

Lecture Schedule

Dynamical programming

- 1 The finite-horizon decision problem
31 January
- 2 Dynamical Programming
7 February
- 3 DP reformulations and introduction to Control
14 February

Control

- 4 Discretization and PID control
21 February
- 5 Direct methods and control by optimization
28 February
- 6 Linear-quadratic problems in control
7 March
- 7 Linearization and iterative LQR
14 March

Syllabus: <https://02465material.pages.compute.dtu.dk/02465public>
Help improve lecture by giving feedback on DTU learn

Reinforcement learning

- 8 Exploration and Bandits
21 March
- 9 Policy and value iteration
4 April
- 10 Monte-carlo methods and TD learning
11 April
- 11 **Model-Free Control with tabular and linear methods**
18 April
- 12 Eligibility traces and value-function approximations
25 April
- 13 Q-learning and deep-Q learning
2 May

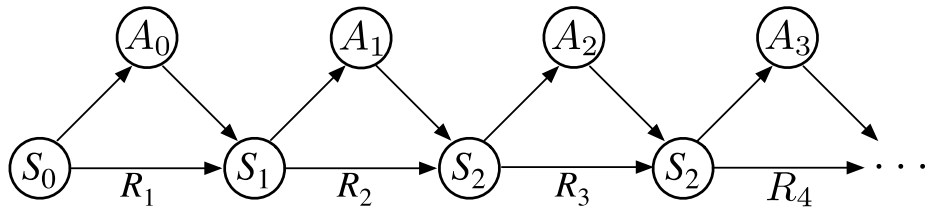
Reading material:

- [SB18, Chapter 6.4-6.5; 7-7.2; 9-9.3; 10.1]

Learning Objectives

- Sarsa on-policy learning
- Q off-policy learning
- the n-step return
- value-function approximations and linear methods

Recap: First-Visit Monte-Carlo value estimation



We want to calculate the value function $v_\pi(s) = \mathbb{E}[G_t | S_t = s]$.

Simulate an episode of experience $s_0, a_0, r_1, s_1, a_1, r_2, \dots, r_T$ using π

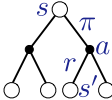
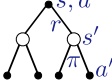
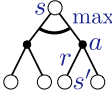
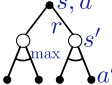
- **First** step t we visit a state s
- Measure return $G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots$ for rest of the episode
- Estimate value function as $v_\pi(s_t) = \mathbb{E}[G_t | S_t = s] \approx \frac{1}{n} \sum_{i=1}^n G_t^{(n)}$
- The average can be computed incrementally:

$$V(s) \leftarrow V(s) + \frac{1}{n} (G_t - V(s))$$

- We use a fixed learning rate α

$$V(s) \leftarrow V(s) + \alpha (G_t - V(s))$$

Dynamical Programming

Bellman equation	Learning algorithm
Bellman expectation equation for v_π $v_\pi(s) = \mathbb{E}_\pi [R + \gamma v_\pi(S') s]$	Iterative policy evaluation to learn v_π $V(s) \leftarrow \mathbb{E}_\pi [R + \gamma V(S') s]$ 
Bellman expectation equation for q_π $q_\pi(s, a) = \mathbb{E}_\pi [R + \gamma q_\pi(S', A') s, a]$	Iterative policy evaluation to learn q_π $Q(s, a) \leftarrow \mathbb{E}_\pi [R + \gamma Q(S', A') s, a]$ 
Policy iteration: Use policy evaluation to estimate v_π or q_π Improve by acting greedily: $\pi'(s) \leftarrow \arg \max_a q_\pi(s, a)$	
Bellman optimality equation for v_* $v_*(s) = \max_a \mathbb{E} [R + \gamma v_*(S') s, a]$	Value iteration $V(s) \leftarrow \max_a \mathbb{E} [R + \gamma V(S') s, a]$ 
Bellman optimality equation for q_* $q_*(s, a) = \mathbb{E} [R + \gamma \max_{a'} q_*(S', a') s, a]$	Q-value iteration $Q(s, a) \leftarrow \mathbb{E} [R + \gamma \max_{a'} Q(S', a') s, a]$ 

TD and MC value estimation

- Recall $v_\pi(s) = \mathbb{E}[G_t | S_t = s]$
- MC learning: G_t estimate of $v_\pi(s)$; update:

$$V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t))$$

- Bellman equation:

$$v_\pi(s) = \mathbb{E}[R_{t+1} + \gamma V(S_{t+1}) | S_t = s]$$

- TD learning: $R_{t+1} + \gamma V(S_{t+1})$ is also an estimate of $v_\pi(s)$; update:

$$V(S_t) \leftarrow V(S_t) + \alpha(R_{t+1} + \gamma V(S_{t+1}) - V(S_t))$$

- TD learning has several advantages
 - Lower variance
 - Don't have to wait for episode to finish
- Natural idea: Apply TD to $Q(s, a)$
 - Still ϵ -greedy policy improvement
 - Update Q estimates at each time step

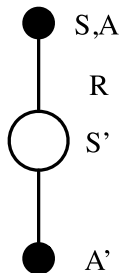
- Bellman equation:

$$q_{\pi}(s, a) = \mathbb{E}[R_{t+1} + \gamma q_{\pi}(S_{t+1}, A_{t+1}) | S_t = s, A_t = a]$$

- Implies $R_{t+1} + \gamma q_{\pi}(S_{t+1}, A_{t+1})$ is an estimate of $q_{\pi}(s, a)$
- Implies the update equation

$$Q(S, A) \leftarrow Q(S, A) + \alpha (R + \gamma Q(S', A') - Q(S, A))$$

- We use bootstrapping (i.e. biased estimate)



Sarsa (on-policy TD control) for estimating $Q \approx q_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$

Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+$, $a \in \mathcal{A}(s)$, arbitrarily except that $Q(\text{terminal}, \cdot) = 0$

Loop for each episode:

 Initialize S

 Choose A from S using policy derived from Q (e.g., ε -greedy)

 Loop for each step of episode:


 Take action A , observe R, S'

 Choose A' from S' using policy derived from Q (e.g., ε -greedy)

$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma Q(S', A') - Q(S, A)]$

$S \leftarrow S'; A \leftarrow A';$

 until S is terminal

 lecture_11_sarsa.py

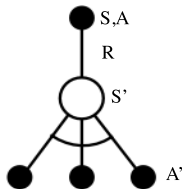
Convergence of Sarsa

Sarsa converge to optimal action-value function $Q \rightarrow q_*$ assuming

- GLIE sequence of policies (decreasing but non-trivial exploration)
- Robbins-Monro sequence of step-sizes α_t

$$\sum_{t=1}^{\infty} \alpha_t = \infty, \quad \sum_{t=1}^{\infty} \alpha_t^2 < \infty$$

Using the Bellman optimality equation



- Bellman equation:

$$q_*(s, a) = \mathbb{E} \left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \mid S_t = s, A_t = a \right]$$

- Implies $R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a')$ is a Monte-Carlo estimate of $q_*(s, a)$
- Implied update equation

$$Q(S, A) \leftarrow Q(S, A) + \alpha \left(R + \gamma \max_{a'} Q(S', a') - Q(S, A) \right)$$

- Note we use bootstrapping (i.e. biased estimate)

$$Q(S, A) \leftarrow Q(S, A) + \alpha \left(R + \gamma \max_{a'} Q(S', a') - Q(S, A) \right)$$

- The **behavior policy** determines which S_t, A_t are visited
- The environment determines what happens next (S')
- The Q -values are updated **without** reference to the **behavior policy**
- Q -learning is therefore **off-policy**

Q-learning (off-policy TD control) for estimating $\pi \approx \pi_*$

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$

Initialize $Q(s, a)$, for all $s \in \mathcal{S}^+$, $a \in \mathcal{A}(s)$, arbitrarily except that $Q(\text{terminal}, \cdot) = 0$

Loop for each episode:

 Initialize S

 Loop for each step of episode:


 Choose A from S using policy derived from Q (e.g., ε -greedy)

 Take action A , observe R, S'

$Q(S, A) \leftarrow Q(S, A) + \alpha [R + \gamma \max_a Q(S', a) - Q(S, A)]$

$S \leftarrow S'$

 until S is terminal

 lecture_11_q.py

- a. The first step in training a Q -learning agent is to compute the set of all states the agent can be in
- b. The Q -table $Q(s, a)$ in Q -learning is a measure of the reward the agent will obtain in the very next step multiplied by γ
- c. Q -learning still works if we initialize the Q -table to -1 , i.e. $Q(s, a) = -1$ for all $s \in \mathcal{S}$
- d. When Q -learning is applied to a deterministic environment, the agent will follow a deterministic policy
- e. Don't know.

Convergence of Q -learning

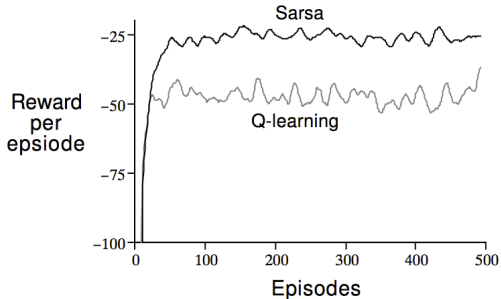
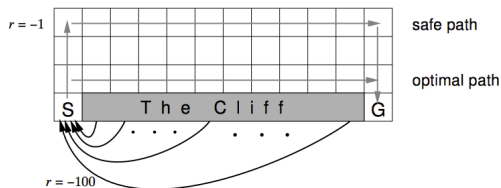
Q -learning converge to optimal action-value function $Q \rightarrow q_*$ assuming



- All s, a pairs visited infinitely often
- Robbins-Monro sequence of step-sizes α_t

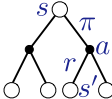

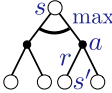
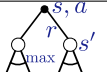
$$\sum_{t=1}^{\infty} \alpha_t = \infty, \quad \sum_{t=1}^{\infty} \alpha_t^2 < \infty$$

Comparing Q-learning and SARSA

- Reward -100 if we fall
- Reward -1 per step
- Both use ϵ -greedy exploration

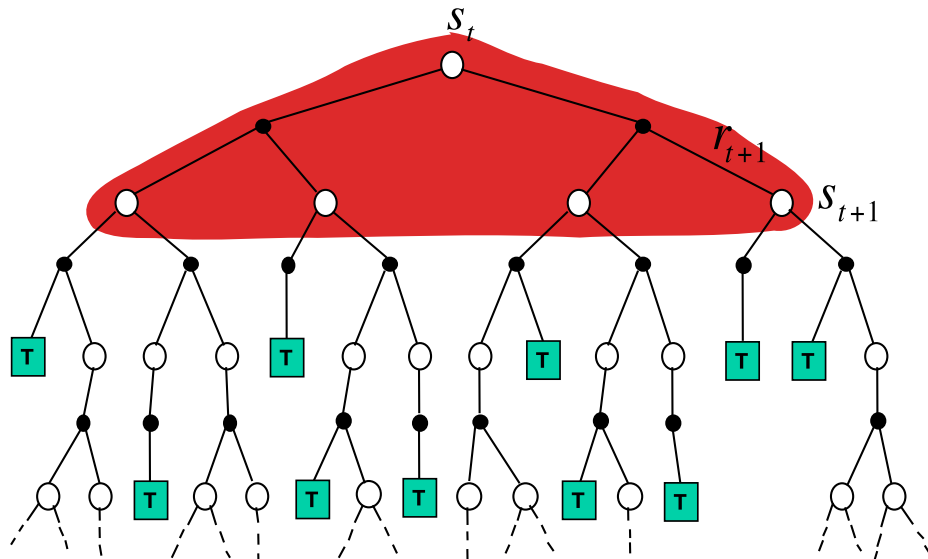


 `lecture_11_sarsa_cliff.py` ,  `lecture_11_q_cliff.py`

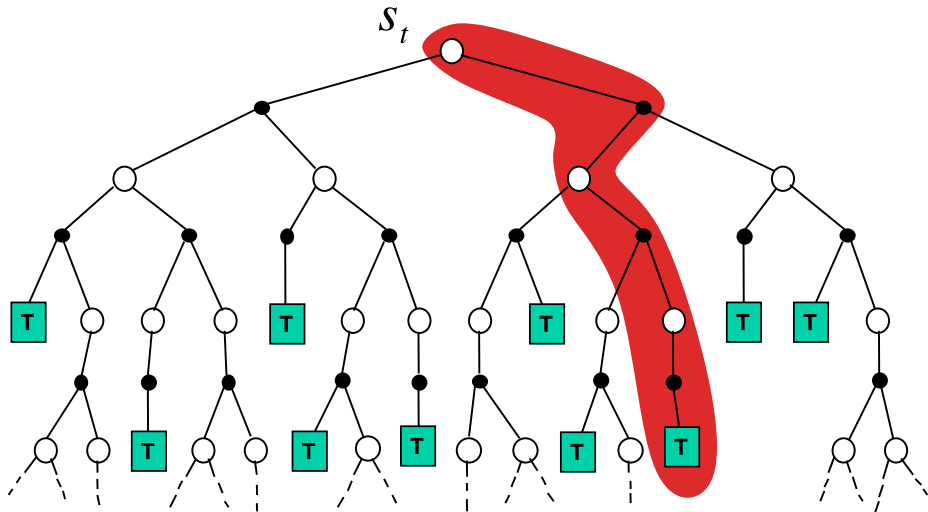
Bellman equation	Learning algorithm	TD Learning $V(S) \stackrel{\alpha}{\leftarrow} R + \gamma V(S')$
Bellman expectation equation for v_π $v_\pi(s) = \mathbb{E}_\pi [R + \gamma v_\pi(S') s]$	Iterative policy evaluation to learn v_π $V(s) \leftarrow \mathbb{E}_\pi [R + \gamma V(S') s]$	
Bellman expectation equation for q_π $q_\pi(s, a) = \mathbb{E}_\pi [R + \gamma q_\pi(S', A') s, a]$	Iterative policy evaluation to learn q_π $Q(s, a) \leftarrow \mathbb{E}_\pi [R + \gamma Q(S', A') s, a]$	
<p>Policy iteration: Use policy evaluation to estimate v_π or q_π</p> <p>Improve by acting greedily: $\pi'(s) \leftarrow \arg \max_a q_\pi(s, a)$</p>		
Bellman optimality equation for v_* $v_*(s) = \max_a \mathbb{E} [R + \gamma v_*(S') s, a]$	Value iteration $V(s) \leftarrow \max_a \mathbb{E} [R + \gamma V(S') s, a]$	
Bellman optimality equation for q_* $q_*(s, a) = \mathbb{E} [R + \gamma \max_{a'} q_*(S', a') s, a]$	Q-value iteration $Q(s, a) \leftarrow \mathbb{E} [R + \gamma \max_{a'} Q(S', a') s, a]$	
<div style="border: 1px solid red; padding: 5px; display: inline-block;"> where $x \stackrel{\alpha}{\leftarrow} y \equiv x \leftarrow x + \alpha(y - x)$ </div>		<div style="border: 1px solid red; padding: 5px; display: inline-block;"> Q-Learning $Q(S, A) \stackrel{\alpha}{\leftarrow} R + \gamma \max_{a' \in \mathcal{A}} Q(S', a')$ </div>

From two weeks ago: DP backups

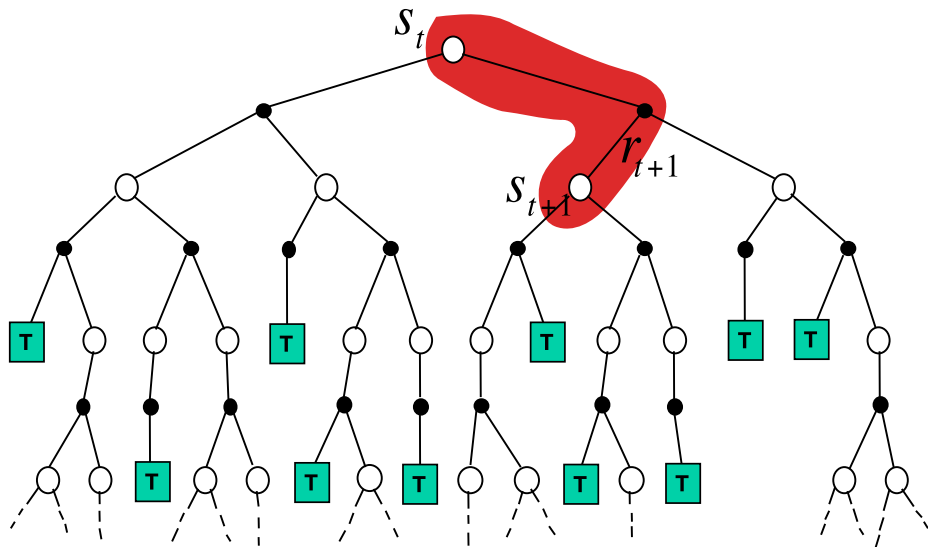
$$V(S_t) \leftarrow \mathbb{E}_\pi [R_{t+1} + \gamma V(S_{t+1})]$$



$$V(S_t) \leftarrow V(S_t) + \alpha(G_t - V(S_t))$$



$$V(S_t) \leftarrow V(S_t) + \alpha (R_{t+1} + \gamma V(S_{t+1}) - V(S_t))$$



- **Bootstrapping:** Update involves an estimate (e.g. V)
 - TD and DP bootstraps
 - MC does not bootstrap
- **Sampling:** Update involves a sample estimate of an expectation
 - MC and TD sample
 - DP does not sample

Let's combine methods and avoid either/or choices

- Recall return is $G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \dots$

$$n = 1: \text{ (TD)} \quad G_t^{(1)} = R_{t+1} + \gamma G_{t+1}$$

$$n = 2: \quad G_t^{(2)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 G_{t+2}$$

$$n: \quad G_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \dots + \gamma^{n-1} R_{t+n} + \gamma^n G_{t+n}$$

$$n = \infty \text{ (MC): } G_t^{(\infty)} = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{T-1} R_T$$

- Using the rules of expectations:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{n-1} R_{t+n} + \gamma^n G_{t+n} | s] \\ &= \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{n-1} R_{t+n} + \mathbb{E}[\gamma^n G_{t+n} | S_{t+n}] | S_t = s] \\ &= \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{n-1} R_{t+n} + \gamma^n v_\pi(S_{t+n}) | S_t = s] \end{aligned}$$

Therefore, the *n*-step return is an estimate of $V(S_t)$

$$G_{t:t+n} = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n})$$

- This gives *n*-step temporal difference update:

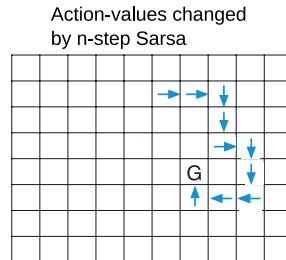
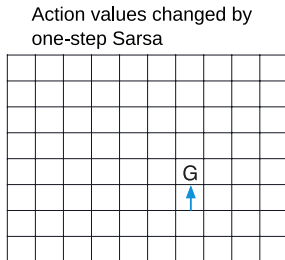
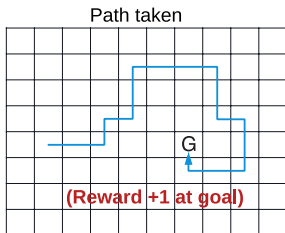
$$V(S_t) \leftarrow V(S_t) + \alpha (G_{t:t+n} - V(S_t))$$

n-step TD: Implementation details

$$G_t^{(n)} = R_{t+1} + \gamma R_{t+1} + \dots + \gamma^{n-1} R_{t+n} + \gamma^n V(S_{t+n})$$

$$V(S_t) \leftarrow V(S_t) + \alpha \left(G_t^{(n)} - V(S_t) \right)$$

- We cannot compute $G_t^{(n)}$ until we have the n next steps episodes
 - Maintain buffer of size n
- At end of episode, we are still missing $n - 1$ updates
 - Do a for-loop and perform missing updates



n-step TD for estimating $V \approx v_\pi$

Input: a policy π

Algorithm parameters: step size $\alpha \in (0, 1]$, a positive integer n

Initialize $V(s)$ arbitrarily, for all $s \in \mathcal{S}$

All store and access operations (for S_t and R_t) can take their index mod $n + 1$

Loop for each episode:

 Initialize and store $S_0 \neq$ terminal

$T \leftarrow \infty$

 Loop for $t = 0, 1, 2, \dots$:

 | If $t < T$, then:

 | Take an action according to $\pi(\cdot|S_t)$

 | Observe and store the next reward as R_{t+1} and the next state as S_{t+1}

 | If S_{t+1} is terminal, then $T \leftarrow t + 1$

 | $\tau \leftarrow t - n + 1$ (τ is the time whose state's estimate is being updated)

 | If $\tau \geq 0$:

 | $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n, T)} \gamma^{i-\tau-1} R_i$

 | If $\tau + n < T$, then: $G \leftarrow G + \gamma^n V(S_{\tau+n})$ ($G_{\tau:\tau+n}$)

 | $V(S_\tau) \leftarrow V(S_\tau) + \alpha [G - V(S_\tau)]$

 Until $\tau = T - 1$

Recall the decomposition:

$$G_t = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{n-1} R_{t+n} + \gamma^n G_{t+n}$$

- As before:

$$\begin{aligned} q_\pi(s, a) &= \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{n-1} R_{t+n} + \gamma^n G_{t+n} | S_t = s, A_t = a] \\ &= \mathbb{E}[R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{n-1} R_{t+n} + \gamma^n q_\pi(S_{t+n}, A_{t+n}) | S_t = s, A_t = a] \end{aligned}$$

- Therefore, the following *n*-step action-value return is an unbiased estimate of q_π

$$q_t^{(n)} = R_{t+1} + \gamma R_{t+2} + \dots + \gamma^{n-1} R_{t+n} + \gamma^n q_\pi(S_{t+n}, A_{t+n})$$

- Suggest the following bootstrap update of the action-value function

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha \left(q_t^{(n)} - Q(S_t, A_t) \right)$$

n-step Sarsa for estimating $Q \approx q_*$ or q_π

Initialize $Q(s, a)$ arbitrarily, for all $s \in \mathcal{S}, a \in \mathcal{A}$

Initialize π to be ε -greedy with respect to Q , or to a fixed given policy

Algorithm parameters: step size $\alpha \in (0, 1]$, small $\varepsilon > 0$, a positive integer n

All store and access operations (for S_t, A_t , and R_t) can take their index mod $n + 1$

Loop for each episode:

 Initialize and store $S_0 \neq$ terminal

 Select and store an action $A_0 \sim \pi(\cdot | S_0)$

$T \leftarrow \infty$

 Loop for $t = 0, 1, 2, \dots$:

 | If $t < T$, then:

 | Take action A_t

 | Observe and store the next reward as R_{t+1} and the next state as S_{t+1}

 | If S_{t+1} is terminal, then:

 | $T \leftarrow t + 1$

 | else:

 | Select and store an action $A_{t+1} \sim \pi(\cdot | S_{t+1})$

 | $\tau \leftarrow t - n + 1$ (τ is the time whose estimate is being updated)

 | If $\tau \geq 0$:

 | $G \leftarrow \sum_{i=\tau+1}^{\min(\tau+n, T)} \gamma^{i-\tau-1} R_i$

 | If $\tau + n < T$, then $G \leftarrow G + \gamma^n Q(S_{\tau+n}, A_{\tau+n})$ ($G_{\tau:\tau+n}$)

 | $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha [G - Q(S_\tau, A_\tau)]$

 | If π is being learned, then ensure that $\pi(\cdot | S_\tau)$ is ε -greedy wrt Q

 Until $\tau = T - 1$

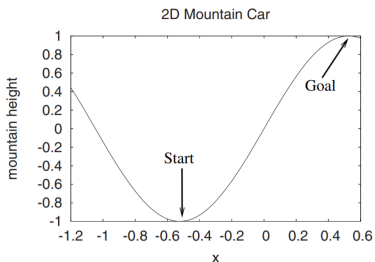
Scaling up reinforcement learning

We want to apply RL to large problems

- Chess: $> 10^{40}$ states
- Go: $> 10^{170}$ states
- Robot arm: continuous state space
- **Example: Mountain-Car** position, velocity. Discrete actions



$$\mathbf{s} = \begin{bmatrix} s_1 \\ s_2 \end{bmatrix} \in \mathbb{R}^2$$



Value-function approximations

Value Function Approximation

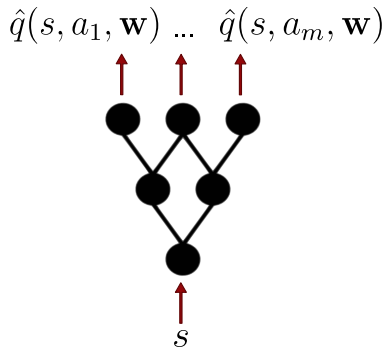
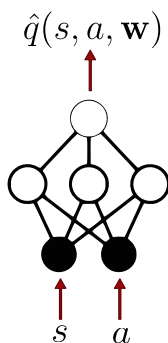
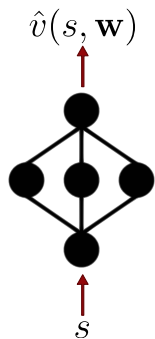


- We have used lookup table representation (stored $Q(s, a)$ as a big table)
 - Every state s has an entry $V(s)$ or
 - Every state-action pair s, a has an entry $Q(s, a)$
- Issues with lookup tables
 - There are too many states and/or actions to store in memory
 - It is too slow to learn the value of each state individually
- Idea:
 - Estimate value function or state-action value with function approximation

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

$$\hat{q}(s, a, \mathbf{w}) \approx q_{\pi}(s, a)$$

- Generalize from seen states to unseen states



Our approximators need to be **differentiable**:

- Neural networks
- Linear combination of features

- Represent value function by a linear combination of features

$$\hat{v}(s, \mathbf{w}) = \mathbf{x}(s)^\top \mathbf{w}, \quad \mathbf{w} \in \mathbb{R}^d$$

Where **feature vector** is defined as:

$$\mathbf{x}(s) = \begin{bmatrix} \mathbf{x}_1(s) \\ \vdots \\ \mathbf{x}_d(s) \end{bmatrix}$$

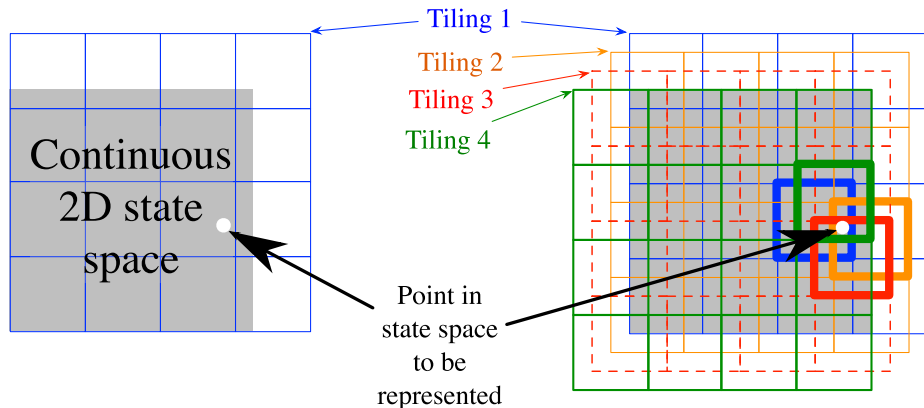
- The gradient is simply:

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

In this case $\hat{q}(s, a, \mathbf{w}) = \mathbf{x}(s, a)^\top \mathbf{w}$

Feature vector construction: Tile coding

- Divide each dimension of s into a number of tiles n_T
- Translate tiles in fraction of tile width to get overlap

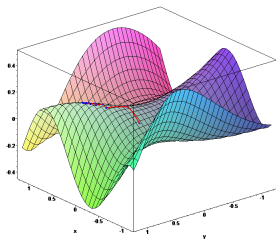


- x has now n_T non-zero elements corresponding to the number of active tiles

Recall from 02450: Gradient Descent

- Let $E(\mathbf{w})$ be a differentiable function of parameter vector \mathbf{w}
- The gradient of $E(\mathbf{w})$ is

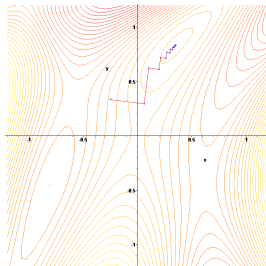
$$\nabla_{\mathbf{w}} E(\mathbf{w}) = \begin{bmatrix} \frac{\partial E(\mathbf{w})}{\partial w_1} \\ \vdots \\ \frac{\partial E(\mathbf{w})}{\partial w_n} \end{bmatrix}$$



- Adjust \mathbf{w} in direction of negative gradient to find a **local minimum** of $E(\mathbf{w})$

$$\mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} E(\mathbf{w})$$

with step-size parameter α (**learning rate**)



- Consider TD learning which implements Bellman equation:

$$v_{\pi}(s) = \mathbb{E}[R + \gamma v(S')|s]$$

- Standard TD update

$$V(s) \leftarrow V(s) + \alpha(r + \gamma V(s') - V(s))$$

- Easy to **plug in** $\hat{v}(s, \mathbf{w})$ instead of $V(s)$ on right-hand side

$$\hat{v}(s, \mathbf{w}) \leftarrow \hat{v}(s, \mathbf{w}) + \alpha(r + \gamma \hat{v}(s', \mathbf{w}) - \hat{v}(s, \mathbf{w}))$$

- **..but how do we update w on the left-hand side so $\hat{v}(s, \mathbf{w})$ agrees with r.h.s.?**

Take a step back: What do we want to do?

- No function approximators: $v(s) = \mathbb{E}[R + \gamma v(S')|s]$
- With function approximators: Find w so that:

$$\hat{v}(s, w) = \mathbb{E}[R + \gamma v(S')|s]$$

- Find w so that:

$$w = \arg \min_w \frac{1}{2} (\hat{v}(s, w) - \mathbb{E}[R + \gamma v(S')|s])^2$$

- Find w using gradient descent:

$$\begin{aligned} w &\leftarrow w + \alpha \nabla_w \frac{1}{2} (\hat{v}(s, w) - \mathbb{E}[R + \gamma v(S')|s])^2 \\ &= w + \alpha (\hat{v}(s, w) - \underbrace{\mathbb{E}[R + \gamma v(S')|s]}_{\approx \frac{1}{B} \sum_{n=1}^B r^{(n)} + v(s'^{(n)})}) \nabla \hat{v}(s, w) \end{aligned}$$

- Use a sample-size of $B = 1$ to compute the average

$$w \leftarrow w + \alpha (\hat{v}(s, w) - r + \gamma v(s')) \nabla \hat{v}(s, w)$$

Summary

- Given $f(x) = \mathbb{E}_z[g(x, z)]$ and approximation-function $\hat{f}(x, \mathbf{w})$
- To find \mathbf{w} such that $\hat{f}(x, \mathbf{w}) \approx f(x)$ iterate:

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \left(g(x, z) - \hat{f}(x, \mathbf{w}) \right) \nabla \hat{f}(x, \mathbf{w})$$

- TD learning: $V(s) = \mathbb{E}[R + \gamma V(S')|s]$ and $\hat{v}(s, \mathbf{w}) \approx v(s)$

$$V(s) \leftarrow V(s) + \alpha (r + \gamma V(s') - V(s))$$

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha (r + \gamma \hat{v}(s', \mathbf{w}) - \hat{v}(s, \mathbf{w})) \nabla \hat{v}(s, \mathbf{w})$$

- Sarsa learning: $q(s, a) = \mathbb{E}[R + \gamma q(S', A')|s, a]$ and $\hat{q}(s, a, \mathbf{w}) \approx q(s, a)$

$$q(s, a) \leftarrow q(s, a) + \alpha (r + \gamma q(s', a') - q(s, a))$$

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha (r + \gamma \hat{q}(s', a', \mathbf{w}) - \hat{q}(s, a, \mathbf{w})) \nabla \hat{q}(s, a, \mathbf{w})$$

- Q-learning: $q(s, a) = \mathbb{E}[R + \gamma \max_{a'} q(S', a')|s, a]$ and $\hat{q}(s, a, \mathbf{w}) \approx q(s, a)$

$$q(s, a) \leftarrow q(s, a) + \alpha (r + \gamma \max_{a'} q(s', a') - q(s, a))$$

$$\mathbf{w} \leftarrow \mathbf{w} + \alpha \left(r + \gamma \max_{a'} \hat{q}(s', a', \mathbf{w}) - \hat{q}(s, a, \mathbf{w}) \right) \nabla \hat{q}(s, a, \mathbf{w})$$

- Remember that $\nabla \hat{q}(s, a, \mathbf{w}) = \mathbf{x}(s, a)$ and $\nabla v(s, \mathbf{w}) = \mathbf{x}(s)$

Quiz: Linear function approximators

Which of the following statements is true about reinforcement learning and linear function approximators?

- a.** Linear function approximators can only be used with continuous state spaces and not with discrete spaces.
- b.** Linear function approximators provide a way to generalize from known states to unknown states, which can be useful in tabular reinforcement learning situations with large state spaces.
- c.** Linear function approximators in SARSA or Q-learning requires that we store all state-action pairs.
- d.** When using linear function approximators the policy will be deterministic
- e.** Don't know.

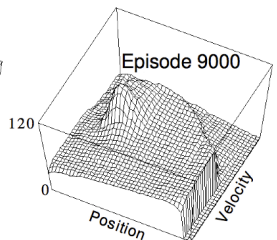
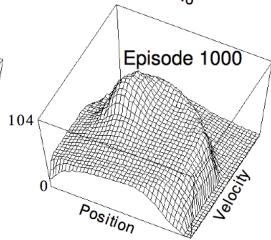
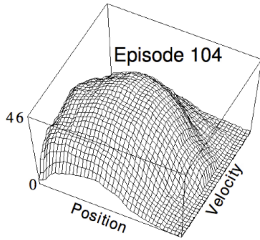
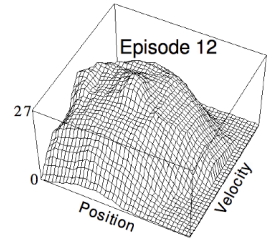
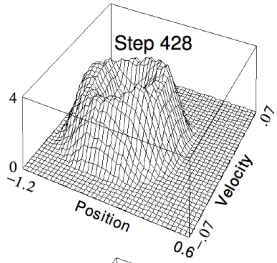
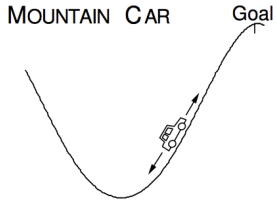
Value-function approximations


Implementing this



```
1 # semi_grad_q.py
2 class LinearSemiGradQAgent(QAgent):
3     def __init__(self, env, gamma=1.0, alpha=0.5, epsilon=0.1, q_encoder=None):
4         """ The Q-values, as implemented using a function approximator, can now be accessed as follows:
5
6         >> self.Q(s,a) # Compute q-value
7         >> self.Q.x(s,a) # Compute gradient of the above expression wrt. w
8         >> self.Q.w # get weight-vector.
9
10        I would recommend inserting a breakpoint and investigating the above expressions yourself;
11        you can of course al check the class LinearQEncoder if you want to see how it is done in practice.
12        """
13        super().__init__(env, gamma, epsilon=epsilon, alpha=alpha)
14        self.Q = LinearQEncoder(env, tilings=8) if q_encoder is None else q_encoder
```

Linear Sarsa with title coding in mountain car



 Richard S. Sutton and Andrew G. Barto.
Reinforcement Learning: An Introduction.
The MIT Press, second edition, 2018.
(Freely available online).

Approximation: The big picture

- Suppose f is a real-valued function $f : \mathcal{X} \mapsto \mathbb{R}$ which happens to be defined using an expectation:

$$f(x) = \mathbb{E}_z [g(x, z)] = \int p(z|x)g(x, z)dz$$

- Assume that $\hat{f}(x, \mathbf{w})$ is a neural network we want to use to approximate f with
- **Problem:** How do we find \mathbf{w} such that $\hat{f}(x, \mathbf{w}) \approx f(x)$?
- **Idea:** Select \mathbf{w} to minimize

$$\mathbf{w}^* = \arg \min_{\mathbf{w}} \mathbb{E}_x \left[\left[\hat{f}(x, \mathbf{w}) - f(x) \right]^2 \right] \quad (1)$$

- Solve this using gradient descent:

$$w \leftarrow w - \alpha \nabla \left(\mathbb{E} \left[f(x) - \hat{f}(x, \mathbf{w}) \right]^2 \right) \quad (2)$$

$$\begin{aligned}
 \nabla \left(\mathbb{E} \left[\hat{f}(x, \mathbf{w}) - f(x) \right]^2 \right) &= \mathbb{E} \left[\nabla \left(\hat{f}(x, \mathbf{w}) - f(x) \right)^2 \right] \\
 &= 2\mathbb{E} \left[\left(\hat{f}(x, \mathbf{w}) - f(x) \right) \nabla \hat{f}(x, \mathbf{w}) \right] \\
 &= 2\mathbb{E} \left[\left(\hat{f}(x, \mathbf{w}) - \mathbb{E}_z [g(x, z)] \right) \nabla \hat{f}(x, \mathbf{w}) \right]
 \end{aligned}$$

Implication: Given samples $x \sim p$ and $z \sim p(z|x)$ then

$$2 \left(\hat{f}(x, \mathbf{w}) - g(x, z) \right) \nabla \hat{f}(x, \mathbf{w})$$

is an **unbiased estimate** of the gradient

Stochastic gradient descent

Given minimization problem $\arg \min F(\mathbf{w})$ and (technical conditions!) then

$$\mathbf{w}_{t+1} \leftarrow \mathbf{w}_t - \alpha_t \hat{g}(\mathbf{w}_t)$$

converge to \mathbf{w}^* provided $\hat{g}(\mathbf{w})$ is an **unbiased estimate** of the gradient $\nabla F(\mathbf{w})$