Lecture 7:
Function approximation in reinforcement learning
(And deep reinforcement learning)

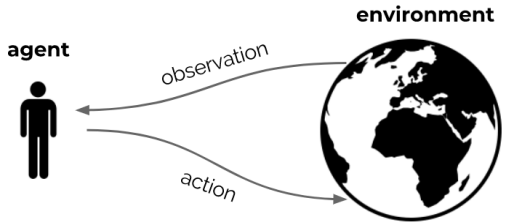
Hado van Hasselt

UCL, Tuesday, February 4th, 2020

Background

Sutton & Barto 2018, Chapters 9 + 10 (+ 11)

Recap



- Reinforcement learning is the science of learning to make decisions
- Agents can learn a **policy**, **value function** and/or a **model**
- The general problem involves taking into account **time** and **consequences**
- Decisions affect the **reward**, the **agent state**, and **environment state**

Function approximation and deep reinforcement learning

- The **policy**, **value function**, **model**, and **agent state update** are all functions
- We want to learn (some of) these from experience
- If there are too many states, we need to approximate
- This is often called **deep reinforcement learning**,
  when using **neural networks** to represent these functions
- The term is fairly new ($\pm$5 years) — the combination is old ($\pm$40-50 years)

Function approximation and deep reinforcement learning

This lecture

- We consider learning **predictions** (value functions)

Next lectures

- Off-policy learning
- Approximate dynamic programming (theory with function approximation)
- Learn explicit policies (policy gradients)
- Model-based RL

Large-Scale Reinforcement Learning

Reinforcement learning can be used to solve **large** problems, e.g.

- Backgammon: 10^{20} states
- Go: 10^{170} states
- Helicopter: continuous state space
- Robots: real world

How can we apply our methods for **prediction** and **control**?

Value Function Approximation

- So far we mostly considered **lookup tables**
  - Every state s has an entry $v(s)$
  - Or every state-action pair s, a has an entry $q(s, a)$
- Problem with large MDPs:
  - There are too many states and/or actions to store in memory
  - It is too slow to learn the value of each state individually
  - Individual states are often **not fully observable**

Value Function Approximation

Solution for large MDPs:

- Estimate value function with **function approximation**

$$v_{\mathbf{w}}(s) \approx v_\pi(s) \qquad (\text{or } v_*(s))$$
$$q_{\mathbf{w}}(s,a) \approx q_\pi(s,a) \qquad (\text{or } q_*(s,a))$$

- Update parameter $\mathbf{w}$ (e.g., using MC or TD learning)
- Generalise from to unseen states

Agent state update

Solution for large MDPs, if the environment state is not fully observable

- Use the **agent state**:

$$\mathbf{s}_t = u_\omega(\mathbf{s}_{t-1}, A_{t-1}, O_t)$$

  with parameters ω (typically $\omega \in \mathbb{R}^n$)
- Henceforth, S_t or $\mathbf{s}_t$ denotes the agent state
- Think of this as either a vector inside the agent,
  or, in the simplest case, just the current observation: $S_t = O_t$

Classes of Function Approximation

- **Tabular**: a table with an entry for each MDP state
- **State aggregation**: Partition environment states into a discrete set
- **Linear function approximation**
  - Fixed feature map $\mathbf{x} : \mathcal{S} \to \mathbb{R}^n$
  - Values are linear function of features: $v_{\mathbf{w}}(s) = \mathbf{w}^\top \mathbf{x}(s)$
  - Note: state aggregation and tabular are special cases of linear FA
- **Differentiable function approximation**
  - $v_{\mathbf{w}}(s)$ is a differentiable function of $\mathbf{w}$, could be **non-linear**
  - E.g., a convolutional neural network that takes pixels as input
  - Another interpretation: features are not fixed, but learnt

Classes of Function Approximation

In principle, **any** function approximator can be used, but RL has specific properties:

- Experience is not i.i.d. — successive time-steps are correlated
- Agent's policy affects the data it receives
- Targets (e.g., values $v_\pi(s)$) can be **non-stationary**
  - ...because of changing policies (which can change the target and the data!)
  - ...because of bootstrapping
  - ...because of non-stationary dynamics (e.g., other learning agents)
  - ...because the world is large (never quite in the same state)

Classes of Function Approximation

Which function approximation should you choose?
This depends on your goals.

- **Tabular**: good theory but does not scale/generalise
- **Linear**: reasonably good theory, but requires good features
- **Non-linear**: less well-understood, but scales well
  Flexible, and less reliant on picking good features first (e.g., by hand)

(Deep) neural nets often perform quite well (if given sufficient experience)

Learning algorithms

This lecture: **prediction** (including value-based control)

Later lecture: policy gradients

Gradient Descent

- Let $J(\mathbf{w})$ be a differentiable function of parameter vector $\mathbf{w}$
- Define the **gradient** of $J(\mathbf{w})$ to be

$$\nabla_{\mathbf{w}} J(\mathbf{w}) = \begin{pmatrix} \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_1} \\ \vdots \\ \frac{\partial J(\mathbf{w})}{\partial \mathbf{w}_n} \end{pmatrix}$$

- Goal: to minimise of $J(\mathbf{w})$
- Method: move $\mathbf{w}$ in the direction of negative gradient

$$\Delta \mathbf{w} = -\frac{1}{2} \alpha \nabla_{\mathbf{w}} J(\mathbf{w})$$

where α is a step-size parameter

Approximate Values By Stochastic Gradient Descent

- Goal: find $\mathbf{w}$ that minimise the difference between $v_{\mathbf{w}}(s)$ and $v_\pi(s)$

$$J(\mathbf{w}) = \mathbb{E}_{S\sim d}[(v_\pi(S) - v_{\mathbf{w}}(S))^2]$$

  where d is a distribution over states (typically induced by the policy and dynamics)
- Gradient descent:

$$\Delta\mathbf{w} = -\frac{1}{2}\alpha\nabla_{\mathbf{w}} J(\mathbf{w}) = \alpha\mathbb{E}_d(v_\pi(S) - v_{\mathbf{w}}(S))\nabla_{\mathbf{w}} v_{\mathbf{w}}(S)$$

- **Stochastic gradient descent** (SGD), sample the gradient:

$$\Delta\mathbf{w} = \alpha(G_t - v_{\mathbf{w}}(S_t))\nabla_{\mathbf{w}} v_{\mathbf{w}}(S_t)$$

  Note: Monte Carlo return G_t is a sample for $v_\pi(S_t)$
- We often write $\nabla v(S_t)$ as short hand for

$$\nabla_{\mathbf{w}} v_{\mathbf{w}}(S_t)|_{\mathbf{w}=\mathbf{w}_t}$$

Linear function approximation

Feature Vectors

- Represent state by a **feature vector**

$$\mathbf{x}(s) = \begin{pmatrix} \mathbf{x}(s)[1] \\ \vdots \\ \mathbf{x}(s)[n] \end{pmatrix}$$

- $\mathbf{x} : \mathcal{S} \to \mathbb{R}^n$ is a fixed mapping from state (e.g., observation) to features
- Short-hand: $\mathbf{x}_t = \mathbf{x}(S_t)$
- For example:
  - Distance of robot from landmarks
  - Trends in the stock market
  - Piece and pawn configurations in chess

Linear Value Function Approximation

- Approximate value function by a linear combination of features

$$v_{\mathbf{w}}(s) = \mathbf{w}^\top \mathbf{x}(s) = \sum_{j=1}^{n} \mathbf{x}_j(s)\mathbf{w}_j$$

- Objective function ('loss') is quadratic in $\mathbf{w}$

$$J(\mathbf{w}) = \mathbb{E}_{S\sim d}[(v_\pi(S) - \mathbf{w}^\top \mathbf{x}(S))^2]$$

- Stochastic gradient descent converges on **global** optimum
- Update rule is simple

$$\nabla_{\mathbf{w}} v_{\mathbf{w}}(S_t) = \mathbf{x}(S_t) = \mathbf{x}_t \qquad \Longrightarrow \qquad \Delta\mathbf{w} = \alpha(v_\pi(S_t) - v_{\mathbf{w}}(S_t))\mathbf{x}_t$$

Update = **step-size** $\times$ **prediction error** $\times$ **feature vector**

Table Lookup Features

- Table lookup is a special case of linear value function approximation
- Let the n states be given by $\mathcal{S} = \{s^{(1)}, \ldots, s^{(n)}\}$.
- Using **table lookup features**

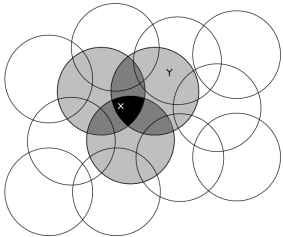
$$\mathbf{x}^{table}(s) = \begin{pmatrix} \mathbf{1}(s = s^{(1)}) \\ \vdots \\ \mathbf{1}(s = s^{(n)}) \end{pmatrix} \qquad \text{(one-hot feature vector)}$$

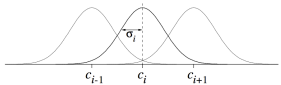
- Parameter vector $\mathbf{w}$ gives value of each individual state

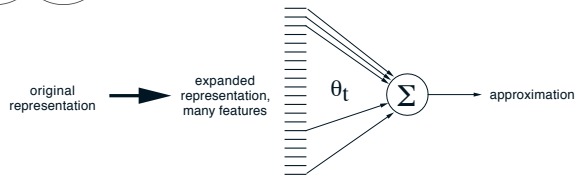
$$V(s) = \begin{pmatrix} \mathbf{1}(s = s^{(1)}) \\ \vdots \\ \mathbf{1}(s = s^{(n)}) \end{pmatrix} \cdot \begin{pmatrix} \mathbf{w}_1 \\ \vdots \\ \mathbf{w}_n \end{pmatrix}$$

Feature construction example: coarse coding

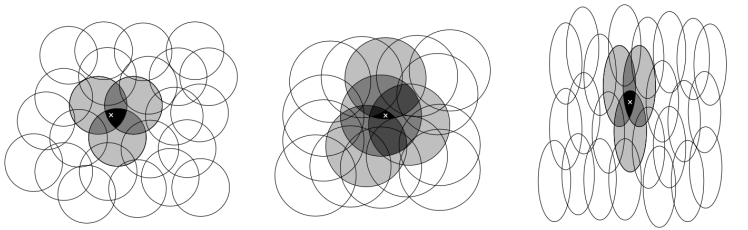
- **Coarse coding** provides large feature vector $\mathbf{x}(s)$
- Parameter vector $\mathbf{w}$ gives a value to each feature







Generalization in Coarse Coding



a) Narrow generalization b) Broad generalization c) Asymmetric generalization

- We aggregate multiple states
- This means the resulting feature vector/agent state is **non-Markovian**
- This is the common case when using function approximation
- Consider whether good solutions exist for given features + function approximation
- Neural networks tend to be more flexible

Linear model-free prediction

Incremental prediction algorithms

- We can't update towards the true value function $v_\pi(s)$
- We substitute a **target** for $v_\pi(s)$
  - For MC, the target is the return G_t

$$\Delta \mathbf{w}_t = \alpha(\boldsymbol{G_t} - v_{\mathbf{w}}(s))\nabla_{\mathbf{w}} v_{\mathbf{w}}(s)$$

  - For TD, the target is the TD target $R_{t+1} + \gamma v_{\mathbf{w}}(S_{t+1})$

$$\Delta \mathbf{w}_t = \alpha(\boldsymbol{R_{t+1} + \gamma v_{\mathbf{w}}(S_{t+1})} - v_{\mathbf{w}}(S_t))\nabla_{\mathbf{w}} v_{\mathbf{w}}(S_t)$$

  - TD(λ):

$$\Delta \mathbf{w}_t = \alpha(\boldsymbol{G_t^\lambda} - v_{\mathbf{w}}(S_t))\nabla_{\mathbf{w}} v_{\mathbf{w}}(S_t)$$

$$G_t^\lambda = R_{t+1} + \gamma\Big((1-\lambda)v_{\mathbf{w}}(S_{t+1}) + \lambda G_{t+1}^\lambda\Big)$$

Monte-Carlo with Value Function Approximation

- The return G_t is an **unbiased** sample of $v_\pi(s)$
- Can therefore apply "supervised learning" to (online) "training data":

$$\{(S_0, G_0), \ldots, (S_t, G_t)\}$$

- For example, using **linear Monte-Carlo policy evaluation**

$$\begin{aligned}\Delta \mathbf{w}_t &= \alpha(\boldsymbol{G_t} - v_{\mathbf{w}}(S_t))\nabla_{\mathbf{w}} v_{\mathbf{w}}(S_t) \\ &= \alpha(G_t - v_{\mathbf{w}}(S_t))\mathbf{x}_t\end{aligned}$$

- Monte-Carlo evaluation converges to a local optimum
- Even when using non-linear value function approximation
- For linear functions, it finds the global optimum

TD Learning with Value Function Approximation

- The TD-target $R_{t+1} + \gamma v_{\mathbf{w}}(S_{t+1})$ is a **biased** sample of true value $v_\pi(S_t)$
- Can still apply supervised learning to "training data":

$$\{(S_0, R_1 + \gamma v_{\mathbf{w}}(S_1)), \ldots (S_t, R_{t+1} + \gamma v_{\mathbf{w}}(S_{t+1}))\}$$

- For example, using **linear TD**

$$\begin{aligned}\Delta \mathbf{w}_t &= \alpha \underbrace{(\boldsymbol{R_{t+1} + \gamma v_{\mathbf{w}}(S_{t+1})} - v_{\mathbf{w}}(S_t))}_{= \delta_t,\ \text{'TD error'}} \nabla_{\mathbf{w}} v_{\mathbf{w}}(S_t) \\ &= \alpha \delta_t \mathbf{x}_t\end{aligned}$$

- This is akin to a non-stationary regression problem
  - But it's a bit different: the non-stationarity in part depends on the updates

Convergence of MC

- With linear functions, MC converges to

$$\mathbf{w}_{\mathrm{MC}} = \operatorname*{argmin}_{\mathbf{w}} \mathbb{E}_\pi[(G_t - v_{\mathbf{w}}(S_t))^2] = \mathbb{E}[\mathbf{x}_t \mathbf{x}_t^\top]^{-1} \mathbb{E}[v_\pi(S_t)\mathbf{x}_t]$$

  (Notation: here the state distribution implicitly depends on π)

- Proof:

$$\begin{aligned}
\nabla_{\mathbf{w}} \mathbb{E}[(G_t - v_{\mathbf{w}}(S_t))^2] = \mathbb{E}[(G_t - v_{\mathbf{w}}(S_t))\mathbf{x}_t] &= 0 \\
\mathbb{E}[(G_t - \mathbf{x}_t^\top \mathbf{w})\mathbf{x}_t] &= 0 \\
\mathbb{E}[G_t \mathbf{x}_t - \mathbf{x}_t \mathbf{x}_t^\top \mathbf{w}] &= 0 \\
\mathbb{E}[\mathbf{x}_t \mathbf{x}_t^\top]\mathbf{w} &= \mathbb{E}[G_t \mathbf{x}_t] \\
\mathbf{w} &= \mathbb{E}[\mathbf{x}_t \mathbf{x}_t^\top]^{-1} \mathbb{E}[v_\pi(S_t)\mathbf{x}_t]
\end{aligned}$$

- **Agent state** S_t does not have to be Markovian — $v_\pi(S_t)$ is defined as the expected return given that we currently have S_t
- This does not have to be the true value of the current **environment state**

Convergence of TD

- With linear functions, TD converges to

$$\mathbf{w}_{\mathrm{TD}} = \mathbb{E}[\mathbf{x}_t(\mathbf{x}_t - \gamma \mathbf{x}_{t+1})^\top]^{-1}\mathbb{E}[R_{t+1}\mathbf{x}_t]$$

  (in continuing problems with fixed γ, and with appropriately decaying $\alpha_t \to 0$)

- Verify:

$$\begin{aligned}
\mathbb{E}[\Delta \mathbf{w}_{\mathrm{TD}}] &= \mathbb{E}[\alpha_t(R_{t+1} + \gamma \mathbf{x}_{t+1}^\top \mathbf{w}_{\mathrm{TD}} - \mathbf{x}_t^\top \mathbf{w}_{\mathrm{TD}})\mathbf{x}_t] \\
&= \mathbb{E}[\alpha_t R_{t+1}\mathbf{x}_t] + \mathbb{E}[\alpha_t \mathbf{x}_t(\gamma \mathbf{x}_{t+1}^\top - \mathbf{x}_t^\top)\mathbf{w}_{\mathrm{TD}}] \\
&= \mathbb{E}[\alpha_t R_{t+1}\mathbf{x}_t] + \mathbb{E}[\alpha_t \mathbf{x}_t(\gamma \mathbf{x}_{t+1}^\top - \mathbf{x}_t^\top)]\mathbb{E}[\mathbf{x}_t(\mathbf{x}_t - \gamma \mathbf{x}_{t+1})^\top]^{-1}\mathbb{E}[R_{t+1}\mathbf{x}_t] \\
&= \mathbb{E}[\alpha_t R_{t+1}\mathbf{x}_t] - \mathbb{E}[\alpha_t R_{t+1}\mathbf{x}_t] \quad \square
\end{aligned}$$

- This **differs** from the MC solution
- Typically, the asymptotic MC solution is preferred (smallest prediction error)
- TD often converges faster (especially intermediate $\lambda \in [0, 1]$ or $n \in \{1, \ldots, \infty\}$)

Convergence of TD

- With linear functions, TD converges to

$$\mathbf{w}_{\mathrm{TD}} = \mathbb{E}[\mathbf{x}_t(\mathbf{x}_t - \gamma \mathbf{x}_{t+1})^\top]^{-1}\mathbb{E}[R_{t+1}\mathbf{x}_t]$$

- Let $\overline{\mathrm{VE}}(\mathbf{w})$ denote the **value error**:

$$\overline{\mathrm{VE}}(\mathbf{w}) = \sum_{s \in \mathcal{S}} d(s)(v_\pi(s) - v_{\mathbf{w}}(s))^2$$

- The Monte Carlo solution minimises the value error

Theorem

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

TD is not a gradient

- The TD update is not a true gradient update:

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

- That's okay: it is a **stochastic approximation** update
- Stochastic approximation algorithms are a broader class than just SGD
- SGD always converges (with bounded noise, decaying step size, stationarity, ...)
- We will see later that this is **not** always true for TD
  (And how to mitigate this)

Residual Bellman updates

TD: $\Delta \mathbf{w}_t = \alpha \delta \nabla v_{\mathbf{w}}(S_t)$ where $\delta_t = R_{t+1} + \gamma v_{\mathbf{w}}(S_{t+1}) - v_{\mathbf{w}}(S_t)$

- This update ignores dependence of $v_{\mathbf{w}}(S_{t+1})$ on $\mathbf{w}$
- Alternative: **Bellman residual gradient** update

loss: $\mathbb{E}[\delta_t^2]$ update: $\Delta \mathbf{w}_t = \alpha \delta_t \nabla_{\mathbf{w}}(v_{\mathbf{w}}(S_t) - \gamma v_{\mathbf{w}}(S_{t+1}))$

- This tends to **work worse** in practice
- Bellman residuals smooth, whereas TD methods predict
- Smoothed values may lead to suboptimal decisions

Residual Bellman updates

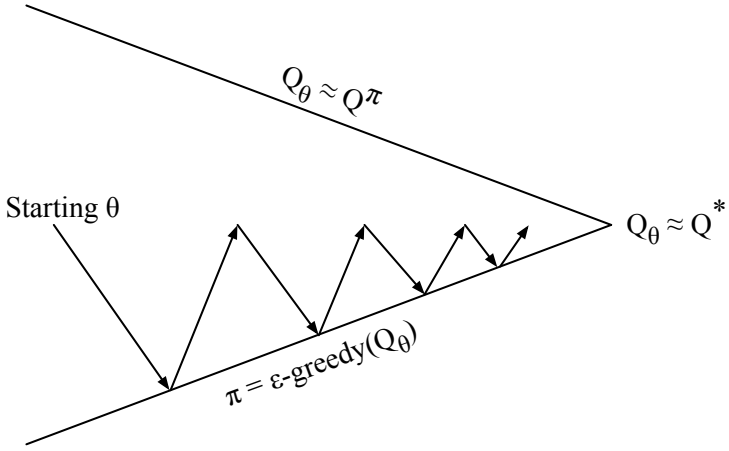
- Alternative: **Bellman residual gradient** update

  loss: $\mathbb{E}[\delta_t]^2$ update: $\Delta \mathbf{w}_t = \alpha \delta_t \nabla_{\mathbf{w}} (v_{\mathbf{w}}(S_t) - \gamma v_{\mathbf{w}}(S'_{t+1}))$

- ...but this requires a second independent sample S'_{t+1}
  (So we can't use this online)

Control with value-function approximation

Control with Value Function Approximation



Policy evaluation **Approximate** policy evaluation, $q_{\mathbf{w}} \approx q_\pi$

Policy improvement E.g., ϵ-greedy policy improvement

Action-Value Function Approximation

- Approximate the action-value function $q_{\mathbf{w}}(s, a) \approx q_\pi(s, a)$
- For instance, with linear function approximation **with state-action features**

$$q_{\mathbf{w}}(s, a) = \mathbf{x}(s, a)^\top \mathbf{w}$$

- Stochastic gradient descent update

$$\begin{aligned} \Delta \mathbf{w} &= \alpha(q_\pi(s, a) - q_{\mathbf{w}}(s, a)) \nabla_{\mathbf{w}} q_{\mathbf{w}}(s, a) \\ &= \alpha(q_\pi(s, a) - q_{\mathbf{w}}(s, a)) \mathbf{x}(s, a) \end{aligned}$$

Action-Value Function Approximation

- Approximate the action-value function $q_{\mathbf{w}}(s, a) \approx q_\pi(s, a)$
- For instance, with linear function approximation **with state features**

$$\mathbf{q}_{\mathbf{w}}(s) = \mathbf{W}\mathbf{x}(s) \qquad (\mathbf{W} \in \mathbb{R}^{m \times n},\ \mathbf{x}(s) \in \mathbb{R}^n \implies \mathbf{q} \in \mathbb{R}^m)$$
$$q_{\mathbf{w}}(s, a) = \mathbf{q}_{\mathbf{w}}(s)[a] = \mathbf{x}(s)^\top \mathbf{w}_a \qquad (\text{where } \mathbf{w}_a = \mathbf{W}_{a\cdot}^\top)$$

- Stochastic gradient descent update

$$\begin{aligned}
\Delta \mathbf{w}_a &= \alpha(q_\pi(s, a) - q_{\mathbf{w}}(s, a)) \nabla_{\mathbf{w}} q_{\mathbf{w}}(s, a) \\
&= \alpha(q_\pi(s, a) - q_{\mathbf{w}}(s, a)) \mathbf{x}(s) \\
\forall a \neq b: \ \Delta \mathbf{w}_b &= 0 \\
\text{Equivalently: } \Delta \mathbf{W} &= \alpha(q_\pi(s, a) - q_{\mathbf{w}}(s, a)) \mathbf{i}_a \mathbf{x}(s)^\top
\end{aligned}$$

where $\mathbf{i}_a = (0, \ldots, 0, 1, 0, \ldots, 0)$ with $\mathbf{i}_a[a] = 1$, $\mathbf{i}_a[b] = 0$ for $b \neq a$

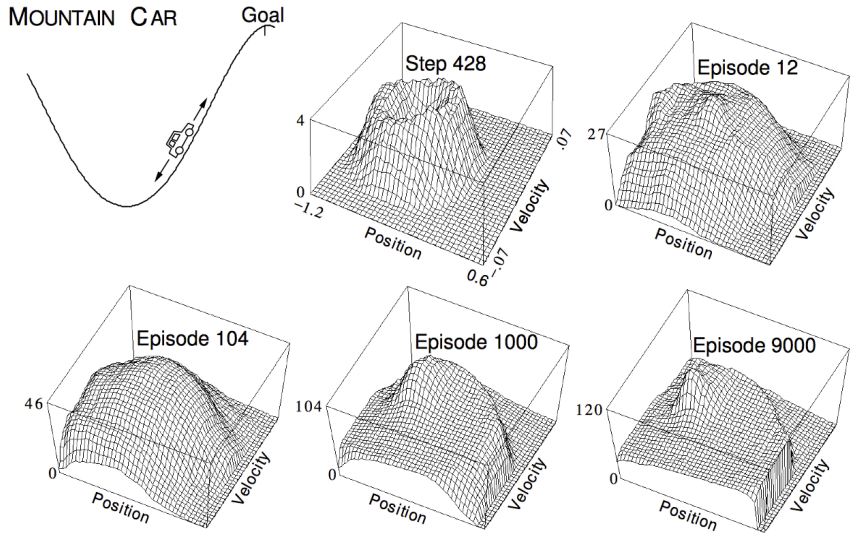
Action-Value Function Approximation

- Should we use action-in, or action-out?
  - Action in: $q_{\mathbf{w}}(s, a) = \mathbf{w}^\top \mathbf{x}(s, a)$
  - Action out: $\mathbf{q}_{\mathbf{w}}(s) = \mathbf{W}\mathbf{x}(s)$ such that $q_{\mathbf{w}}(s, a) = \mathbf{q}_{\mathbf{w}}(s)[a]$
- One reuses the same weights, the other the same features
- Unclear which is better in general
- If we want to use continuous actions, action-in is easier (later lecture)
- For (small) discrete action spaces, action-out is common (e.g., DQN)

Action-Value Function Approximation

- SARSA is TD applied to state-action pairs
- $\implies$ Inherits same properties
- But easier to do policy optimisation, and therefore policy iteration

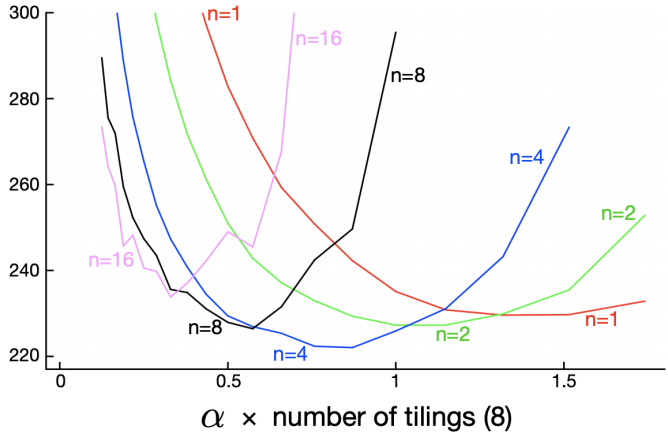
Linear Sarsa with Coarse Coding in Mountain Car



Linear Sarsa with Tile Coding

Mountain Car

Steps per episode
averaged over
first 50 episodes
and 100 runs



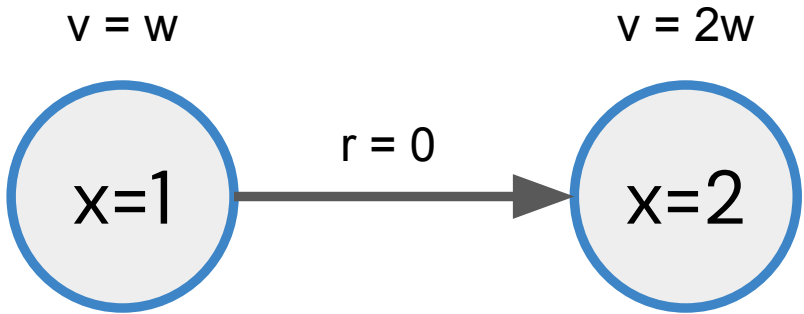
Tile coding is similar to coarse coding:
Overlaying different discretisations of the state space

Convergence and divergence

Convergence Questions

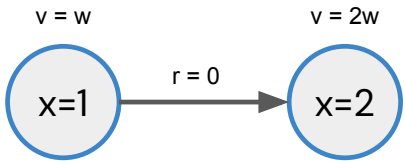
- When do incremental prediction algorithms converge?
  - When using **bootstrapping** (i.e. TD)?
  - When using (e.g., linear) value **function approximation**?
  - When using **off-policy** learning?
- Ideally, we would like algorithms that converge in all cases
- Alternatively, we want to understand when algorithms do, or do not, converge

Example of divergence



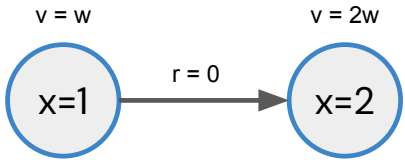
What if we use TD only on this transition?

Example of divergence



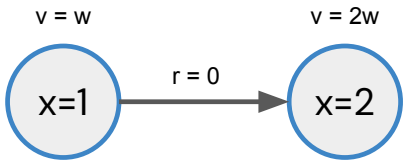
$$w_{t+1} = w_t + \alpha_t(r + \gamma v(s') - v(s))\nabla v(s)$$

Example of divergence



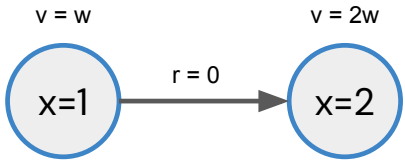
$$
\begin{aligned}
w_{t+1} &= w_t + \alpha_t(r + \gamma v(s') - v(s))\nabla v(s) \\
&= w_t + \alpha_t(r + \gamma v(s') - v(s))x(s)
\end{aligned}
$$

Example of divergence



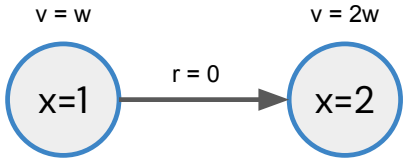
$$
\begin{aligned}
w_{t+1} &= w_t + \alpha_t(r + \gamma v(s') - v(s))\nabla v(s) \\
&= w_t + \alpha_t(r + \gamma v(s') - v(s))x(s) \\
&= w_t + \alpha_t(0 + \gamma 2 w_t - w_t)
\end{aligned}
$$

Example of divergence



$$
\begin{aligned}
w_{t+1} &= w_t + \alpha_t(r + \gamma v(s') - v(s))\nabla v(s) \\
&= w_t + \alpha_t(r + \gamma v(s') - v(s))x(s) \\
&= w_t + \alpha_t(0 + \gamma 2 w_t - w_t) \\
&= w_t + \alpha_t(2\gamma - 1)w_t
\end{aligned}
$$

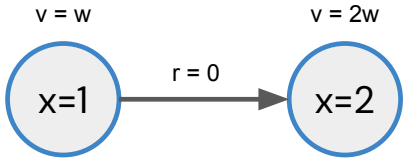
Example of divergence



$$
\begin{aligned}
w_{t+1} &= w_t + \alpha_t(r + \gamma v(s') - v(s))\nabla v(s) \\
&= w_t + \alpha_t(r + \gamma v(s') - v(s))x(s) \\
&= w_t + \alpha_t(0 + \gamma 2 w_t - w_t) \\
&= w_t + \alpha_t(2\gamma - 1)w_t
\end{aligned}
$$

Consider $w_t > 0$. If $\gamma > \frac{1}{2}$, then $w_{t+1} > w_t$.
$\implies \lim_{t\to\infty} w_t = \infty$

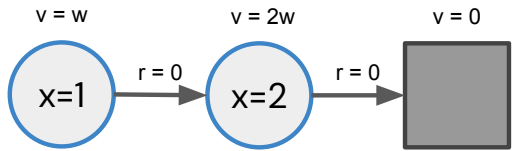
Example of divergence



- Algorithms that combine
  - **bootstrapping**
  - **off-policy learning**, and
  - **function approximation**

  ...may diverge
- This is sometimes called the **deadly triad**

Deadly triad

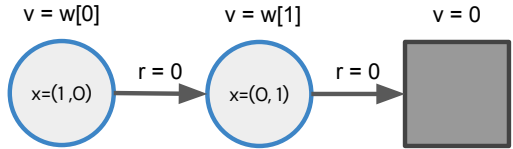


- Consider sampling **on-policy**, over an episode. Update:

$$\begin{aligned}\Delta w &= \alpha(0 + 2\gamma w - w) + \alpha(0 + \gamma 0 - 2w) \\ &= \alpha(2\gamma - 3)w\end{aligned}$$

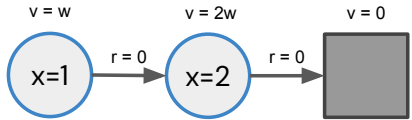
- The multiplier is negative, for all $\gamma \in [0, 1]$
- This implies convergence (move w towards zero)

Deadly triad



- With tabular features, this is just regression
- Answer may be sub-optimal, but no divergence occurs
- Specifically, if we only update $v(s)$ (=left-most state):
  - $v(s) = w[0]$ will converge to $\gamma v(s')$
  - $v(s') = w[1]$ will stay where it was initialised

Deadly triad



- What if we use multi-step returns?
- Still consider only updating the left-most state

$$\begin{aligned}\Delta w &= \alpha(r + \gamma(G_t^\lambda - v(s)) \\ &= \alpha(r + \gamma((1-\lambda)v(s') + \lambda(r' + v(s'')) - v(s)) \qquad (r = r' = v(s'') = 0) \\ &= \alpha(2\gamma(1-\lambda) - 1)w\end{aligned}$$

- The multiplier is negative when $2\gamma(1-\lambda) < 1 \implies \lambda > 1 - \frac{1}{2\gamma}$
- E.g., when $\gamma = 0.9$, then we need $\lambda > 4/9 \approx 0.45$

Convergence of Prediction Algorithms

On/Off-Policy	Algorithm	Table Lookup	Linear	Non-Linear
On-Policy	MC	✓	✓	✓
	TD	✓	✓	✗
Off-Policy	MC	✓	✓	✓
	TD	✓	✗	✗