Function Approximation

CMPUT 366: Intelligent Systems

S&B §9.0-9.5.4

Lecture Outline

1. Recap & Logistics
2. Parameterized Value Functions
3. Gradient Descent
4. Approximation Schemes

Assignments

- **Assignment #3** was due last Friday
- Late submissions (for 20% penalty) will be accepted until **Tuesday** (Mar 29) at **11:59pm**
  - This is a firm deadline
- **Assignment #4** is available now on eclass
  - Deadline: Friday Apr 15, 11:59pm

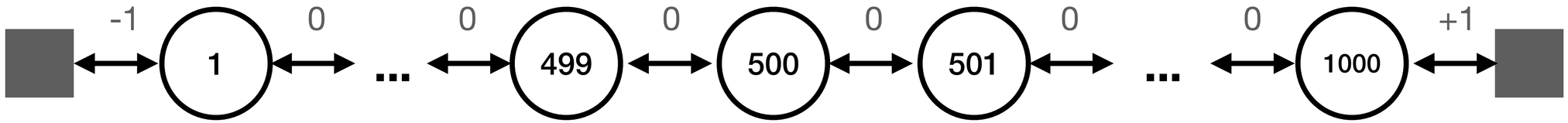
Recap: TD Learning

- 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



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

- **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?

(Image: Sutton & Barto, 2018)

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 the feature weights
  - $\hat{v}$ could be a **neural network**: $\mathbf{w}$ is the 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**: $\left(S_t, v_\pi(S_t)\right)$
- 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**:

target

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

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 $v_\pi(S_t)$: $\mathbb{E}[U_t \mid S_t = s] = v_\pi(s)$

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

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S} \times \mathbb{R}^d \to \mathbb{R}$
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, $t = 0, 1, \ldots, T-1$:
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[G_t - \hat{v}(S_t,\mathbf{w})\big]\nabla\hat{v}(S_t,\mathbf{w})$

Semi-gradient

- **TD(0) target:** $U_t = R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t)$
- Bootstrapping targets like TD(0) depend on the **current value** of $\mathbf{w}_t$, so they are **not unbiased**
- Gradient $\nabla \hat{v}(s, \mathbf{w}_t)$ accounts for change in the **estimate** from change in $\mathbf{w}_t$
- But updates to $\mathbf{w}$ change both the **estimate** *and* the **target**
- We call these updates **semi-gradient** updates

Semi-gradient TD(0)

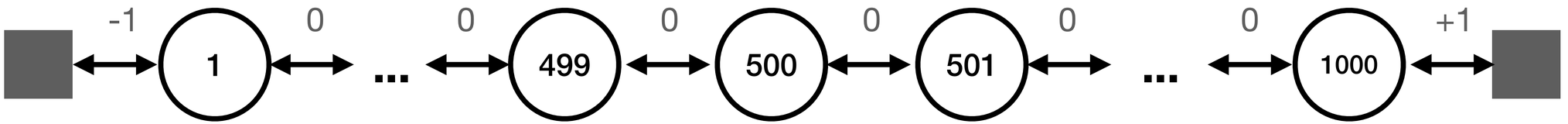
- **TD(0) target:** $U_t = R_{t+1} + \gamma \hat{v}(S_{t+1}, \mathbf{w}_t)$

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

State Aggregation



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

- 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 $x(s) = \left\lfloor \frac{s}{100} \right\rfloor$

(Image: Sutton & Barto, 2018)

State Aggregation Performance

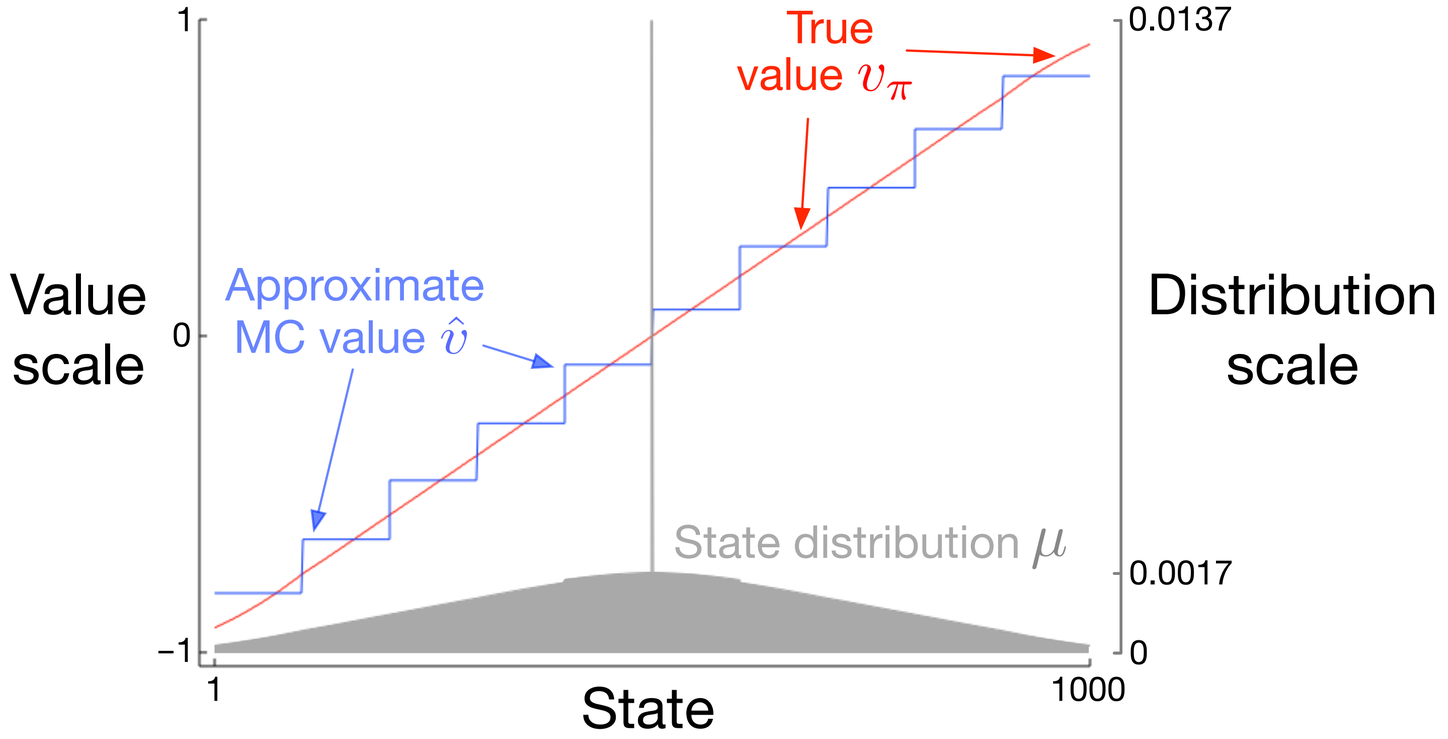


Figure 9.1: Function approximation by state aggregation on the 1000-state random walk task, using the gradient Monte Carlo algorithm (page 202).

(Image: Sutton & Barto, 2018)

Linear Approximation

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

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

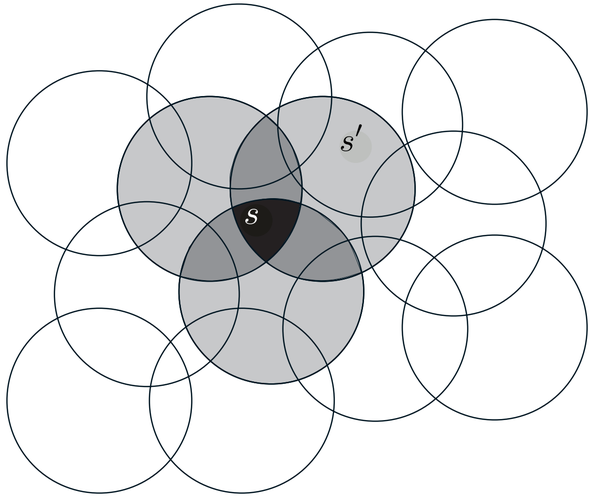
- State-value function approximation:

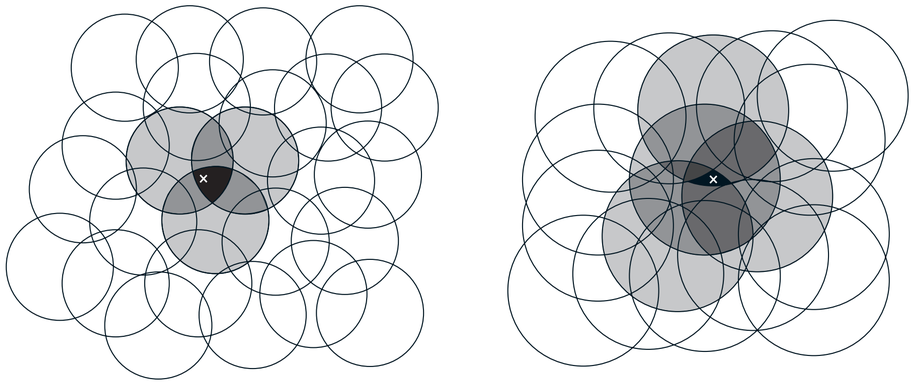
$$\hat{v}(s, \mathbf{w}) \doteq \mathbf{w}^T \mathbf{x}(s) = \sum_{i=1}^{d} w_i x_i(s)$$

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





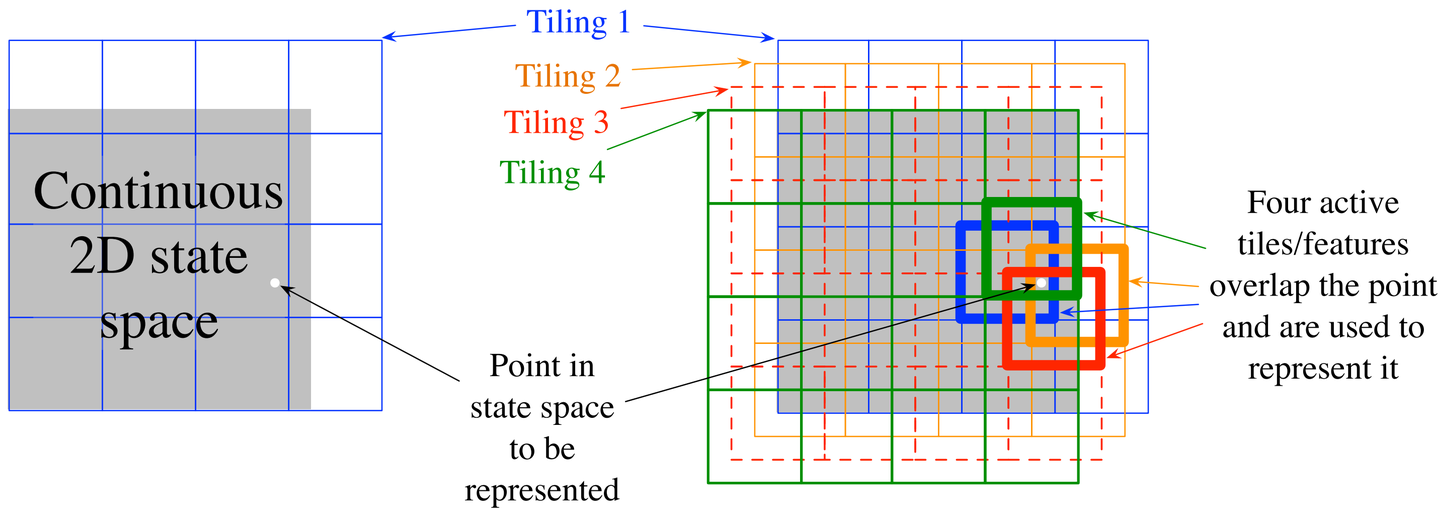
Narrow generalization

Broad generalization

(Image: Sutton & Barto, 2018)

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**



(Image: Sutton & Barto, 2018)

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**
- Most efficient forms of approximation:
  Linear approximations, especially **coarse coding** and **tile coding**