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

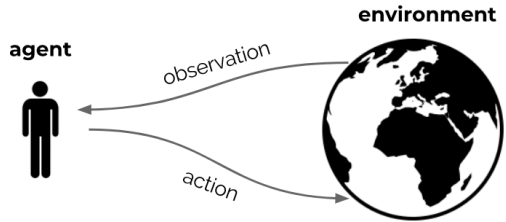
Hado van Hasselt

UCL, 2021

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**

Why function approximation?

Function approximation and deep reinforcement learning

- The **policy**, **value function**, **model**, and **agent state update** are all functions
- We want to learn 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 (±7-8 years) — the combination is fairly old (±50 years)

Function approximation and deep reinforcement learning

This lecture

- We consider learning **predictions** (value functions; including value-based control)

Upcoming 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

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 environment states are often **not fully observable**

Value Function Approximation

Solution for large MDPs:

- Estimate value function with **function approximation**

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

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

Agent state update

When the environment state is not fully observable ($S_t^{\text{env}} \neq O_t$)

- 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: $S_t = O_t$

Function classes

Classes of Function Approximation

- **Tabular**: a table with an entry for each MDP state
- **State aggregation**: Partition environment states (or observations) into a discrete set
- **Linear function approximation**
  - Consider fixed agent state update (e.g., $S_t = O_t$)
  - 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
- Regression targets 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, and remain a popular choice

Gradient-based algorithms

Gradient Descent

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

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

- 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 $v_\pi(s)$

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

  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 $\boldsymbol{\nabla v(S_t)}$ as short hand for

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

Linear function approximation

Feature Vectors

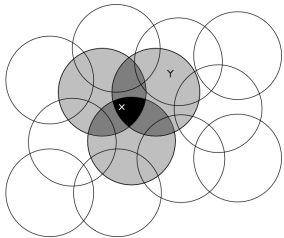
- Represent state by a **feature vector**

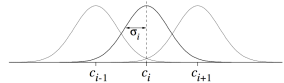
$$\mathbf{x}(s) = \begin{pmatrix} x_1(s) \\ \vdots \\ x_n(s) \end{pmatrix}$$

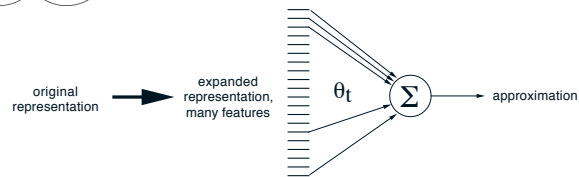
- $\mathbf{x} : \mathcal{S} \to \mathbb{R}^n$ is a fixed mapping from state (e.g., observation) to features
- Short-hand: $\mathbf{x}_t = \mathbf{x}(S_t)$
- For example:
  - Distance of robot from landmarks
  - Trends in the stock market
  - Piece and pawn configurations in chess

Feature construction example: coarse coding

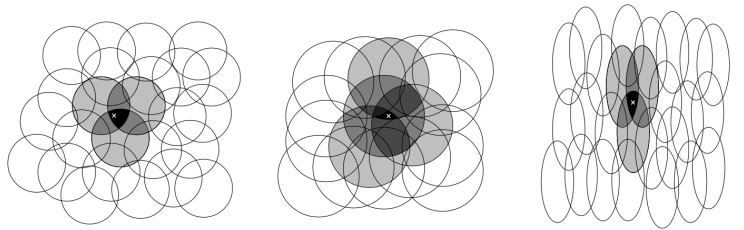
- **Coarse coding** provides large feature vector $\mathbf{x}(s)$
- Parameter vector $\mathbf{w}$ gives a value to each feature







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

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** × **prediction error** × **feature vector**

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):

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

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

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**

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

- Linear Monte-Carlo evaluation converges to the global optimum
- Even when using non-linear value function approximation it converges (but perhaps to a local 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":

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

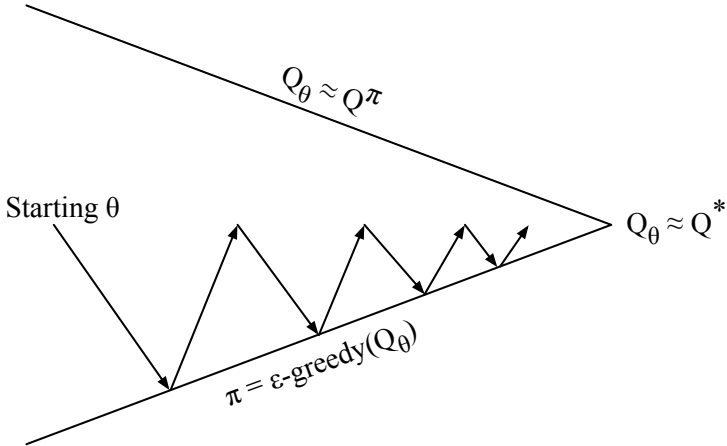
- For example, using **linear TD**

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

- This is akin to a non-stationary regression problem
- But it's a bit different: the target depends on our parameters!

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 $b \neq a$

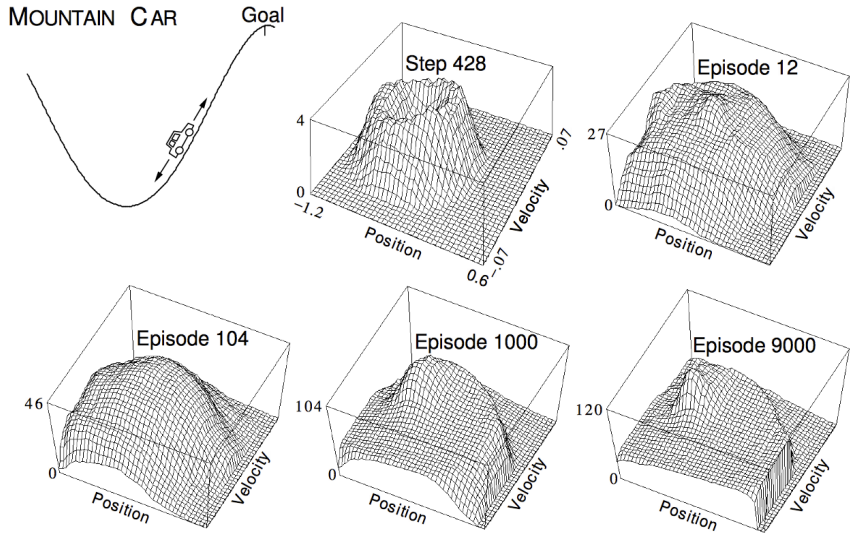
Action-Value Function Approximation

- 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)

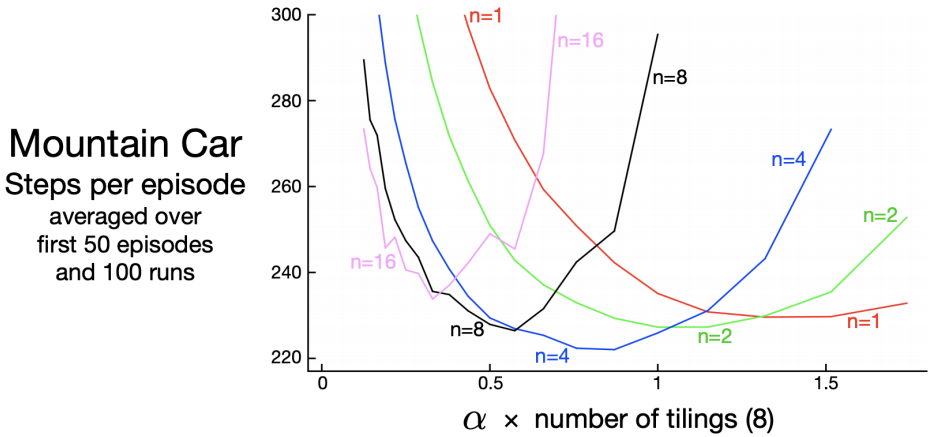
Action-Value Function Approximation

- SARSA is TD applied to state-action pairs
- $\Longrightarrow$ 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



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

Convergence of MC

- With linear functions (and suitably decaying step size), MC converges to

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

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

- Verifying the fixed point:

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

- **Agent state** S_t does not have to be Markov:
  the fixed point only depends on observed data (and features)

Convergence of TD

- With linear functions, TD converges to

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

  (in continuing problems with fixed $\gamma < 1$, and with appropriately decaying $\alpha_t \to 0$)

- Verify (assuming α_t does not correlate with $R_{t+1}, \mathbf{x}_t, \mathbf{x}_{t+1}$):

$$\begin{aligned}
\mathbb{E}[\Delta\mathbf{w}_{\text{TD}}] = 0 &= \mathbb{E}[\alpha_t(R_{t+1} + \gamma\mathbf{x}_{t+1}^\top\mathbf{w}_{\text{TD}} - \mathbf{x}_t^\top\mathbf{w}_{\text{TD}})\mathbf{x}_t] \\
0 &= \mathbb{E}[\alpha_t R_{t+1}\mathbf{x}_t] + \mathbb{E}[\alpha_t\mathbf{x}_t(\gamma\mathbf{x}_{t+1}^\top - \mathbf{x}_t^\top)\mathbf{w}_{\text{TD}}] \\
\mathbb{E}[\alpha_t\mathbf{x}_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})^\top]\mathbf{w}_{\text{TD}} &= \mathbb{E}[\alpha_t R_{t+1}\mathbf{x}_t] \\
\mathbf{w}_{\text{TD}} &= \mathbb{E}[\mathbf{x}_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})^\top]^{-1}\mathbb{E}[R_{t+1}\mathbf{x}_t]
\end{aligned}$$

- 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

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

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

$$\overline{\text{VE}}(\mathbf{w}) = \|v_\pi - v_{\mathbf{w}}\|_{d_\pi} = \sum_{s \in \mathcal{S}} d_\pi(s)(v_\pi(s) - v_{\mathbf{w}}(s))^2$$

- The Monte Carlo solution minimises the value error

Theorem

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

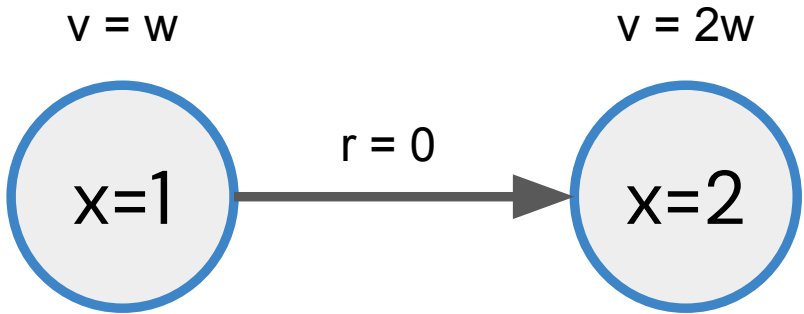
TD is not a gradient

- The TD update is not a true gradient update:

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

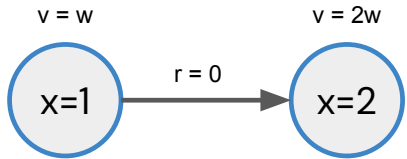
- That's okay: it is a **stochastic approximation** update
- Stochastic approximation algorithms are a broader class than just SGD
- SGD always converges (with bounded noise, decaying step size, stationarity, ...)
- TD does **not** always converge...

Example of divergence



What if we use TD only on this transition?

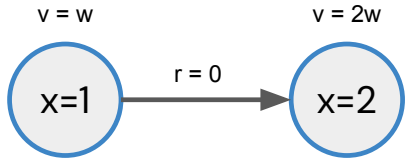
Example of divergence



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

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

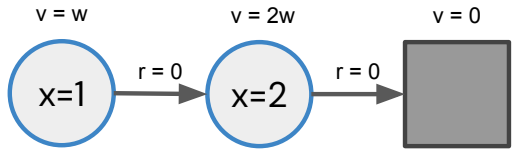
Example of divergence



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

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

Deadly triad

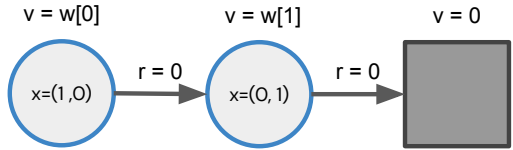


- Consider sampling **on-policy**, over an episode. Update:

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

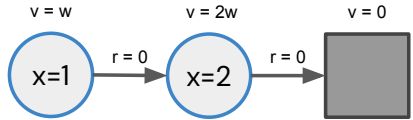
- The multiplier is negative, for all $\gamma \in [0, 1]$
- $\implies$ convergence (w goes to zero, which is optimal here)

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 $\lambda > 4/9 \approx 0.45$

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: minimise the **Bellman error**

  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 requires a second independent sample S'_{t+1} which could (randomly) differ from S_{t+1} (So we can't use this online)

Convergence of Prediction Algorithms

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

Convergence of Control Algorithms

- Tabular control learning algorithms (e.g., Q-learning) can be extended to FA (e.g., Deep Q Network — DQN)
- The theory of control with function approximation is not fully developed
- **Tracking** is often preferred to convergence
  (I.e., continually adapting the policy instead of converging to a fixed policy)

Batch Methods

Batch Reinforcement Learning

- Gradient descent is simple and appealing
- But it is not **sample** efficient
- Batch methods seek to find the best fitting value function for a given a set of past experience (“training data")

Least Squares Temporal Difference

- Which parameters $\mathbf{w}$ give the **best fitting** linear value function $v_{\mathbf{w}}(s) = \mathbf{w}^\top \mathbf{x}(s)$? Recall:

$$\mathbb{E}[(R_{t+1} + \gamma v_{\mathbf{w}}(S_{t+1}) - v_{\mathbf{w}}(S_t))\mathbf{x}_t] = \mathbf{0}$$
$$\implies \mathbf{w}_{\text{TD}} = \mathbb{E}[\mathbf{x}_t(\mathbf{x}_t - \gamma \mathbf{x}_{t+1})^\top]^{-1}\mathbb{E}[R_{t+1}\mathbf{x}_t]$$

- We can use a closed-form **empirical loss**:

$$\frac{1}{t}\sum_{i=0}^{t}(R_{i+1} + \gamma v_{\mathbf{w}}(S_{i+1}) - v_{\mathbf{w}}(S_i))\mathbf{x}_i = \mathbf{0}$$
$$\implies \mathbf{w}_{\text{LSTD}} = \left(\sum_{i=0}^{t}\mathbf{x}_i(\mathbf{x}_i - \gamma \mathbf{x}_{i+1})^\top\right)^{-1}\left(\sum_{i=0}^{t}R_{i+1}\mathbf{x}_i\right)$$

- This is called least-squares TD (**LSTD**)

Least Squares Temporal Difference

$$\mathbf{w}_t = \underbrace{\left(\sum_{i=0}^{t} \mathbf{x}_i(\mathbf{x}_i - \gamma\mathbf{x}_{i+1})^\top\right)^{-1}}_{=\mathbf{A}_t^{-1}} \underbrace{\left(\sum_{i=0}^{t} R_{i+1}\mathbf{x}_i\right)}_{=\mathbf{b}_t} \qquad \text{(LSTD estimate)}$$

- We can update $\mathbf{b}_t$ and $\mathbf{A}_t^{-1}$ incrementally **online**
- **Naive approach** ($O(n^3)$)

$$\mathbf{A}_{t+1} = \mathbf{A}_t + \mathbf{x}_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})^\top \qquad \mathbf{b}_{t+1} = \mathbf{b}_t + R_{t+1}\mathbf{x}_t$$

- **Faster approach** ($O(n^2)$): directly update $\mathbf{A}^{-1}$ with Sherman-Morrison:

$$\mathbf{A}_{t+1}^{-1} = \mathbf{A}_t^{-1} - \frac{\mathbf{A}_t^{-1}\mathbf{x}_t(\mathbf{x}_t - \gamma\mathbf{x}_{t+1})^\top\mathbf{A}_t^{-1}}{1 + (\mathbf{x}_t - \gamma\mathbf{x}_{t+1})^\top\mathbf{A}_t^{-1}\mathbf{x}_t} \qquad \mathbf{b}_{t+1} = \mathbf{b}_t + R_{t+1}\mathbf{x}_t$$

- Still more compute per step than TD ($O(n)$)

Least Squares Temporal Difference

- In the limit, LSTD and TD converge to the same fixed point
- We can extend LSTD to multi-step returns: LSTD(λ)
- We can extend LSTD to action values: LSTDQ
- We can also interlace with policy improvement:
  least-squares policy iteration (LSPI)

Experience Replay

Given experience consisting of trajectories of experience

$$\mathcal{D} = \{S_0, A_0, R_1, S_1, \ldots, S_t\}$$

Repeat:

1. Sample transition(s), e.g., $(S_n, A_n, R_{n+1}, S_{n+1})$ for $n \leq t$
2. Apply stochastic gradient descent update

$$\Delta \mathbf{w} = \alpha(R_{n+1} + \gamma v_{\mathbf{w}}(S_{n+1}) - v_{\mathbf{w}}(S_n))\nabla_{\mathbf{w}} v_{\mathbf{w}}(S_n)$$

3. Can re-use old data

This is also a form of batch learning
Beware: the data may be off-policy if the policy changes

Deep reinforcement learning

(briefly, more later)

Deep neural networks (blackboard)

Deep reinforcement learning

- Many ideas immediately transfer when using deep neural networks:
  - TD and MC
  - Double learning (e.g., double Q-learning)
  - Experience replay
  - ...
- Some ideas do not easily transfer
  - UCB
  - Least squares TD/MC

Example: neural Q-learning

- Online neural Q-learning may include:
  - **Neural network**: $O_t \mapsto \mathbf{q_w}$ (action-out)
  - **Exploration policy**: $\pi_t = \epsilon$-greedy($\mathbf{q}_t$), and then $A_t \sim \pi_t$
  - **Weight update**: for instance Q-learning

$$\Delta \mathbf{w} \propto \left(R_{t+1} + \gamma \max_a q_{\mathbf{w}}(S_{t+1}, a) - q_{\mathbf{w}}(S_t, A_t) \right) \nabla_{\mathbf{w}} q_{\mathbf{w}}(S_t, A_t)$$

  - An **optimizer** to minimize the loss (e.g., SGD, RMSProp, Adam)
- Often, we implement the weight update via a 'loss'

$$L(\mathbf{w}) = \frac{1}{2} \left(R_{t+1} + \gamma \left[\!\!\left[\max_a q_{\mathbf{w}}(S_{t+1}, a) \right]\!\!\right] - q_{\mathbf{w}}(S_t, A_t) \right)^2$$

  where $[\![\cdot]\!]$ denotes stopping the gradient, so that the **semi-gradient** is $\Delta \mathbf{w}$
- Note that $L(\mathbf{w})$ is not a real loss, it just happens to have the right gradient

Example: DQN

- DQN (Mnih et al. 2013, 2015) includes:
  - A **neural network**: $O_t \mapsto \mathbf{q}_\mathbf{w}$ (action-out)
  - An **exploration policy**: $\pi_t = \epsilon$-greedy$(\mathbf{q}_t)$, and then $A_t \sim \pi_t$
  - A **replay buffer** to store and sample past transitions $(S_i, A_i, R_{i+1}, S_{i+1})$
  - **Target network parameters** $\mathbf{w}^-$
  - A Q-learning **weight update** on $\mathbf{w}$ (uses replay and target network)

$$\Delta\mathbf{w} = \left(R_{i+1} + \gamma \max_a q_{\mathbf{w}^-}(S_{i+1}, a) - q_\mathbf{w}(S_i, A_i)\right) \nabla_\mathbf{w} q_\mathbf{w}(S_i, A_i)$$

  - Update $\mathbf{w}_t^- \leftarrow \mathbf{w}_t$ occasionally (e.g., every 10000 steps)
  - An **optimizer** to minimize the loss (e.g., SGD, RMSprop, or Adam)
- Replay and target networks make RL look more like supervised learning
- Neither is strictly necessary, but they helped for DQN
- "DL-aware RL"

End of Lecture