02465: Introduction to reinforcement learning and control

Model-Free Control with tabular and linear methods

Tue Herlau

DTU Compute, Technical University of Denmark (DTU)

Lecture Schedule

Dynamical programming

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

Control

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

Reinforcement learning

8. Exploration and Bandits
   28 March
9. Bellmans equations and exact planning
   4 April
10. Monte-carlo methods and TD learning
    11 April
11. **Model-Free Control with tabular and linear methods**
    25 April
12. Eligibility traces
    2 May
13. Deep-Q learning
    9 May

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

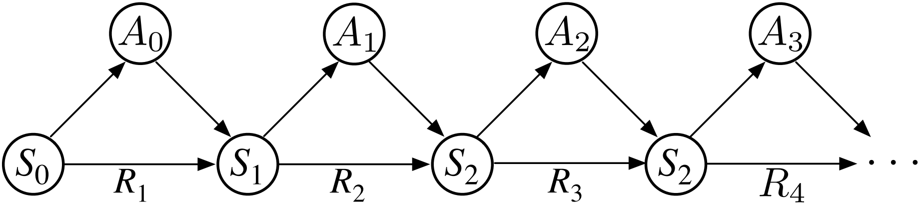
Reading material:

- [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 $s_0, a_0, r_1, s_1, a_1, r_2, \ldots, r_T$ using π

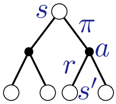
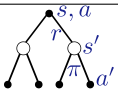
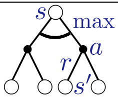
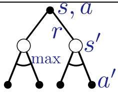
- **First** step t we visit a state s
- Measure return $G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots$ for rest of the episode
- Estimate value function as $v_\pi(s_t) = \mathbb{E}[G_t | S_t = s] \approx \frac{1}{n} \sum_{i=1}^{n} G_t^{(n)}$
- The average can be computed incrementally:

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

- 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 $v_\pi(s) = \mathbb{E}_\pi\left[R + \gamma v_\pi\left(S'\right)\vert s\right]$	**Iterative policy evaluation** to learn v_π $V(s) \leftarrow \mathbb{E}_\pi\left[R + \gamma V\left(S'\right)\vert s\right]$	
Bellman expectation equation for q_π $q_\pi(s,a) = \mathbb{E}_\pi\left[R + \gamma q_\pi\left(S', A'\right)\vert s, a\right]$	**Iterative policy evaluation** to learn q_π $Q(s,a) \leftarrow \mathbb{E}_\pi\left[R + \gamma Q\left(S', A'\right)\vert s, a\right]$	
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_* $v_*(s) = \max_a \mathbb{E}\left[R + \gamma v_*(S')\vert s, a\right]$	**Value iteration** $V(s) \leftarrow \max_a \mathbb{E}\left[R + \gamma V(S')\vert s, a\right]$	
Bellman optimality equation for q_* $q_*(s,a) = \mathbb{E}\left[R + \gamma \max_{a'} q_*(S', a')\vert s, a\right]$	**Q-value iteration** $Q(s,a) \leftarrow \mathbb{E}\left[R + \gamma \max_{a'} Q(S', a')\vert s, a\right]$	

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:

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

- 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:

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

- 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

Sarsa estimation of action-value function

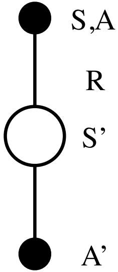
- Bellman equation:

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

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

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

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



Sarsa control

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 $Q(\mathit{terminal}, \cdot) = 0$

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)
        $Q(S, A) \leftarrow Q(S, A) + \alpha\big[R + \gamma Q(S', A') - Q(S, A)\big]$
        $S \leftarrow S'$; $A \leftarrow A'$;
    until S is terminal

`lecture_11_sarsa.py`

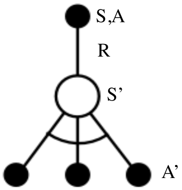
Convergence of Sarsa

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

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

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

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

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

Q-learning is off-policy

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

- 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

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 $Q(\mathit{terminal}, \cdot) = 0$

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

`lecture_11_q.py`

Exam question: Q-learning

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

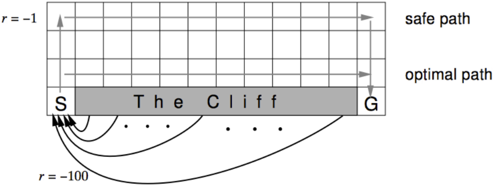
Q-learning converge to optimal action-value function $Q \to q_*$ assuming

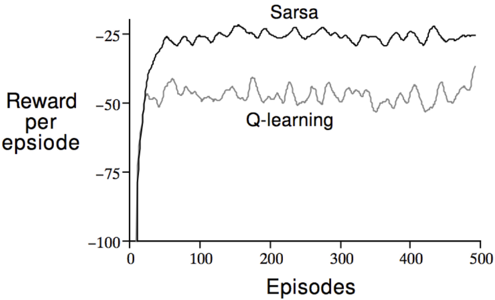
- All s, a pairs visited infinitely often
- Robbins-Monro sequence of step-sizes α_t

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

Comparing Q-learning and SARSA

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

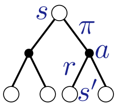


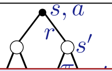


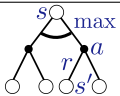
`lecture_11_sarsa_cliff.py`, `lecture_11_q_cliff.py`

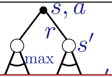
Algorithms so far

Bellman equation	Learning algorithm
Bellman expectation equation for v_π $v_\pi(s) = \mathbb{E}_\pi\left[R + \gamma v_\pi(S') \,\vert\, s\right]$	**Iterative policy evaluation** to learn v_π $V(s) \leftarrow \mathbb{E}_\pi\left[R + \gamma V(S') \,\vert\, s\right]$
Bellman expectation equation for q_π $q_\pi(s,a) = \mathbb{E}_\pi\left[R + \gamma q_\pi(S', A') \,\vert\, s, a\right]$	**Iterative policy evaluation** to learn q_π $Q(s,a) \leftarrow \mathbb{E}_\pi\left[R + \gamma Q(S'\right.$ [illegible]
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_* $v_*(s) = \max_a \mathbb{E}\left[R + \gamma v_*(S') \vert s, a\right]$	**Value iteration** $V(s) \leftarrow \max_a \mathbb{E}\left[R + \gamma V(S') \vert s, a\right]$
Bellman optimality equation for q_* $q_*(s,a) = \mathbb{E}\left[R + \gamma \max_{a'} q_*(S', a') \vert s, a\right]$	**Q-value iteration** $Q(s,a) \leftarrow \mathbb{E}\left[R + \gamma \max_{a'} Q(\right.$ [illegible]









TD Learning
$V(S) \overset{\alpha}{\leftarrow} R + \gamma V(S')$

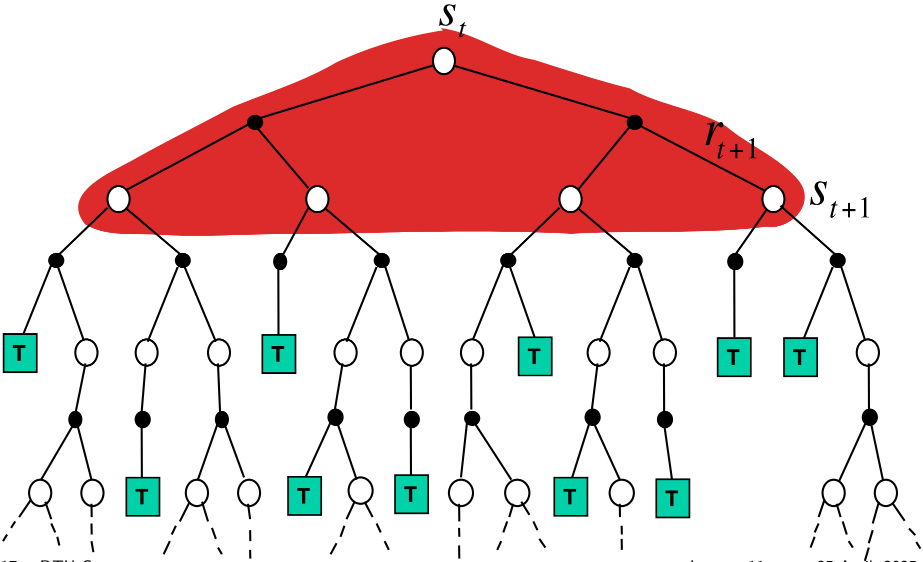
Sarsa
$Q(S,A) \overset{\alpha}{\leftarrow} R + \gamma Q(S', A')$

Q-Learning
$Q(S,A) \overset{\alpha}{\leftarrow} R + \gamma \max_{a' \in \mathcal{A}} Q(S', a')$

where $x \overset{\alpha}{\leftarrow} y \equiv x \leftarrow x + \alpha(y - x)$

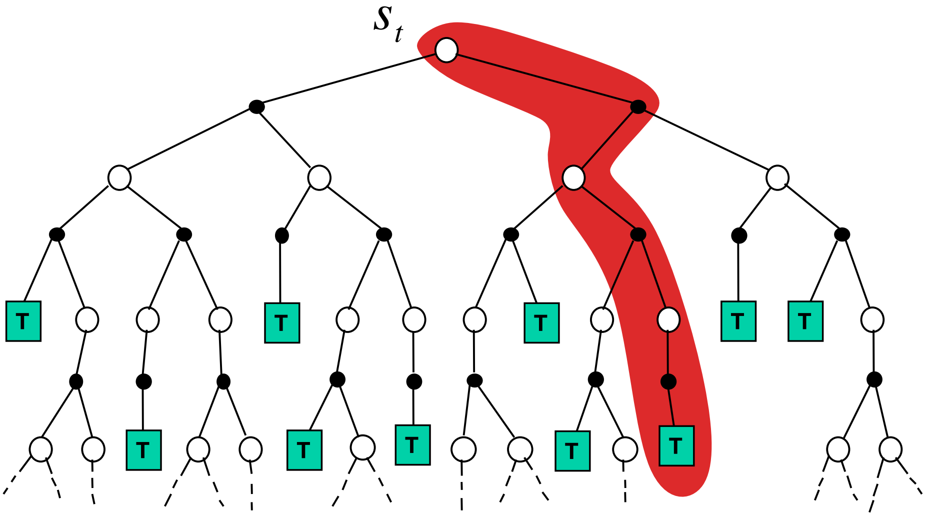
From two weeks ago: DP backups

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



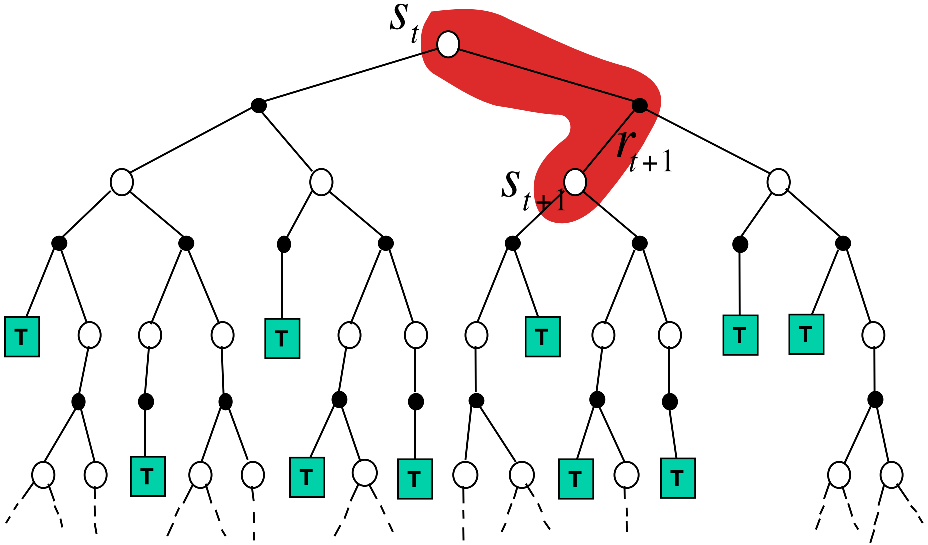
Last week: MC backups

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



Last week: TD backups

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



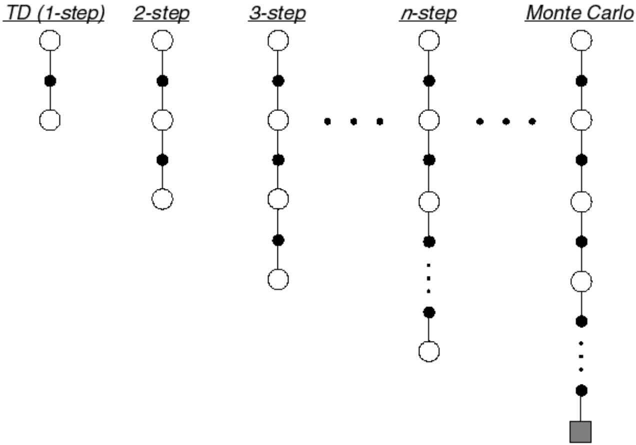
Comparisons

- **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

n-step predictions

- Let TD target look n steps into the future



n-step return

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

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

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

- Using the rules of expectations:

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

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

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

- This gives n-step temporal difference update:

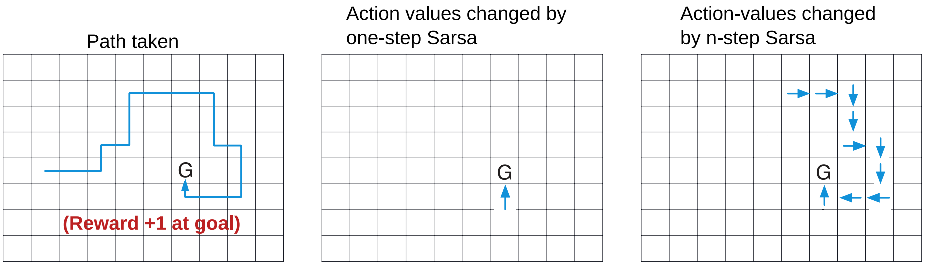
$$
V(S_t) \leftarrow V(S_t) + \alpha \left(G_{t:t+n} - V(S_t)\right)
$$

n-step TD: Implementation details

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

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

- 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 Sarsa

Recall the decomposition:

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

- As before:

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

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

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

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

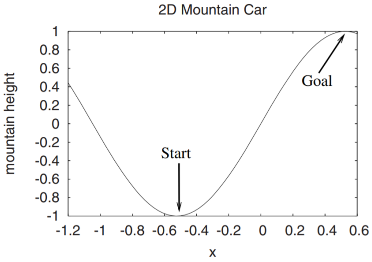
$$Q\left(S_t, A_t\right) \leftarrow Q\left(S_t, A_t\right) + \alpha\left(q_t^{(n)} - Q\left(S_t, A_t\right)\right)$$

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

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



`lecture_11_mountaincar_nolearn.py`

Value Function Approximation

- We have used loopup 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

Feature Vectors and linear representations

- Represent value function by a linear combination of features

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

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:

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

- For Q-values we only need to change the feature vector:

$$\hat{q}(s, a, \boldsymbol{w}) = \boldsymbol{x}(s, a)^\top \boldsymbol{w}$$

Example: Mountain-car

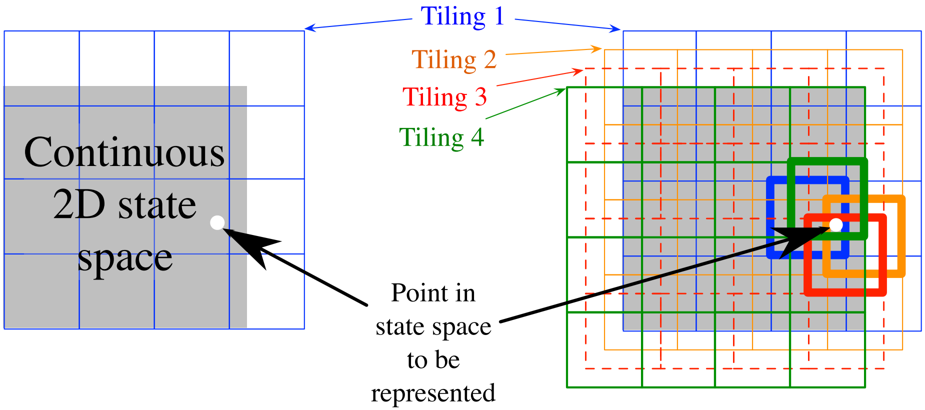
- Mountain-car has two **actions** $a = 0, 1$ (left, right); the **reward** is $R_t = -1$.
- The **state** is two-dimensional $s = (\text{position}, \text{velocity}) = (s_1, s_2)$
- **Naive example 1:** $\boldsymbol{x}(s) = \begin{bmatrix} s_1 \\ s_2 \end{bmatrix}$. **Naive example 2:** $\boldsymbol{x}(s) = \begin{bmatrix} s_1 \\ s_2 \\ \sin(s_1)\cos(s_2) \\ \vdots \end{bmatrix}$
- How about $\boldsymbol{x}(s, a)$? Idea: Re-use $\boldsymbol{x}(s)$ by zero-padding:

$$\boldsymbol{x}(s, a = 0)^\top = [\underbrace{0, 0, 0, 0, 0, \dots, 0}_{\text{pad with } |x(s)| \text{ zeros}}, \ \boldsymbol{x}(s)^\top]$$

$$\boldsymbol{x}(s, a = 1)^\top = [\boldsymbol{x}(s)^\top, \quad \underbrace{0, 0, 0, 0, 0, \dots, 0,}_{\text{pad with } |x(s)| \text{ zeros}}]$$

Details: Tile coding

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

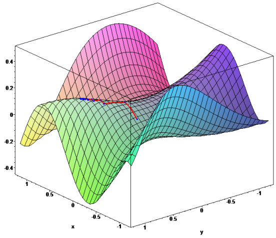


- x has now n_T non-zero elements corresponding to the number of active tiles

`lecture_11_mountaincar_random_weights.py`

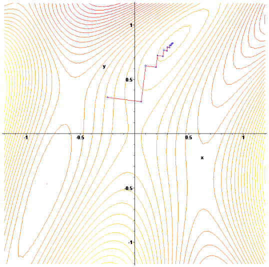
Recall from 02450: Gradient Descent

- 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 $E(\mathbf{w})$

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

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

Using the approximations

- 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** $\hat{v}(s, \boldsymbol{w})$ instead of $V(s)$ on right-hand side

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

- **..but how do we update $\boldsymbol{w}$ on the left-hand side so $\hat{v}(s, \boldsymbol{w})$ 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 $\boldsymbol{w}$ so that:

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

- Find $\boldsymbol{w}$ so that:

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

- Find $\boldsymbol{w}$ using gradient descent:

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

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

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

Summary

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

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

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

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

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

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

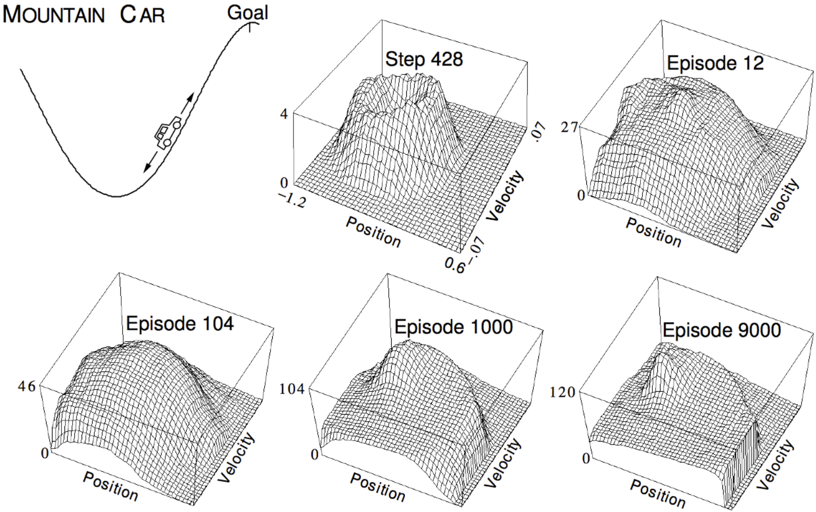
- Q-learning: $q(s,s) = \mathbb{E}[R + \gamma \max_{a'} q(S',a')|s,a]$ and $\hat{q}(s,a,\boldsymbol{w}) \approx q(s,a)$

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

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

Linear Sarsa with tile coding in mountain car

- Sarsa with linear tile-encoding ($|\boldsymbol{x}(s)| \approx 2000$)
- We plot $\max_a \hat{q}(s, a; \boldsymbol{w})$; `lecture_11_mountaincar_feature_space.py`



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.

Implementing this

```python
# semi_grad_q.py
class LinearSemiGradQAgent(QAgent):
    def __init__(self, env, gamma=1.0, alpha=0.5, epsilon=0.1, q_encoder=None):
        """ The Q-values, as implemented using a function approximator, can now be accessed as follows:

        >> self.Q(s,a) # Compute q-value
        >> self.Q.x(s,a) # Compute gradient of the above expression wrt. w
        >> self.Q.w # get weight-vector.

        I would recommend inserting a breakpoint and investigating the above expressions yourself;
        you can of course al check the class LinearQEncoder if you want to see how it is done in practice.
        """
        super().__init__(env, gamma, epsilon=epsilon, alpha=alpha)
        self.Q = LinearQEncoder(env, tilings=8) if q_encoder is None else q_encoder
```

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:

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

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

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

- Solve this using gradient descent:

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

Evaluating the gradient

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

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

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

is an **unbiased estimate** of the gradient

Stochastic gradient descent

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

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

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