Introduction to Reinforcement Learning

Harm van Seijen
Microsoft Research Montreal

SPCOM 2018 Tutorial

Microsoft

What is Reinforcement Learning

Reinforcement learning considers the task of learning behaviour in an initially unknown environment.

- Initially, actions are tried at random
- Good behaviour is *reinforcement* with a positive reward

A bit of history

Reinforcement learning is about 30 years old….

TD gammon
(Tesauro, 1992)

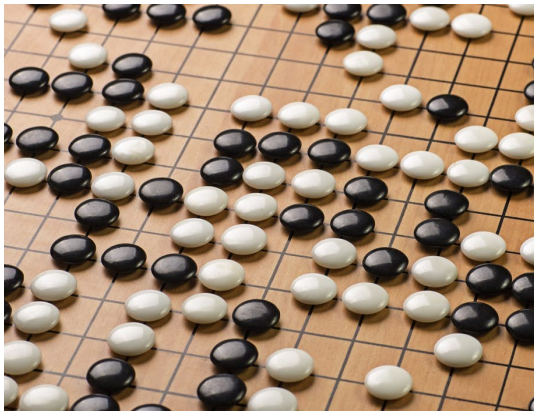
Autonomous helicopter
(Ng et al., 2004)

….but the field really took off in the last 3 years

Some recent results…

DQN

(Mnih et al, 2015)



AlphaGo Zero

(D. Silver, 2017)

Overview Tutorial

Part 1: Reinforcement Learning Theory

Part 2: Deep Reinforcement Learning

Part 3: Open Problems

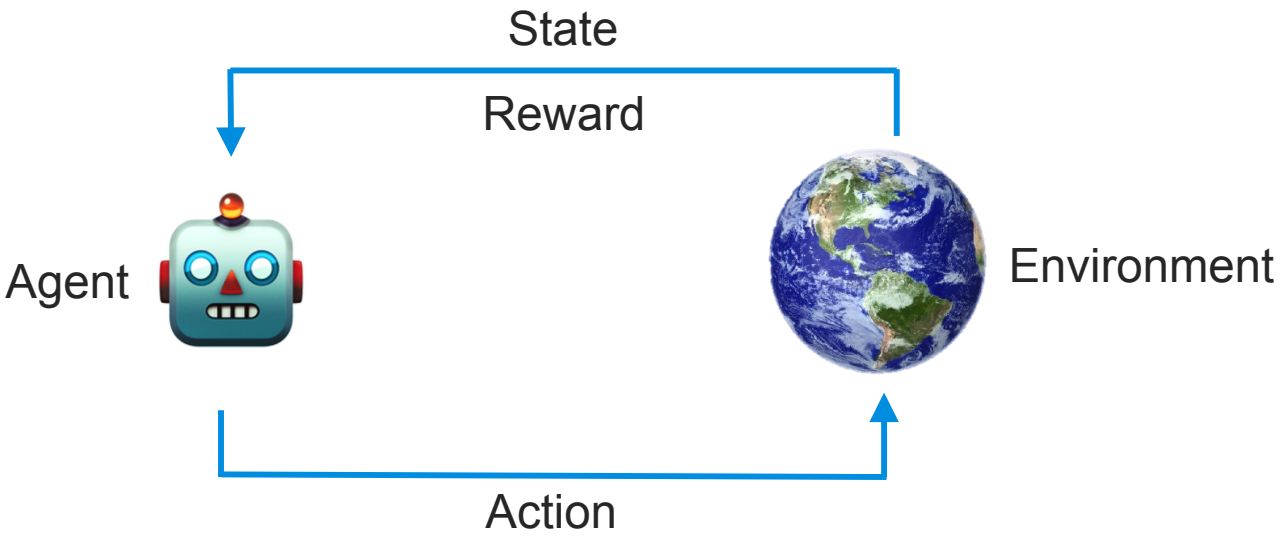
Reinforcement Learning Theory

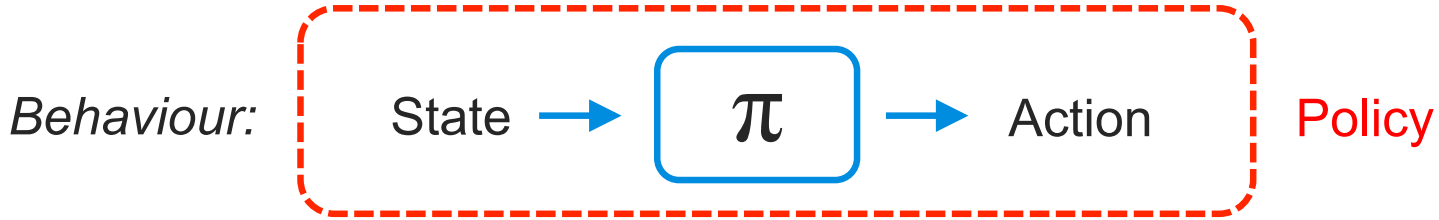
1. Markov decision processes
2. Monte Carlo
3. Temporal-Difference learning
4. Multi-Step methods
5. Function Approximation
6. Other Approaches

credit:
course: Reinforcement Learning for Artificial Intelligence
book: Introduction to Reinforcement Learning

Markov Decision Processes

Agent-Environment Interface





Goal: find a policy that results in the highest sum of rewards

finite Markov Decision Process

- discrete time $t = 1, 2, 3, \ldots$
- finite set of states
- finite set of actions
- finite set of rewards
- life is a trajectory:

$$\ldots S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1}, R_{t+2}, S_{t+2}, \ldots$$

- with arbitrary Markov (stochastic, state-dependent) dynamics:

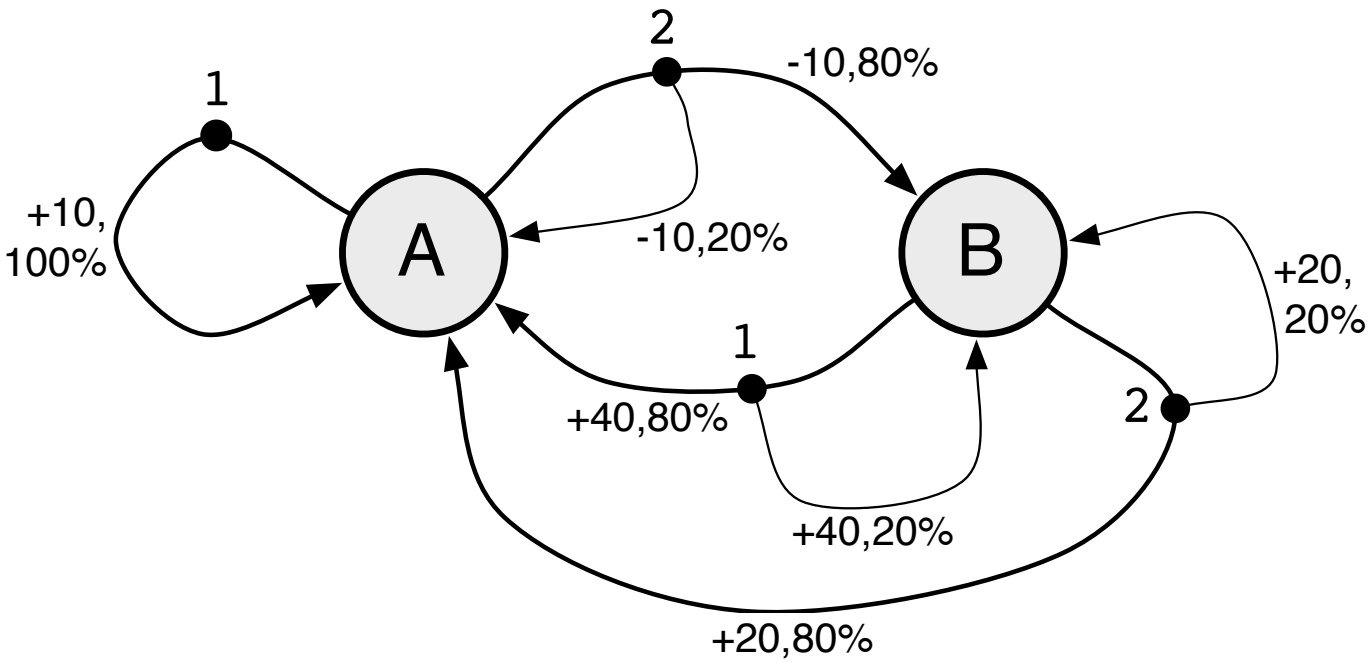
$$p(r, s'|s, a) = Prob\Big[R_{t+1} = r, S_{t+1} = s' \;\Big|\; S_t = s, A_t = a\Big]$$

transition function: $p(s'|s, a)$ reward function: $r(s, a, s')$

Episodic Tasks vs Continuing Tasks

- Some tasks have a natural end-point; we call these episodic tasks.
  Example: a game of Chess.
- Other tasks are more continuing of nature; we call these continuing tasks.
  Example: regulating the temperature in a room.
- Episodic tasks are modelled using an MDP containing special states, called *terminal states*
- The MDP of a continuing tasks does not contain any terminal states.

finite Markov Decision Process



Agent behaviour

- deterministic policy: $\pi : \mathcal{S} \to \mathcal{A}$
- stochastic policy: $\pi : \mathcal{S} \times \mathcal{A} \to [0, 1]$
- The agent's goal is to maximize the discounted sum or rewards, called the *return*:

continuing task:

discount factor $\gamma \in [0, 1]$

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1}$$

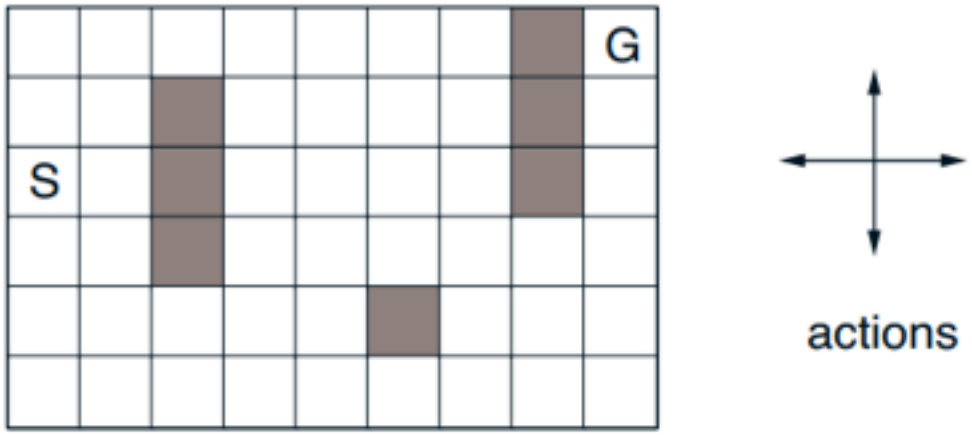
episodic task:

$$G_t = R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{T-t-1} R_T = \sum_{k=0}^{T-t-1} \gamma^k R_{t+k+1}$$

Agent is told *what* to achieve, but not *how*

- The reward function $r(s, a, s')$ together with the discount factor γ specify the goal of the agent
- The agent *learns* how to take actions that achieve this goal

Episodic Task Example



Goal: reach the goal-state as quickly as possible

Possible reward function: -1 at every step, $\gamma = 1$

Alternatively: 0 at every step; +1 on reaching goal-state, $\gamma < 1$

Value Functions

- The **value of a state** is the expected return starting from that state; depends on the agent's policy:

State - value function for policy π :

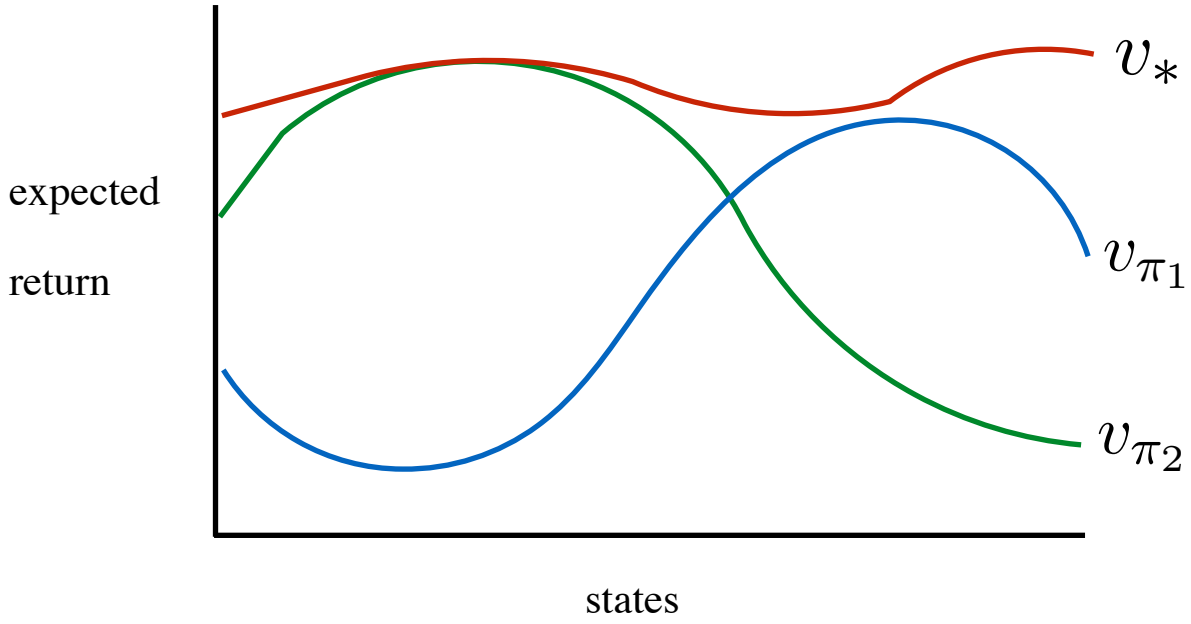
$$v_\pi(s) = E_\pi\left\{G_t \mid S_t = s\right\} = E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s\right\}$$

- The **value of an action (in a state)** is the expected return starting after taking that action from that state; depends on the agent's policy:

Action - value function for policy π :

$$q_\pi(s,a) = E_\pi\left\{G_t \mid S_t = s, A_t = a\right\} = E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a\right\}$$

Evaluating policies



Given an MDP, there always exists a policy that is at least as

good as all other policies for *each* state. This is called the optimal policy π_* .

Optimal Polices

- A policy π_* is optimal iff it maximizes the value functions:

$$v_{\pi_*}(s) = \max_{\pi} v_{\pi}(s) = v_*(s) \qquad \text{for all } s$$

$$q_{\pi_*}(s,a) = \max_{\pi} q_{\pi}(s,a) = q_*(s,a) \qquad \text{for all } s,a$$

- For each MDP there is a unique v_* and q_* , but there can be multiple optimal policies
- Given an optimal action-value function, the optimal policy can be easily derived as follows:

$$\pi_*(s) = \arg\max_{a} q_*(s,a)$$

Two Tasks

- **evaluation/prediction:**

Given a policy π, compute v_π and/or q_π

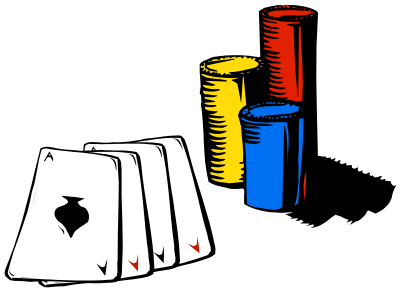
- **control:**

Find an optimal policy π_*

Monte Carlo

Blackjack example

- *Object:* Have your card sum be greater than the dealer's without exceeding 21.
- *States* (200 of them):
  - current sum (12-21)
  - dealer's showing card (ace-10)
  - do I have a useable ace?



- *Reward:* +1 for winning, 0 for a draw, -1 for losing
- *Actions:* stick (stop receiving cards), hit (receive another card)

Monte Carlo policy evaluation

- Goal: for policy π determine the function v_π
- full returns are sampled
- the estimate of $v_\pi(s)$ after episode *t*, $V_t(s)$, is the average of the observed returns from state *s* after episode *t*.
- at the end of each new episode, the estimates for states visited during the episode are updated

first-visit Monte Carlo policy evaluation

Initialize:

$\pi \leftarrow$ policy to be evaluated

$V \leftarrow$ an arbitrary state-value function

$Returns(s) \leftarrow$ an empty list, for all $s \in \mathcal{S}$

Repeat forever:

Generate an episode using π

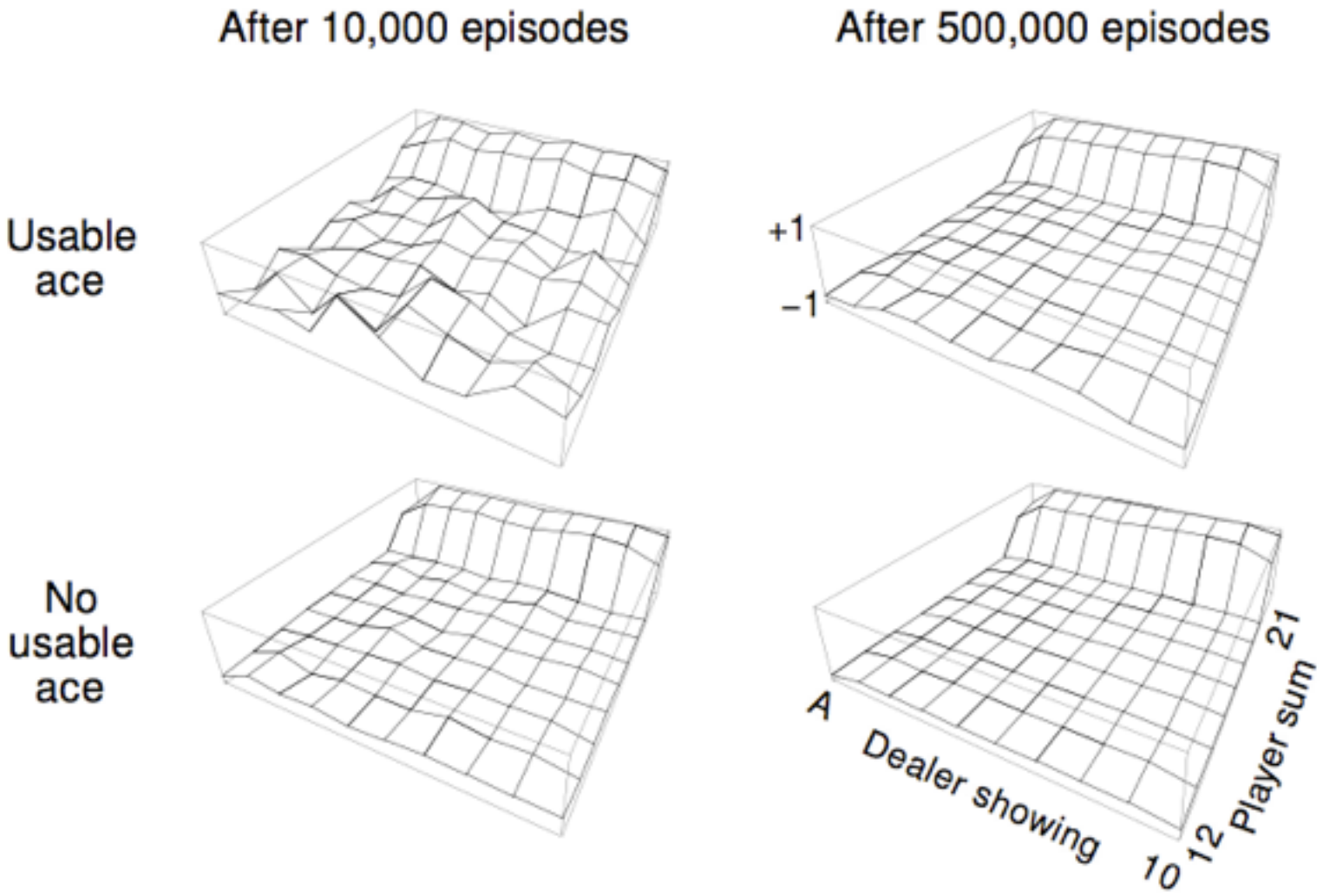
For each state s appearing in the episode:

$G \leftarrow$ return following the first occurrence of s

Append G to $Returns(s)$

$V(s) \leftarrow$ average$(Returns(s))$

Learned blackjack state-value functions



Naive approach to finding optimal policy

- evaluate the value function for all policies
- use definition: $v_{\pi_*}(s) = \max_{\pi} v_\pi(s) = v_*(s) \qquad \text{for all } s$

issue: number of possible policies is huge: $2^{200} \approx 1.6 \cdot 10^{60}$

Policy Improvement Theorem

- Given the value function for *any policy* π:

$$q_\pi(s, a) \qquad \text{for all } s, a$$

- It can always be greedified to obtain a *better policy*:

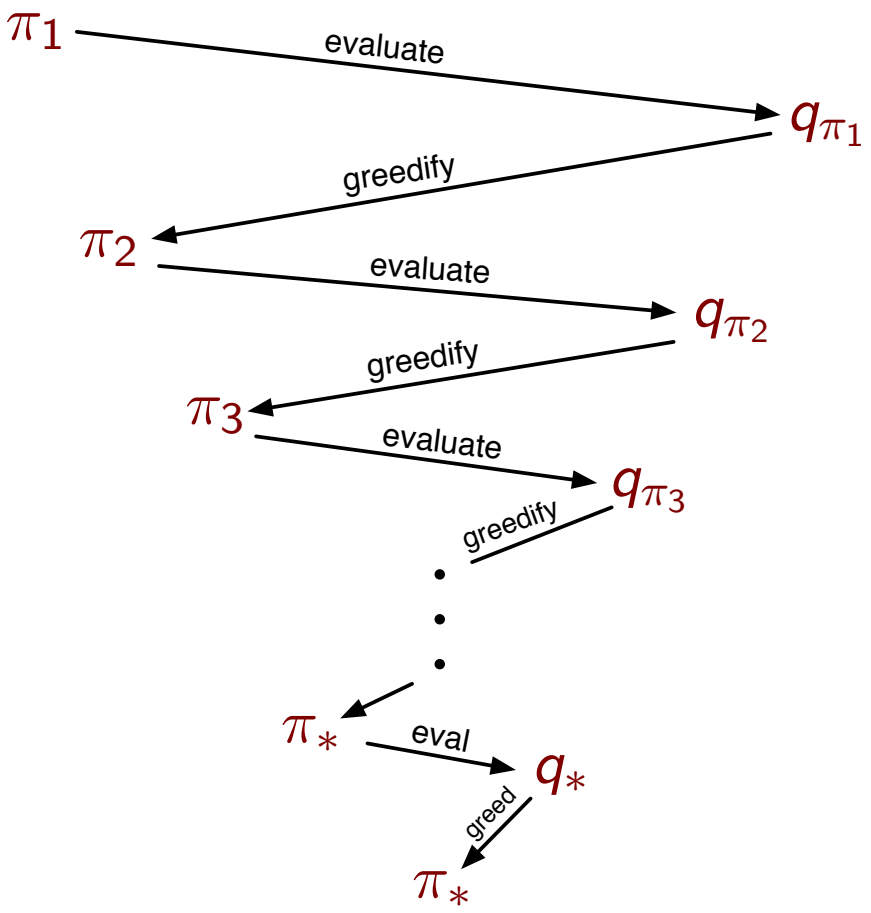
$$\pi'(s) = \arg\max_a q_\pi(s, a) \qquad (\pi' \text{ is not unique})$$

- where better means:

$$q_{\pi'}(s, a) \geq q_\pi(s, a) \qquad \text{for all } s, a$$

- with equality only if both policies are optimal

Policy Iteration



Any policy evaluates to a unique value function, which can be greedified to produce a better policy

That in turn evaluates to a value function

which can in turn be greedified…

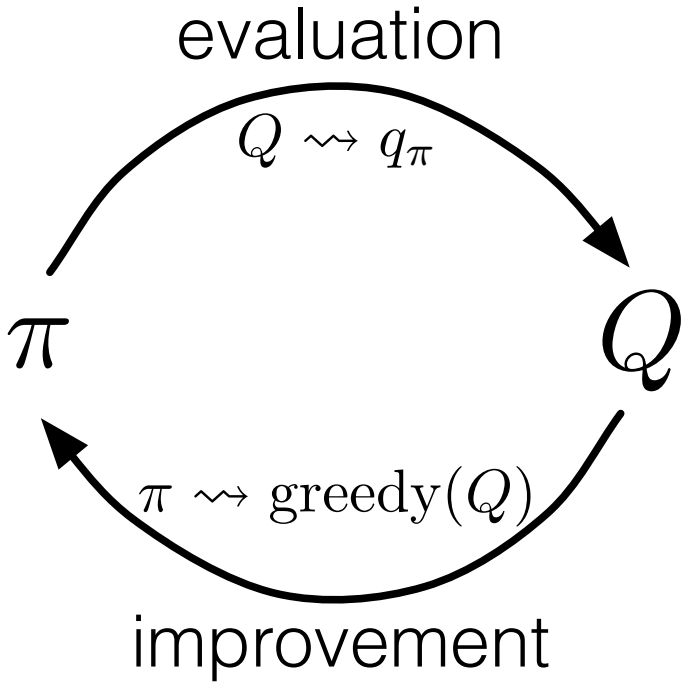
Each policy is *strictly better* than the previous, until *eventually both are optimal*

There are *no local optima*

The dance converges in a finite number of steps, usually very few

Generalized Policy Iteration

Generalized Policy Iteration (GPI):
any interaction of policy evaluation and policy improvement, independent of their granularity.



Monte Carlo Control

- $Q_t(s,a)$ is the average of observed return for state-action pair *(s,a)* after episode *t*
- $Q_t(s,a)$ is an estimate of $q_{\pi_t}(s,a)$, with π_t the policy used to generate episode *t*
- At the end of an episode:
  - $Q_t(s,a)$ is updated for the visited state-action pairs
  - a new greedy policy is computed:

$$\pi_{t+1}(s) = \arg\max_a Q_t(s,a)$$

Exploration

- An RL agent is (partly) in control of the samples that it sees
- To get a proper evaluation, MC should observe returns for *all* state-action pairs
- Ensuring sufficient exploration of the state-action space is a big challenge for any RL method
- For now, we consider an extremely simplistic exploration strategy: exploring starts
- Exploring starts means the initial state is drawn at random and the initial action as well

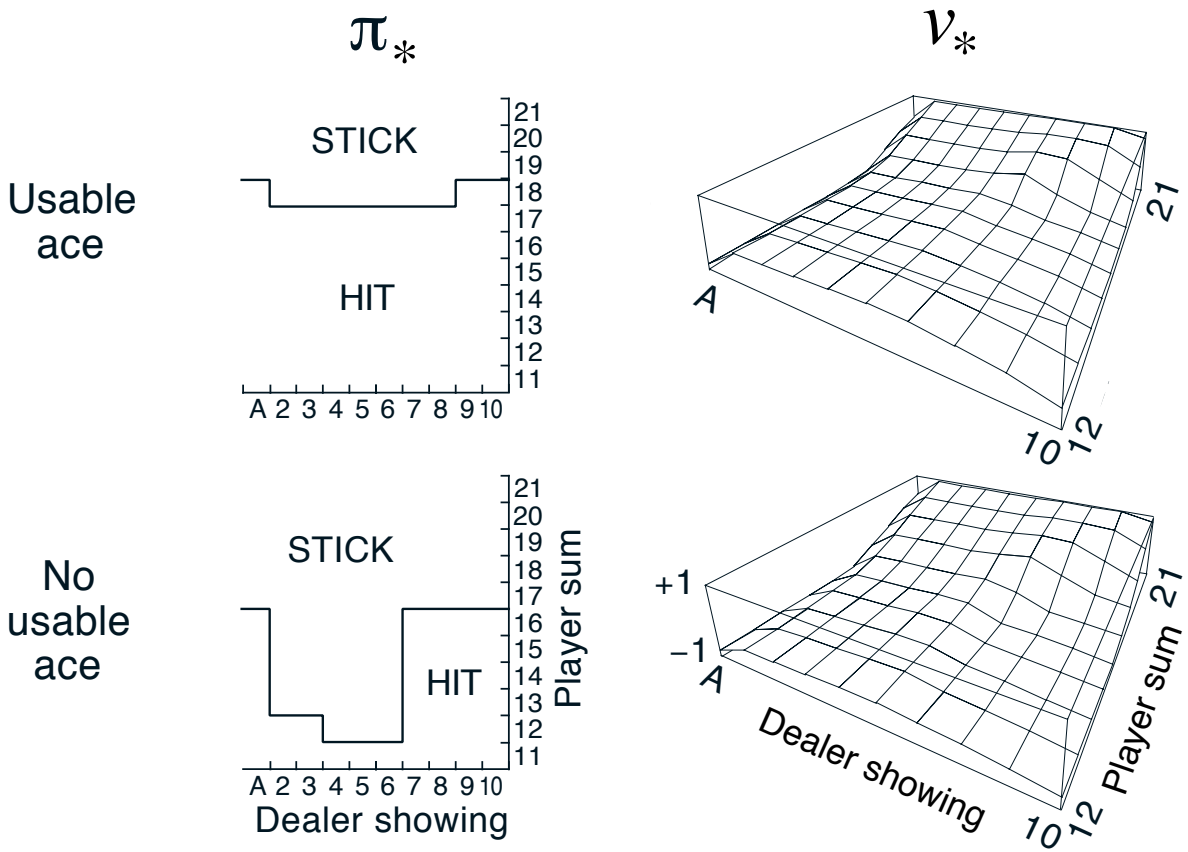
Monte Carlo Exploring Starts

Initialize, for all $s \in \mathcal{S}$, $a \in \mathcal{A}(s)$:
    $Q(s,a) \leftarrow$ arbitrary
    $\pi(s) \leftarrow$ arbitrary
    $Returns(s,a) \leftarrow$ empty list

Repeat forever:
    Choose $S_0 \in \mathcal{S}$ and $A_0 \in \mathcal{A}(S_0)$ s.t. all pairs have probability > 0
    Generate an episode starting from S_0, A_0, following π
    For each pair s, a appearing in the episode:
        $G \leftarrow$ return following the first occurrence of s, a
        Append G to $Returns(s,a)$
        $Q(s,a) \leftarrow$ average$(Returns(s,a))$
    For each s in the episode:
        $\pi(s) \leftarrow \arg\max_a Q(s,a)$

Blackjack example continued

- Exploring starts
- Initial policy as described before



Better update rule

- When doing control, maintaining an average of all returns per state-action pairs can result in very slow learning
- A better update rule is to take a weighted average of returns, with higher weights for more recent returns
- This can be achieved by update rule:

$$Q_{k+1}(s,a) = Q_k(s,a) + \alpha(G_k - Q_k(s,a))$$

or $$Q_{k+1}(s,a) = (1-\alpha)Q_k(s,a) + \alpha G_k$$

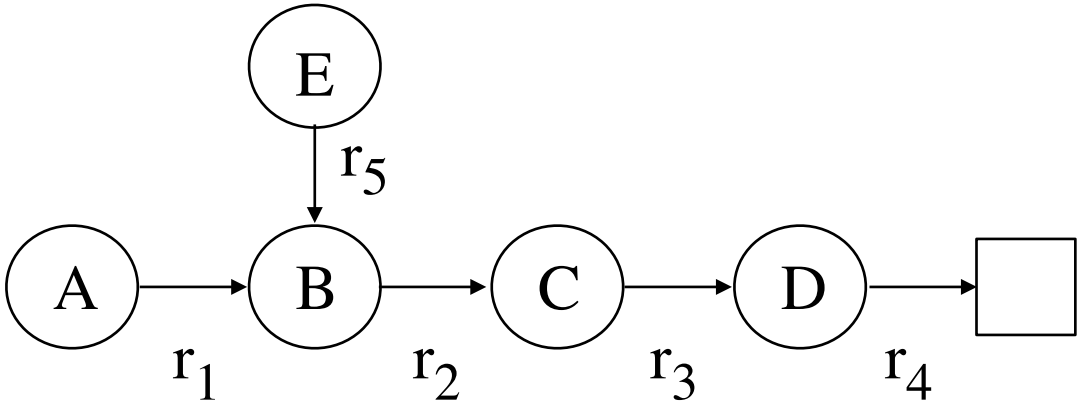
- The step-size $\alpha \in (0,1]$ determines how much weight is put on recent observations

Disadvantages of Monte Carlo

- Updates only occur at the end of an episode, which can delay learning for long episodes
- Cannot be used for continuing tasks
- In stochastic environments, the variance can be very high

Temporal Difference Learning

Motivating Example



Consider:

- $\gamma = 1$
- all rewards are stochastic
- state-values for state A and B:

$$v(A) = \mathbb{E}\{r_1 + r_2 + r_3 + r_4\}$$
$$v(B) = \mathbb{E}\{r_2 + r_3 + r_4\}$$

- state-value evaluation for state E:

$$\begin{aligned} v(E) &= \mathbb{E}\{r_5 + r_2 + r_3 + r_4\} \\ &= \mathbb{E}\{r_5\} + \mathbb{E}\{r_2 + r_3 + r_4\} \\ &= \mathbb{E}\{r_5\} + v(B) \end{aligned}$$

Temporal difference learning

- Instead of using the full return as update target, an update target is used that bootstraps from other value estimates
- Updates happen at each step of an episode

Bellman Equation for a Policy π

The basic idea:

$$\begin{aligned} G_t &= R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \gamma^3 R_{t+4} + \cdots \\ &= R_{t+1} + \gamma \left(R_{t+2} + \gamma R_{t+3} + \gamma^2 R_{t+4} + \cdots \right) \\ &= R_{t+1} + \gamma G_{t+1} \end{aligned}$$

So:

$$\begin{aligned} v_\pi(s) &= E_\pi \left\{ G_t \,\middle|\, S_t = s \right\} \\ &= E_\pi \left\{ R_{t+1} + \gamma v_\pi \left(S_{t+1} \right) \middle|\, S_t = s \right\} \end{aligned}$$

Or, without the expectation operator:

$$v_\pi(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \left[r + \gamma v_\pi(s') \right]$$

TD Prediction

Policy Evaluation (the prediction problem):

for a given policy π, compute the state-value function v_π

Recall: Simple every-visit Monte Carlo method:

$$V(S_t) \leftarrow V(S_t) + \alpha\Big[G_t - V(S_t)\Big]$$

target: the actual return after time *t*

The simplest temporal-difference method TD(0):

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

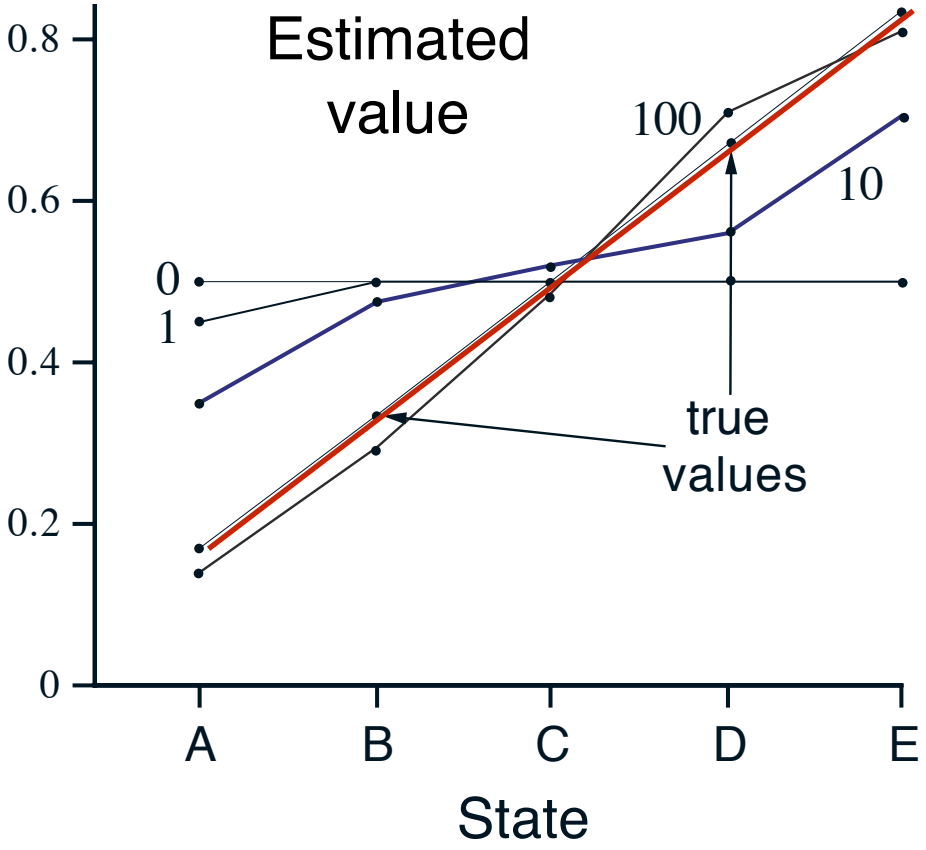
target: an estimate of the return

TD Prediction

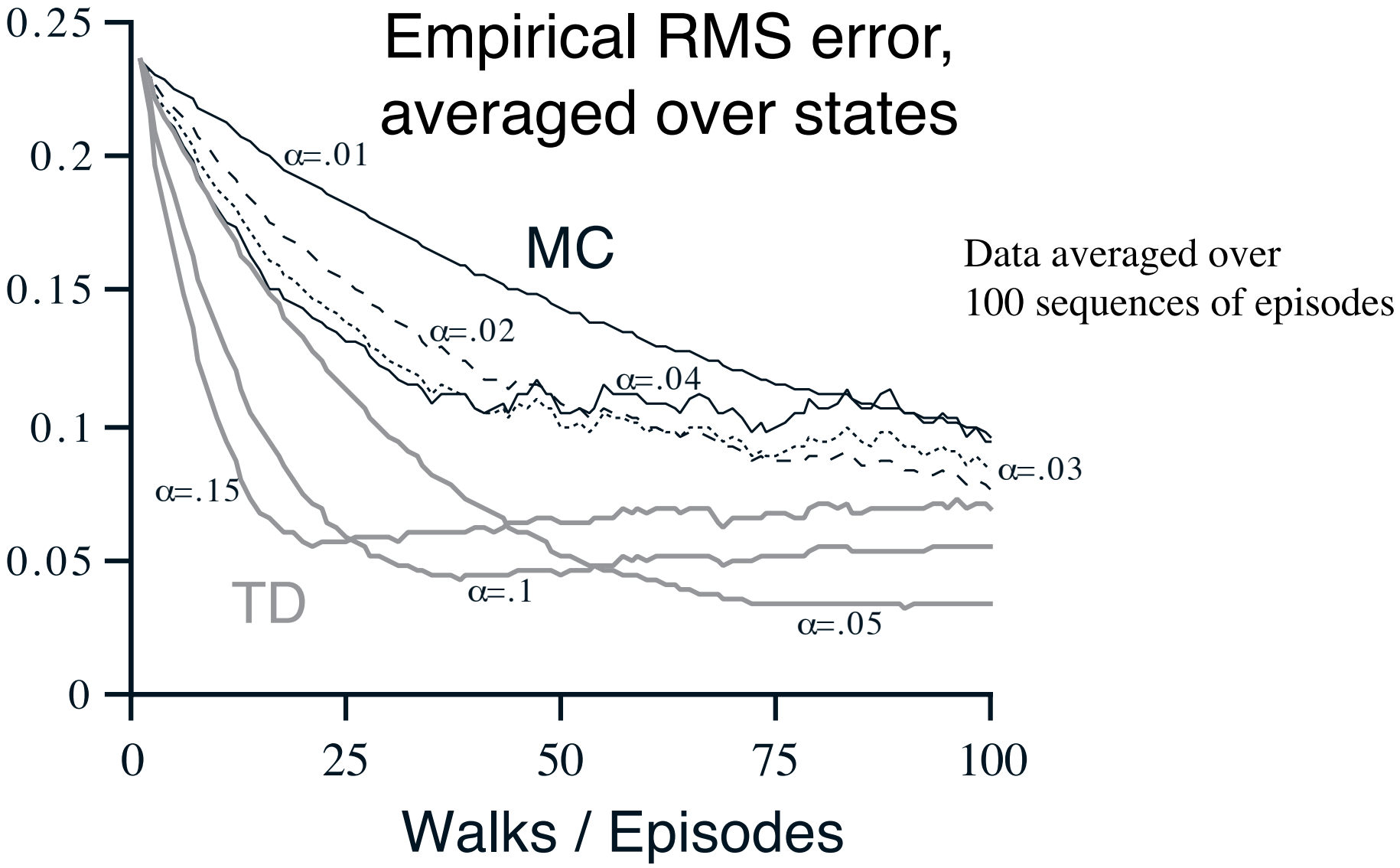
Tabular TD(0) for estimating v_π

Input: the policy π to be evaluated
Initialize $V(s)$ arbitrarily (e.g., $V(s) = 0$, for all $s \in \mathcal{S}^+$)
Repeat (for each episode):
    Initialize S
    Repeat (for each step of episode):
        $A \leftarrow$ action given by π for S
        Take action A, observe R, S'
        $V(S) \leftarrow V(S) + \alpha\big[R + \gamma V(S') - V(S)\big]$
        $S \leftarrow S'$
    until S is terminal

Values learned by TD from one run, after various numbers of episodes

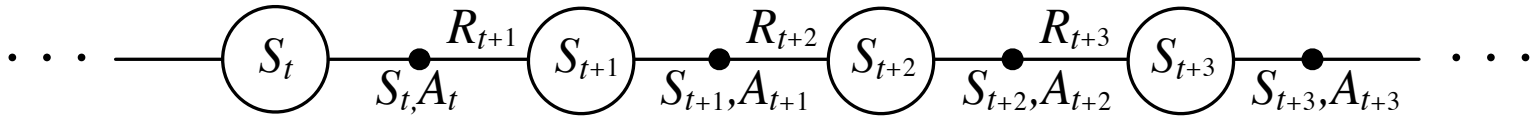


TD and MC on the Random Walk



Learning An Action-Value Function

Estimate q_π for the current policy π



After every transition from a nonterminal state, S_t, do this:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\left[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\right]$$

If S_{t+1} is terminal, then define $Q(S_{t+1}, A_{t+1}) = 0$

Exploration

- To ensure sufficient exploration of all state-action pairs, typically a stochastic policy is used to generate samples
- For example, an ε-greedy policy select with 1-ε probability the greedy action, and selects with ε probability an action uniformly at random

Sarsa(0): TD Control

Initialize $Q(s,a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
    Initialize S
    Choose A from S using policy derived from Q (e.g., ε-greedy)
    Repeat (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

Note: because a stochastic policy is used (e.g. ε-greedy), Sarsa does not converge to the optimal policy, unless the stochasticity is decreased slowly over time.

On-policy versus off-policy control

- behaviour policy: policy that generates the samples
- estimation policy: policy that is being estimated

- on-policy learning: behaviour policy = estimation policy
- off-policy learning: behaviour policy $\neq$ estimation policy

- up to now, we only considered on-policy learning
- off-policy learning offers more much more flexibility

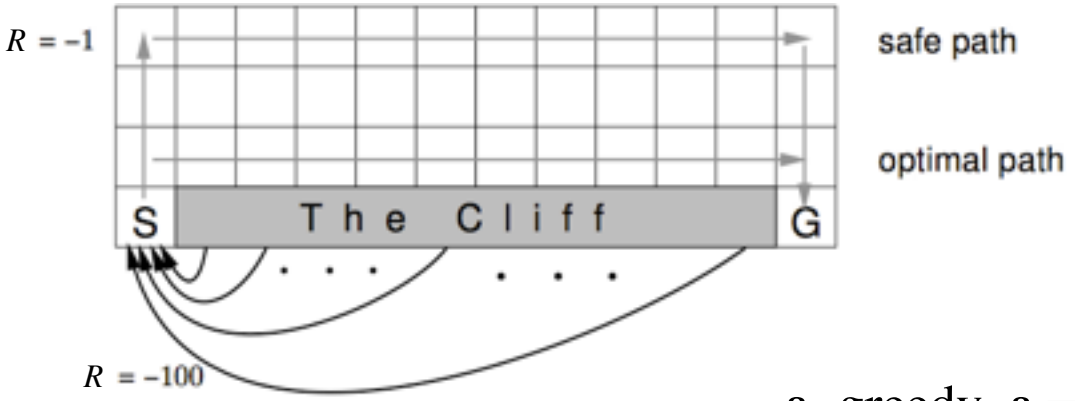
Q-Learning: Off-Policy TD Control

One-step Q-learning:

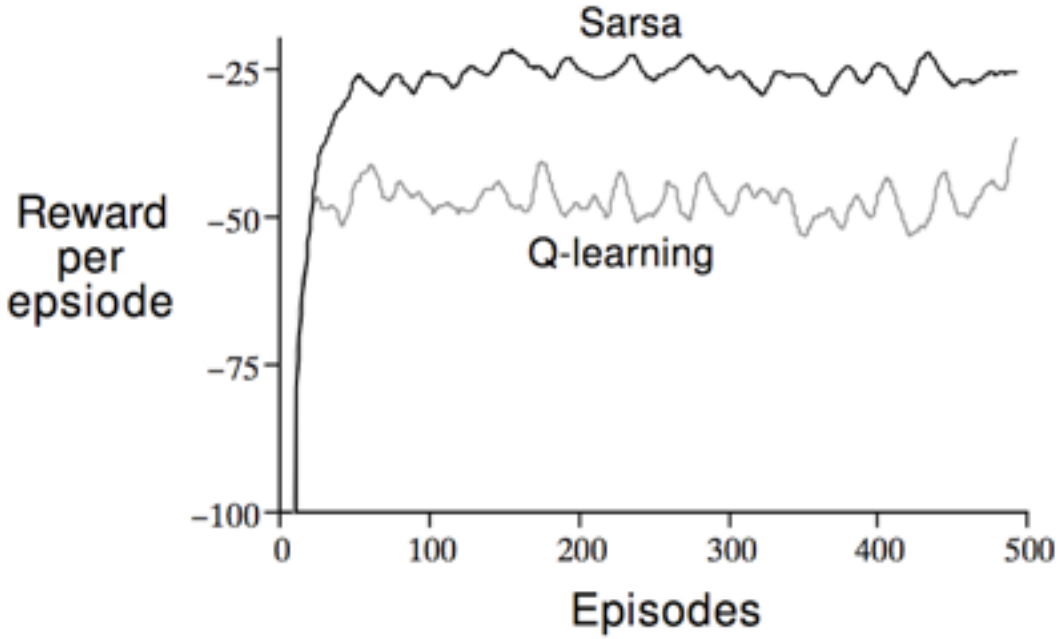
$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha\Big[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)\Big]$$

Initialize $Q(s, a), \forall s \in \mathcal{S}, a \in \mathcal{A}(s)$, arbitrarily, and $Q(\textit{terminal-state}, \cdot) = 0$
Repeat (for each episode):
    Initialize S
    Repeat (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

Cliffwalking



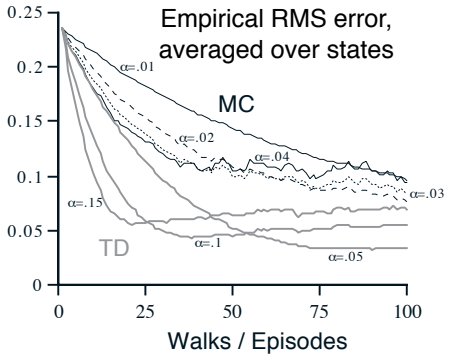
ε−greedy, ε = 0.1



Multi-step methods

Monte Carlo vs TD

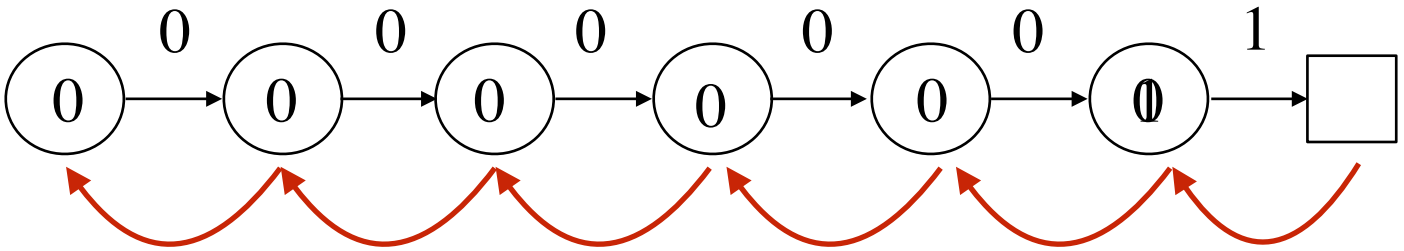
- Monte Carlo: $G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T$
- TD: $G_{t:t+1} \doteq R_{t+1} + \gamma V_t(S_{t+1})$
  - Use V_t to estimate remaining return
- On random walk task, TD outperformed MC



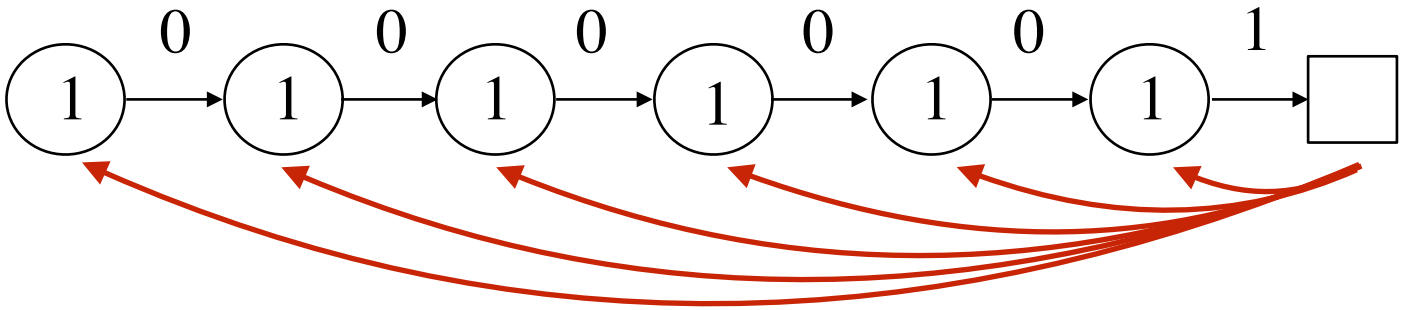
Is TD always better?

Consider: $\gamma = 1, \alpha = 1$

TD:



MC:



variance-bias trade-off

	variance	bias
ideal	low	low
TD	low	high
MC	high*	low

*In tasks with high environment stochasticity and/or policy stochasticity.

n-step Update Targets

- Monte Carlo: $G_t \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T$
- TD: $G_{t:t+1} \doteq R_{t+1} + \gamma V_t(S_{t+1})$
  - Use V_t to estimate remaining return
- *n*-step TD:
  - 2-step return: $G_{t:t+2} \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 V_{t+1}(S_{t+2})$
  - *n*-step return: $G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^n V_{t+n-1}(S_{t+n})$

    with $G_{t:t+n} \doteq G_t$ if $t+n \geq T$)

n-step TD

- Recall the *n*-step return:

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

- Of course, this is not available until time *t+n*

- The natural algorithm is thus to wait until then:

$$V_{t+n}(S_t) \doteq V_{t+n-1}(S_t) + \alpha\big[G_{t:t+n} - V_{t+n-1}(S_t)\big], \qquad 0 \le t < T$$

- This is called *n*-step TD

n-step TD

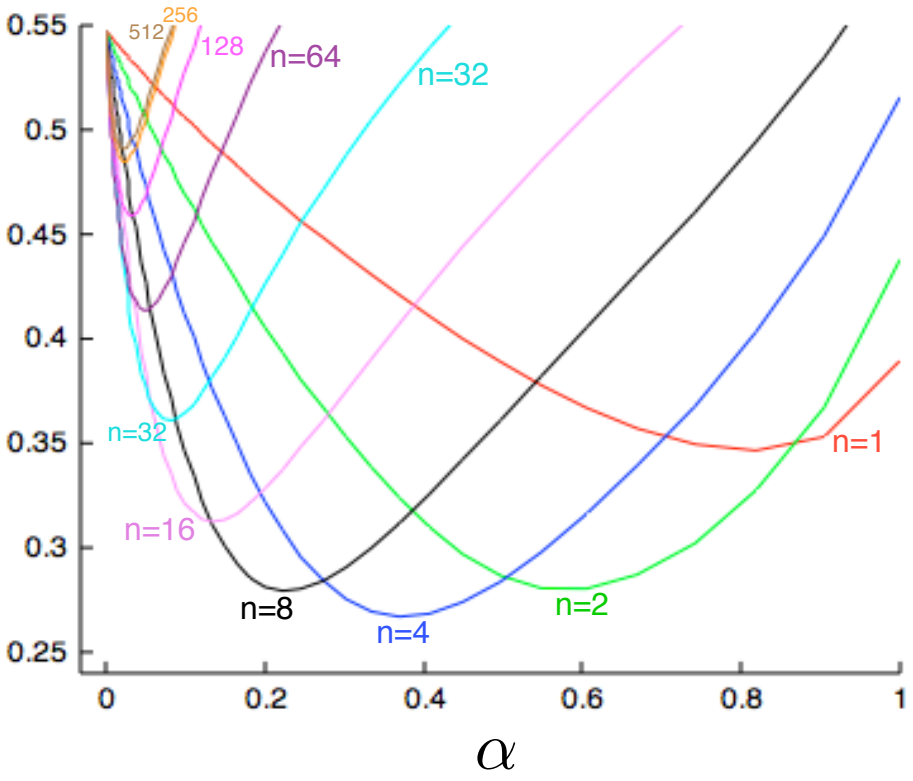
n-step TD for estimating $V \approx v_\pi$

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 $t = 0, 1, 2, \ldots$:
  | 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 $\tau = T - 1$

19-state Random Walk

Average RMS error over 19 states and first 10 episodes



n-step TD results

- An intermediate α is best
- An intermediate n is best

On-policy *n*-step control is straightforward

- Action-value form of *n*-step return

$$G_{t:t+n} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{n-1} R_{t+n} + \gamma^{n} Q_{t+n-1}(S_{t+n}, A_{t+n})$$

- *n*-step Sarsa:

$$Q_{t+n}(S_t, A_t) \doteq Q_{t+n-1}(S_t, A_t) + \alpha \left[G_{t:t+n} - Q_{t+n-1}(S_t, A_t) \right]$$

Off-policy *n*-step control is challenging

- Recall: off-policy updating involves a behaviour policy, b, and an estimation policy, π.
- Off-policy multi-step updates is possible using *Importance Sampling*
- Each update target is multiplied by importance sampling ratio: $$\rho_{t:h} \doteq \prod_{k=t}^{\min(h,T-1)} \frac{\pi(A_k|S_k)}{b(A_k|S_k)}$$
- Off-policy *n*-step TD:

$$V_{t+n}(S_t) \doteq V_{t+n-1}(S_t) + \alpha\,\rho_{t:t+n-1}\,[G_{t:t+n} - V_{t+n-1}(S_t)]$$

- Off-policy *n*-step updates suffer from a large variance

Value Function Approximation

Motivation

- So far, we discussed tabular methods, where values are stored and estimated on a per-state basis
- This approach only works when the total number of states is relatively small
- RL suffers from the *curse of dimensionality:* the number of states tends to grow exponentially with the problem size
- Value function approximation scales RL to large (or continuous) state-spaces

Value Function Approximation

transpose

*i*th components

$$\cancel{V(s) \approx} \; v_\pi(s) \;\approx\; \hat{v}(s,\mathbf{w}) \;\doteq\; \mathbf{w}^\top \mathbf{x}(s) \;\doteq\; \sum_{i=1}^{d} w_i \cdot x_i(s) = 1.71$$

inner product

parameter vector

$$\mathbf{w} \in \mathbb{R}^d, \quad e.g., \quad \mathbf{w} = \begin{bmatrix} 2.1 \\ 0.01 \\ -1.1 \\ 1.2 \\ -0.1 \\ 0.01 \\ 4.93 \\ 0.5 \end{bmatrix}, \qquad \mathbf{x}(s) = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ 0 \\ 0 \\ 0 \\ 1 \end{bmatrix}, \quad \mathbf{x} : \mathcal{S} \to \mathbb{R}^d$$

feature vector

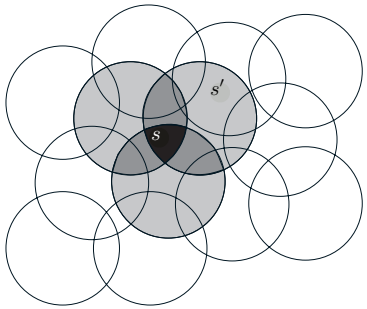
$$\cancel{Q(s,a) \approx} \; q_\pi(s,a) \;\approx\; \hat{q}(s,a,\mathbf{w}) \;\doteq\; \mathbf{w}^\top \mathbf{x}(s,a) \;\doteq\; \sum_{i=1}^{d} w_i \cdot x_i(s,a)$$

Stochastic Gradient Descent (SGD) is the idea behind most approximate learning

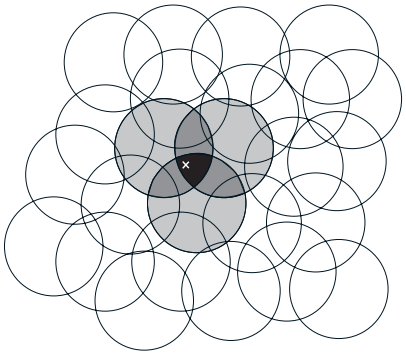
$$
\begin{aligned}
\text{General SGD:} \quad & \mathbf{w} \leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} \, Error_t^2 \\
\text{For VFA:} \quad & \leftarrow \mathbf{w} - \alpha \nabla_{\mathbf{w}} \left[Target_t - \hat{v}(S_t, \mathbf{w}) \right]^2 \\
\text{Chain rule:} \quad & \leftarrow \mathbf{w} - 2\alpha \left[Target_t - \hat{v}(S_t, \mathbf{w}) \right] \nabla_{\mathbf{w}} \left[Target_t - \hat{v}(S_t, \mathbf{w}) \right] \\
\text{Semi-gradient:} \quad & \leftarrow \mathbf{w} + \alpha \left[Target_t - \hat{v}(S_t, \mathbf{w}) \right] \nabla_{\mathbf{w}} \hat{v}(S_t, \mathbf{w}) \\
\text{Linear case:} \quad & \leftarrow \mathbf{w} + \alpha \left[Target_t - \hat{v}(S_t, \mathbf{w}) \right] \mathbf{x}(S_t)
\end{aligned}
$$

$$
\text{Action-value form:} \quad \mathbf{w} \leftarrow \mathbf{w} + \alpha \left[Target_t - \hat{q}(S_t, A_t, \mathbf{w}) \right] \mathbf{x}(S_t, A_t)
$$

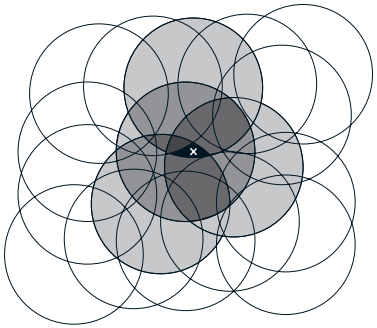
Different features give different generalization



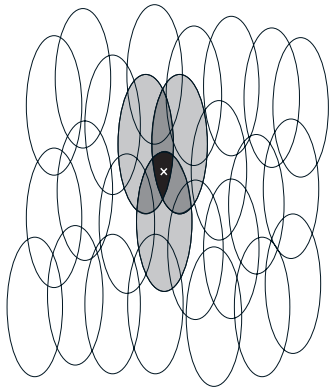
Idea



a) Narrow generalization



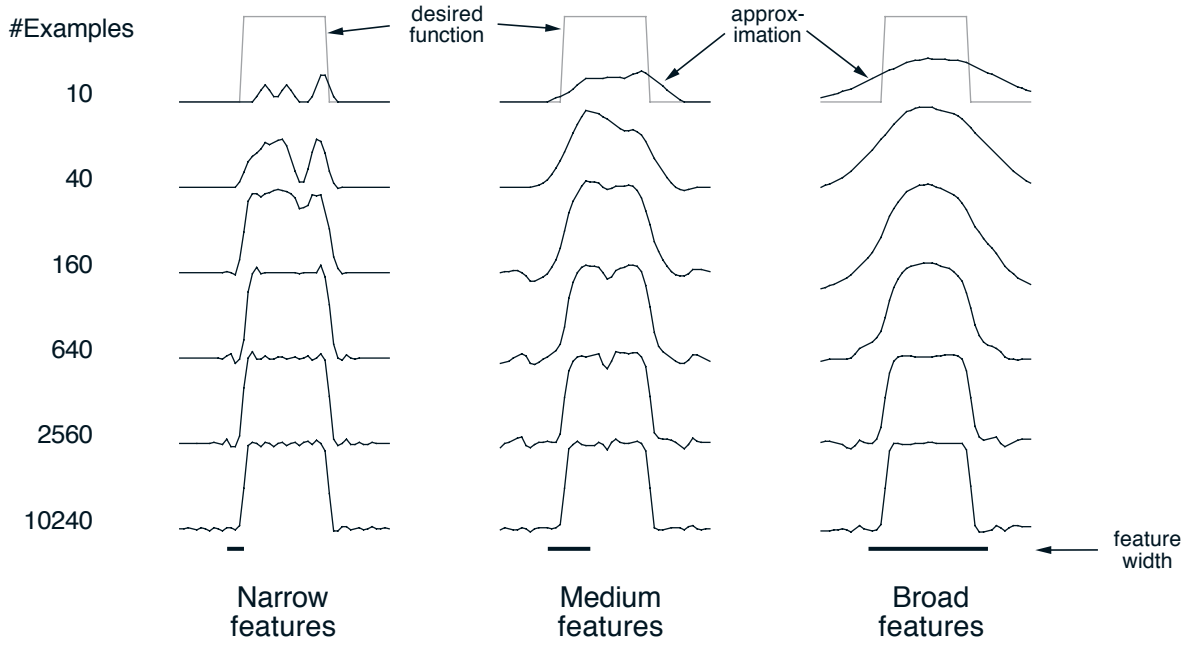
b) Broad generalization



c) Asymmetric generalization

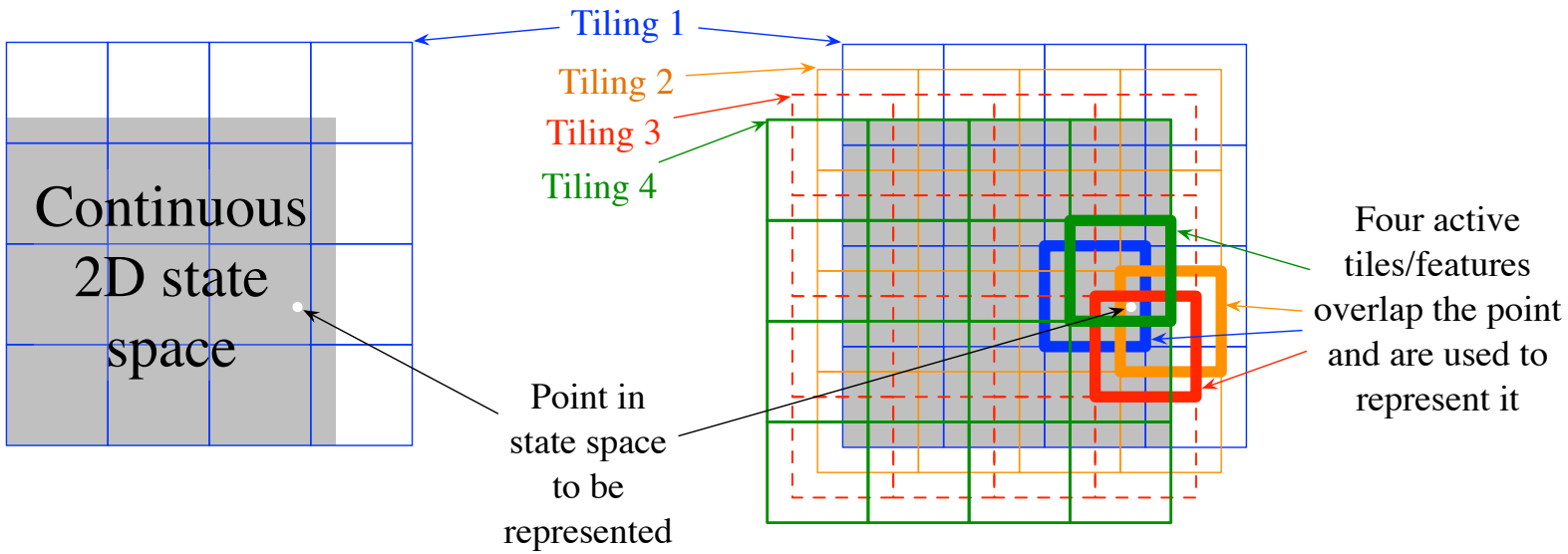
The width of the receptive fields determines breadth of generalization

1D example,
supervised training

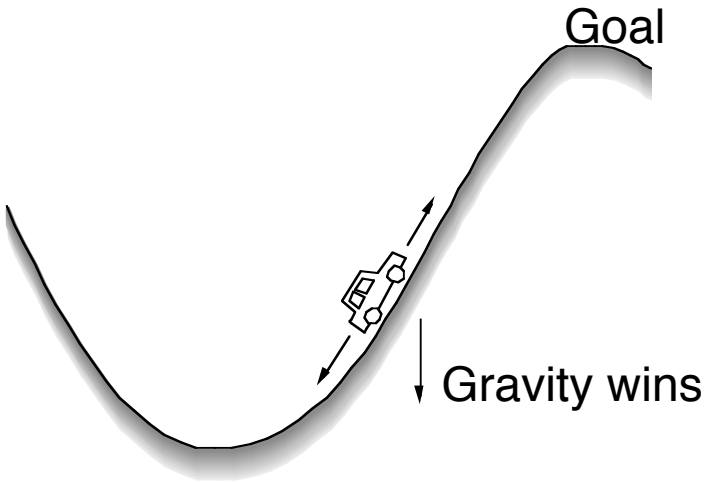


Tile coding is coarse coding for digital computers, with rectangular receptive fields, controlled overlap

2D example



Example: The Mountain-Car problem



Minimum-Time-to-Goal Problem

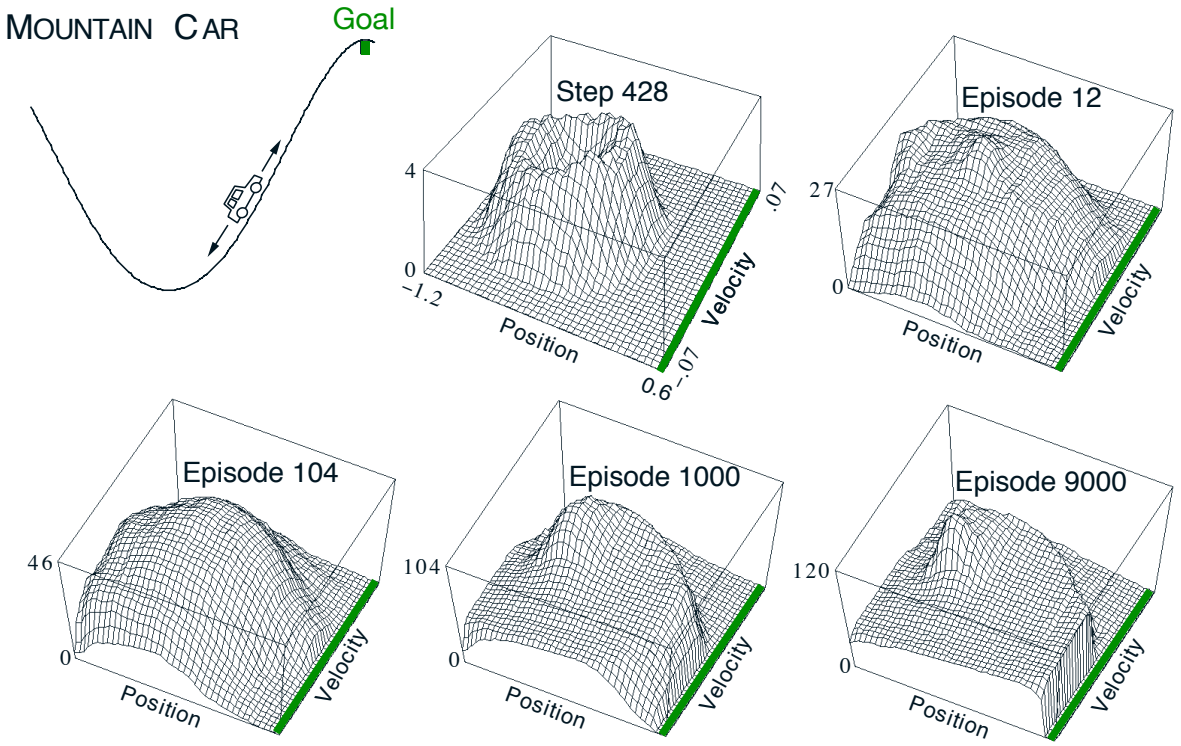
SITUATIONS:
car's position and velocity

ACTIONS:
three thrusts: forward, reverse, none

REWARDS:
always –1 until car reaches the goal

Episodic, No Discounting,
γ=1

Values learned while solving Mountain-Car with tile coding function approximation



$-\max_a \hat{q}(s, a, \mathbf{w})$

Other Approaches

Other Approaches

- So far, we discussed methods that learn a value function directly from samples with the policy being simplicity defined (f.e. greedy w.r.t. the value function)
- There are other approaches to learning a policy:
  - policy-gradient methods directly update a parameterized policy using gradient ascent
  - model-based RL methods estimate a model of the environment dynamics from samples and then use any planning method to find a policy based on the estimated model

Policy-gradient setup

Given a policy parameterization:

$$\pi(a|s, \boldsymbol{\theta})$$

And objective:

$$J(\boldsymbol{\theta}) \doteq v_{\pi_{\boldsymbol{\theta}}}(s_0)$$

Approximate stochastic gradient ascent:

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha \widehat{\nabla J(\boldsymbol{\theta}_t)}$$

Typically, based on the Policy-Gradient Theorem:

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a q_\pi(s, a) \nabla_{\boldsymbol{\theta}} \pi(a|s, \boldsymbol{\theta})$$

on-policy distribution

REINFORCE with baseline

Policy-gradient theorem with baseline:

$$\nabla J(\boldsymbol{\theta}) \propto \sum_s \mu(s) \sum_a q_\pi(s,a) \nabla_{\boldsymbol{\theta}} \pi(a|s,\boldsymbol{\theta})$$

$$= \sum_s \mu(s) \sum_a \Big(q_\pi(s,a) - b(s) \Big) \nabla_{\boldsymbol{\theta}} \pi(a|s,\boldsymbol{\theta})$$

any function of state, not action

Because

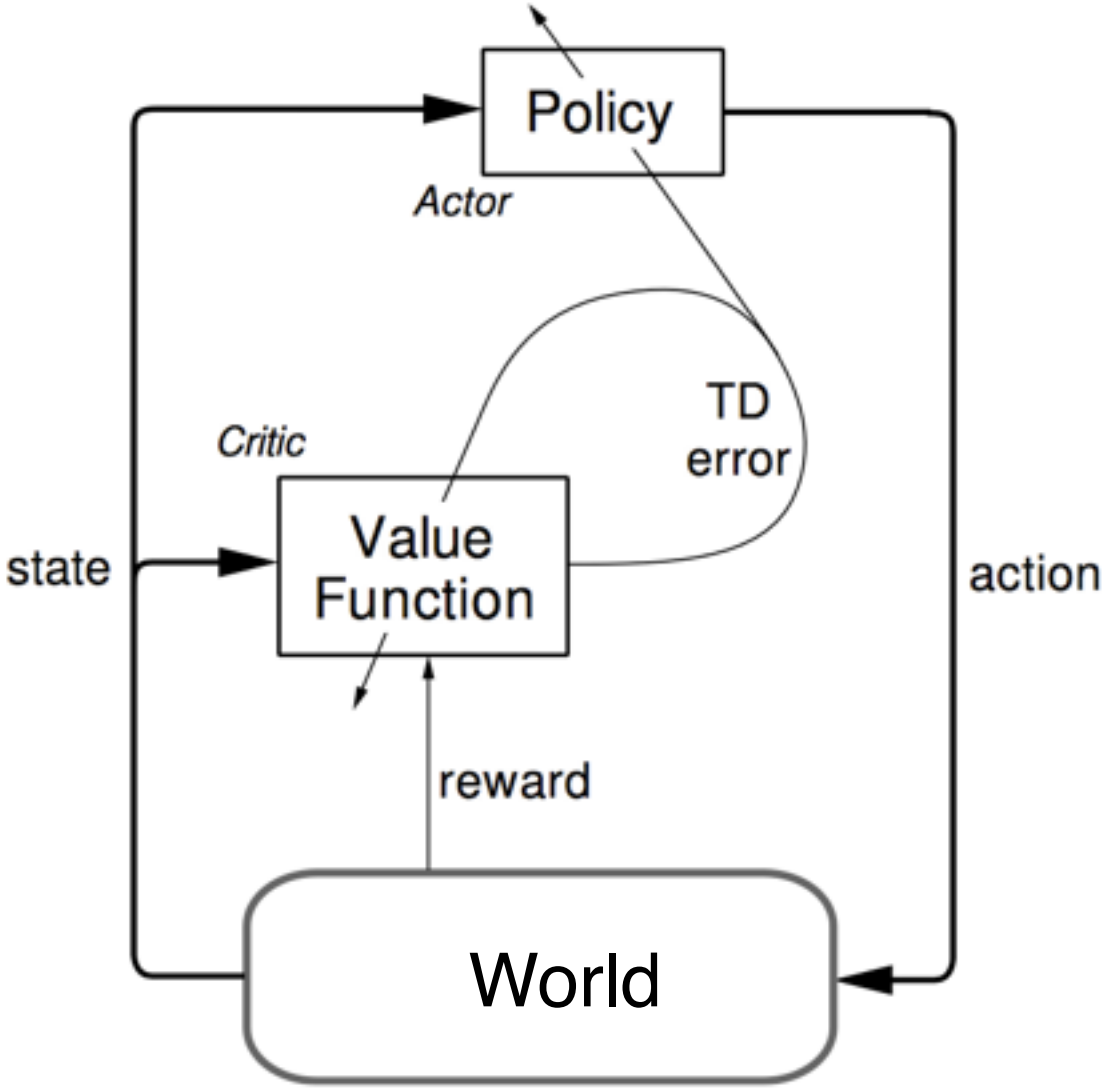
$$\sum_a b(s) \nabla_{\boldsymbol{\theta}} \pi(a|s,\boldsymbol{\theta}) = b(s) \nabla_{\boldsymbol{\theta}} \sum_a \pi(a|s,\boldsymbol{\theta}) = b(s) \nabla_{\boldsymbol{\theta}} 1 = 0 \qquad \forall s \in \mathcal{S}$$

Thus

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha \Big(G_t - b(S_t) \Big) \frac{\nabla_{\boldsymbol{\theta}} \pi(A_t|S_t,\boldsymbol{\theta}_t)}{\pi(A_t|S_t,\boldsymbol{\theta}_t)}$$

e.g., $b(s) = \hat{v}(s,\mathbf{w})$

Actor-critic architecture



Actor-Critic methods

REINFORCE with baseline:

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha\Big(G_t - b(S_t)\Big)\frac{\nabla_{\boldsymbol{\theta}}\pi(A_t|S_t,\boldsymbol{\theta}_t)}{\pi(A_t|S_t,\boldsymbol{\theta}_t)}$$

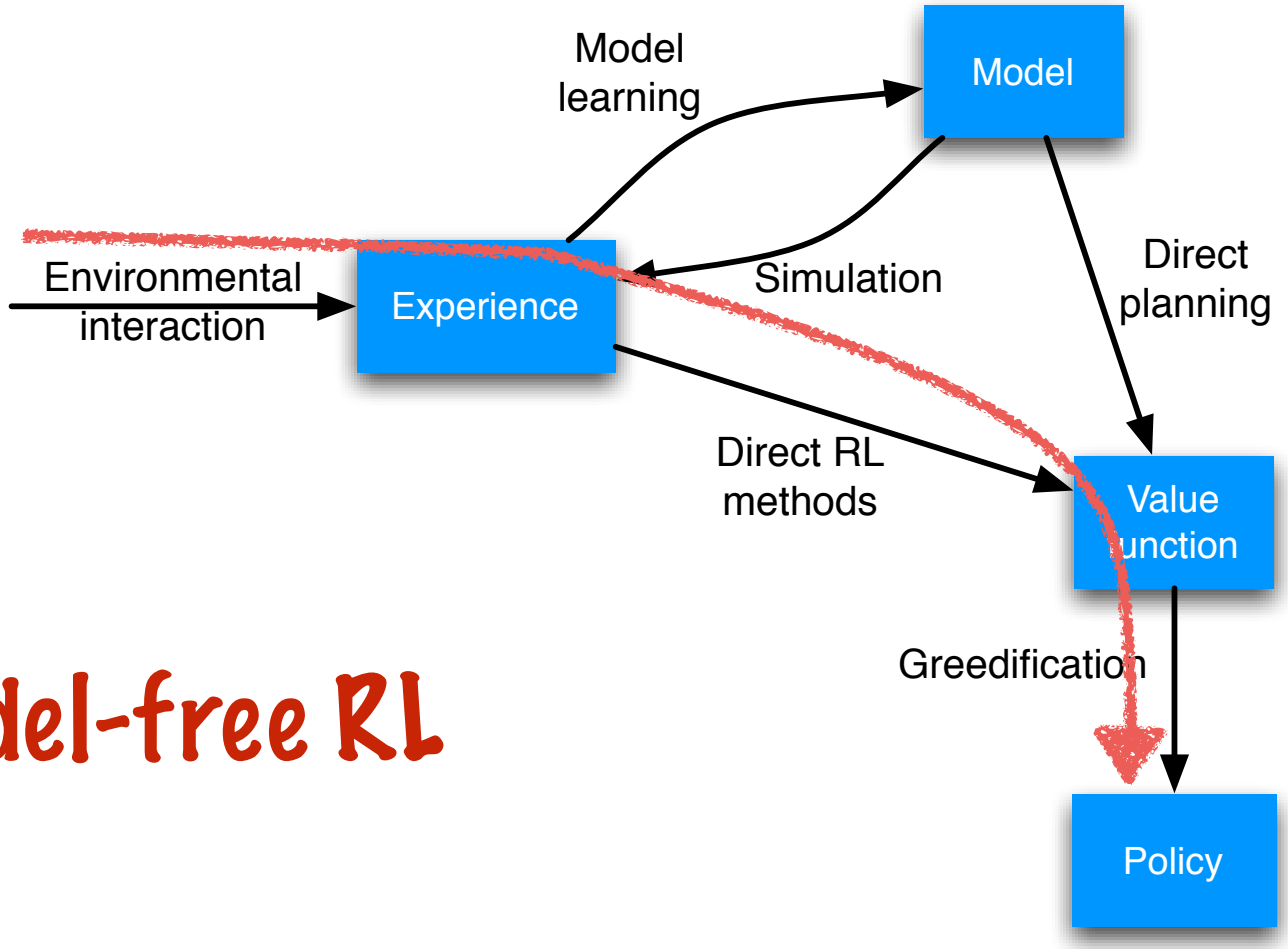
Actor-Critic method:

$$\begin{aligned}\boldsymbol{\theta}_{t+1} &\doteq \boldsymbol{\theta}_t + \alpha\Big(G_{t:t+1} - \hat{v}(S_t,\mathbf{w})\Big)\frac{\nabla_{\boldsymbol{\theta}}\pi(A_t|S_t,\boldsymbol{\theta}_t)}{\pi(A_t|S_t,\boldsymbol{\theta}_t)}\\ &= \boldsymbol{\theta}_t + \alpha\Big(R_{t+1} + \gamma\hat{v}(S_{t+1},\mathbf{w}) - \hat{v}(S_t,\mathbf{w})\Big)\frac{\nabla_{\boldsymbol{\theta}}\pi(A_t|S_t,\boldsymbol{\theta}_t)}{\pi(A_t|S_t,\boldsymbol{\theta}_t)}\\ &= \boldsymbol{\theta}_t + \alpha\delta_t\frac{\nabla_{\boldsymbol{\theta}}\pi(A_t|S_t,\boldsymbol{\theta}_t)}{\pi(A_t|S_t,\boldsymbol{\theta}_t)}.\end{aligned}$$

Model-based methods

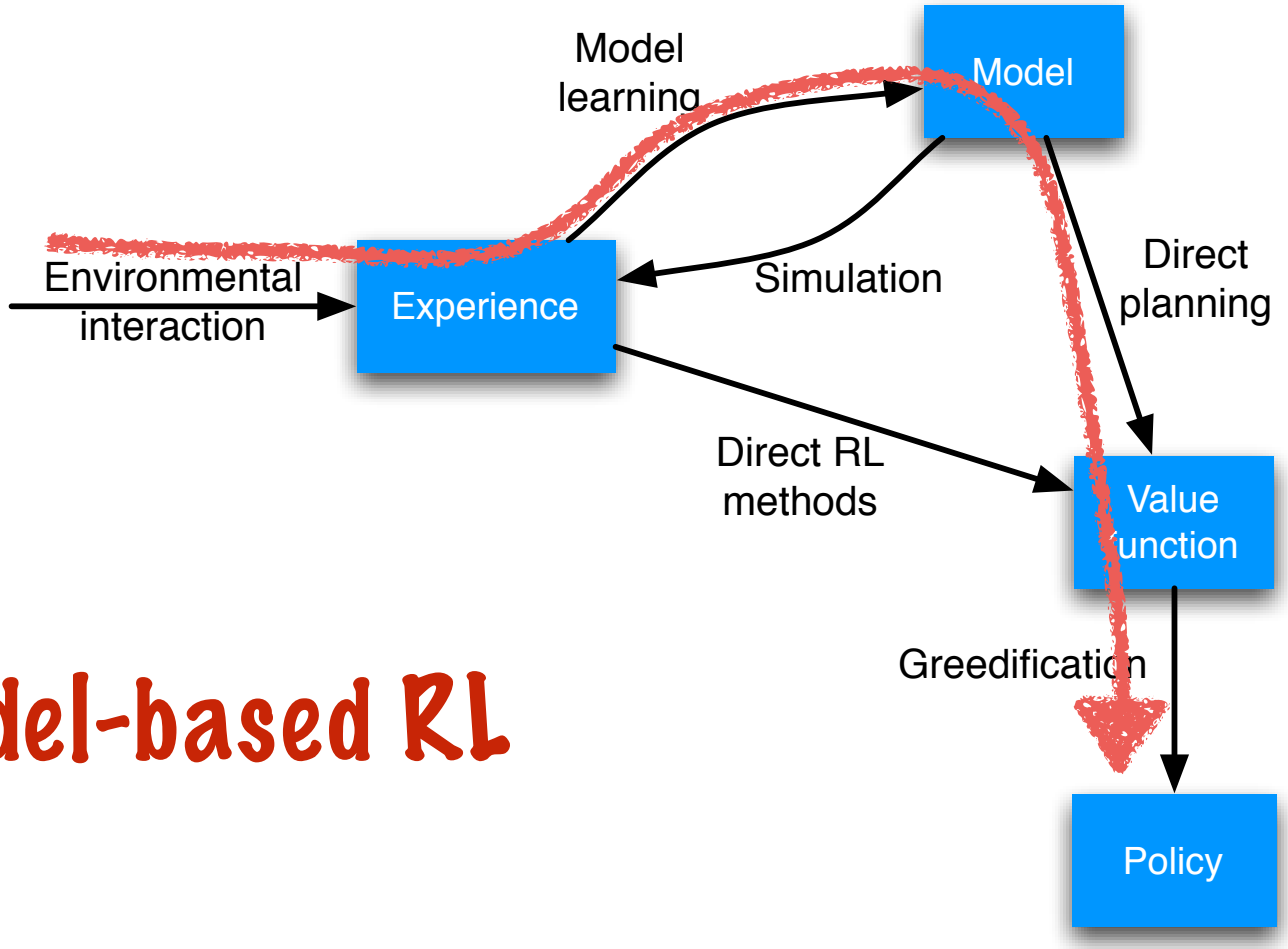
- Model: anything the agent can use to predict how the environment will respond to its actions
- A distribution model aims to estimate the one-step dynamics of the environment. That is, it aims to estimate $p(s', r \mid s, a)$ for all s, a, s', r
- Any planning method can be used to find a policy given the estimated model, such as, dynamic programming, tree search, etc.
- Sample model, a.k.a. a simulation model
  - produces sample experiences for given s, a
    - sampled according to the probabilities

Paths to a policy



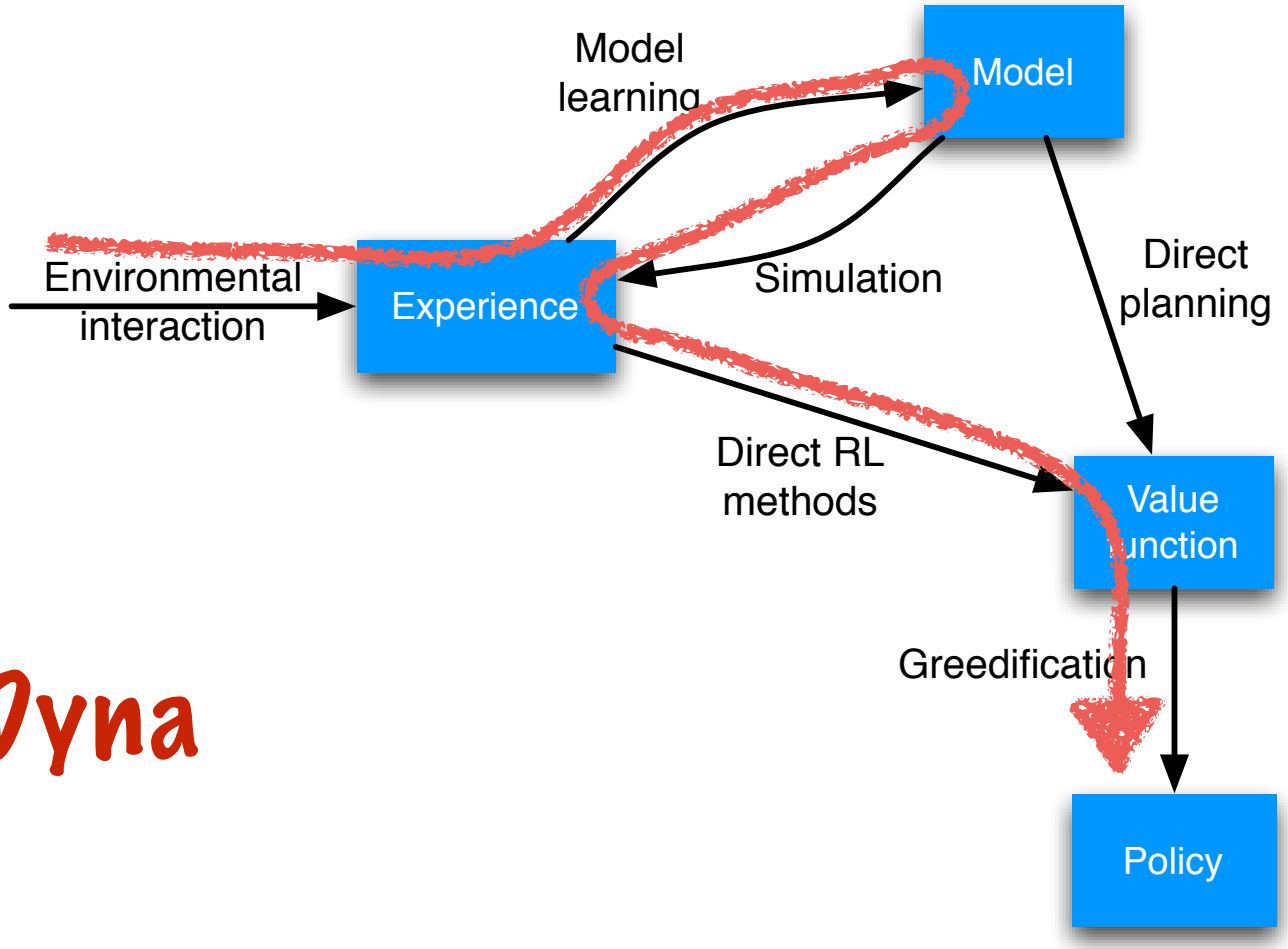
Model-free RL

Paths to a policy



Model-based RL

Paths to a policy



Dyna

Overview Tutorial

Part 1: Reinforcement Learning Theory

Part 2: Deep Reinforcement Learning

Part 3: Open Problems

deep RL methods

1. DQN
2. Prioritized Sweeping
3. A3C
4. AlphaGo Zero

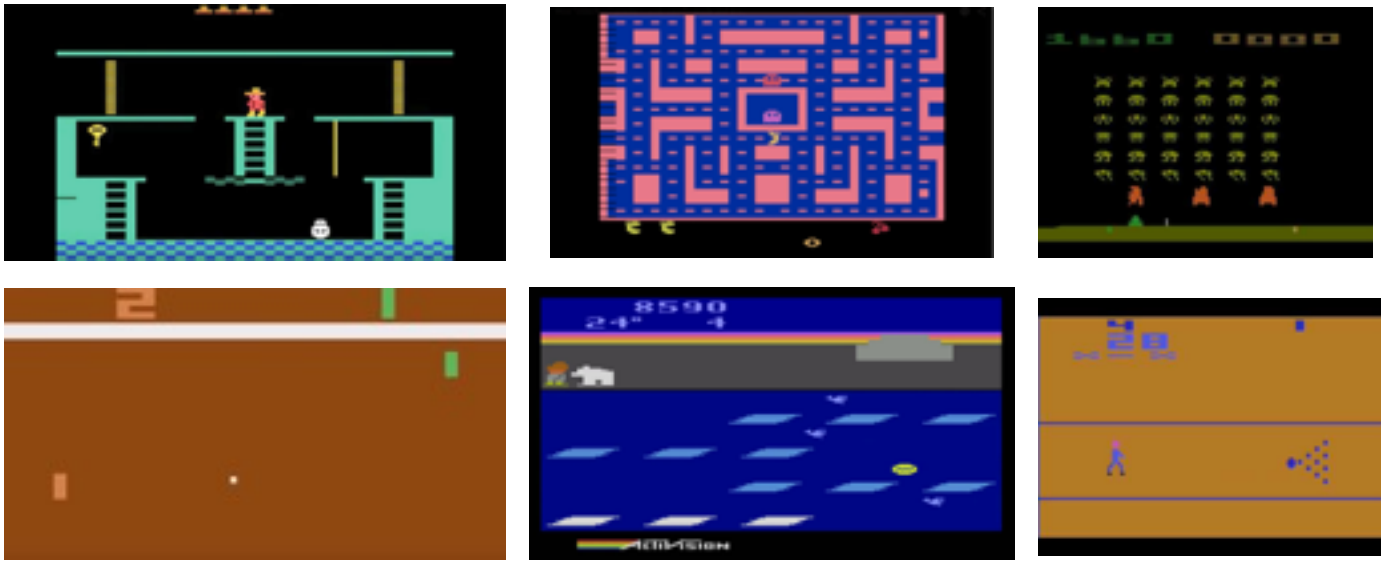
DQN

Deep Q-networks (DQN)

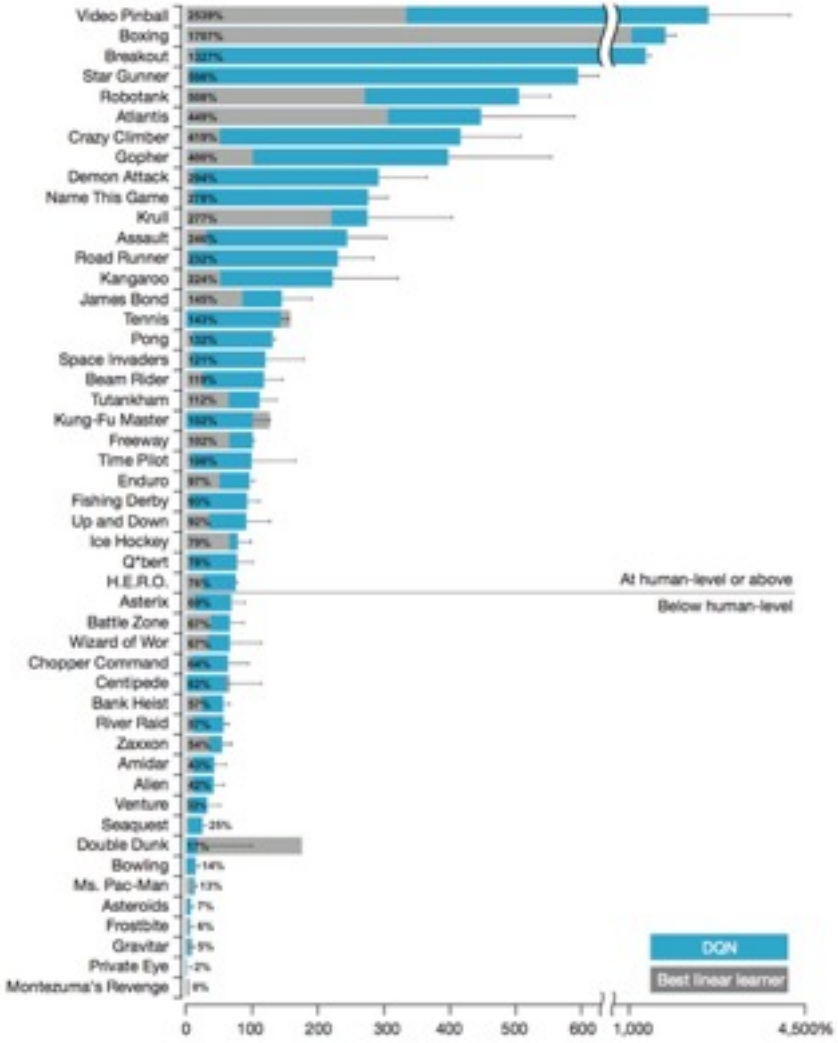
- Nature paper published in February 2015.
- The algorithm achieved above-human performance on a large number of Atari 2600 games
- Combines standard RL techniques with convolutional neural networks

Arcade Learning Environment (ALE)

- A software framework for testing AI algorithms by providing an interface to Atari 2600 games, introduced in June 2013
- Motivation behind ALE was to provide an uniform evaluation platform for building general agents
- For evaluation, a fixed set of 49 (later: 57) games is considered



Performance DQN



from: Human-level control through deep reinforcement learning, V. Mnih, et al., 2015

Main contribution

- DQN was not the first method that combines RL with neural networks (c.f. TD-Gammon, 1992).
- DQN was also not the first method that achieved good performance on the ALE set (c.f. linear methods)
- However, DQN was the first method that successfully combined modern deep neural networks with RL, resulting in a huge jump in performance on the ALE set

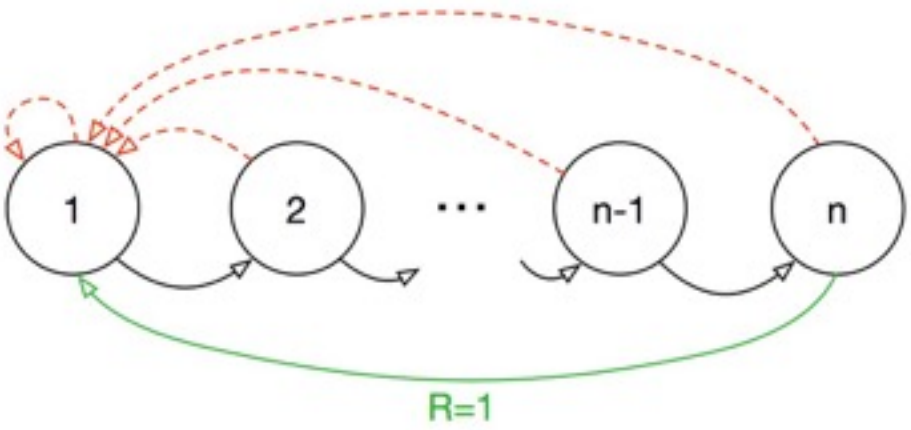
DQN Algorithm Breakdown

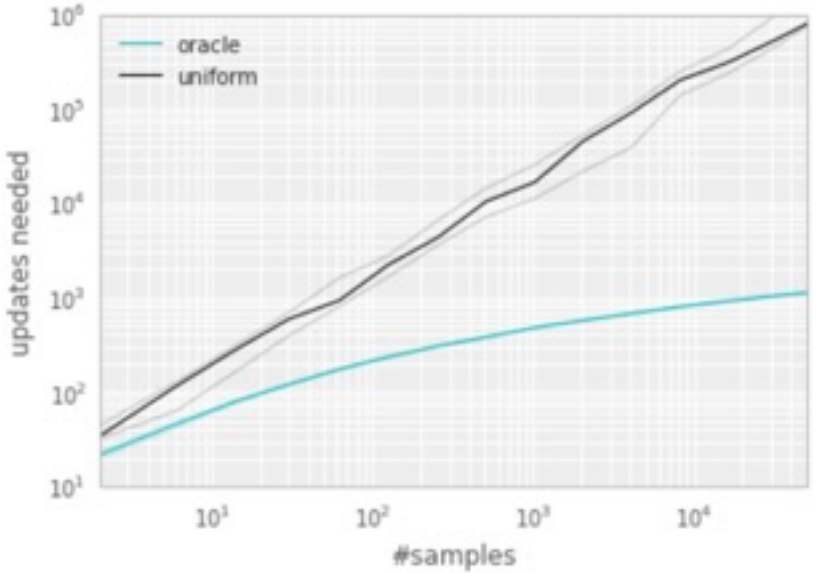
- Standard Q-learning method with two main modifications
- Modification 1: large experience replay buffer to mitigate the effect of correlated samples and catastrophic forgetting
- Modification 2: two networks to stability learning:
  - a target network: used for update targets, updated at low frequency
  - an online network: used for action selection, updated at high frequency

Prioritized Experience Replay

Pririorized Experience Replay

- DQN draws experience samples from the replay memory uniformly at random
- Prioritized experience replay aims to replay important experience samples more often





from: Prioritized Experience Replay, T. Schaul et al., 2016

How to determine importance of a sample

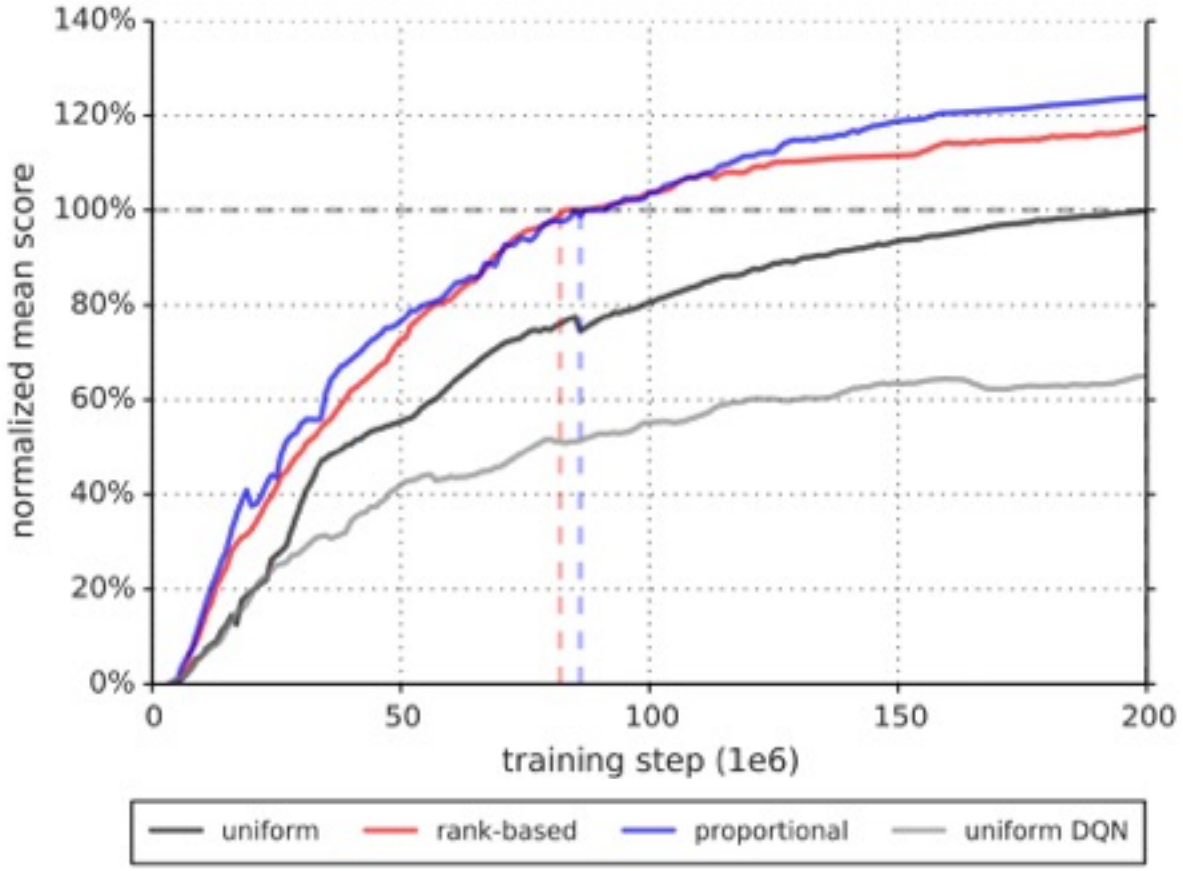
- The TD-error is a good proxy:

$$\delta_t \quad := \quad R_t + \gamma_t \max_a Q(S_t, a) - Q(S_{t-1}, A_{t-1})$$

- Greedy sample selection based on only TD error has issues:
  - samples that initially have low TD error never get selected
  - in the presence of noisy rewards, TD-error is not a good indicator of importance

- Prioritized experience replay uses a mix between pure greedy prioritization and uniform random sampling

Performance across ALE set



from: Prioritized Experience Replay, T. Schaul et al., 2016

A3C

A3C

- **A**synchronous **A**dvantage **A**ctor-**C**ritic
- Multiple parallel learners that—asynchronously— update the same value function/policy
- Because there are multiple parallel learners, samples are de-correlated even without experience replay
- Without experience replay buffer, multi-step updates can be used.

AlphaZero

Background

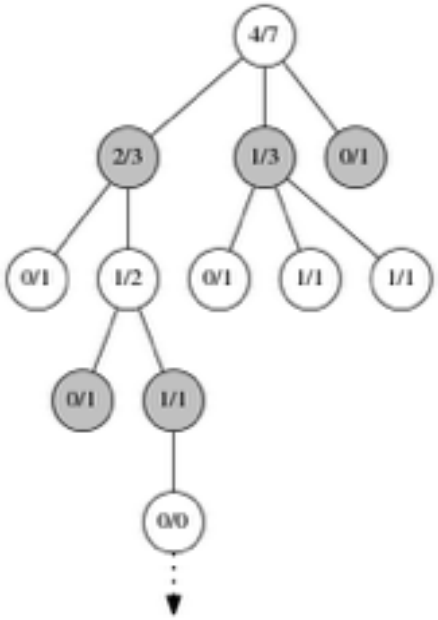
- After IBM's Deep Blue defeated the world champion in Chess, Garry Kasparov in 1997, researchers started focusing on the ancient game Go
- Go is much harder than Chess, among others due to its gigantic state-space size and large branching factor

AlphaGo Zero

- In January 2016, DeepMind published Nature paper on AlphaGo, a method that was able to beat defeat 18-time world champion, Lee Sedol
- AlphaGo was pre-trained on a large database of human Go games; deep RL was used to further improve performance
- In October 2017, DeepMind published a second Nature paper, introducing AlphaGo Zero
- AlphaGo Zero did no longer rely on a database of human Go games, but was trained fully from scratch using RL
- AlphaGo Zero also no longer makes use of some hand-engineered features, but only uses stone-positions as input

Key Component: Monte-Carlo Tree Search

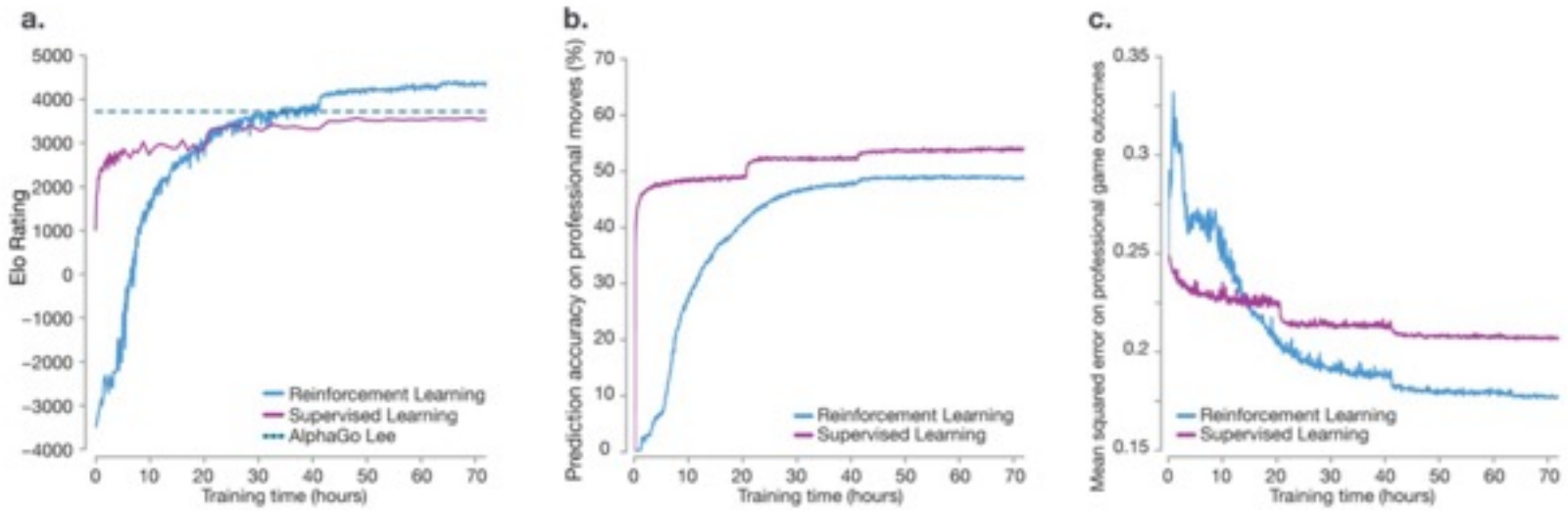
- Heuristic search algorithm that uses many random rollout to the end of the game to determine how good a particular move is
- by focussing rollouts on moves that appear to be more successful, the efficiency is improved significantly
- From the current state, a tree is build up that stores intermediate evaluations and guides which moves is evaluated next



AlphaGo Zero

- AlphaGo Zero uses MCTS, but instead of using random rollouts, it relies on deep value network
- Because no random roll-outs are performed it is much more computationally efficient

Results



from: Mastering the Game of Go without Human Knowledge, D. Silver et al. Nature '17

Main Conclusions

- AlphaGo Zero unambiguously shows the power of making value predictions using (very) deep neural networks
- The results also highlight the power of learning through self-play
- Self-play is believed to be so effective, because it is a natural form of *curriculum learning*

curriculum learning: providing a learning agent with increasingly challenging tasks in order to speed up learning of complex behaviour

Some limitations

- The environment dynamics (i.e., the rules of the game) are not learned, but provided
- Hence, the problem AlphaGo Zero considers is very different from the standard RL setting
- Because AlphaGo Zero relies on MCTS, it cannot be easily be adapted to domains where the environment dynamics are unknown (like ALE for example)

Remaining Challenges / Open Issues

1. Partial Observability
2. Long Decision Horizons
3. Exploration
4. Fast learning / adaptation
5. Limitations of Reward Function

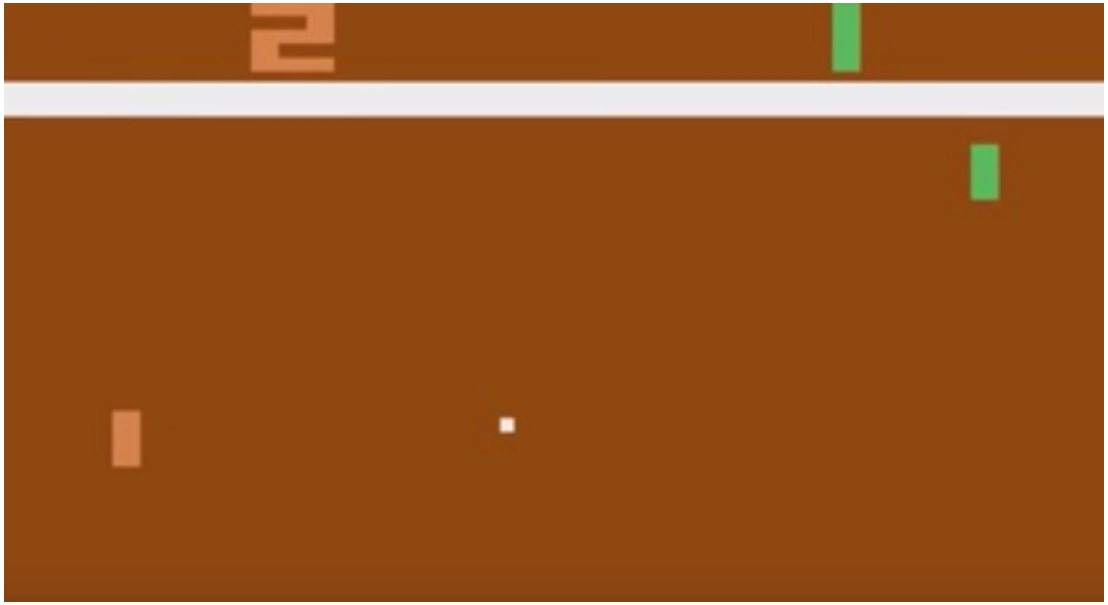
Partial Observability

Partial Observability

- Typically, the environment state is not directly observed
- Instead, a feature vector is observed that is correlated with the state, but does not fully disambiguate between different states
- If the state ambiguity is significant, it becomes hard to make accurate predictions about the return, and hence determine a good policy.

Partial Observability: Example 1

Is Pong partial observable?



reward:

- +1 if controllable agents scores point
- -1 if opponent scores
- whichever player reaches 21 points first wins

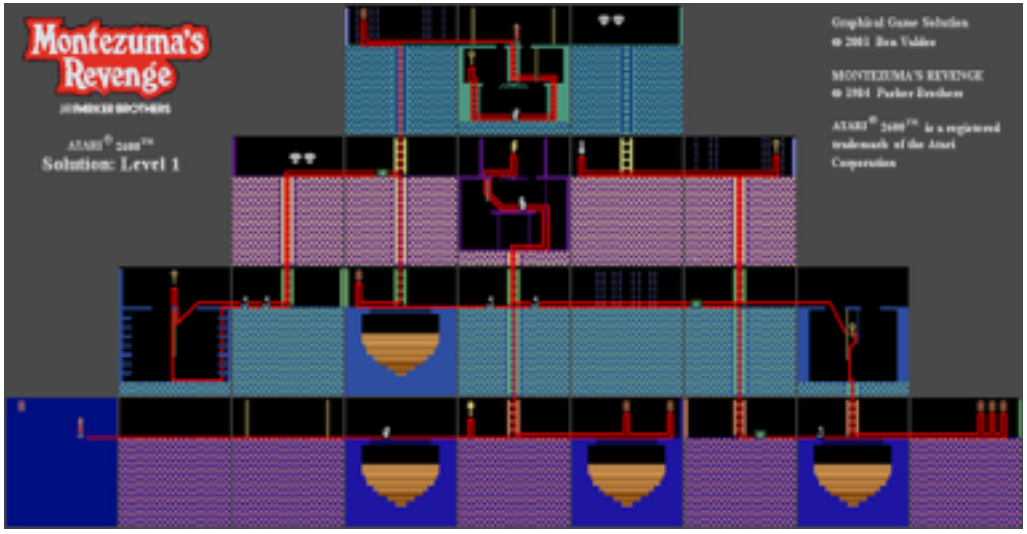
Partial Observability: Example 1

How about now?

reward:

- +1 if controllable agents scores point
- -1 if opponent scores
- whichever player reaches 21 points first wins

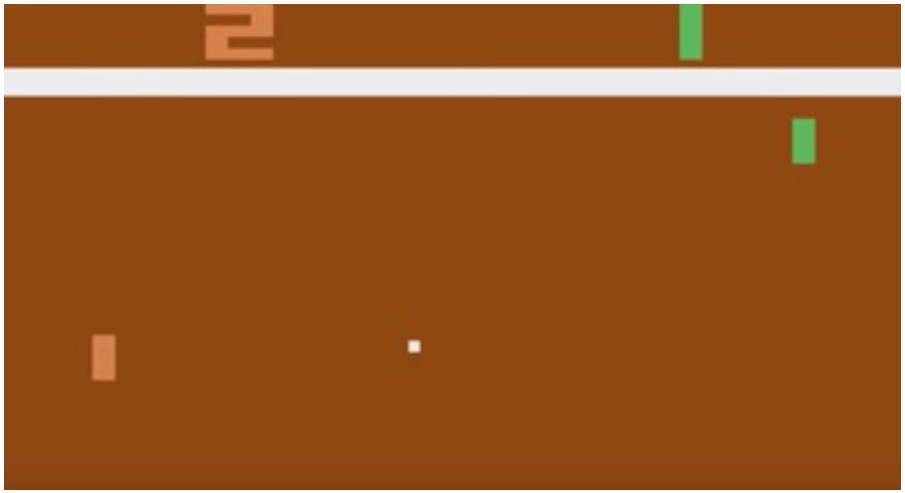
Partial Observability: Example 2



Long Decision Horizon

Decision Horizon

- The decision horizon determines how far an agent has to look into the future to determine the optimal action
- How far the agent looks into the future is controlled by the discount factor
- Often, even when the aim is to optimize the reward over many time steps, the decision horizon is much smaller and hence a smaller discount factor can be used



Long Decision Horizon

Long Decision Horizon

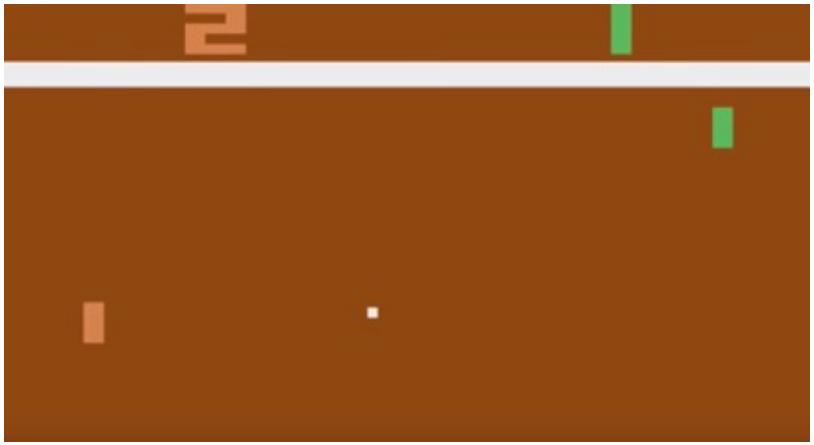
Long decision horizons are hard, because:

- It requires information to be propagated back in time over a long distance
- Can result in more challenging function approximation

Long Decision Horizon

What features are relevant for making accurate predictions:

- small discount factor:
  - values: between -1 and 1
  - relevant features: peddle-positions, ball position + direction
- large discount factor
  - values: between -21 and 21
  - relevant features: peddle-positions, ball position + direction, score

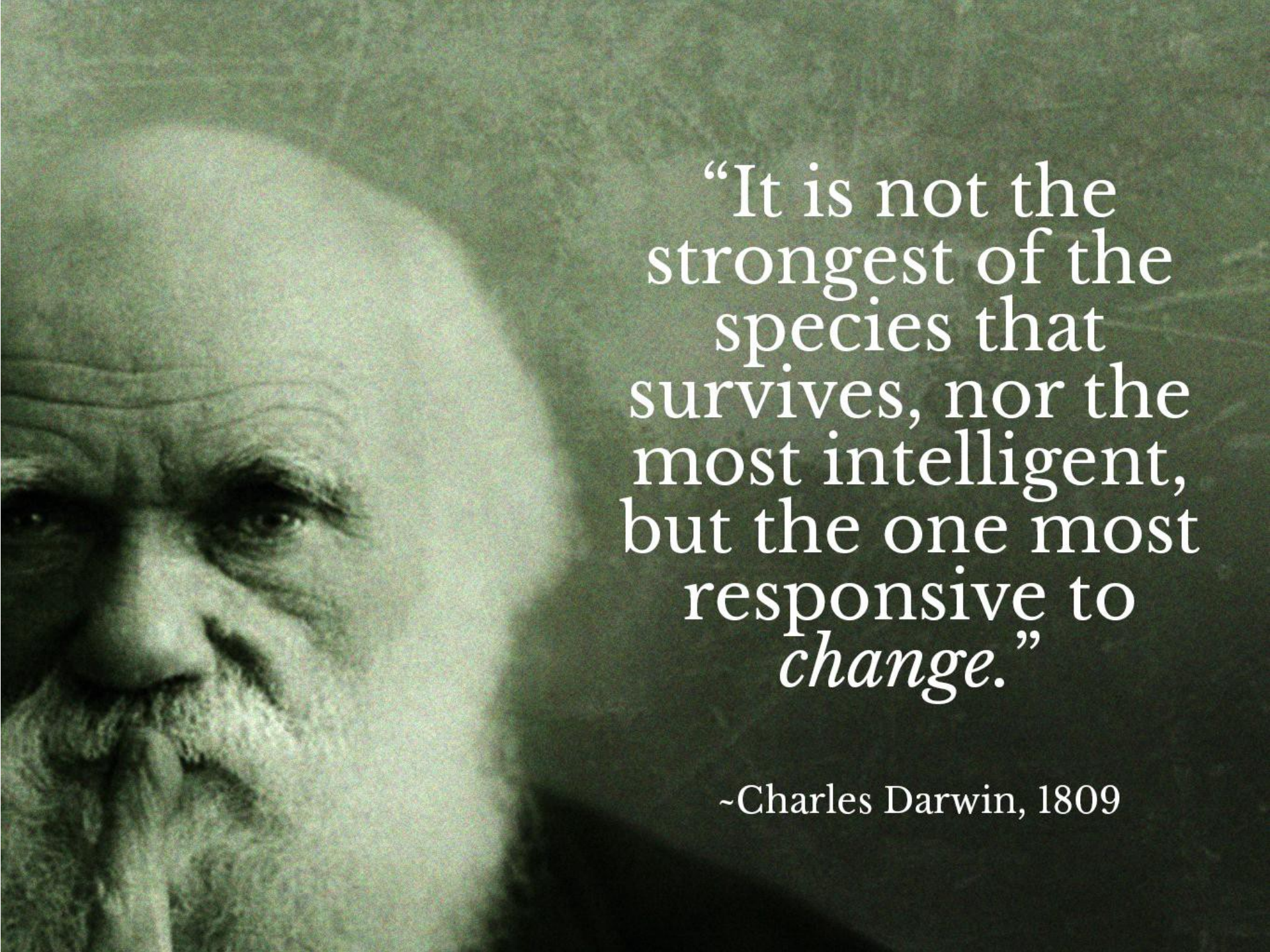


Exploration

Exploration

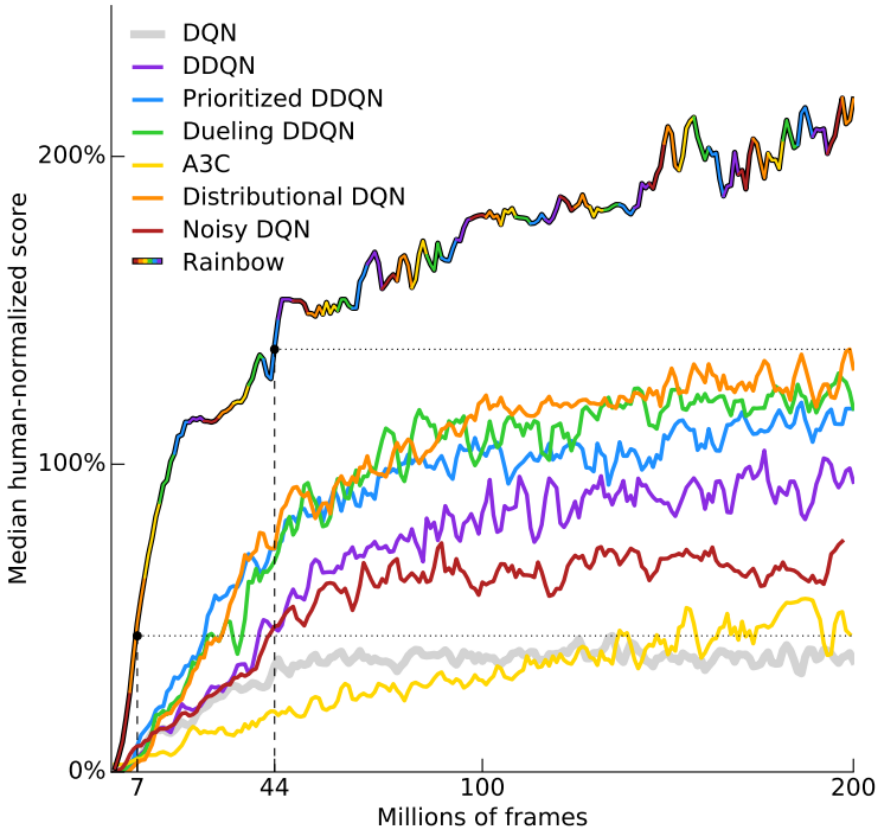
- Rewards drive policy improvement
- When the chance of observing a reward under a random policy is low, it becomes hard to make policy improvements
- This can happen, for example, when rewards are sparse

Fast learning / adaptation


“It is not the strongest of the species that survives, nor the most intelligent, but the one most responsive to *change*.”
~Charles Darwin, 1809

Current methods require huge amount of data

average training time per game:



100% median human-normalized score:
15 million frames ~ 27 human-play days*

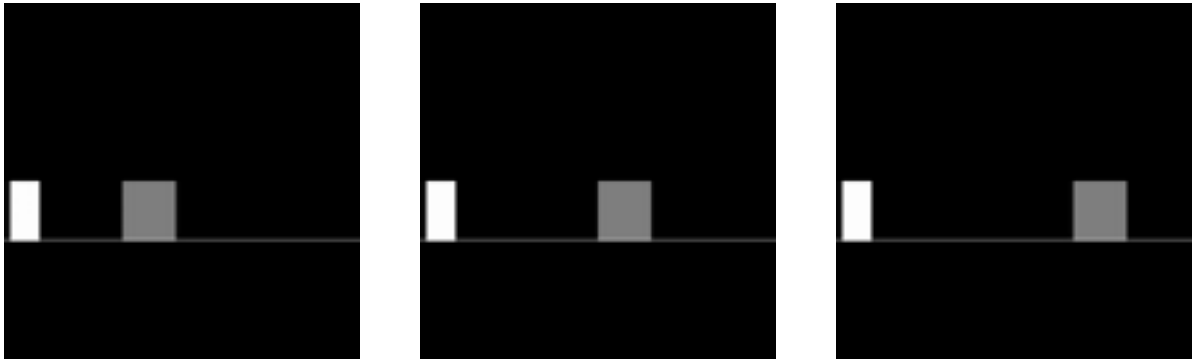
Full training period:
200 million frames ~ 370 human-play days*

* assuming:
- framerate: 60 frames/s
- skip-frame: 4
- human plays 10 hours per day

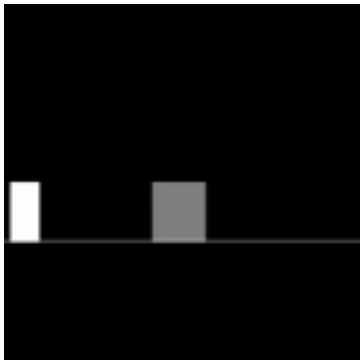
from: Rainbow: Combining Improvements in Deep Reinforcement Learning. M Hessel, et al. 2017

Adaptation to Small Task Changes is Hard

Training set:



Test set:



from: Learning invariances for Policy Generalization. R. Tachet des Combes et al., 2018

Limitations of Reward Function

Limitations Reward Function

- The reward function specifies how we want our agents to behave
- However, not all behavioural goals can be easily captured by a reward function, for example, having a meaningful conversation
- Furthermore, in a complex world, even specifying a seemingly well-defined task like ‘get a cup of coffee’ can be challenging

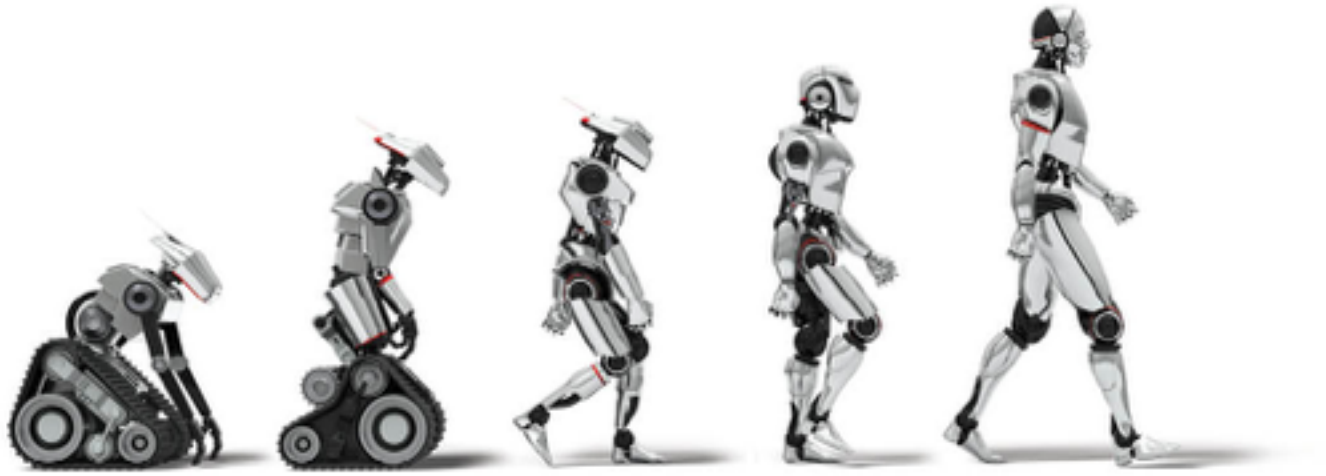
How far are we?

After 3 decades of RL research…

easy

hard

…. but progress is non-linear



Thank you!