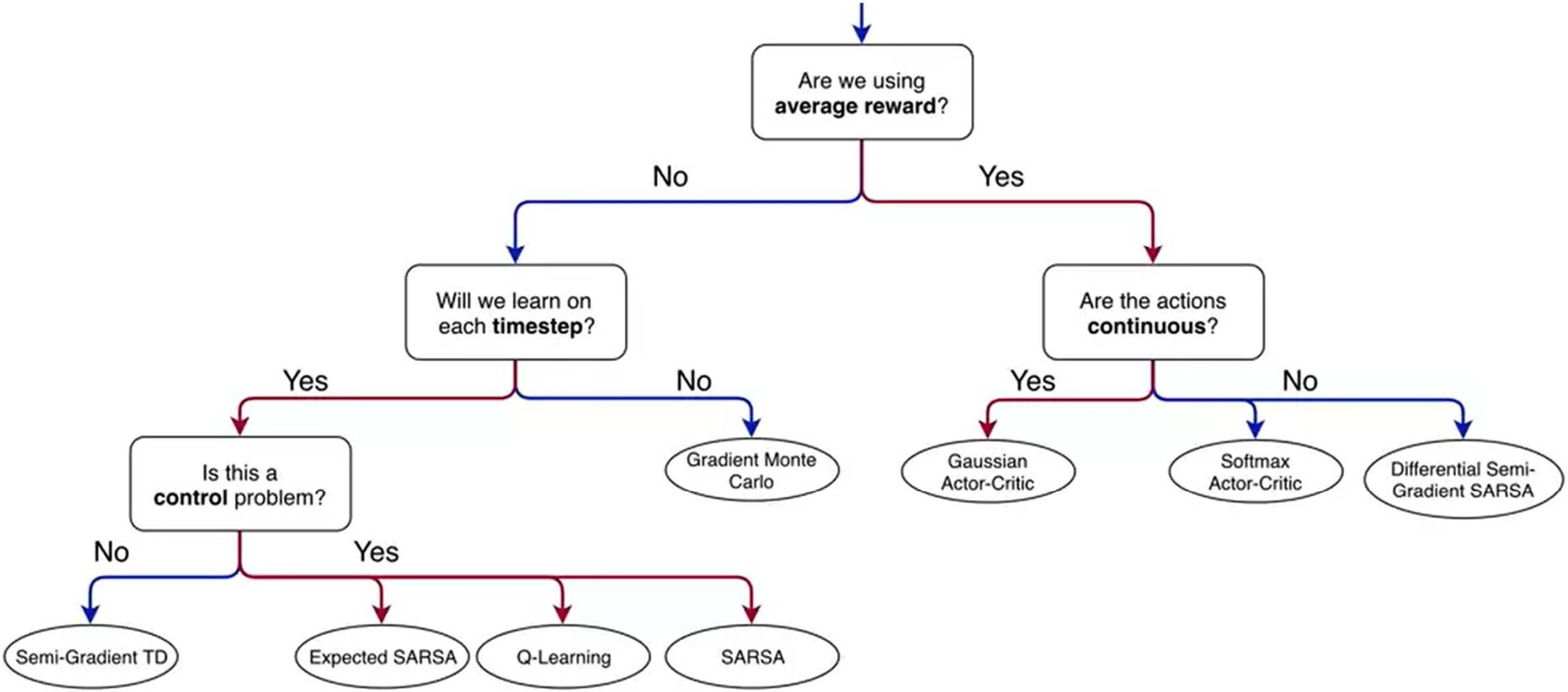
References:

https://mitpress.mit.edu/books/reinforcement-learning

https://www.coursera.org/specializations/reinforcement-learning


Can we represent the value function with a **table**?
No
Yes
Are we using **average reward**?
Do we already have access to a **model**?
No
Yes
Yes
No
Will we learn on each **timestep**?
Are the actions **continuous**?
Is this a **control** problem?
Will we learn a **model**?
Yes
No
Yes
No
Yes
No
Yes
No
Gradient Monte Carlo
Gaussian Actor-Critic
Softmax Actor-Critic
Differential Semi-Gradient SARSA
Value Iteration
Policy Iteration
Iterative Policy Evaluation
Q-Planning
Dyna-Q+
Dyna-Q
Will we learn on each **timestep**?
Is this a **control** problem?
No
Yes
Yes
No
Yes
No
Semi-Gradient TD
Expected SARSA
Q-Learning
SARSA
Q-Learning
Expected SARSA
SARSA
TD
Is this a **control** problem?
No
Yes
Off-Policy MC
MC Prediction
e-soft MC
Exploring Starts MC


Do we already have access to a **model**?
Yes
No
Is this a **control** problem?
Will we learn a **model**?
Yes
No
Yes
No
Value Iteration
Policy Iteration
Iterative Policy Evaluation
Q-Planning
Dyna-Q+
Dyna-Q
Will we learn on each **timestep**?
Yes
No
Is this a **control** problem?
Is this a **control** problem?
Yes
No
No
Yes
Q-Learning
Expected SARSA
SARSA
TD
Off-Policy MC
MC Prediction
e-soft MC
Exploring Starts MC

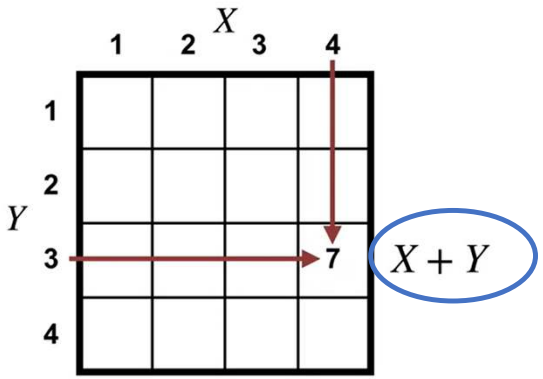

Are we using **average reward**?
No
Yes
Will we learn on each **timestep**?
Are the actions **continuous**?
Yes
No
Yes
No
Is this a **control** problem?
Gradient Monte Carlo
Gaussian Actor-Critic
Softmax Actor-Critic
Differential Semi-Gradient SARSA
No
Yes
Semi-Gradient TD
Expected SARSA
Q-Learning
SARSA

Moving to Parametrized Functions

Up until now, we've learned about tabular methods.
Methods that store table containing separately learn values, for each possible state.
In real-world problems, these tables will become intractably large.
Imagine a robot that sees the world through a camera.
We clearly cannot store a table entry for every possible image.
Luckily, this is not the only possibility.
We'll talk about more general ways to approximate value function.

A value function can be represented in many different ways

State	Value
s_1	-4
s_2	6
s_3	12
s_4	5
s_5	53
...	
s_{16}	-9



In the second case we have represented value function as sum of features.
This may not be the correct value function, but it's just an illustration of the way value function can be represented as **function of features of states**

Parameterizing the Value Function

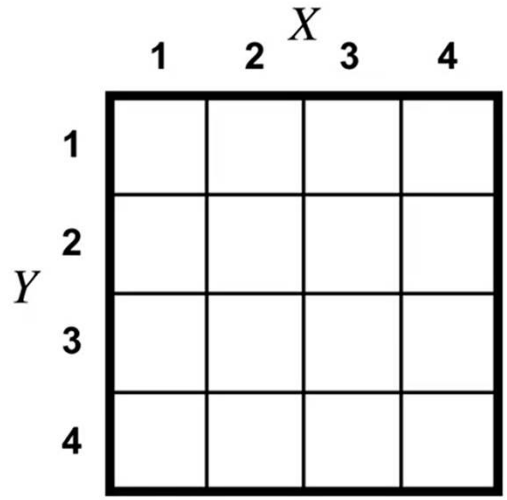
$$\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$$

weights

Example of Parameterized Value Function

$$\hat{v}(s, \mathbf{w}) \doteq w_1 X + w_2 Y$$

We only have to store the two weights



How Changes to the Weight Impact the Value Function

$w_1 = 1$
$w_2 = 1$

Y \ X	1	2	3	4
1	2	3	4	5
2	3	4	5	6
3	4	5	6	7
4	5	6	7	8

$w_1 = 2$
$w_2 = 1$

Y \ X	1	2	3	4
1	3	5	7	9
2	4	6	8	10
3	5	7	9	11
4	6	8	10	12

The tabular representation modifying the value estimates for one state, leaves the others unchanged. This is no longer the case with a parameterized function.

Linear Value Function Approximation

Features

$$\hat{v}(s, \mathbf{w}) \doteq \sum w_i x_i(s)$$

$$= < \mathbf{w}, \mathbf{x}(s) >$$

Limitations of Linear Value Function Approximation

$$\hat{v}(s, \mathbf{w}) \doteq \sum w_i x_i(s)$$

Y \ X	1	2	3	4
1	0	0	0	0
2	0	5	5	0
3	0	5	5	0
4	0	0	0	0

In this case we cannot approximate value function as a linear function of X and Y. Basically, X and Y are not good features to represent value function approximation.
We need to have better features.

Tabular value functions are linear functions

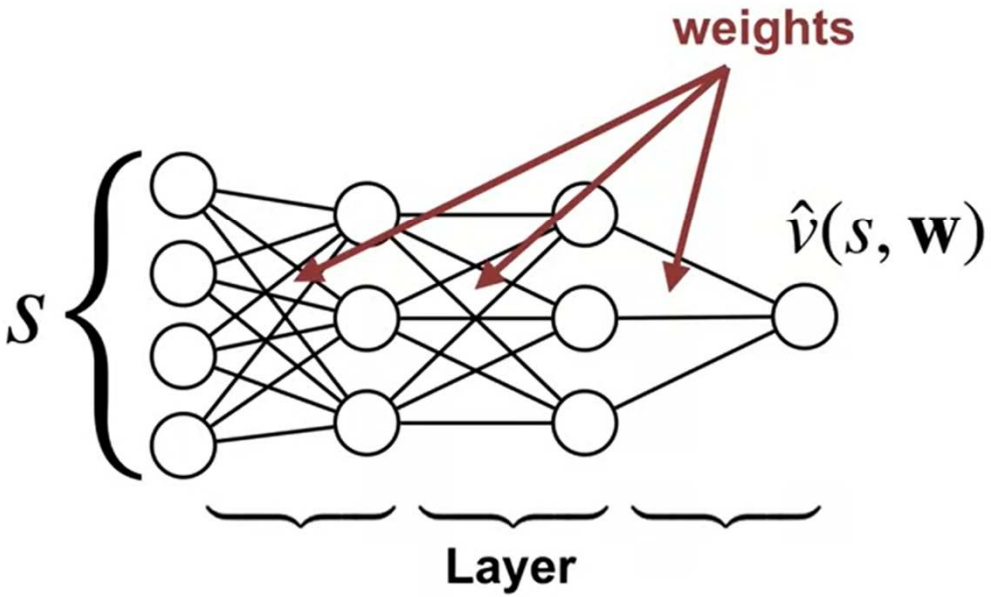
State	Value
s_1	w_1
s_2	w_2
s_3	w_3
$\cdots$	$\cdots$
s_i	w_i
$\cdots$	$\cdots$
s_{16}	w_{16}

$$\hat{v}(s_i, \mathbf{w}) \doteq < \mathbf{w}, \mathbf{x}(s_i) >$$
$$= w_i$$

$$\mathbf{x}(s_i) = \begin{bmatrix} 0 \\ 0 \\ \cdots \\ 0 \\ 1 \\ 0 \\ \cdots \\ 0 \end{bmatrix}$$

i-th element

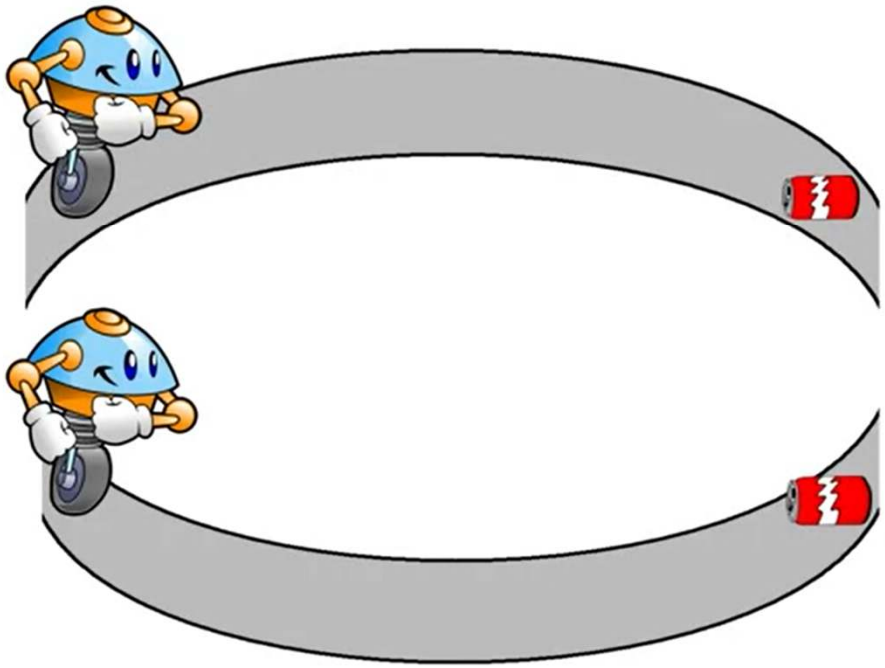
Nonlinear Function Approximation



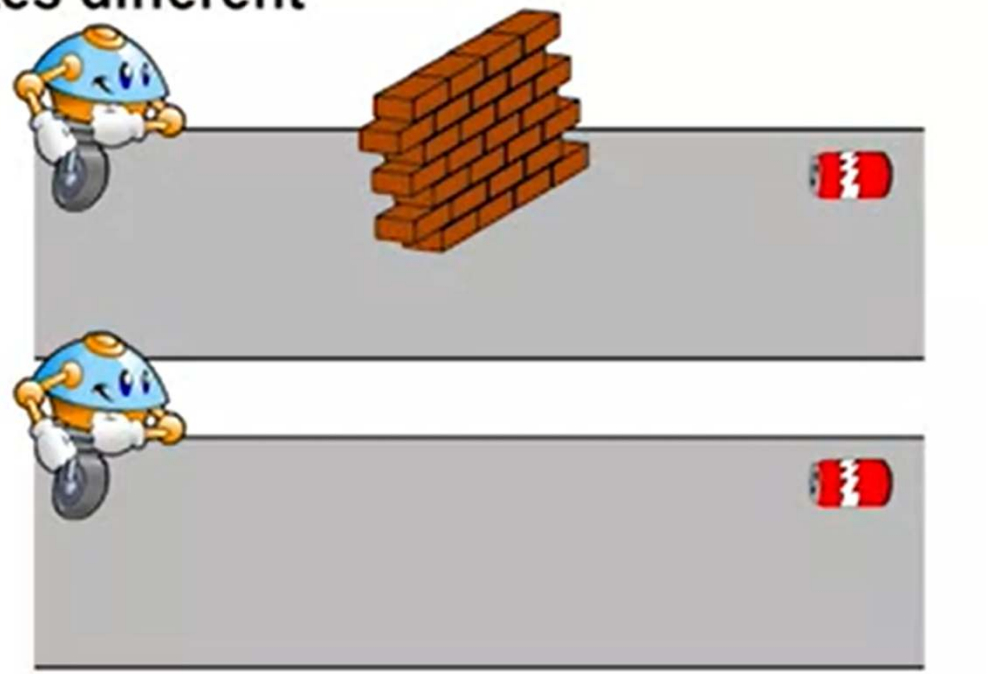
Neural Network

Generalization: Updates to One State Affect the Value of Other States

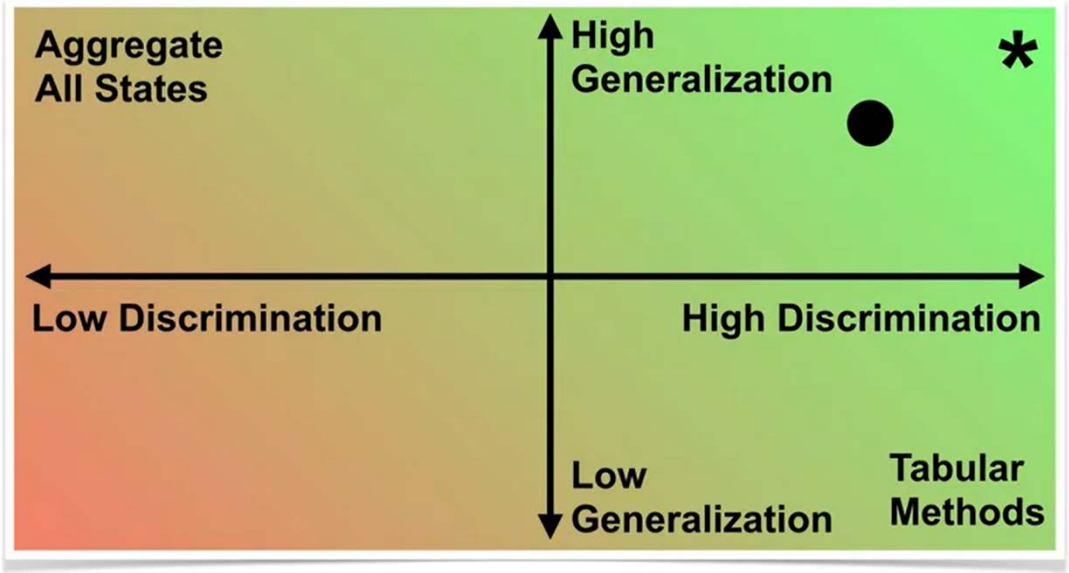
State	Value
s_1	-3
s_2	-2
s_3	2
s_4	-1.7
s_5	4



Discrimination: The ability to make the value of two states different



Categorizing Methods Based on Generalization and Discrimination

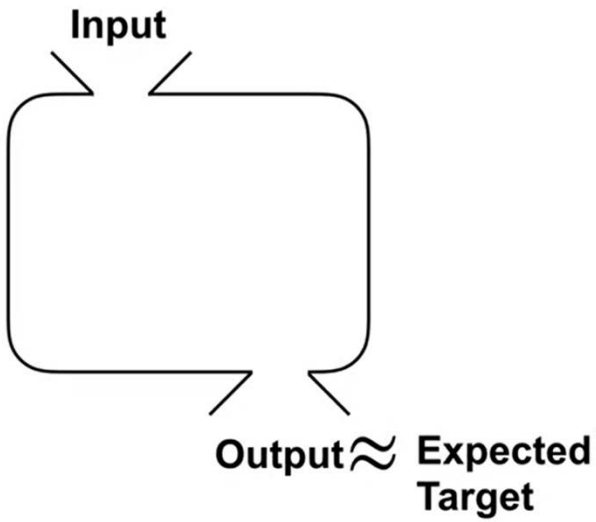


- Tabular representations have provided good discrimination, but no generalization
- Generalization is important for faster learning
- Having both generalization and discrimination is ideal

Supervised learning methods can be useful for handling parts of the reinforcement learning problem. Value estimation can be framed as a supervised learning problem.

Supervised Learning

{(Input=(2 bed, 103 m², …), Target=$300000),
(Input=(1 bed, 80 m², …), Target=$170000),
…,
(Input=(2 bed, 103 m², …), Target=$515000)}



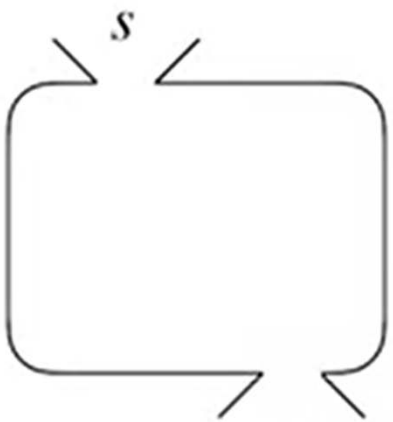
Framing Policy Evaluation as Supervised Learning

{(S_1, G_1),
(S_2, G_2),
(S_3, G_3),
...}

Monte-Carlo

{(S_1, $R_2 + \gamma \hat{v}(S_2, \mathbf{w})$),
(S_2, $R_3 + \gamma \hat{v}(S_3, \mathbf{w})$),
(S_3, $R_4 + \gamma \hat{v}(S_4, \mathbf{w})$),
...}

TD



Value Estimate $\approx v_\pi(s)$

In principle, any function approximation technique from supervised learning can be applied to the policy evaluation task.
However, not all are equally well-suited.
In reinforcement learning, an agent interacts with an environment and continually generates new data.
This is often called the online setting.
To distinguish it from the offline setting where the full dataset is available from the start and remains fixed throughout learning.
If we want to use a function approximation technique, we should make sure it can work in the online setting.
Some methods are not compatible with the online setting because they are either designed for a fixed batch of data or there are not designed for temporally correlated data, and the data in reinforcement learning is always correlated.
TD methods introduce an additional complication when applying techniques from supervised learning.
TD methods use bootstrapping, meaning that our targets now depend on our own estimates.
These estimates change as learning progresses and so our targets continually change.
This is different than supervised learning where we have access to a ground truth label as the target.

The Function Approximator Should be Compatible with Online Updates

$$(S_1, G_1),\ (S_2, G_2),\ (S_3, G_3),\ (S_4, G_4),\ (S_5, G_5),\ \ldots$$

The Function Approximator should be Compatible with Bootstrapping

Target depends on w

$\{(S_1, R_2 + \gamma\hat{v}(S_2, \mathbf{w})),$

$(S_2, R_3 + \gamma\hat{v}(S_3, \mathbf{w})),$ TD

$(S_3, R_4 + \gamma\hat{v}(S_4, \mathbf{w})), \ldots\}$

Target is fixed and given

{((2 bed, 103 m², …), $300000),

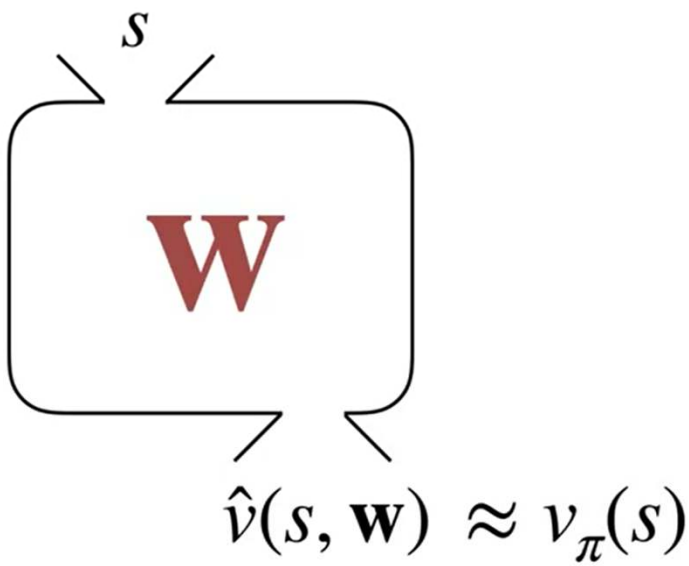
((1 bed, 80 m², …), $170000), Supervised Learning

…,

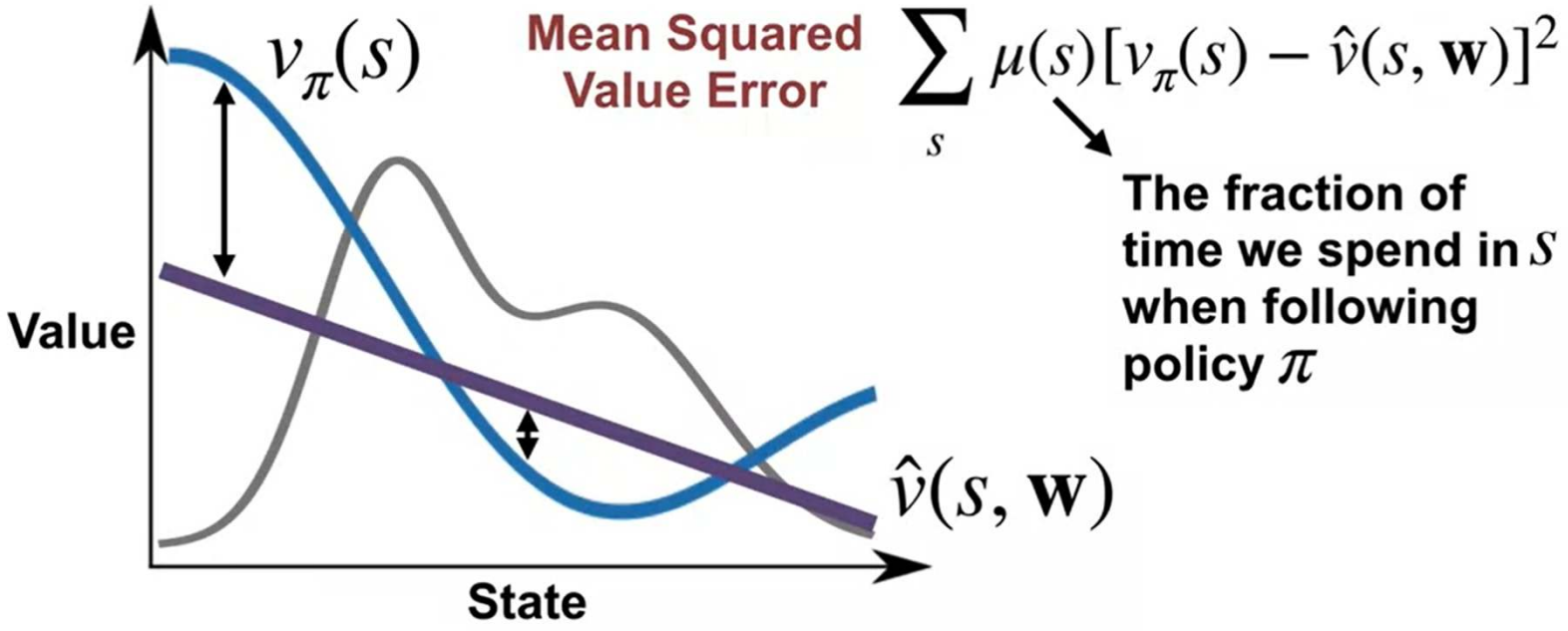
((2 bed, 103 m², …), $515000)}

An Idealized Scenario

$\{(S_1, v_\pi(S_1)), (S_2, v_\pi(S_2)), (S_3, v_\pi(S_3)), \ldots\}$



The Mean Squared Value Error Objective



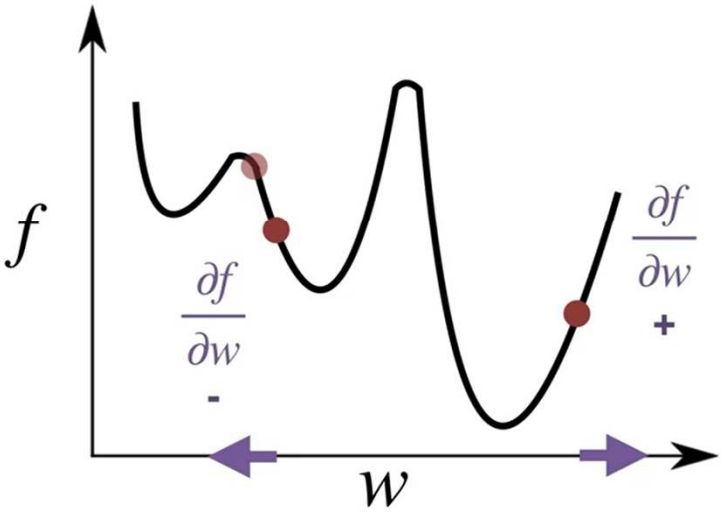
Grey curve gives μ(s), which is a probability distribution

Adapting the Weights to Minimize the Mean Squared Value Error Objective

$$\overline{VE} = \sum_s \mu(s)[v_\pi(s) - \hat{v}(s, \mathbf{w})]^2$$

$$\mathbf{w} = \begin{bmatrix} w_1 \\ w_2 \\ \cdots \\ w_d \end{bmatrix}$$

Understanding Derivatives



Gradient: Derivatives in Multiple Dimensions

$$\mathbf{w} \doteq \begin{bmatrix} w_1 \\ w_2 \\ \cdots \\ w_d \end{bmatrix} \qquad \nabla f \doteq \begin{bmatrix} \frac{\partial f}{\partial w_1} \\ \frac{\partial f}{\partial w_2} \\ \cdots \\ \frac{\partial f}{\partial w_d} \end{bmatrix}$$

The direction to change w_2 in order to increase f

How quickly f changes

The gradient gives the direction of steepest ascent

Example: Gradient of a Linear Value Function

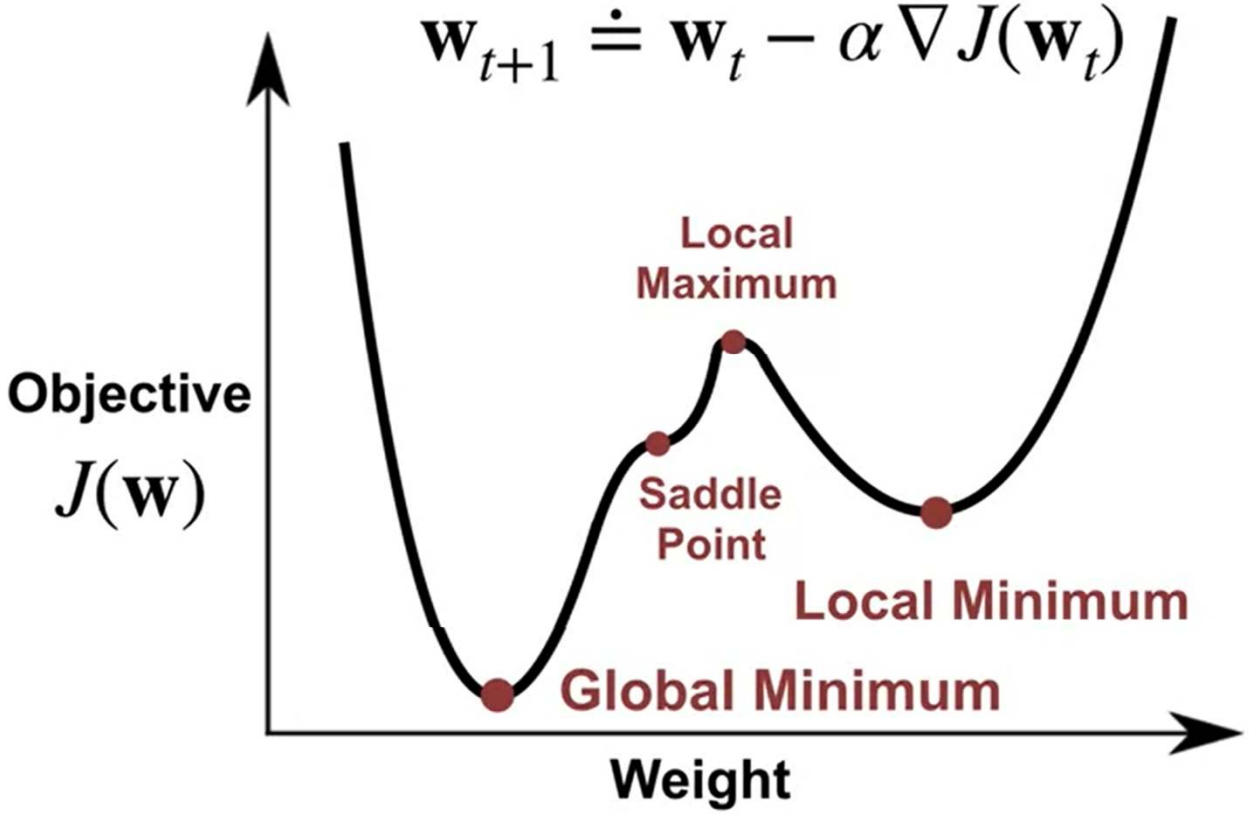
$$\hat{v}(s, \mathbf{w}) \doteq \sum w_i x_i(s)$$

$$\frac{\partial \hat{v}(s, \mathbf{w})}{\partial w_i} = x_i(s)$$

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

Gradient Descent

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



Global Minima and Solution Quality

$$\hat{v} \neq v_\pi$$

$$\mathbf{x}(s) \doteq [1]$$

Note that a global minimum does not necessarily correspond to the true value function.
It is limited by our choice of function parameterization and depends on our choice of objective.
Imagine a feature vector which just contains a single element that is always one, no matter what state you are in.
The approximate value function that minimizes mean squared value error, will converge to the average value over all the states.
This is not a very good value function.
However, this is still the best we can do under that parameterization in terms of the value error objective.

Gradient of the Mean Squared Value Error Objective

$$\nabla \sum_{s \in \mathcal{S}} \mu(s)[v_\pi(s) - \hat{v}(s, \mathbf{w})]^2$$

$$= \sum_{s \in \mathcal{S}} \mu(s) \nabla [v_\pi(s) - \hat{v}(s, \mathbf{w})]^2$$

$$= - \sum_{s \in \mathcal{S}} \mu(s) 2 \underbrace{[v_\pi(s) - \hat{v}(s, \mathbf{w})]}_{\text{+ / -}} \nabla \hat{v}(s, \mathbf{w})$$

$$\Delta \mathbf{w} \propto \sum_{s \in \mathcal{S}} \mu(s)[v_\pi(s) - \hat{v}(s, \mathbf{w})] \nabla \hat{v}(s, \mathbf{w})$$

$$\hat{v}(s, \mathbf{w}) \doteq < \mathbf{w}, \mathbf{x}(s) >$$

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

If the difference is positive, it means the true value is higher than our estimate, so we should change the weights in the direction that increases our estimate.

If the current error is negative, we should change the weights in the opposite direction.

Computing the gradient for the mean squared value error requires summing over all states.
This is generally not feasible

From Gradient Descent to Stochastic Gradient Descent

$$\sum_{s \in \mathcal{S}} \mu(s) 2[v_\pi(s) - \hat{v}(s, \mathbf{w})] \nabla \hat{v}(s, \mathbf{w})$$

$$(S_1, v_\pi(S_1)), (S_2, v_\pi(S_2)), (S_3, v_\pi(S_3)), \ldots$$

$$\mathbf{w}_2 \doteq \mathbf{w}_1 + \alpha[v_\pi(S_1) - \hat{v}(S_1, \mathbf{w}_1)] \nabla \hat{v}(S_1, \mathbf{w}_1)$$

$$(S_1, v_\pi(S_1)), (S_2, v_\pi(S_2)), (S_3, v_\pi(S_3)), \ldots$$

$$\mathbf{w}_3 \doteq \mathbf{w}_2 + \alpha[v_\pi(S_2) - \hat{v}(S_2, \mathbf{w}_2)] \nabla \hat{v}(S_2, \mathbf{w}_2)$$

Gradient Monte Carlo

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)] \nabla \hat{v}(S_t, \mathbf{w}_t)$$

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha[\ G_t\ - \hat{v}(S_t, \mathbf{w}_t)] \nabla \hat{v}(s, \mathbf{w}_t)$$

Recall that

$$v_\pi(s) \doteq \mathbb{E}_\pi\left[G_t \mid S_t = s\right]$$

$$\mathbb{E}_\pi\left[2[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w})] \nabla \hat{v}(S_t, \mathbf{w})\right]$$

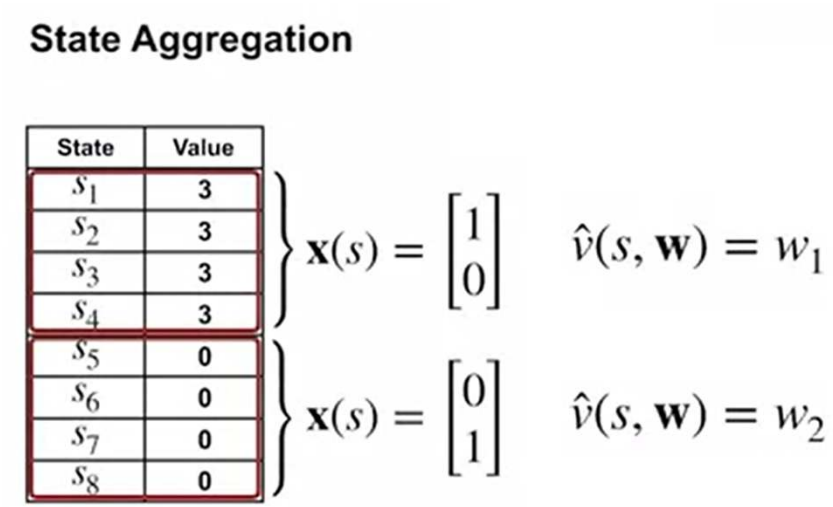
$$= \mathbb{E}_\pi\left[2[\ G_t\ - \hat{v}(S_t, \mathbf{w})] \nabla \hat{v}(S_t, \mathbf{w})\right]$$

Gradient Monte Carlo Algorithm for Estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S} \times \mathbb{R}^d \to \mathbb{R}$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

Loop forever (for each episode):
    Generate an episode $S_0, A_0, R_1, S_1, A_1, \ldots, R_T, S_T$ using π
    Loop for each step of episode, $t = 0, 1, \ldots, T-1$:
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha \big[G_t - \hat{v}(S_t, \mathbf{w}) \big] \nabla \hat{v}(S_t, \mathbf{w})$

State Aggregation with Monte Carlo



State aggregation is another example of linear function approximation.

There is one feature for each group of states.

Each feature will be 1 if the current state belongs to the associated group, and 0 otherwise.

The approximate value of a state is the weight associated with the group that state belongs to.

How to Compute the Gradient for Monte Carlo with State Aggregation

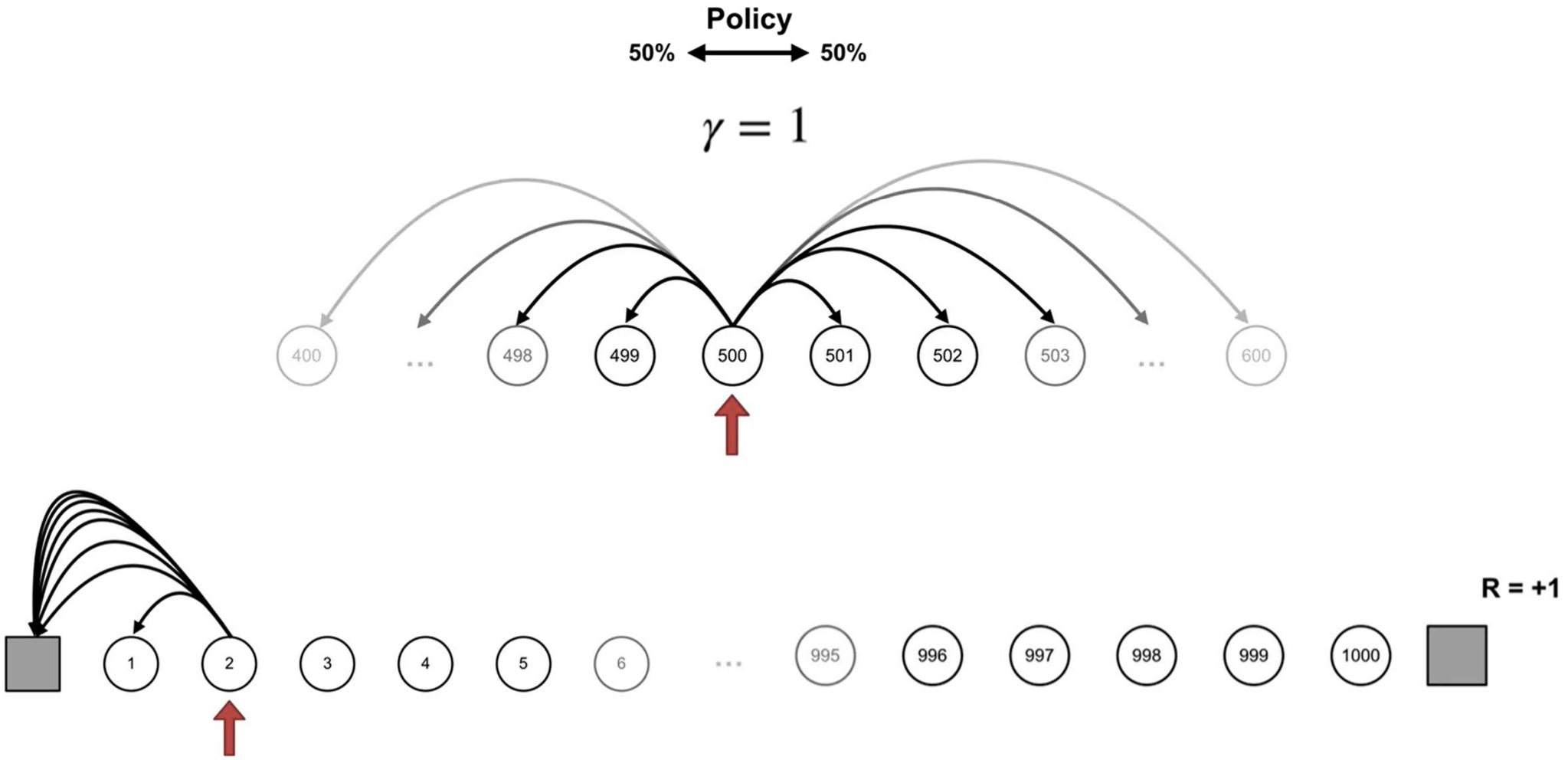
$$\mathbf{w} \leftarrow \mathbf{w} + \alpha[G_t - \hat{v}(S_t, \mathbf{w})] \nabla \hat{v}(S_t, \mathbf{w})$$

$$w_i \leftarrow w_i + \alpha[G_t - \hat{v}(S_t, \mathbf{w})]0$$

$$w_j \leftarrow w_j + \alpha[G_t - \hat{v}(S_t, \mathbf{w})]1$$

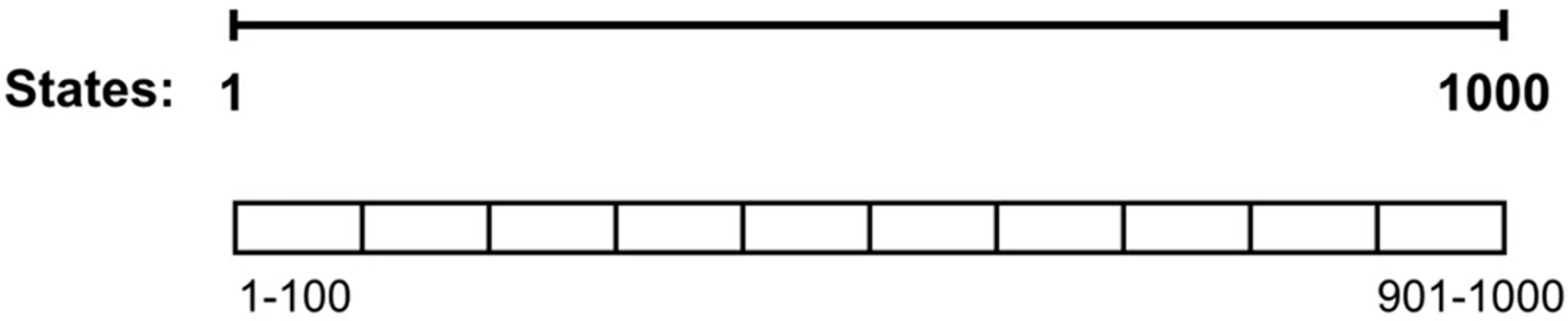
$$\nabla \hat{v}(S_t, \mathbf{w}) = \mathbf{x}(S_t) = \begin{bmatrix} 0 \\ 0 \\ \vdots \\ 0 \\ 1 \\ 0 \\ \vdots \\ 0 \end{bmatrix}$$

Random Walk Example



- The states are numbered from 1 to 1,000 and arranged in a line from left to right.
- All episodes will begin in the middle of the line, at state 500.
- There are two actions, left and right.
- The left action moves the agent to the left, but not necessarily just one place.
- The agent jumps left to any state within 100 neighboring states, uniformly at random.
- Likewise, the right action jumps to the right, to one of the neighboring 100 states.
- Let's evaluate the uniform random policy, which moves either left or right with equal probability.
- States close to the terminal state, on the left, have fewer than 100 neighbors to the left.
- All actions that would jump past the end of the chain go to the terminal state instead.
- Likewise on the right.
- Terminating on the left gives a reward of minus 1,
- terminating the right gives a reward of plus 1.
- All other transitions give reward of 0.
- The discount gamma is 1.
- This task seems fairly simple, but learning the values for 1,000 states might take a lot of time.
- We might benefit from some kind of function approximation.

Constructing a State Aggregation for the Random Walk



Monte Carlo Updates for a Single Episode

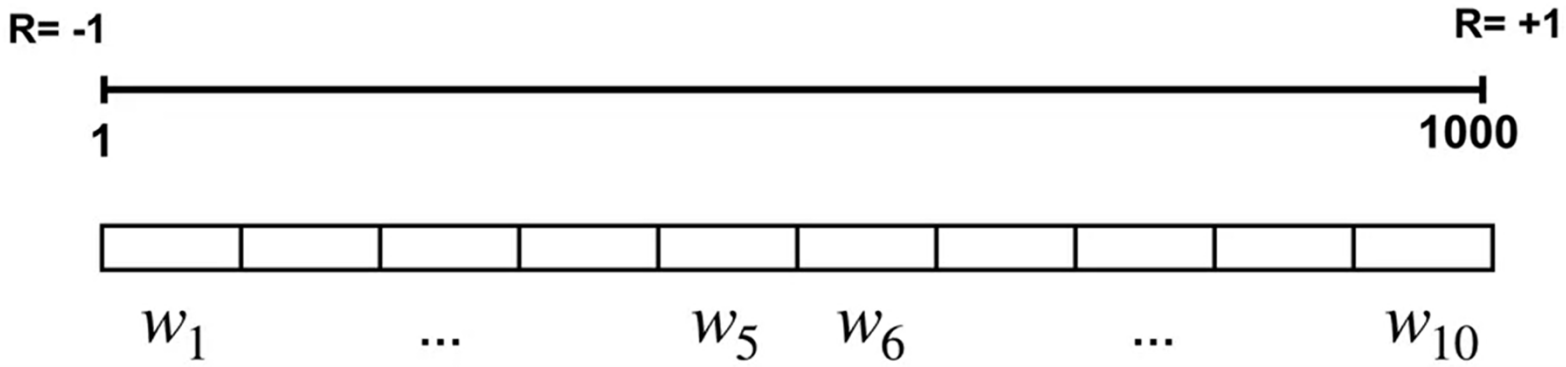
Return: 1 , 1 , 1 , …, 1

Visited states: 500, 423, 482, …, 936

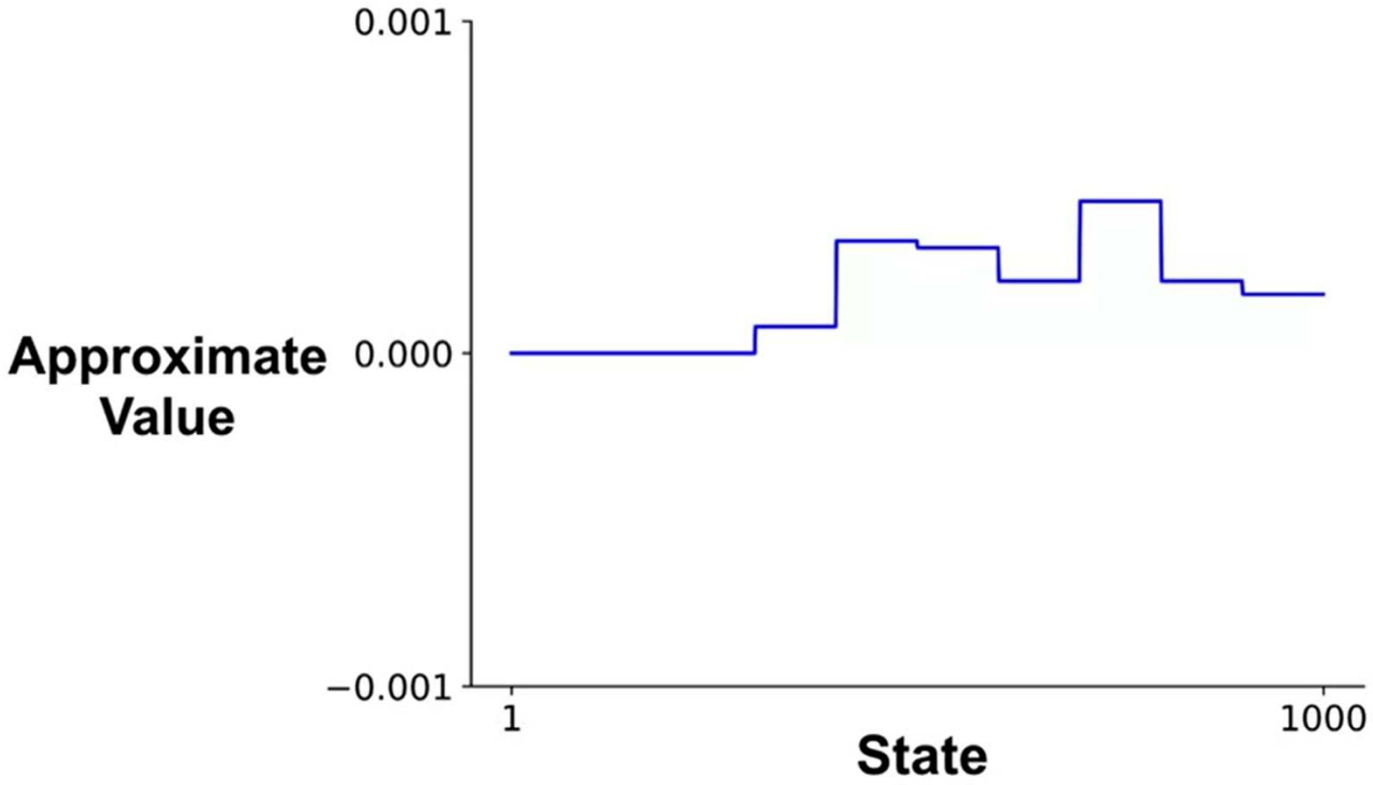
Policy

50% ⟷ **50%**

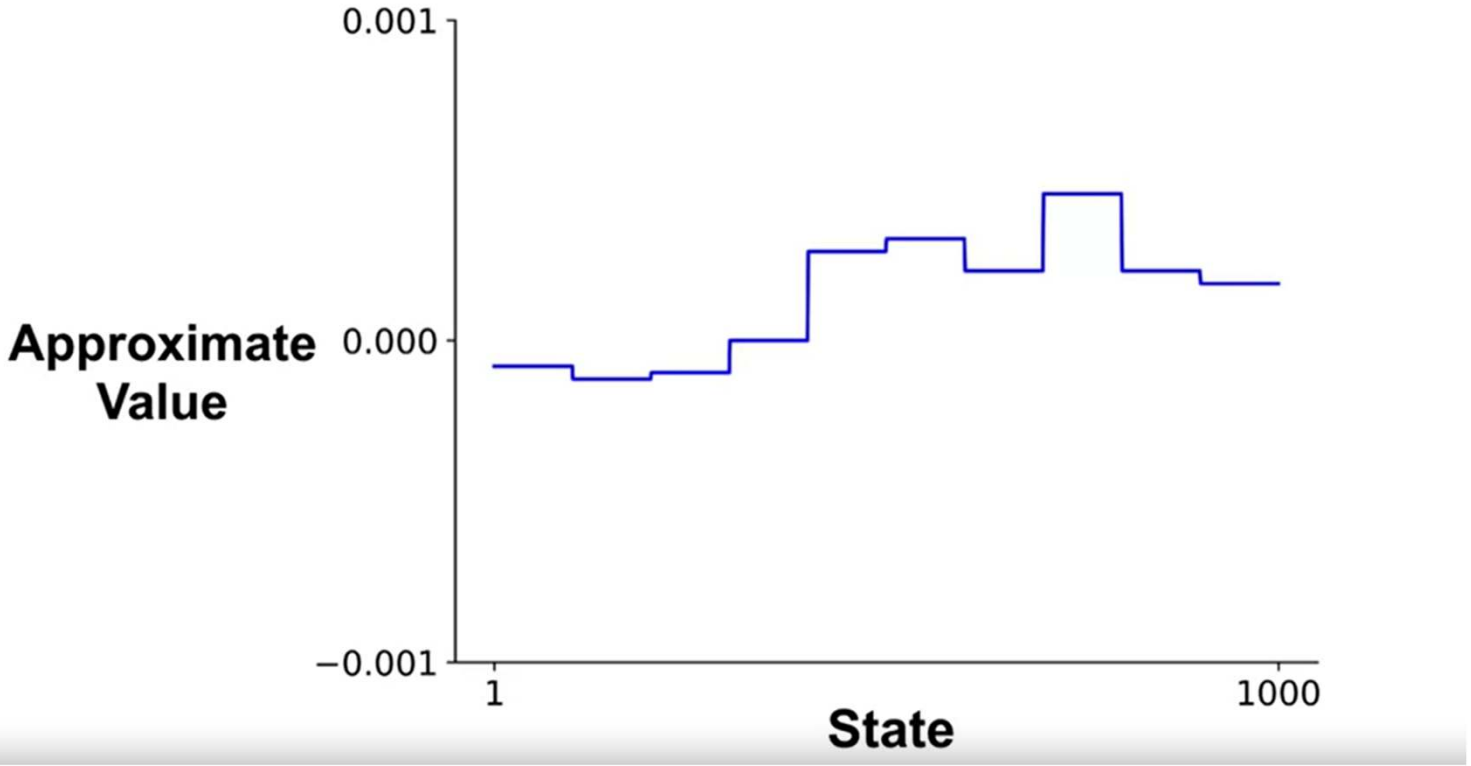
$\alpha = 2 * 10^{-5}$



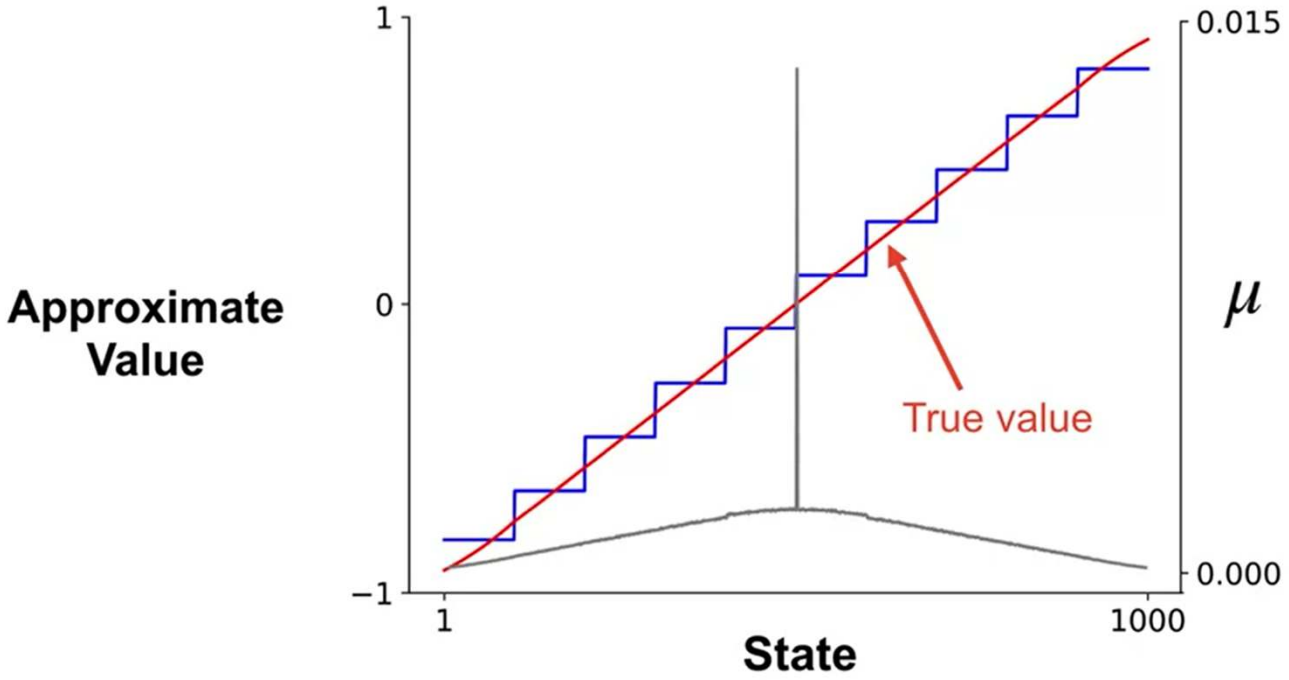
Value Estimates After One Episode



Value Estimates After Two Episodes



Final Value Estimates



After running for many episodes, we get state values that look something like this.
Each step in the plot corresponds to our group of states that share the same approximate value.
For comparison, here's the true value function.
Although we have heavily approximated by aggregating many states together, our value estimate is not that far off.
Why doesn't the red line pass directly through the center of all the blue steps?
This is where mu plays an important role.
Recall mu of s is the visitation frequency for state s.
States near the center of the chain are visited more often than those near the terminal states, as depicted here.
For example, let's look at the leftmost group of states, states 1 to 100.
States near state 100 are visited much more than states near state 1.
The approximate value is skewed towards the true value for states near state 100.
This is because mu weights these states higher in the value error.

Semi-Gradient TD for Policy Evaluation

Gradient Monte Carlo

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha[G_t - \hat{v}(S_t, \mathbf{w})] \nabla \hat{v}(S_t, \mathbf{w})$$

The TD Update for Function Approximation

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha[U_t - \hat{v}(S_t, \mathbf{w})] \nabla \hat{v}(S_t, \mathbf{w})$$

$$U_t \doteq R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})$$

U_t biased → w may not converge to a local optimum

U_t unbiased → w will converge to a local optimum

We can replace the return in the Gradient Monte Carlo update with any estimate of the value.
Let's call this estimate Ut.

If Ut is an unbiased estimate of the true value, then our function approximator will converge to a local optimum under the appropriate conditions.

This was the case for the return, but we can also replace Ut with a bootstrap target, such as the one step TD target.

This is still an estimate of the return, but in this case, the estimate is biased.

The TD target uses our current value estimate,
which will likely not equal the true value function.

Because of this we cannot guarantee this algorithm will converge to a local minimum of the value error.

The upside is that the TD target often has lower variance than the sample of the return.
This means TD will tend to converge in fewer updates.

TD is a semi-gradient method

$$\nabla \frac{1}{2}[U_t - \hat{v}(S_t, \mathbf{w})]^2$$

$$U_t \doteq R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})$$

$$= (U_t - \hat{v}(S_t, \mathbf{w}))(\nabla U_t - \nabla \hat{v}(S_t, \mathbf{w})) \neq -(U_t - \hat{v}(S_t, \mathbf{w}))\nabla \hat{v}(S_t, \mathbf{w}) \qquad \nabla U_t = 0$$

The TD Update

For TD:

$$\nabla U_t = \nabla (R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}))$$

$$= \gamma \nabla \hat{v}(S_{t+1}, \mathbf{w})$$

$$\neq 0$$

TD is not performing gradient descent updates on the squared error

In TD the target contains an estimate of the value, which depends on the weights.

This means the gradient of Ut is not 0 as shown here, therefore, TD is not performing gradient descent updates on the squared error.

We call it a semi-gradient method.

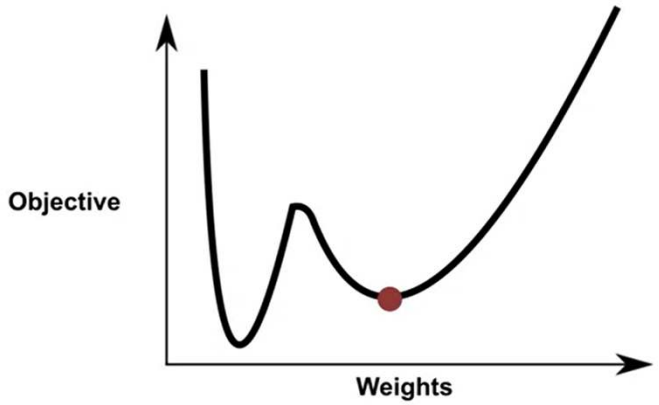
Semi-gradient TD(0) for estimating $\hat{v} \approx v_\pi$

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

Loop for each episode:
    Initialize S
    Loop for each step of episode:
        Choose $A \sim \pi(\cdot|S)$
        Take action A, observe R, S'
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[R + \gamma\hat{v}(S',\mathbf{w}) - \hat{v}(S,\mathbf{w})\big]\nabla\hat{v}(S,\mathbf{w})$
        $S \leftarrow S'$
    until S is terminal

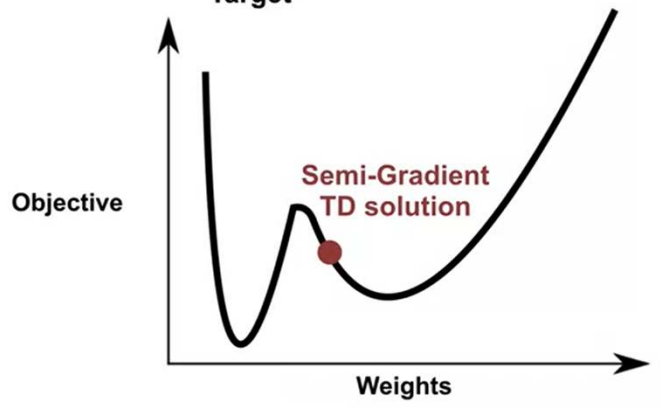
Gradient Monte Carlo will converge to a local minimum of the Mean Squared Value Error

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha[\underbrace{G_t}_{\text{Target}} - \hat{v}(S_t, \mathbf{w})] \nabla \hat{v}(S_t, \mathbf{w})$$

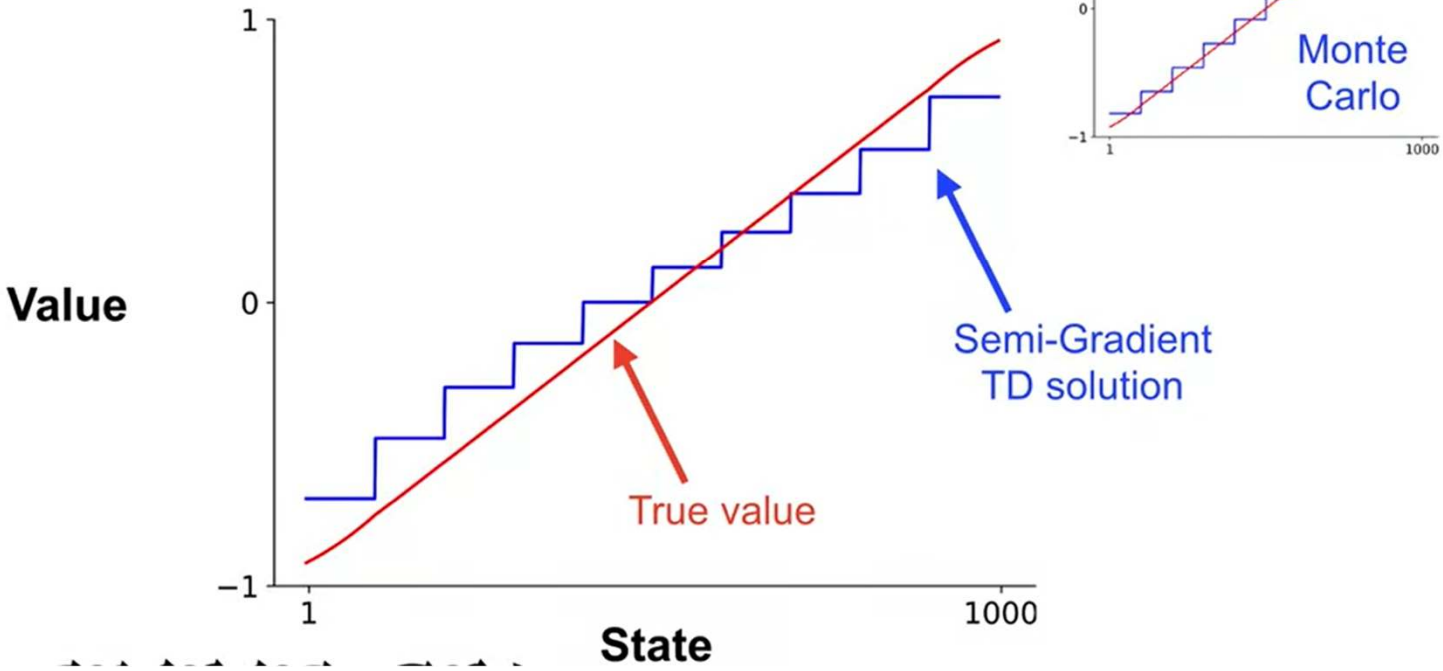


Semi-Gradient TD will not necessarily converge to a local minimum of the Mean Squared Value Error

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha[\underbrace{R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})}_{\text{Target}} - \hat{v}(S_t, \mathbf{w})] \nabla \hat{v}(S_t, \mathbf{w})$$



Long Run Performance of TD on Random Walk

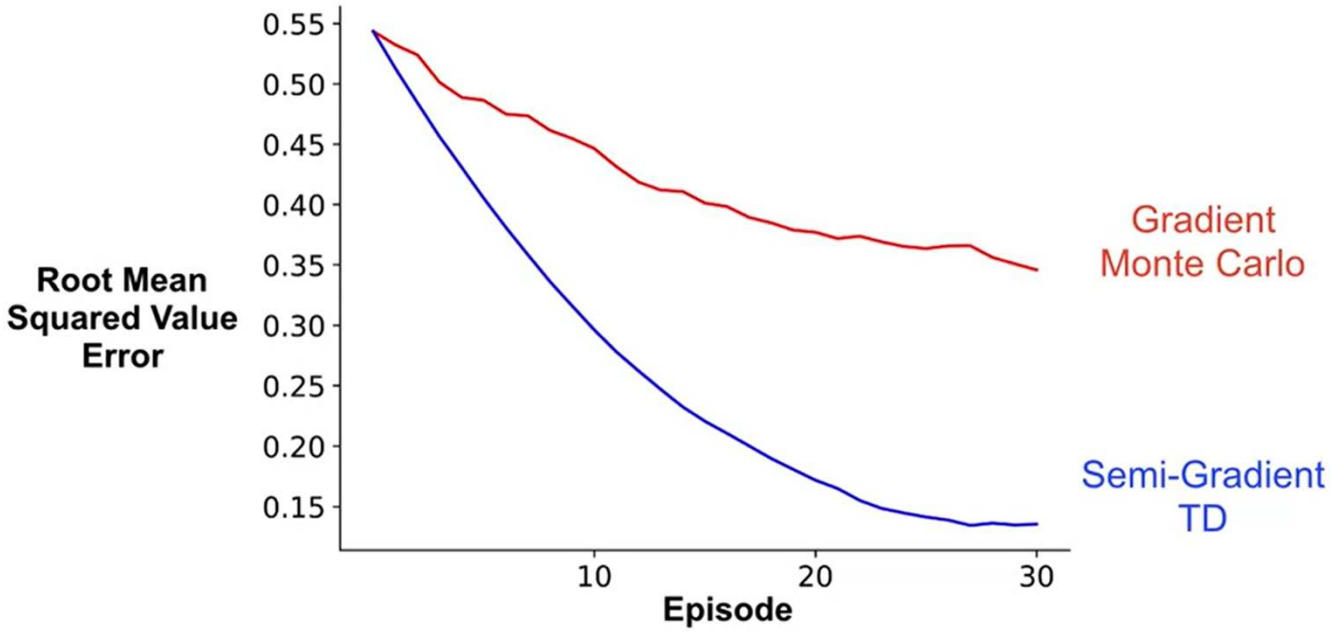


Which algorithm learns faster?

- 30 episodes
- 100 evenly spaced values of α between 0 and 1 with each algorithm
- The best α
  - For TD: 0.22
  - For Monte Carlo: 0.01

- The TD update for function approximation can be biased
- We often prefer TD learning over Monte Carlo anyway because it can converge more quickly

TD converges faster



TD Update with Linear Function Approximation

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta_t \nabla \hat{v}(S_t, \mathbf{w})$$

$$\delta_t \doteq R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w})$$

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha\delta_t \mathbf{x}(S_t)$$

$$\hat{v}(S_t, \mathbf{w}) \doteq \mathbf{w}^T \mathbf{x}(S_t)$$

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

Tabular TD is a special case of linear TD

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha[R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w})]\mathbf{x}(S_t)$$

$$w_i \leftarrow w_i + \alpha[R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}) - \hat{v}(S_t, \mathbf{w})]1$$

$$\mathbf{x}(s_i) = \begin{bmatrix} 0 \\ 0 \\ \cdots \\ 0 \\ 1 \\ 0 \\ \cdots \\ 0 \end{bmatrix}$$

i-th element

$$\hat{v}(s_i, \mathbf{w}) = w_i$$

The Utility of Linear Function Approximation

- Linear methods are simpler to understand and analyze mathematically
- With good features, linear methods can learn quickly and achieve good prediction accuracy

The Expected TD Update

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha[R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t)]\mathbf{x}(S_t) \qquad \hat{v}(s, \mathbf{w}) \doteq \mathbf{w}^T \mathbf{x}(s)$$

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha[R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t) - \hat{v}(S_t, \mathbf{w}_t)]\mathbf{x}_t$$

$$= \mathbf{w}_t + \alpha[R_{t+1}\mathbf{x}_t - \mathbf{x}_t(\mathbf{x}_t - \gamma \mathbf{x}_{t+1})^T \mathbf{w}_t]$$

$$\mathbb{E}[\Delta \mathbf{w}_t] = \alpha(\mathbf{b} - \mathbf{A}\mathbf{w}_t)$$

$$\mathbf{b} \doteq \mathbb{E}[R_{t+1}\mathbf{x}_t] \qquad \mathbf{A} \doteq \mathbb{E}[\mathbf{x}_t(\mathbf{x}_t - \gamma \mathbf{x}_{t+1})^T]$$

The TD Fixed Point

$$\mathbb{E}[\Delta \mathbf{w}_{TD}] = \alpha(\mathbf{b} - \mathbf{A}\mathbf{w}_{TD}) = 0$$

$$\Longrightarrow \quad \mathbf{w}_{TD} = \mathbf{A}^{-1}\mathbf{b}$$

$\mathbf{w}_{TD}$ **minimizes** $(\mathbf{b} - \mathbf{A}\mathbf{w})^T(\mathbf{b} - \mathbf{A}\mathbf{w})$

Relating the TD Fixed Point and the Minimum of the Value Error

$$\overline{VE}(\mathbf{w}_{TD}) \leq \frac{1}{1-\gamma} \min_{\mathbf{w}} \overline{VE}(\mathbf{w})$$

Summary

- Linear semi-gradient TD is guaranteed to converge to a fixed point, called the TD fixed point
- The TD fixed point relates to the minimum mean squared value error