Reinforcement learning

Dr. Aissa Boulmerka

The National School of Artificial Intelligence
aissa.boulmerka@ensia.edu.dz

2024-2025

Chapter 8
On-policy Prediction with Approximation

Outline

- Value-function Approximation
- Generalization and Discrimination
- The Prediction Objective ($\overline{\mathbf{VE}}$)
- Stochastic-gradient and Semi-gradient Methods
- Linear Methods
- Feature Construction for Linear Methods
- Nonlinear Function Approximation

Introduction

Introduction

- In this chapter, the **approximate value function** is represented **not as a table** but as a **parameterized functional form** with **weight vector** $\mathbf{w} \in \mathbb{R}^d$.
- We will write $\hat{v}(s, \mathbf{w}) \approx v_\pi(s)$ for the **approximate** value of **state** s given the **weight** vector $\mathbf{w}$. Note that $\mathbf{w}$ is the **vector of weights**.
- For example, $\hat{v}$ might be a **linear function** in features of the state. More generally, $\hat{v}$ might be a **non-linear function** computed by a **multi-layer artificial neural network**.
- Typically, the number of weights (the dimensionality of $\mathbf{w}$) is much less than the number of **states** ($d \ll |\mathcal{S}|$), and changing one weight changes the estimated value of many states.
- Consequently, when a single state is updated, the change **generalizes** from that state to affect the values of many other states. Such ***generalization*** makes the learning potentially more **powerful** but also potentially more **difficult** to manage and understand.

Value-Function Approximation

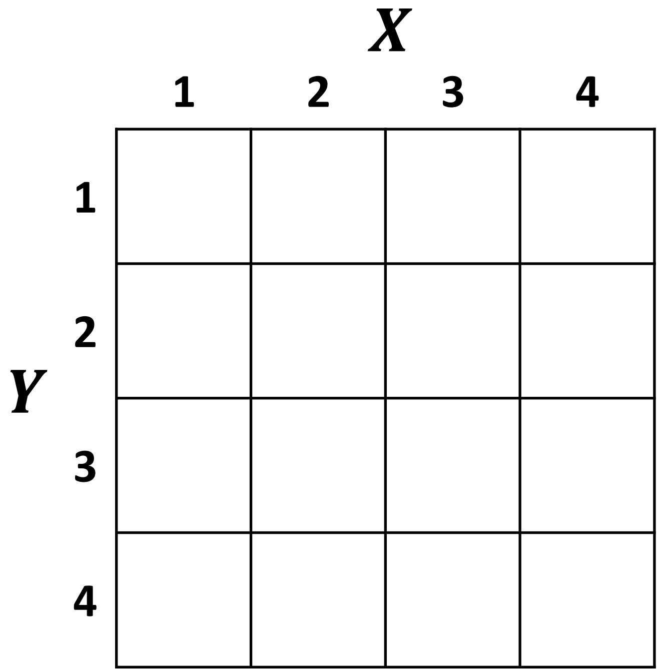
Parameterized value-function

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

Example of parameterized value-function

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

We only have to store two weights w_1 and w_2



How change the weight impact the value-function

$$w_1 = 1$$
$$w_2 = 1$$

X

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

How change the weight impact the value-function

$$w_1 = 2$$
$$w_2 = 1$$

$$\begin{array}{c|c|cccc}
 & & & X & & \\
 & & 1 & 2 & 3 & 4 \\
\hline
 & 1 & 3 & 5 & 7 & 9 \\
Y & 2 & 4 & 6 & 8 & 10 \\
 & 3 & 5 & 7 & 9 & 11 \\
 & 4 & 6 & 8 & 10 & 12
\end{array}$$

How change the weight impact the value-function

$$w_1 = -1$$
$$w_2 = 1$$

X

Y	1	2	3	4
1	0	-1	-2	-3
2	1	0	-1	-2
3	2	1	0	-1
4	3	2	1	0

How change the weight impact the value-function

$$w_1 = 4$$
$$w_2 = 1$$

$$\begin{array}{c|cccc} Y \backslash X & 1 & 2 & 3 & 4 \\ \hline 1 & 5 & 9 & 13 & 17 \\ 2 & 6 & 10 & 14 & 18 \\ 3 & 7 & 11 & 15 & 19 \\ 4 & 8 & 12 & 16 & 20 \end{array}$$

Linear value-function approximation

$$\hat{v}(s, \mathbf{w}) \doteq \sum \boldsymbol{w_i x_i(s)} = < \mathbf{w}, \mathbf{x}(\boldsymbol{s}) >$$

- The value of each state is represented by a **linear function** of the **weights**.
- This simply means that the value of each state, is computed as the **sum of the weights** multiplied by some fixed **attributes** of the state called **features**.
- $< \mathbf{w}, \mathbf{x}(\boldsymbol{s}) >$ is the **dot product (inner product)** of **weight vector** $\mathbf{w}$ and the **feature vector** $\mathbf{x}(\boldsymbol{s})$.

Limitations of Linear Value Function Approximation

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

- We cannot represent this as a **linear function** of $\boldsymbol{X}$ and $\boldsymbol{Y}$.
- Here $\boldsymbol{X}$ and $\boldsymbol{Y}$ are not **good features** for this problem.
- There are many **powerful** methods to **construct features**.

$$\begin{array}{c|c|cccc}
 & & & \boldsymbol{X} & & \\
 & & 1 & 2 & 3 & 4 \\
\hline
 & 1 & 0 & 0 & 0 & 0 \\
\boldsymbol{Y} & 2 & 0 & 5 & 5 & 0 \\
 & 3 & 0 & 5 & 5 & 0 \\
 & 4 & 0 & 0 & 0 & 0
\end{array}$$

Tabular value-functions are linear functions

State	Value
S_1	
S_2	
S_3	
$\dots$	
S_i	
$\dots$	
S_{16}	

- Linear function **approximation** is actually very general.
- In fact, even a **tabular representation** is a special case of **linear function approximation**.
- How to implement that?

Tabular value-functions are linear functions

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

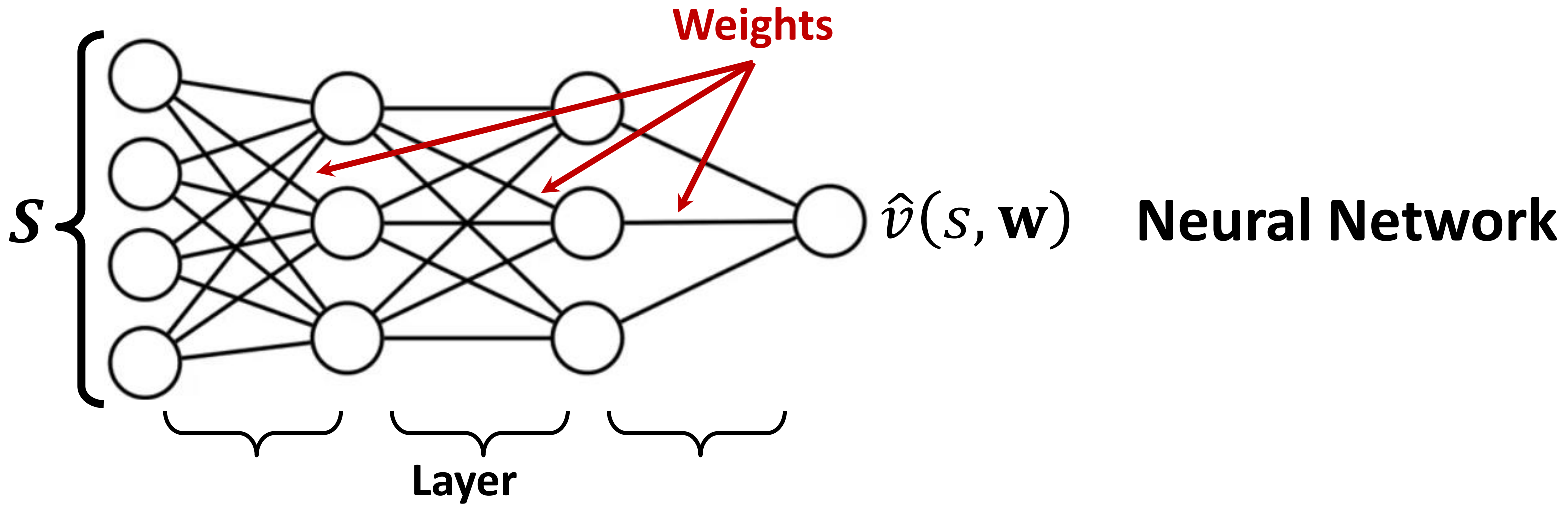
- Let's choose our **features** to be **indicator functions** for particular states.
- For state s_i, feature i is one and the remaining features are 0.
- We have **16 features**, one for each state.

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

***i*-th element**

$$\begin{aligned} \hat{v}(s, \mathbf{w}) &\doteq < \mathbf{w}, \mathbf{x}(\boldsymbol{s}) > \\ &= w_i \end{aligned}$$

Nonlinear function approximation



- Neural networks are an example of a **nonlinear function** of state. The output of the network is an **approximate value** for a given **state**.
- The state is passed to the network as the **input**. All the **connections** in the network correspond to real valued **weights**.
- This **process** transforms the **input state** through a **sequence of layers** to finally produce the **value estimate**.

Generalization and Discrimination

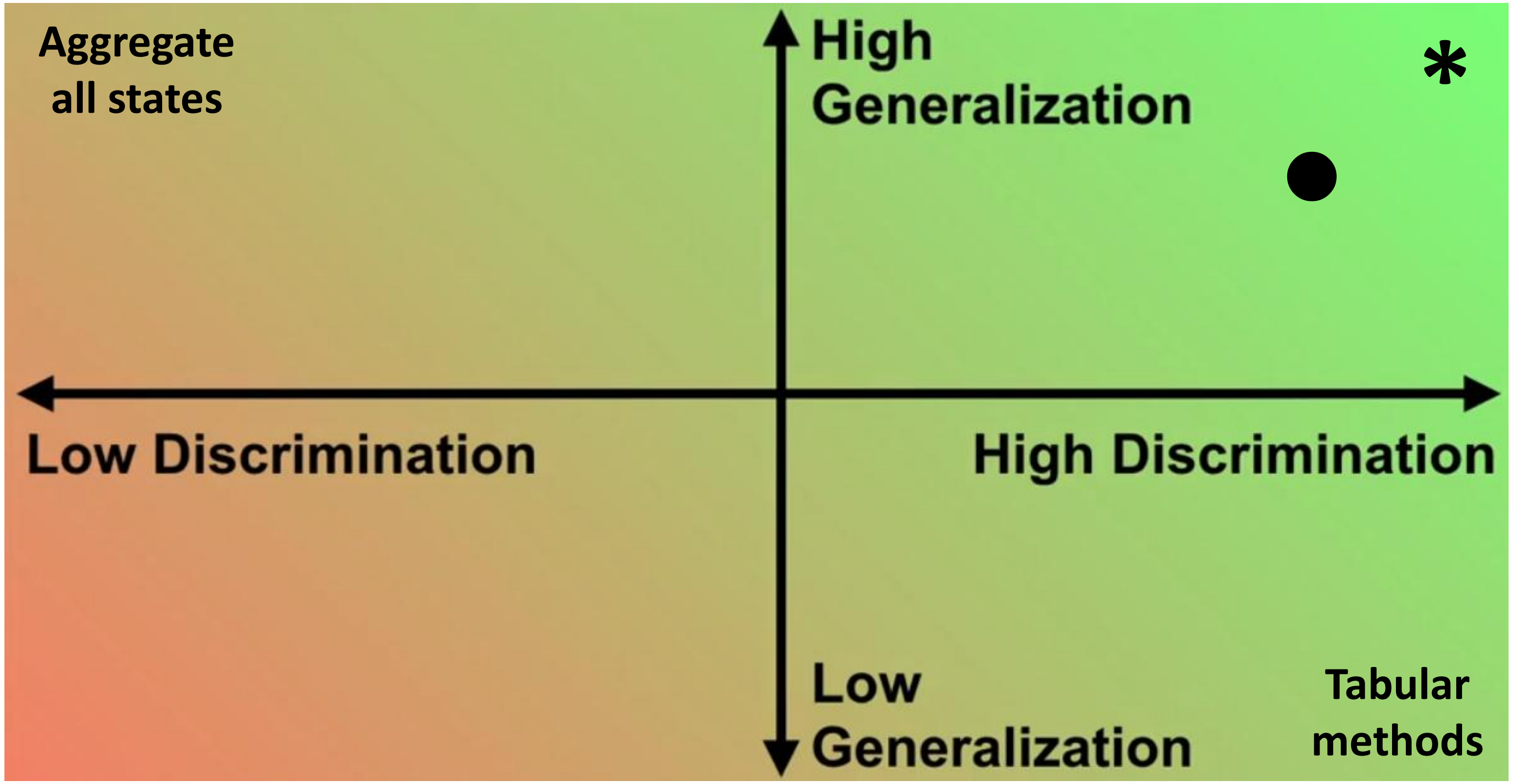
Generalization

- Generalization intuitively means applying knowledge about **specific situations** to draw **conclusions** about a wider variety of situations.
- Generalization, in the context of **policy evaluation**, means that updates to the **value estimate** of one state **influence** the value of **other states**.
- Imagine a **robot**, observing the world through a set of **distance sensors**. In many locations, it would take the same amount of time to drive to the **nearest object**. Even though they correspond to different sensor readings, these locations have similar values. Thus, we might want the **value function** to **generalize** across those states.
- Generalization can **speed learning** by making better use of the **experience** we have.
- You may not have to **visit every state** as much to get this values correct if we can **learn its value from similar states**.

Discrimination

- On the other hand, **discrimination** means the ability to make the values for two states different to **distinguish** between the **values for these two states**.
- Going back to the **example of a robot**, imagine it is in a state where an object is **three feet away**, but **behind a wall**.
- Compare this to a state where an object is **three feet away**, but with **a clear paths to reach it**.
- The robot would want to **assign different values** to these states.
- So while it is useful to **generalize** between states with **similar distance** to the nearest object, it is also important that we **discriminate** between states based on other information when it is likely to impact their value.

Generalization and Discrimination



Frame value estimation as supervised learning

- Supervised learning methods learn a function from an **offline dataset**. This is very different from **reinforcement learning**. But **supervised learning** methods can be useful for handling parts of the reinforcement learning problem.
- In reinforcement learning, an agent **interacts** with an **environment** and continually **generates** new data. This is often called the **online setting**. The proposed **function approximation** technique should work in the **online setting**.
- 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**. So our **targets** continually **change**. This is different than supervised learning where we have access to a **ground truth** label as the **target**.

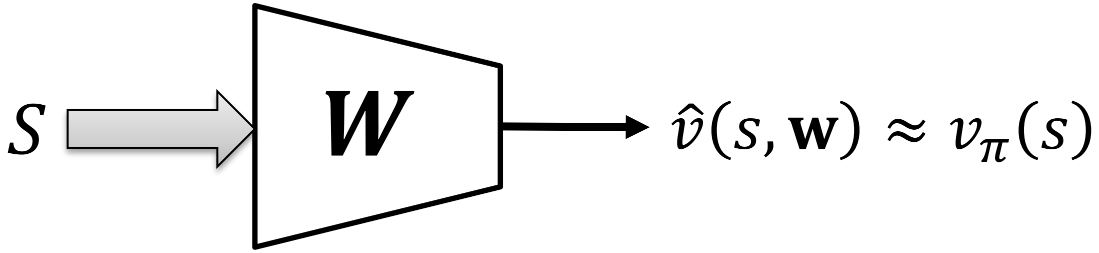
The Prediction Objective

The Prediction Objective $\overline{\mathbf{VE}}$

- Suppose the following idealized scenario. We get a sequence of **pairs of states and true values**.

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

- We want to use this **data** to find a **parameterized function** that closely **approximates** v_π.



- We will do this by adjusting the **weights** so that the output of the function **approximate** the associated value for a given state.

- To make our goal precise, we need to specify some **measure** of how **close our approximation is to the value function**.

The Mean Squared Value Error Objective

- Let's specify a **state distribution** $\mu(s) \geq 0, \sum_s \mu(s) = 1$, representing how much we **care about the error** in each state s.
- The **error** in a state s is computed using the **square of the difference** between the approximate value $\hat{v}(s, \mathbf{w})$ and the true value $v_\pi(s)$.
- Weighting this over the state space by μ, we obtain a natural **objective function**, the **Mean Squared Value Error**, denoted $\overline{VE}$:

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

- Often $\mu(s)$ is chosen to be the **fraction of time spent** in s. Under on-policy training this is called the **on-policy distribution**.
- The goal of defining this objective is to adapt the weights to **minimize** the mean squared value error.

Stochastic-Gradient and Semi-Gradient Methods

Stochastic-gradient methods

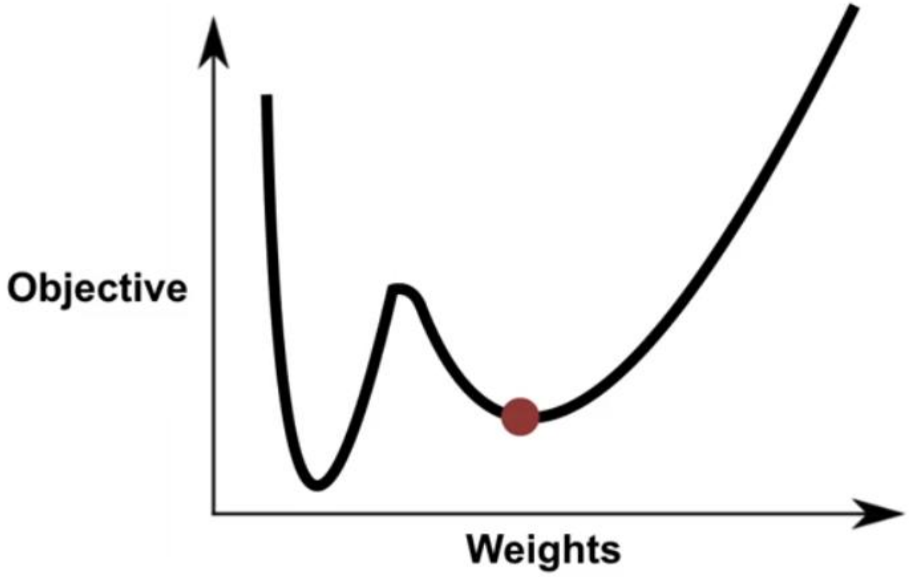
- Consider the weight vector $w \doteq (w_1, w_2, \cdots, w_d)^T$, and the **approximate value function** $\hat{v}(s, \mathbf{w})$ is a **differentiable** function of $\mathbf{w}$ for all $s \in \mathcal{S}$.
- We will update $\mathbf{w}$ at each of a series of **discrete time steps,** $t = 0, 1, 2, 3, \cdots$. Also, consider w_t for the weight vector at each step t.
- *Stochastic gradient-descent* (SGD) methods are particularly well suited to **online** reinforcement learning. **Stochastic gradient-descent (SGD)** methods **minimize** error on the observed examples by adjusting the **weight vector** after each example by a **small amount** in the **direction that most reduce the error** on that example:

$$\begin{aligned}\mathbf{w}_{t+1} &\doteq \mathbf{w}_t - \frac{1}{2}\alpha\nabla\left[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)\right]^2 \\ &= \mathbf{w}_t + \alpha\left[v_\pi(S_t) - \hat{v}(S_t, \mathbf{w}_t)\right]\nabla\hat{v}(S_t, \mathbf{w}_t)\end{aligned}$$

From gradient descent to stochastic gradient descent

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

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



Gradient of the MSVE objective

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

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

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

Linear value function

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

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

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

Gradient Monte Carlo

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

- Recall that :

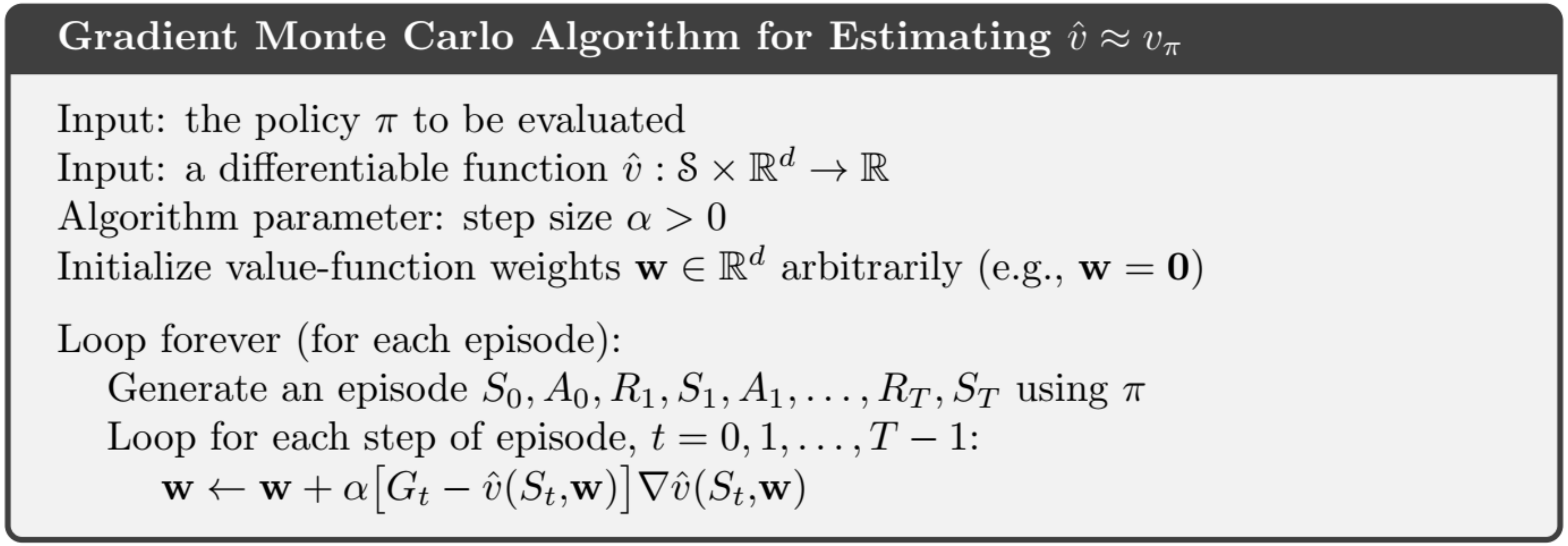
$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t | S_t = s]$$

- The **expectation** of the **gradient** when we use a **sampled return** in place of the **true value**, is still equal to the **gradient of the MSVE**.

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

Gradient Monte Carlo state-value prediction

- Suppose the states in the examples are the states generated by **interaction** with the **environment** using **policy** π.
- The SGD method converges to a **locally optimal approximation** to $v_\pi(S_t)$.
- So, the gradient-descent version of Monte Carlo state-value prediction is guaranteed to find a **locally optimal solution**.



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

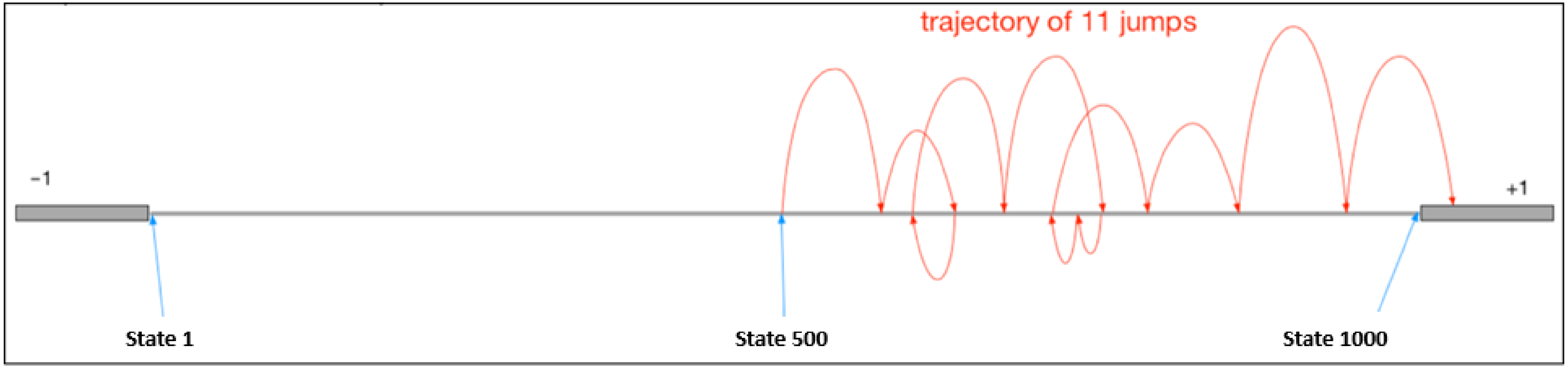
```
Input: the policy $\pi$ to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S} \times \mathbb{R}^d \rightarrow \mathbb{R}$
Algorithm parameter: step size $\alpha > 0$
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

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

Example: Random Walk

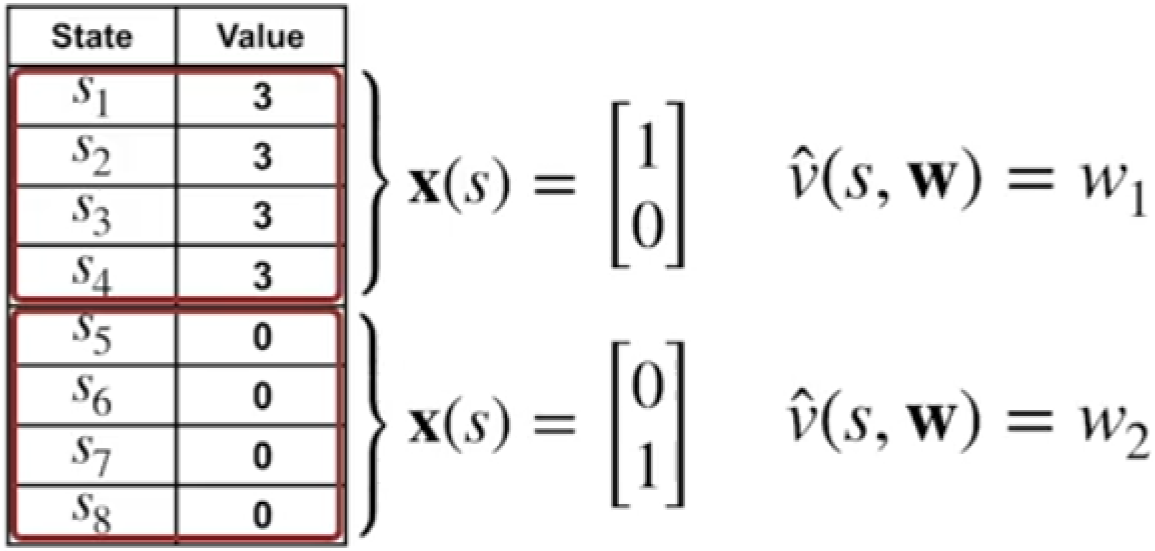
- Consider a **1000-state version** of the **random walk task**. The states are numbered from 1 to 1000, **left to right**, and all episodes begin near the **center**, in state 500.
- State transitions are from the current state to one of the **100 neighboring** states to its left, or to one of the **100 neighboring** states to its right, all with **equal probability**.
- If the current state is **near an edge**, then there may be fewer than 100 neighbors on that side of it. In this case, all the probability that would have gone into those missing neighbors goes into the **probability of terminating** on that side (example, state 1 has a 0.5 chance of terminating on the left, and state 950 has a 0.25 chance of terminating on the right).
- Termination on the **left** produces a **reward** of -1, and termination on the **right** produces a **reward** of $+1$. All other **transitions** have a **reward** of zero.

Example: Random Walk



Example: Random Walk

State aggregation



State	Value
s_1	3
s_2	3
s_3	3
s_4	3
s_5	0
s_6	0
s_7	0
s_8	0

$$\mathbf{x}(s) = \begin{bmatrix}1\\0\end{bmatrix} \qquad \hat{v}(s,\mathbf{w}) = w_1$$

$$\mathbf{x}(s) = \begin{bmatrix}0\\1\end{bmatrix} \qquad \hat{v}(s,\mathbf{w}) = w_2$$

Example: Random Walk

How to compute the gradient for Monte Carlo with state aggregation?

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

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

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

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

Example: Random Walk

Monte Carlo update for a single episode

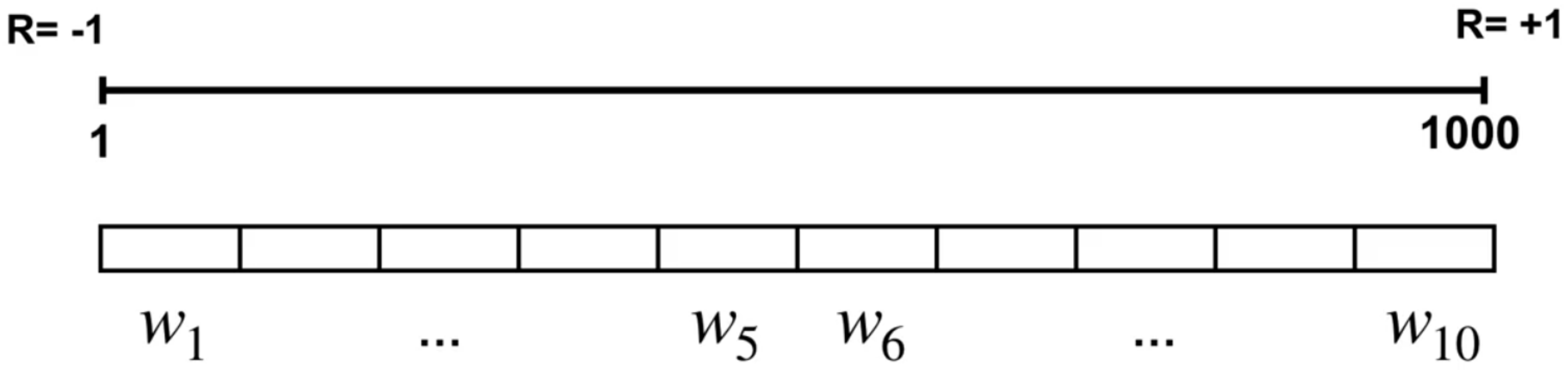
Return: 1 , 1 , 1 , …, 1

Visited states: 500, 423, 482, …, 936

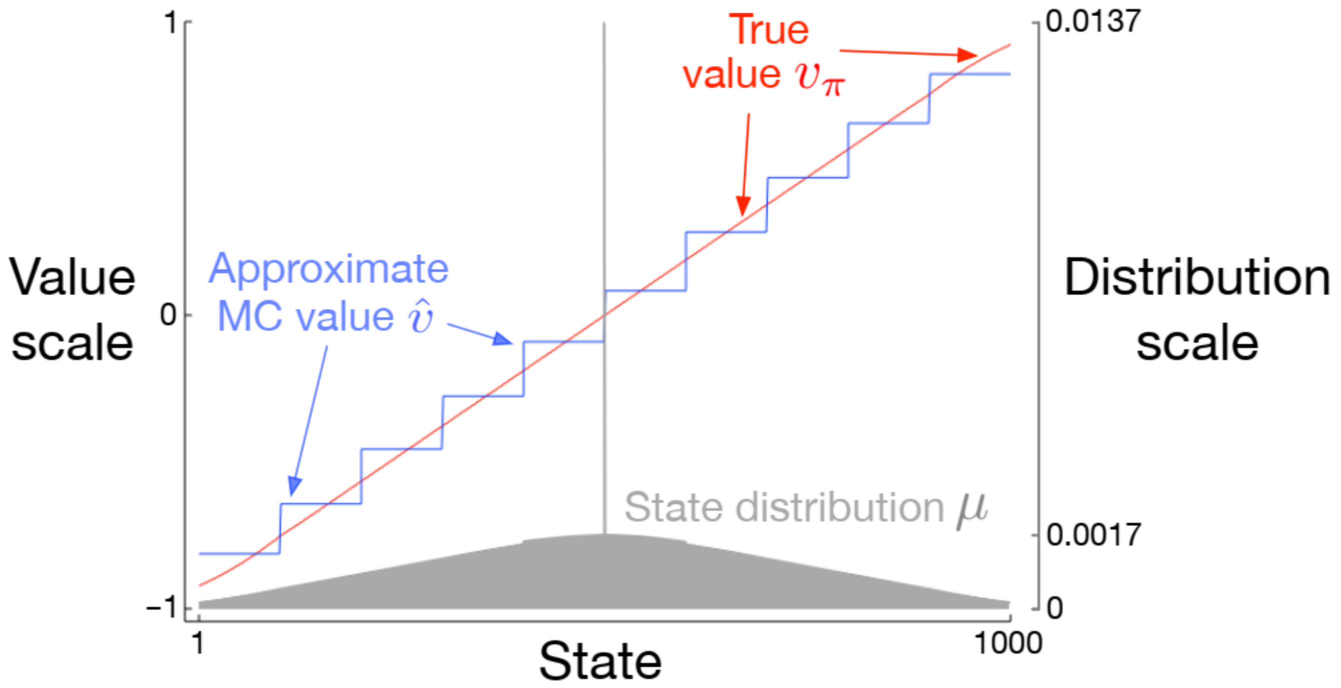
Policy

50% ⟷ **50%**

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



Example: Random Walk



Function approximation by state aggregation on the 1000-state random walk task, using the gradient Monte Carlo algorithm.

Semi-gradient TD(0)

The TD update for function approximation

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

- We can also replace U_t with a **bootstrap target**, such as the **one step TD target**.

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

- 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**.

Semi-gradient TD(0)

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

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

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

$\neq [U_t - \hat{v}(S_t, \mathbf{w}_t)]\nabla \hat{v}(S_t, \mathbf{w}_t)$ (The TD update)

This is true only if $\nabla U_t = 0$

For TD:

$$\begin{aligned} \nabla U_t &= \nabla(R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w})) \\ &= \gamma \nabla \hat{v}(S_{t+1}, \mathbf{w}) \\ &\neq \mathbf{0} \end{aligned}$$

Semi-gradient TD(0)

- Bootstrapping methods are not in fact instances of **true gradient descent**. They take into account the effect of **changing the weight vector** $\mathbf{w}_t$ on the estimate, but ignore its effect on the **target**. They include only a **part of the gradient** and, accordingly, we call them **semi-gradient methods**.
- Although **semi-gradient (bootstrapping)** methods do not converge as robustly as gradient methods, they do converge reliably in important cases such as the **linear** case.
- One advantage for this is that they typically enable significantly **faster learning**.
- Another advantage, is that they enable **learning to be continual and online**, without waiting for the **end of an episode**. This enables them to deal with **continuing problems** and provides **computational advantages**.

Semi-gradient TD(0)

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

```
Input: the policy $\pi$ 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
```

Comparing TD and MC with state aggregation

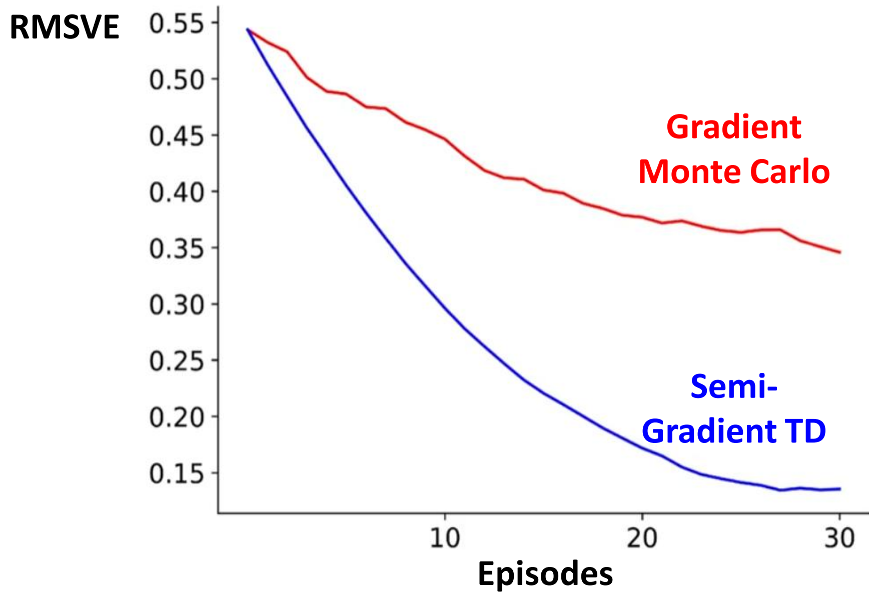
$$\mathbf{w} \leftarrow \mathbf{w} + \alpha[U_t - \hat{v}(S_t, \mathbf{w})]\nabla\hat{v}(S_t, \mathbf{w})$$

Gradient Monte Carlo	Semi-Gradient TD
Target : $U_t = G_t$	**Target :** $U_t = R_{t+1} + \gamma\hat{v}(S_{t+1}, \mathbf{w})$
▪ Gradient MC uses an **unbiased** estimate of the gradient of the value error. ▪ It will approach a **local minimum** of the Mean Squared Value Error with more and more samples. ▪ Gradient Monte Carlo **will converge** to a **local minimum** of the mean squared value error.	▪ The update could be **biased** because the estimate in the target may not be accurate. ▪ Since the value approximation will **never be perfect** even in the limit, the target may remain biased. ▪ Semi-gradient TD **cannot guarantee** to **converge** to a **local minimum** at the Mean Squared value error.

Comparing TD and MC with state aggregation

Experiment settings:

- 30 episodes
- 100 evenly spaced values of α between 0 and 1 with each algorithm.
- The best α for TD : 0.22 and for MC : 0.01.



Conclusions:

- We can conclude that TD often learns **faster** than Monte Carlo. This is because TD can **learn** during the episode and has **lower variance** updates.
- Monte Carlo is better on **long-run performance**, it's not always the main concern.
- We can never run our experiments to achieve **asymptotic performance**.
- Early learning is perhaps **more important** in practice.

Linear Methods

Linear function approximation

- One of the most important special cases of **function approximation** is that in which the approximate function, $\hat{v}(s, \mathbf{w})$, is a **linear function** of the **weight** vector, $\mathbf{w}$.

- Corresponding to every **state** s, there is a real-valued **vector**

$$\mathbf{x}(s) \doteq (x_1(s), x_2(s), \cdots, x_d(s))^T$$

- Linear methods approximate state-value function by the **inner product** between $\mathbf{w}$ and $\mathbf{x}(s)$:

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

- In this case the **approximate value function** is said to be **linear in the weights**, or simply **linear**.

Linear function approximation

- The vector $\mathbf{x}(s)$ is called a **feature vector** representing state s.
- Each component $x_i(s)$ of $\mathbf{x}(s)$ is the value of a function $x_i: \mathcal{S} \rightarrow \mathbb{R}$.
- We think of a **feature** as the entirety of one of these functions, and we call its value for a state s a **feature of s**.
- For linear methods, features are **basis functions** because they form a linear basis for the set of approximate functions.
- Constructing **d-dimensional** feature vectors to represent states is the same as selecting a set of d basis functions.
- Features may be defined in **many different ways**; we cover a few possibilities later in this course.

SGD update of linear function approximation

- It is natural to use **SGD** updates with **linear function approximation**. The **gradient** of the approximate value function with respect to $\mathbf{w}$ in this case is:

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

- In this case the general **SGD update** reduces to a particularly simple form:

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

- Because it is so **simple**, the linear SGD case is one of the most **favorable** for mathematical analysis.
- Almost all useful **convergence** results for **learning systems** of all kinds are for **linear** (or simpler) function approximation methods.

TD update of linear function approximation

- The semi-gradient TD(0) algorithm also **converges** under linear function approximation. The weight vector converged to is also not the **global optimum**, but rather a point near the **local optimum**.

- The update at each time t is

$$\mathbf{w}_{t+1} \doteq \mathbf{w}_t + \alpha \delta_t \mathbf{x}(S_t)$$

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

Where δ_t is the **TD error**.

- This fixed basis given by the **expert design features** has a large impact on the update. If well-designed, we can get **effective value function approximation** with a simple update.

Tabular TD is a special case of linear TD

- We can show that **linear TD** is a strict **generalization** of both tabular TD and TD with state aggregation.

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

- In the update, the **feature** vector $\mathbf{x}(S_t)$ selects a single weight associated with the current state.

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

- We can use the **same analysis** to show that TD with state aggregation is also a **special case of linear TD**.

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

***i*-th element**

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

The Expected TD update

- We can expand the **TD update** like the following formula, note that we have used the notational shorthand $\mathbf{x}_t = \mathbf{x}(S_t)$.

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

- The **expected update** characterizes the **expected change** in the weight from one time step to the next:

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

Where $\mathbf{b} \doteq \mathbb{E}[R_{t+1}\mathbf{x}_t] \in \mathbb{R}^d$ and $\mathbf{A} \doteq \mathbb{E}[\mathbf{x}_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})^T] \in \mathbb{R}^d \times \mathbb{R}^d$

The TD fixed point

- It is clear that, if the system **converges**, it must converge to the weight vector $\mathbf{w}_{TD}$ at which

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

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

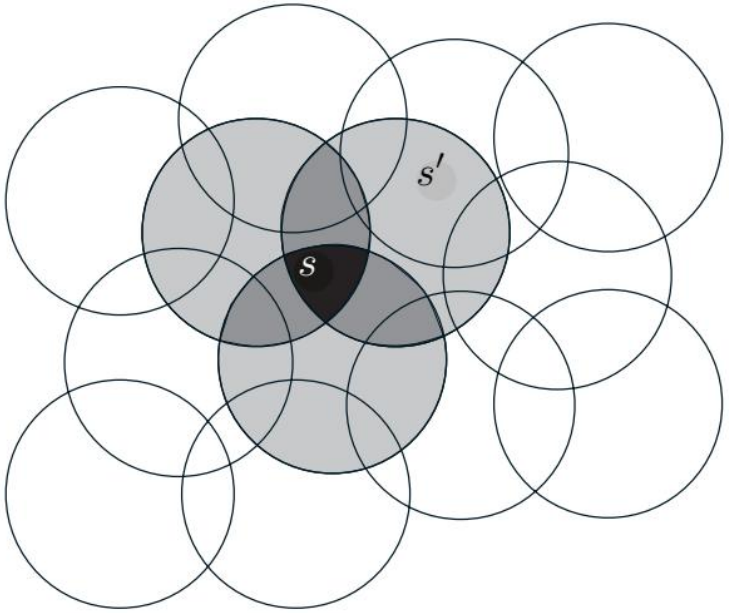
- If $\mathbf{A}$ is **invertible**, $\mathbf{w}_{TD}$ is a **solution** to this **linear system**. We call this solution the **TD fixed point**. In fact linear semi-gradient TD(0) **converges** to this point.
- $\mathbf{w}_{TD}$minimizes $(\mathbf{b} - \mathbf{A}\mathbf{w})^T(\mathbf{b} - \mathbf{A}\mathbf{w})$. This **objective** extends the connection between **TD and Bellman equations**, to the **function approximation setting**.
- At the TD **fixed point**, the $\overline{VE}$ is **bounded**

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

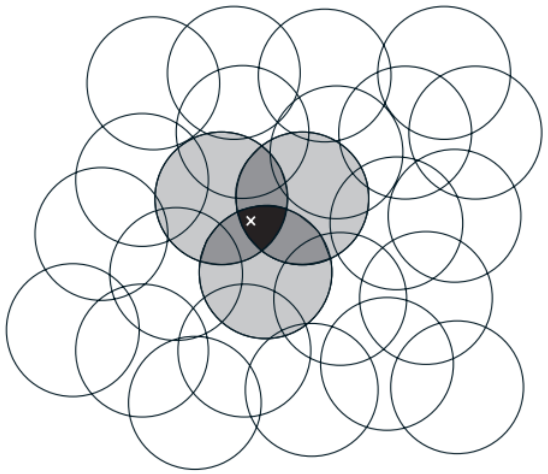
Feature Construction for Linear Methods

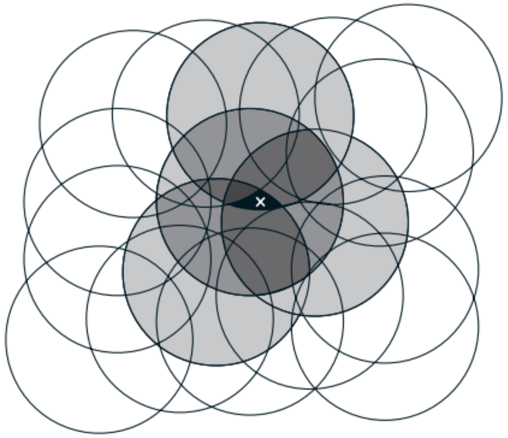
Coarse coding

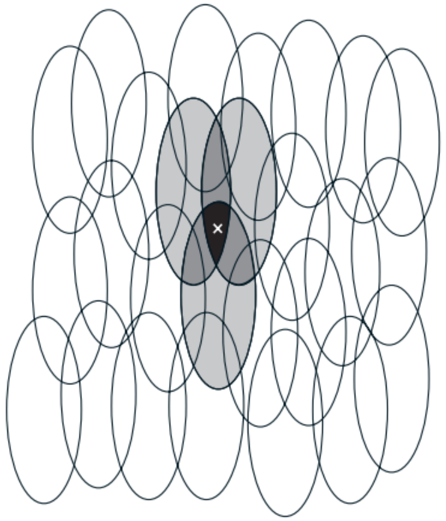
- Features corresponding to **circles** in **state space**.
- If the state is **inside** a circle, then the corresponding feature has the value 1 and is said to be **present**; otherwise the **feature** is 0 and is said to be **absent**.
- This kind of 1–0-valued feature is called a **binary feature**.
- Representing a state with features that overlap in this way is known **as coarse coding.**



Coarse coding generalization

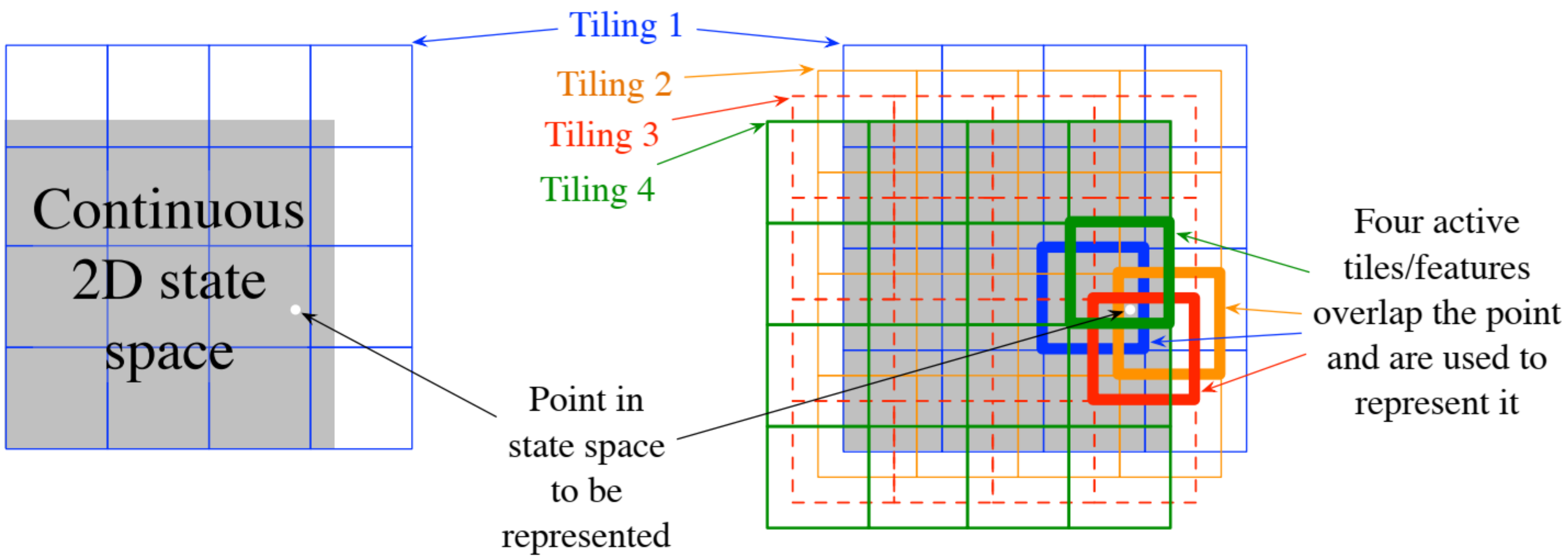

Narrow generalization


Broad generalization


Asymmetric generalization

- Generalization in **linear** function approximation methods is determined by the **sizes and shapes** of the features' **receptive fields**.
- All three of these cases have roughly the same number and **density of features**.

Tile coding

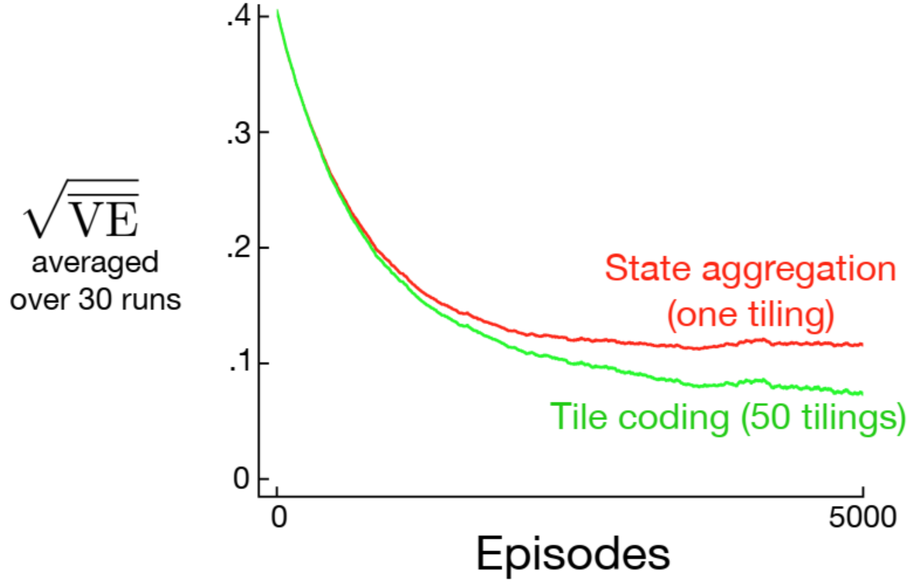

Tiling 1
Tiling 2
Tiling 3
Tiling 4
Continuous 2D state space
Point in state space to be represented
Four active tiles/features overlap the point and are used to represent it

- In **tile coding** the receptive fields of the features are grouped into partitions of the state space.
- Each such partition is called a **tiling**, and each element of the partition is called a **tile**.

Why we use coarse coding?

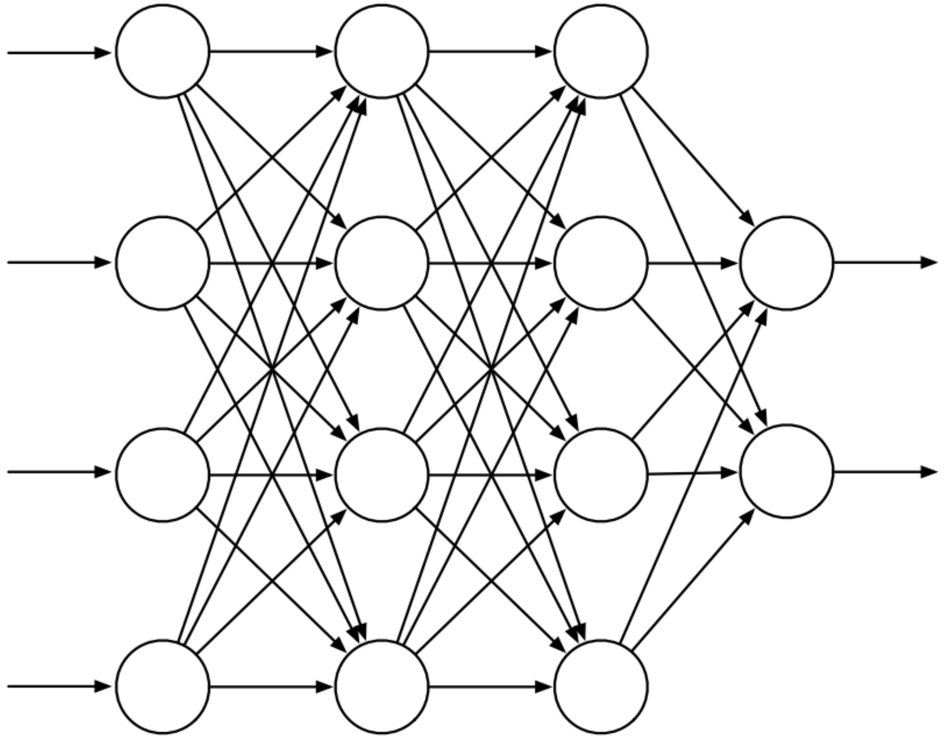
1000-state random walk example for the gradient Monte Carlo algorithm with a single tiling and with multiple tilings.

- The space of **1000 states** was treated as a single **continuous dimension**, covered with tiles each **200 states wide**.
- The multiple **tilings** were **offset** from each other by 4 states.
- The **step-size** parameter was set so that the initial learning rate in the two cases was the same,
  - $\alpha = 0.0001$ for the single tiling
  - $\alpha = 0.0001/50$ for the 50 tilings.

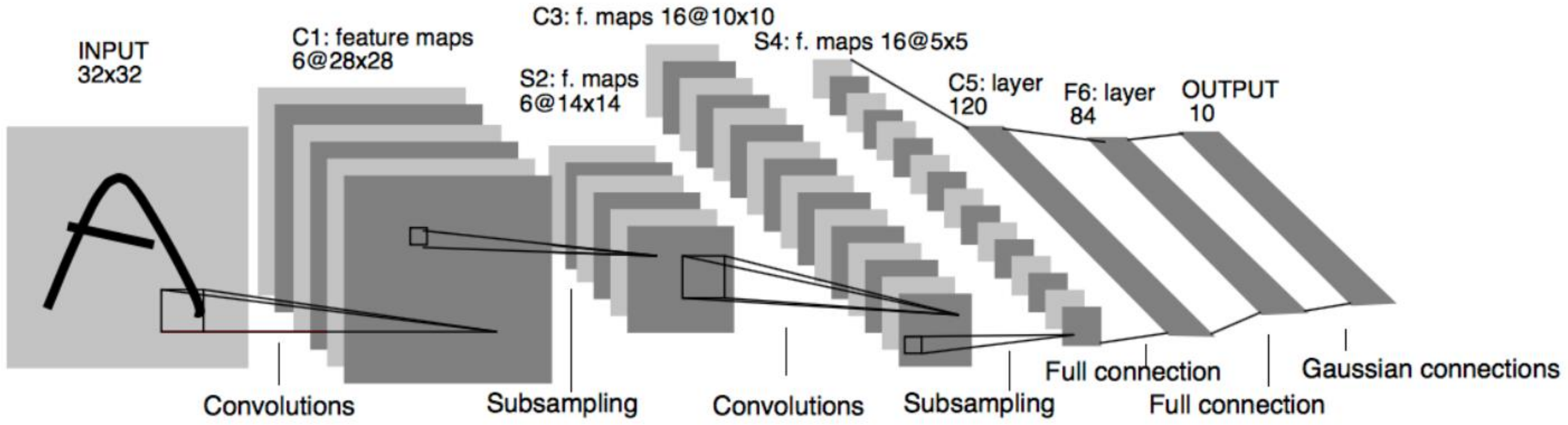


Nonlinear Function Approximation

Artificial Neural Network



Deep Neural Network



Deep Neural Network

- In theory, a **neural network** need not be deep. A neural network with a **single hidden layer** can approximate any continuous function given that is sufficiently wide.
- We call this the **universal approximation** property.
- Practical experience and theory suggests that **deep neural networks** may make it easier to approximate **complex functions**.
- One reason for this is that the depth allows **composition of features**.
- Composition can produce more **specialized features** by combining modular components.
- Overall, **depth in a network** can significantly improve our agent's ability to **learn features**.

Thank you!
Q/A