

Function Approximation & Policy Gradient Methods

CMPUT 261: Introduction to Artificial Intelligence

S&B §9.0-9.5.4, 13.0-13.3

Lecture Outline

1. Recap & Logistics
2. Parameterized Value Functions
3. Gradient Descent
4. Approximation Schemes
5. Parameterized Policies
6. Policy Gradient Theorem
7. REINFORCE algorithm

After this lecture, you should be able to:

- explain why function approximation is useful
- define tile coding
- explain the difference between action-value and policy gradient methods for control
- state the Policy Gradient Theorem and explain why it is important
- trace an execution of the REINFORCE algorithm

Logistics

- **Assignment #4** is due **December 4** at 11:59pm (this Thursday!)
 - Late submissions for 20% deduction
 - No EA or remarking for empty submissions
- **SPOT** (formerly USRI) surveys are now available
 - Available until **December 10**
 - You should have gotten an email
- Next lecture: Game theory for multiagent systems

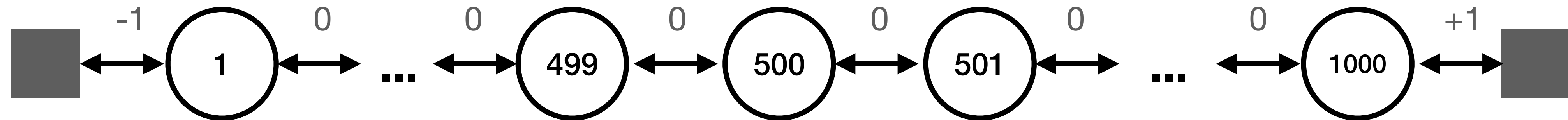
Recap: TD Learning

- Temporal Difference Learning **bootstraps** *and* learns from **experience**
 - Dynamic programming bootstraps, but doesn't learn from experience (requires full dynamics)
 - Monte Carlo learns from experience, but doesn't bootstrap
- Prediction: **TD(0) algorithm**
- **Sarsa** estimates action-values of **actual ϵ -greedy policy**
- **Q-Learning** estimates action-values of **optimal** policy while **executing** an **ϵ -greedy** policy

Tabular Value Functions

- We have been assuming a **tabular representation** for value function estimates $V(s)$ and $Q(s, a)$
 - We can **separately** set the value of $V(s)$ or $Q(s, a)$ for every possible $s \in \mathcal{S}$ and $a \in \mathcal{A}$
- This implicitly means that we **must** store a separate value for every possible input for the value function
- **Question:** What should we do if there are **too many states** to store a value for each? (e.g., **pixel values** in the Atari setting)
- **Question:** What should we do if the state **isn't fully observable**?

Example: Number Line Walk



$$\pi(a | s) = 0.5 \quad \forall s \in \mathcal{S}, a \in \{\text{left}, \text{right}\}$$

1D gridworld with states 1 to 1000 with double-ended arrows labelled 0 between them. Left of 1 is a terminal state with an arrow to it labelled -1. Right of 1000 is a terminal state with an arrow to it labelled +1.

- **Question:** Would dynamic programming, Monte Carlo, or TD(0) work to estimate v_π ?
- **Question:** How much **storage** would that require?
- **Question:** What could we do instead?

Parameterized Value Functions

- A **parameterized value function**'s values are set by setting the values of a **weight vector** $\mathbf{w} \in \mathbb{R}^d$:

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

- $\hat{v}$ could be a **linear function**: $\mathbf{w}$ is feature weights for state **features** $\mathbf{x}(s)$
- $\hat{v}$ could be a **neural network**: $\mathbf{w}$ is weights, biases, kernels, etc.
- Many fewer weights than states: $d \ll |\mathcal{S}|$
 - Changing **one weight** changes the estimated value of **many states**
 - Updating a single state **generalizes** to affect many other states' values

Decoupled Estimates

- With **tabular** estimates:
 - Can update the value of a single state **individually**
 - Estimates can be **exactly correct** for **each state**
- For **parameterized** estimates:
 - Estimates cannot necessarily be correct for each state (e.g., when two states have identical features but different values)
 - Cannot independently adjust state values

Prediction Objective

- Since we cannot guarantee that every state will be correct, we must **trade off** estimation quality of one state vs. another
- We will use a distribution $\mu(s)$ to specify **how much we care** about the quality of our value estimate for each state
- We will optimize the **mean squared value error**:

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

- **Note:** If we knew v_{π} , this would be a **supervised learning problem** with a **loss** of $\overline{VE}$
- **Question:** What should we use for $\mu(s)$?

Stochastic Gradient Descent with Known True Values

- Suppose we are given a **new example**: $(S_t, v_\pi(S_t))$
- How should we update our weight vector $\mathbf{w}$?
- **Stochastic Gradient Descent:** After each example, adjust weights a tiny bit in **direction** that would most **reduce error** on **that example**:

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

target

Stochastic Gradient Descent with Unknown True Values

- If we knew $v_{\pi}(s)$, we would be done!
- Instead, we will update toward an **approximate target** U_t :

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

- U_t can be any of our update targets from previous lectures

Gradient Monte Carlo

- **Monte Carlo target:** $U_t = G_t$
- U_t is an **unbiased** estimate of $v_\pi(S_t)$: $\mathbb{E}[U_t | S_t = s] = v_\pi(s)$

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

Input: the policy π to be evaluated

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

Algorithm parameter: step size $\alpha > 0$

Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ arbitrarily (e.g., $\mathbf{w} = \mathbf{0}$)

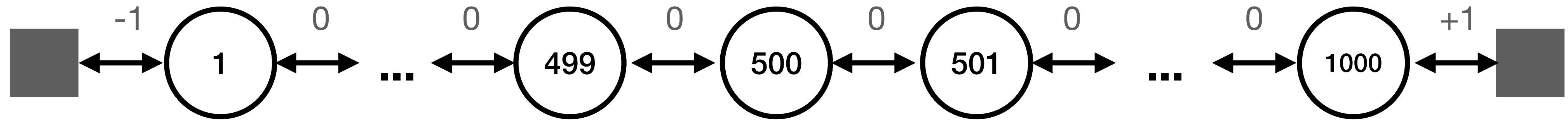
Loop forever (for each episode):

 Generate an episode $S_0, A_0, R_1, S_1, A_1, \dots, R_T, S_T$ using π

 Loop for each step of episode, $t = 0, 1, \dots, T - 1$:

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

State Aggregation



$$\pi(a | s) = 0.5 \quad \forall s \in \mathcal{S}, a \in \{\text{left}, \text{right}\}$$

1D gridworld with states 1 to 1000 with double-ended arrows labelled 0 between them. Left of 1 is a terminal state with an arrow to it labelled -1. Right of 1000 is a terminal state with an arrow to it labelled +1.

- One easy way to reduce the memory usage for a large state space is to **aggregate** states together
- In the Number Line Walk example, we could group the states into 10 groups of 100 states each
- $\mathbf{w}$ is a 10-element vector

- $\hat{v}(s, \mathbf{w}) = \mathbf{w}_{x(s)}$, where $x(s) = \left\lfloor \frac{s}{100} \right\rfloor$

State Aggregation Performance

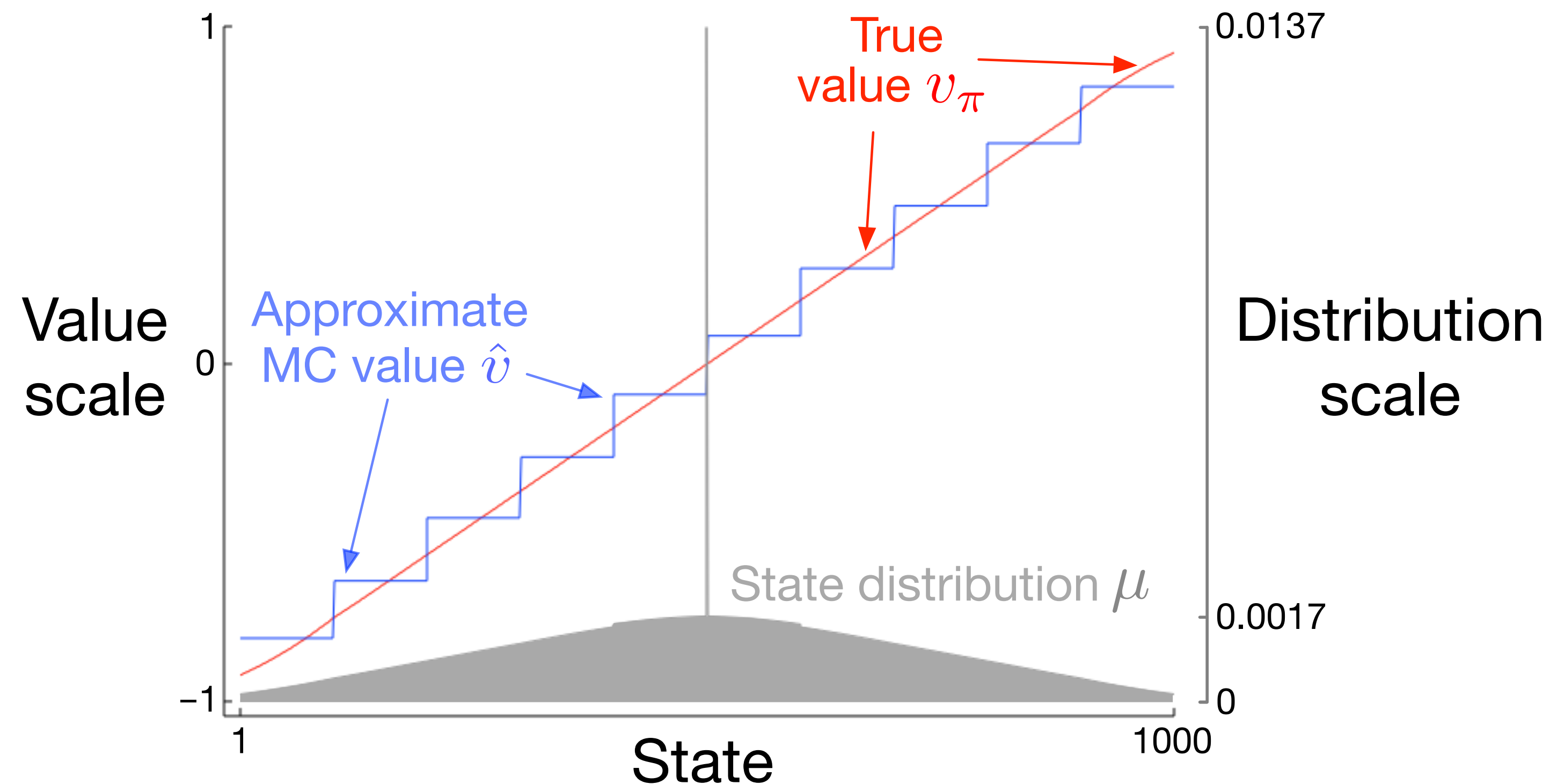


Figure 9.1: Function approximation by state aggregation on the 1000-state random walk task, using the gradient Monte Carlo algorithm (page 202).

Linear Approximation

- Every state $s \in \mathcal{S}$ is assigned a **feature vector** $\mathbf{x}(s)$

$$\mathbf{x}(s) \doteq (x_1(s), x_2(s), \dots, x_d(s))$$

- State-value function approximation:

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

- **Gradient** is easy:

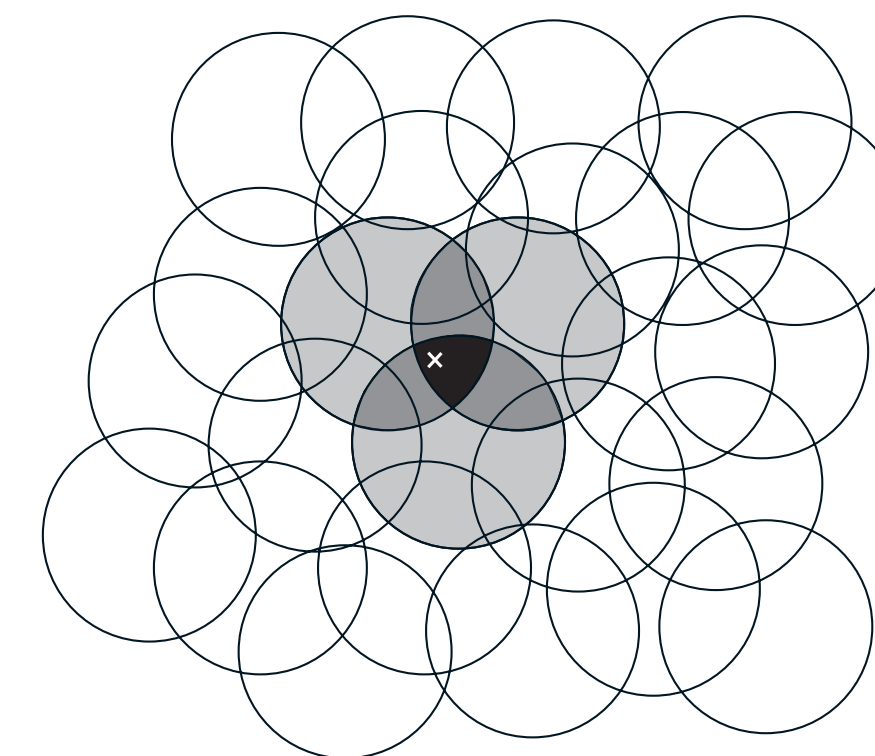
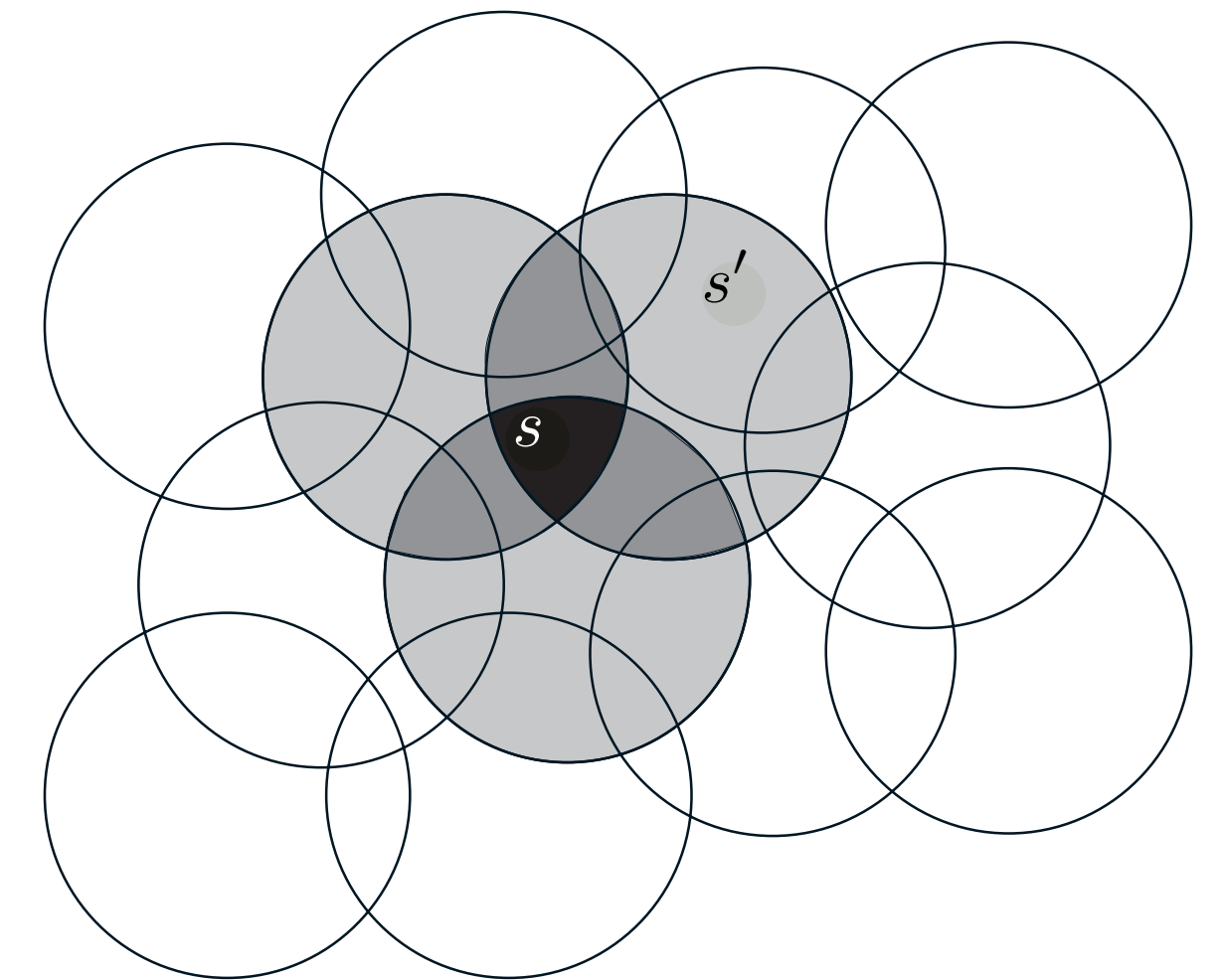
$$\nabla \hat{v}(s, \mathbf{w}) = \mathbf{x}(s)$$

- **Gradient updates** are easy: $\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t + \alpha [U_t - \hat{v}(s, \mathbf{w}_t)] \mathbf{x}(s)$

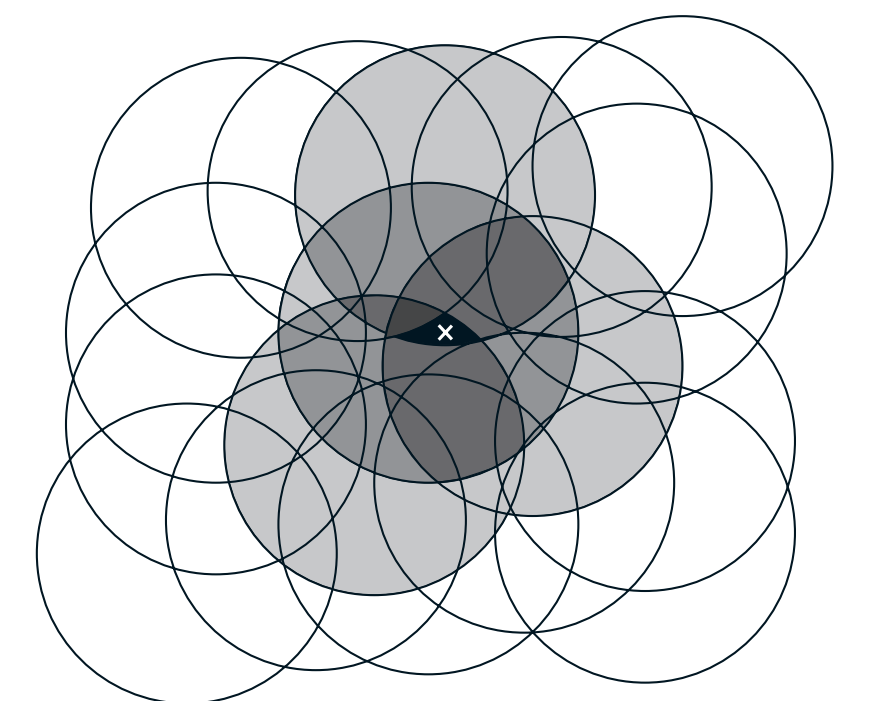
- State aggregation is a **special case** of linear approximation (**why?**)

Feature Construction: Coarse Coding

- Divide state space up into **overlapping cells**
- One **indicator feature** for each cell, set to 1 if the state is in the cell
- This is another form of **state aggregation**
- Updating one state **generalizes** to other states that **share a cell**



Narrow generalization

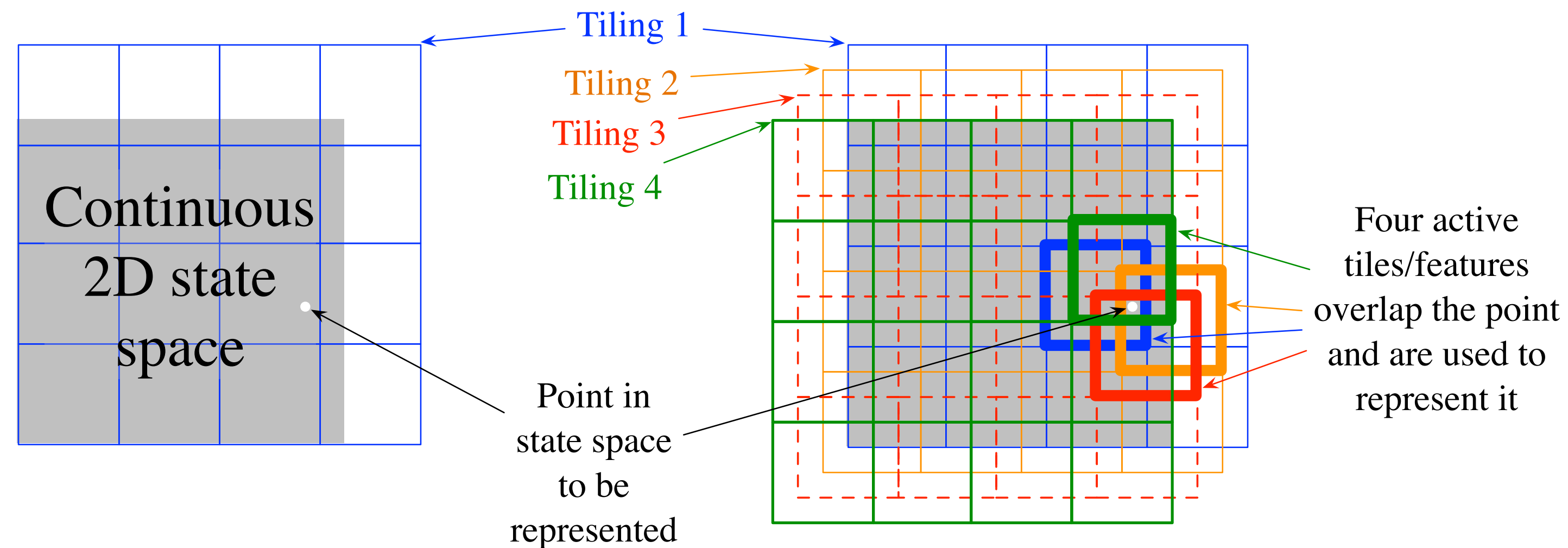


Broad generalization

Another two figures with circles and coloured intersections; the left has many small circles that overlap less and is labelled "Narrow generalization", the right has larger circles that overlap more and is labelled "Broad generalization".

Tile Coding

- The most practical form of coarse coding
- Partition state space into a uniform grid called a **tiling**
 - Use **multiple** tilings that are **offset**



Example of tile coding in a 2D feature space. 4 identical uniform grids, but with random offsets, are overlaid; each grid square that contains a specific point is drawn thicker.

Approaches to Control

1. **Action-value methods** (all previous approaches)
 - Learn the value of **each action** in **each state**: $q_{\pi}(s, a)$
 - Pick the **max-value action** (usually): $\arg \max_a q_{\pi}(s, a)$
2. **Function approximation** (just now)
 - **Prediction**: Learn the **parameters** $\mathbf{w}$ of state-value function $\hat{v}(s, \mathbf{w})$
 - **Control**: Learn the **parameters** $\mathbf{w}$ of action-value function $\hat{q}(s, \mathbf{w})$
3. **Policy-gradient methods** (rest of today)
 - Learn the **parameters** θ of a **policy** $\pi(a \mid s, \theta)$
 - Update by **gradient ascent** in performance

Parameterized Policies

- The action probabilities of a **parameterized policy** $\pi(a \mid s, \theta)$ are set by setting the values of a **parameter vector** $\theta \in \mathbb{R}^{d'}$
- Common approach: **softmax in action preferences**
 - Learn an **action preference function** $h(s, a, \theta)$
 - **Softmax** over action preferences gives action probabilities:

$$\pi(a \mid s, \theta) \doteq \frac{e^{h(s,a,\theta)}}{\sum_{a'} e^{h(s,a',\theta)}}$$

Action Preferences

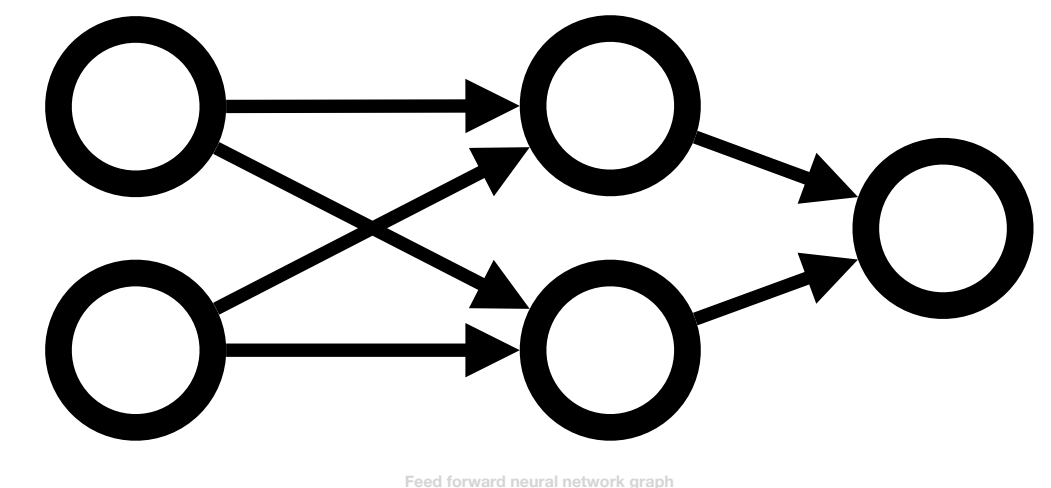
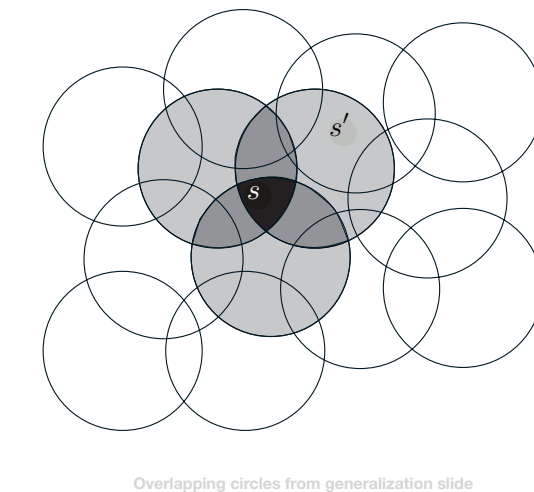
- **Question:** What **functional forms** can we use for action preferences?
- Anything we could have used for $\hat{v}$:

- **Linear approximations:**

$$h(s, a, \theta) \doteq \theta^T \mathbf{x}(s) = \sum_{i=1}^d \theta_i x_i(s)$$

- Including state aggregation, coarse coding, tile coding

- **Neural network:** θ are weights, offsets, kernels, etc.



Parameterized Policies Advantage: Deterministic Action

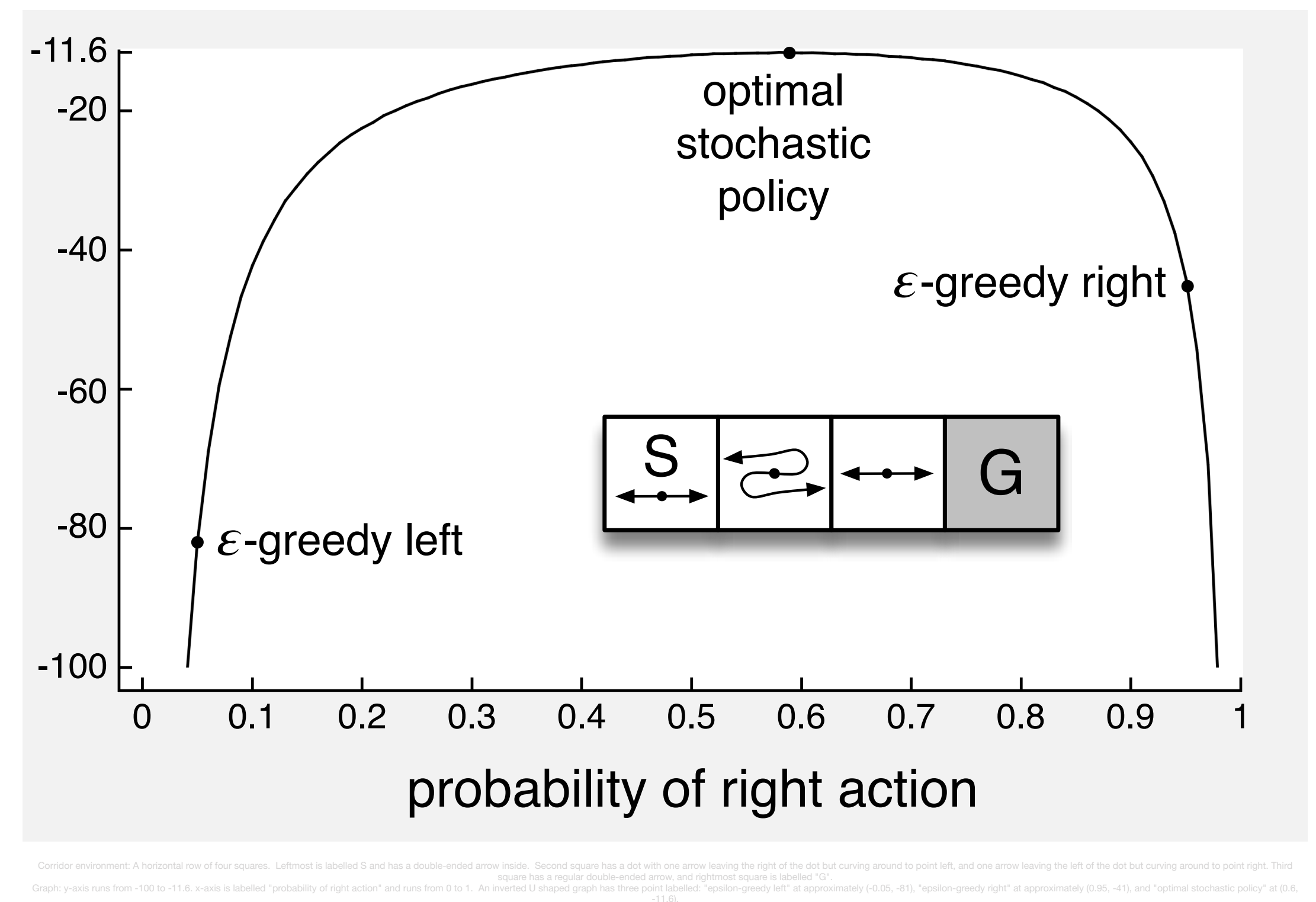
- The **optimal policy** $\pi^*(a \mid s) = \arg \max_a q^*(s, a)$ is typically **deterministic**
- If we run an ϵ -soft policy, we cannot get to an optimal policy
 - **Every action** is played either with probability ϵ or $(1 - \epsilon)$
- Softmax in **action preference policies** can learn **arbitrary probabilities**, because $h(s, a, \theta)$ is completely **unconstrained**:

$$\pi(a \mid s, \theta) \doteq \frac{e^{h(s, a, \theta)}}{\sum_{a'} e^{h(s, a', \theta)}}$$

- **Question:** How can a softmax in action preferences policy converge to a deterministic policy?
- **Question:** Can you get the same results with $h(s, a, \theta) = \hat{q}(s, a, \theta)$? (**why?**)

Example: Switcheroo Corridor

- Actions **left** and **right** have usual effect
- Except in one state they are **reversed**!
- Function approximation makes **all** the states look **identical**
- **Optimal policy** is **stochastic**, with $\Pr(\text{right}) \approx 0.59$
- But ϵ -greedy policies can only pick $\Pr(\text{right})$ of ϵ or $(1 - \epsilon)$!



Parameterized Policies Advantage: Stochastic Actions

- Optimal policies are **deterministic**, but only when there is no **state aggregation**
- When **function approximation** makes states look the same, or when states are **imperfectly observable**, the optimal policy might be an **arbitrary probability distribution**
- Parameterized policies can represent **arbitrary** distributions
 - Although not necessarily arbitrary distributions in **every possible state** (**why not?**)

Policy Performance

- We choose the policy parameters θ in order to maximize the **performance** of the policy: $J(\theta)$
- **Question:** What should $J(\theta)$ be in episodic cases?
- **Expected returns** to the policy specified by θ :

$$J(\theta) \doteq \mathbb{E}_{\pi_{\theta}} [G_0]$$

- With special **single starting state** s_0 :

$$J(\theta) \doteq v_{\pi_{\theta}}(s_0)$$

Policy Gradient Ascent

1. Want to **maximize performance**: $J(\theta) = v_{\pi_\theta}(s_0)$
2. Gradient gives direction that **J increases**: $\nabla_\theta J(\theta)$
3. Update parameters in **direction of the gradient**:

$$\begin{aligned}\theta_{t+1} &\leftarrow \theta_t + \alpha \nabla_\theta J(\theta_t) \\ &= \theta_t + \alpha \boxed{\nabla_\theta v_{\pi_\theta}(s_0)}\end{aligned}$$

Oops!

Policy Gradient Theorem

- The **gradient of the policy** $\nabla J(\theta)$ is just the gradient of the value function with respect to the policy $v_{\pi_\theta}(s_0)$
- But we **don't know** the gradient of the **value function**!

Policy Gradient Theorem:

$$\nabla J(\theta) \propto \sum_s \mu(s) \sum_a q_\pi(s, a) \nabla \pi(a | s, \theta)$$

on-policy stationary distribution true action values gradient of **policy**

Monte Carlo Policy Gradient

$$\begin{aligned}
 \nabla J(\theta) &\propto \sum_s \mu(s) \sum_a q_\pi(s, a) \nabla \pi(a | s, \theta) && \sum_s \Pr(s) f(s) = \mathbb{E}[f(S)] \\
 &= \mathbb{E}_\pi \left[\sum_a q_\pi(S_t, a) \nabla \pi(a | S_t, \theta) \right] \\
 &= \mathbb{E}_\pi \left[\sum_a q_\pi(S_t, a) \nabla \pi(a | S_t, \theta) \frac{\pi(a | S_t, \theta)}{\pi(a | S_t, \theta)} \right] \\
 &= \mathbb{E}_\pi \left[\sum_a \pi(a | S_t, \theta) q_\pi(S_t, a) \frac{\nabla \pi(a | S_t, \theta)}{\pi(a | S_t, \theta)} \right] && \sum_a \Pr(a) f(a) = \mathbb{E}[f(A)] \\
 &= \mathbb{E}_\pi \left[q_\pi(S_t, A_t) \frac{\nabla \pi(A_t | S_t, \theta)}{\pi(A_t | S_t, \theta)} \right] \\
 &= \mathbb{E}_\pi \left[G_t \frac{\nabla \pi(A_t | S_t, \theta)}{\pi(A_t | S_t, \theta)} \right] && \mathbb{E} [\mathbb{E}[f(A)]] = \mathbb{E}[f(A)]
 \end{aligned}$$

$f(s)$
 $f(a)$

Monte Carlo Policy Gradient

Algorithm: REINFORCE

$$\text{REINFORCE Update: } \theta_{t+1} \leftarrow \theta_t + \alpha G_t \frac{\nabla \pi(A_t | S_t, \theta_t)}{\pi(A_t | S_t, \theta_t)}$$

REINFORCE: Monte-Carlo Policy-Gradient Control (episodic) for π_*

Input: a differentiable policy parameterization $\pi(a|s, \theta)$

Algorithm parameter: step size $\alpha > 0$

Initialize policy parameter $\theta \in \mathbb{R}^{d'}$ (e.g., to $\mathbf{0}$)

Loop forever (for each episode):

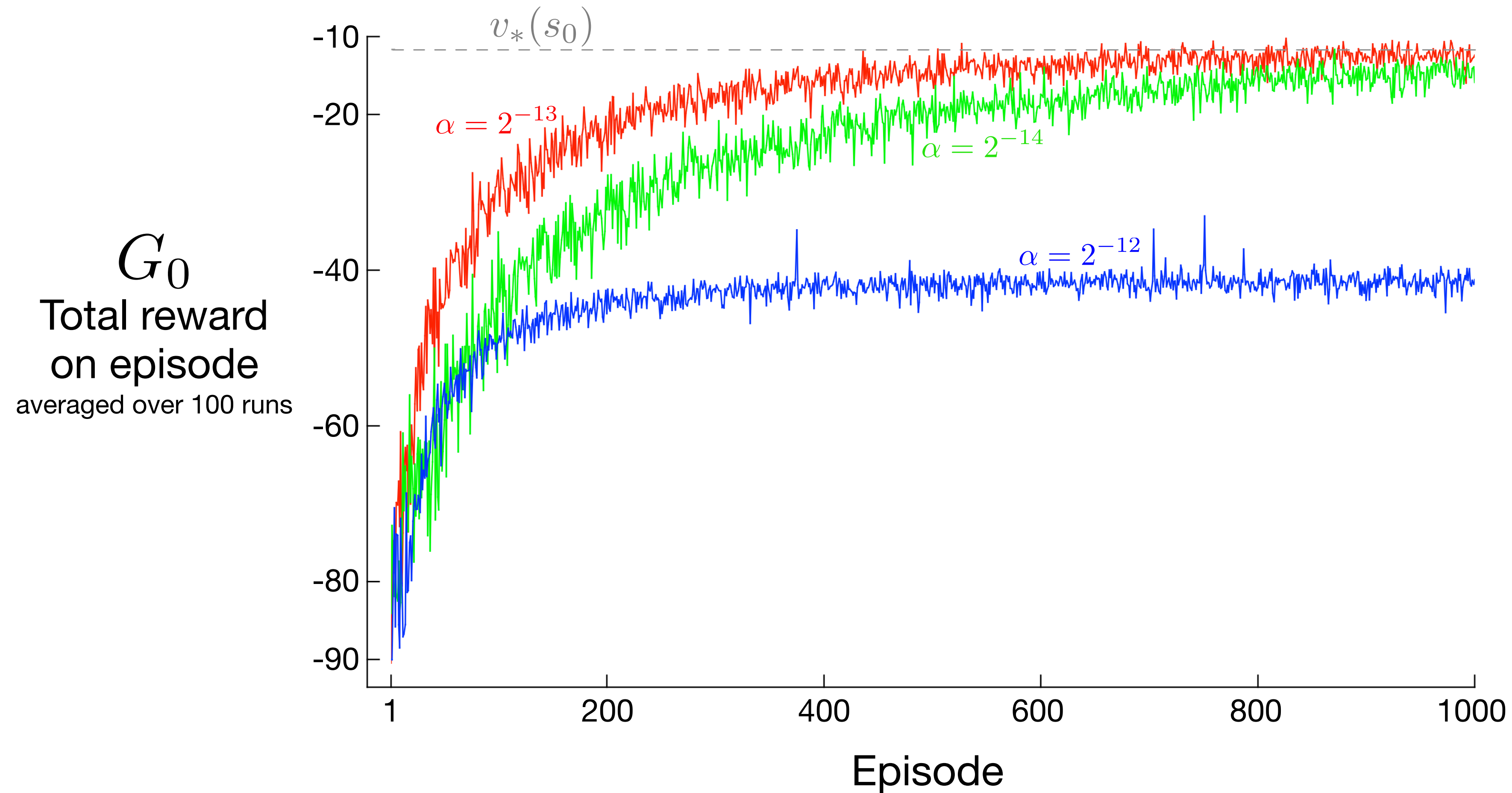
Generate an episode $S_0, A_0, R_1, \dots, S_{T-1}, A_{T-1}, R_T$, following $\pi(\cdot|\cdot, \theta)$

Loop for each step of the episode $t = 0, 1, \dots, T - 1$:

$$G \leftarrow \sum_{k=t+1}^T \gamma^{k-t-1} R_k \quad (G_t)$$
$$\theta \leftarrow \theta + \alpha \gamma^t G \underbrace{\nabla \ln \pi(A_t | S_t, \theta)}_{\text{"eligibility function"}}$$

$$\frac{\nabla \pi(A_t | S_t, \theta)}{\pi(A_t | S_t, \theta)} \quad \text{"eligibility function"} \quad \left(\nabla \ln x = \frac{\nabla x}{x} \right)$$

REINFORCE Performance in Switcheroo Corridor



Graph with x-axis labelled "Episode" and y-axis labelled " G_0 Total reward on episode averaged over 100 runs". One trace labelled $\alpha=2^{-12}$ converges to -45 after approximately 150 episodes; another labelled $\alpha=2^{-14}$ converges to approximately -11 after 1000 episodes; another labelled $\alpha=2^{-13}$ converges to approximately -11 after approximately 700 episodes. A horizontal dashed line at approximately -11 is labelled $v_*(s_0)$.

Summary

- It is often impractical to track the estimated value for **every possible state** and/or action
- **Parameterized value function** $\hat{v}(s, \mathbf{w})$ uses weights $\mathbf{w} \in \mathbb{R}^d$ to specify the values of states
 - Weights can be set using **gradient descent** and **semi-gradient descent**
- All our previous control algorithms were **action-value** methods
 1. Approximate the action-value $q^*(s, a)$
 2. Choose maximal-value action at every state
- **Policy gradient** methods:
 1. Represent policies using **parametric policy** $\pi(s \mid a, \theta)$
 2. **Directly optimize** performance $J(\theta)$ by adjusting θ
- **Policy Gradient Theorem** lets us restate $J(\theta)$ in terms of quantities that we **know** ($\nabla \pi$) or can **approximate** (q_π)
- REINFORCE uses a particular **estimation scheme** for policy gradients