Introduction to Reinforcement Learning
Part 1: Prediction, Value-Based, Model-free, Control (including DQN)

Doina Precup
McGill University/Mila and DeepMind Montreal
dprecup@cs.mcgill.ca

With thanks to Rich Sutton, Reasoning & Learning Lab at McGill/Mila, DeepMind

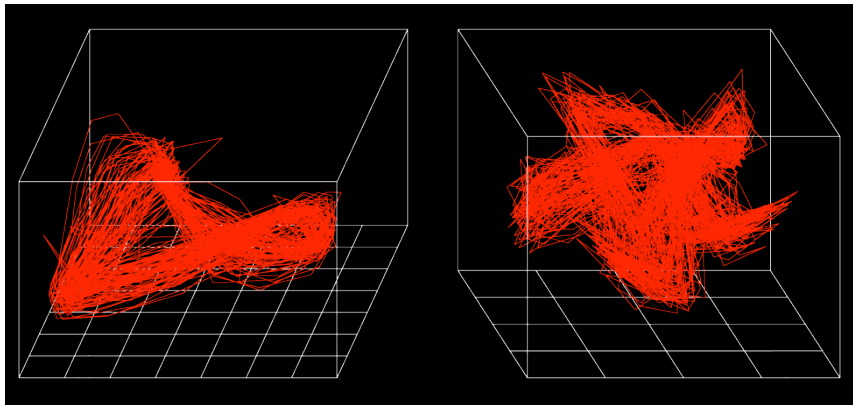
Based on Sutton & Barto, Reinforcement Learning: An Introduction (2nd ed), 2019

Supervised Learning

Given: Input data and desired output
Eg: Images and parts of interest

Goal: find a function that can be used for new inputs, and that matches the provided examples

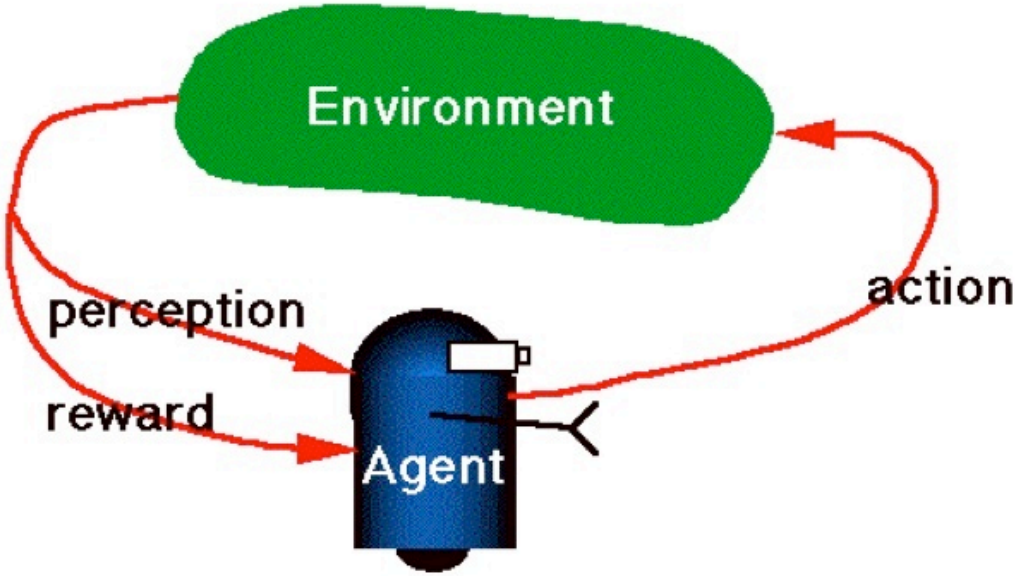
Unsupervised Learning



Given: Just data!
Eg: accelerometer information from a mobile phone

Goal: find “interesting” patterns
Often there is no single correct answer

Reinforcement Learning

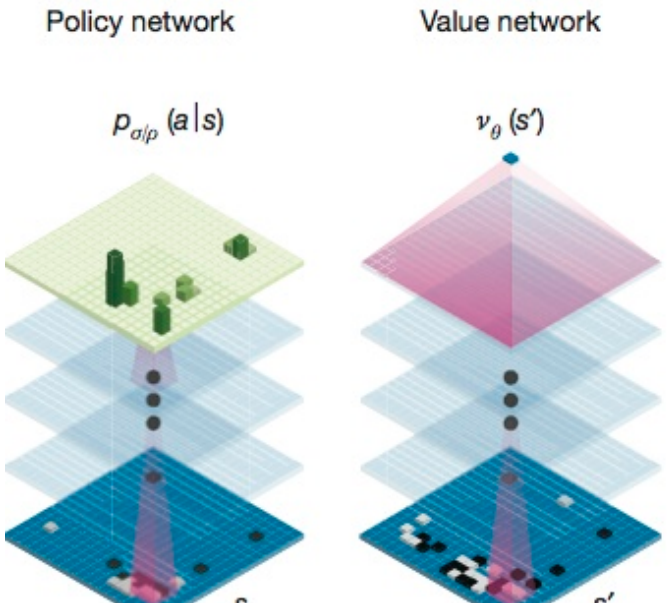


Reward: Food or shock

Reward: Positive and negative numbers

- Learning by trial-and-error
- Reward is often delayed

Example: AlphaGo & AlphaZero



- Perceptions: state of the board
- Actions: legal moves
- Reward: +1 or -1 at the end of the game
- Trained by playing games against itself
- Invented new ways of playing which seem superior

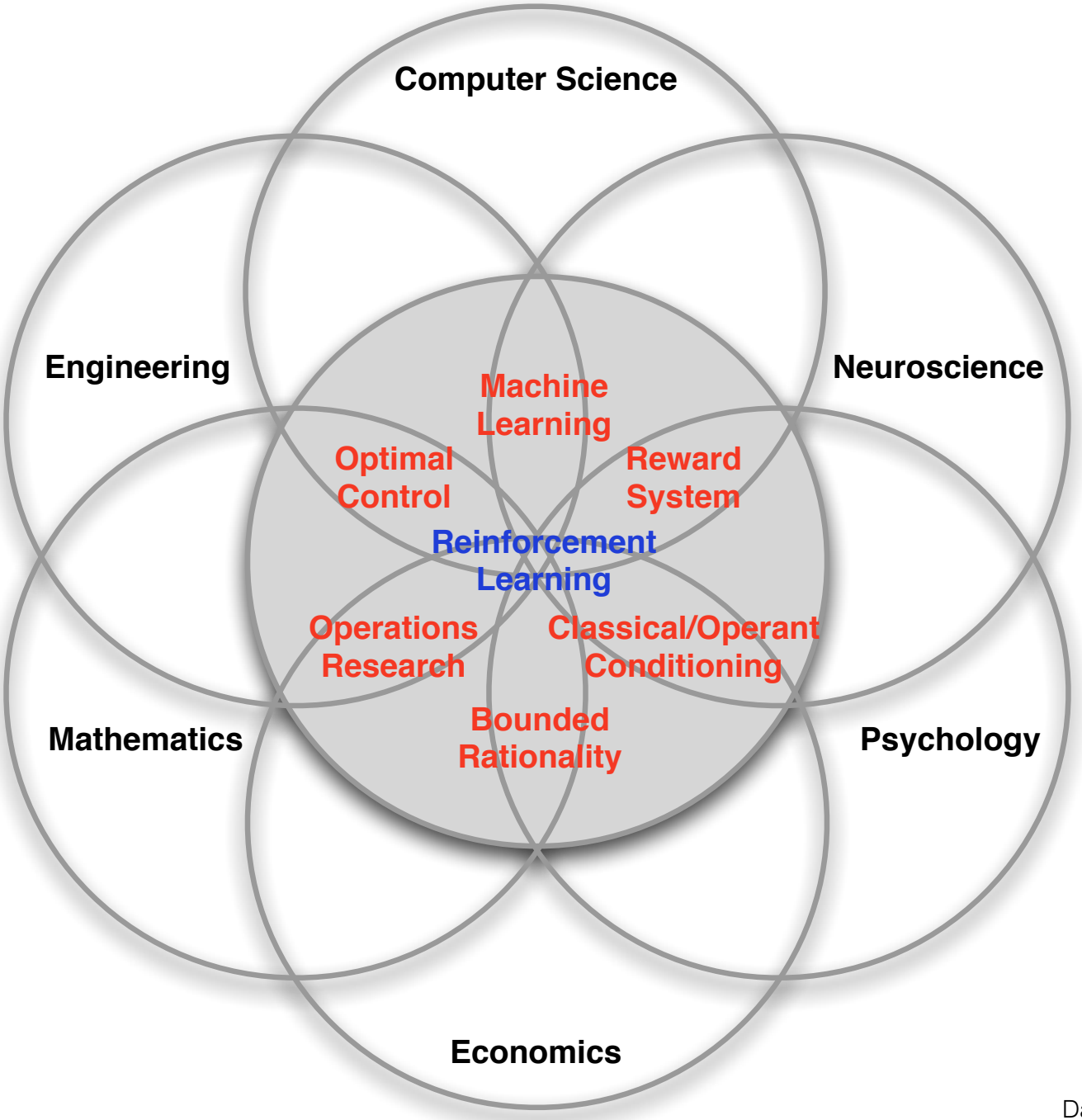
Key Features of RL

- The learner is not told what actions to take, instead it find finds out what to do by *trial-and-error search*

  Eg. Players trained by playing thousands of simulated games, with no expert input on what are good or bad moves
- The environment is *stochastic*
- The *reward may be delayed*, so the learner may need to sacrifice short-term gains for greater long-term gains

  Eg. Player might get reward only at the end of the game, and needs to assign credit to moves along the way
- The learner has to balance the need to *explore* its environment and the need to *exploit* its current knowledge

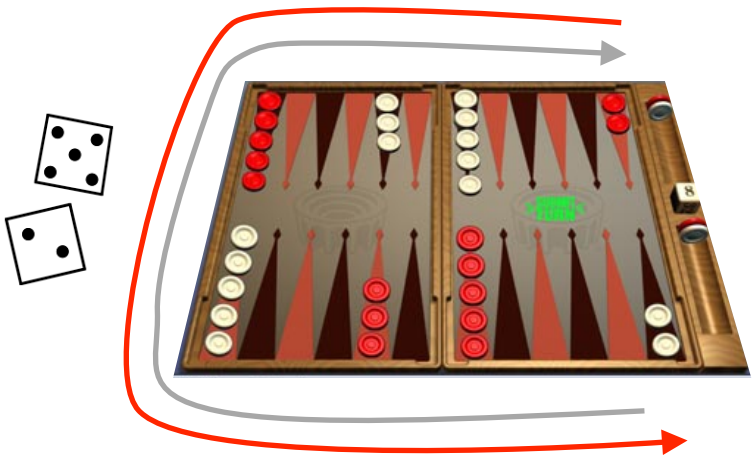
  Eg. One has to try new strategies but also to win games

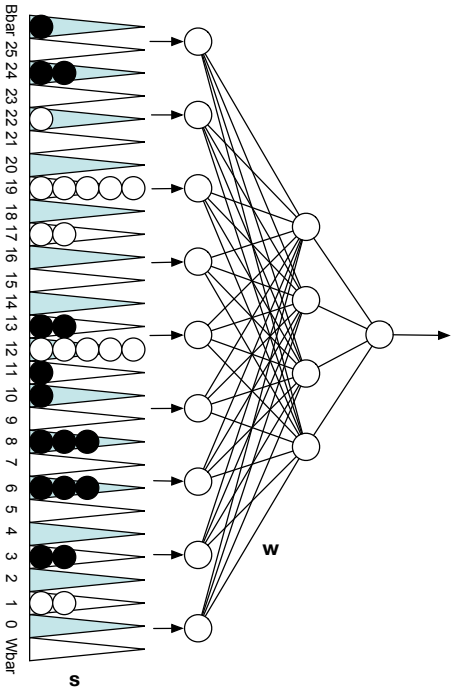


David Silver 2015

Example: TD-Gammon

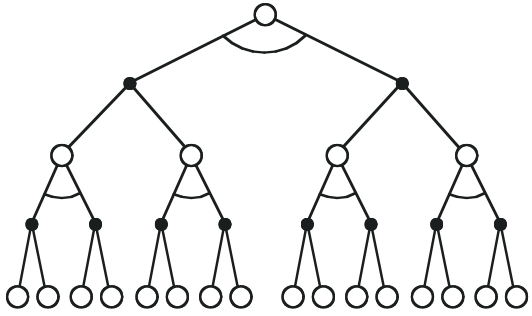
Tesauro, 1992-1995





estimated state value
(≈ prob of winning)

Action selection
by a shallow search



Start with a random Network

Play millions of games against itself

Learn a value function from this simulated experience

Six weeks later it's the best player of backgammon in the world

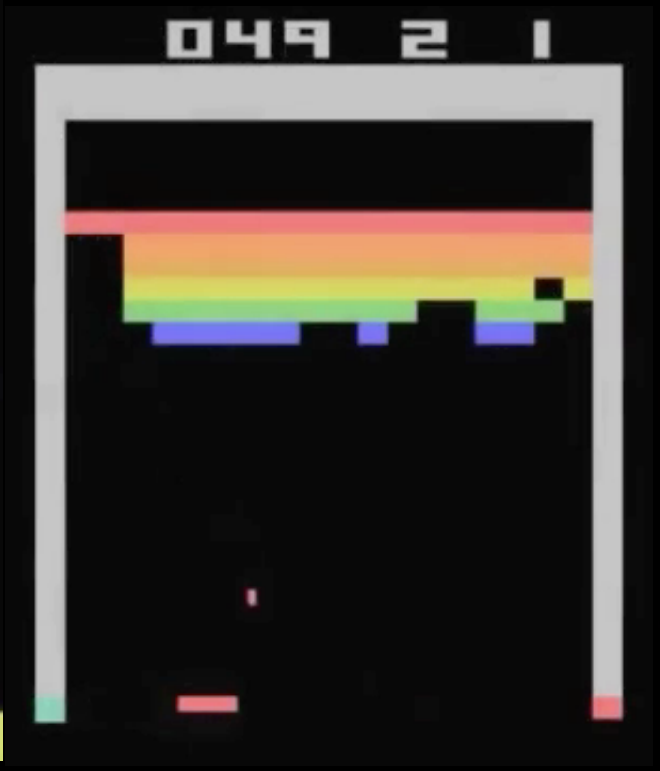
Originally used expert handcrafted features, later repeated with raw board positions

Some RL Successes

- Learned the world's best player of Backgammon (Tesauro 1995)
- Learned acrobatic helicopter autopilots (Ng, Abbeel, Coates et al 2006+)
- Widely used in the placement and selection of advertisements and pages on the web (e.g., A-B tests)
- Used to make strategic decisions in *Jeopardy!* (IBM's Watson 2011)
- Achieved human-level performance on Atari games from pixel-level visual input, in conjunction with deep learning (Google Deepmind 2015)
- In all these cases, performance was better than could be obtained by any other method, and was obtained without human instruction

RL + Deep Learing Performance on Atari Games

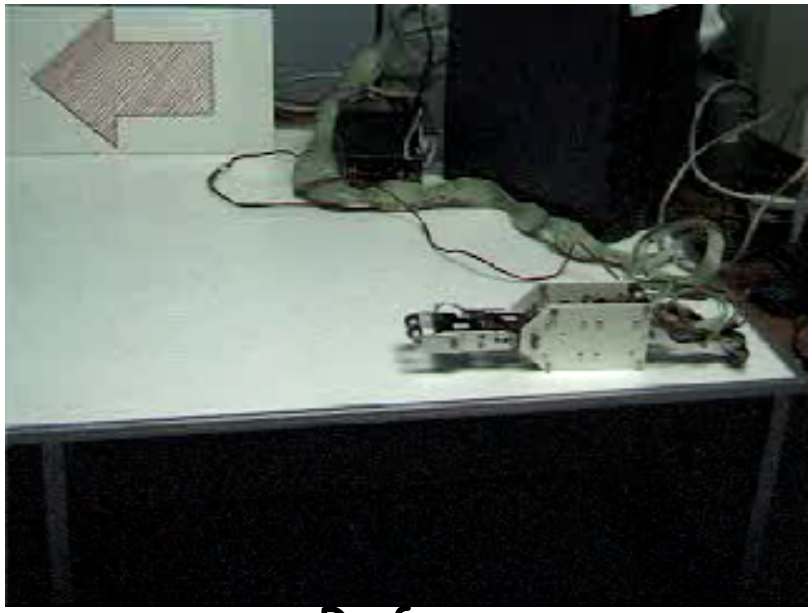
Space Invaders



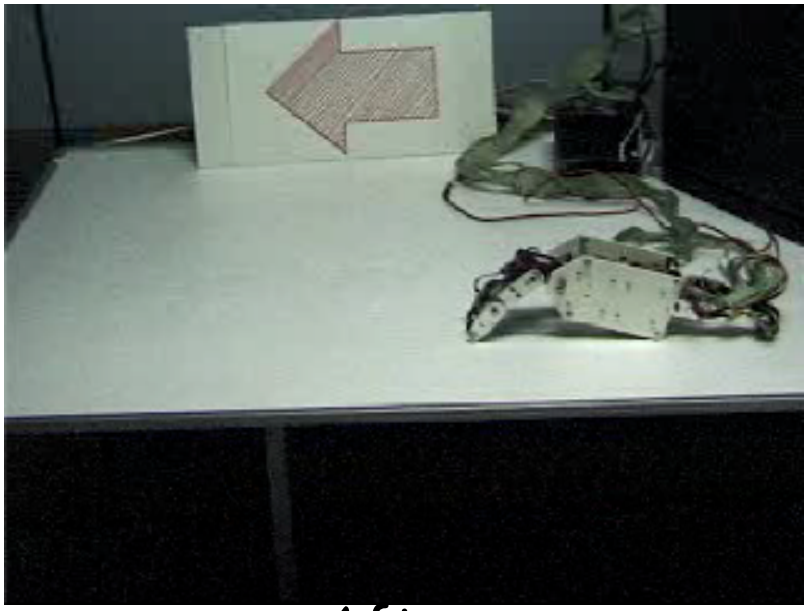
Breakout

Enduro

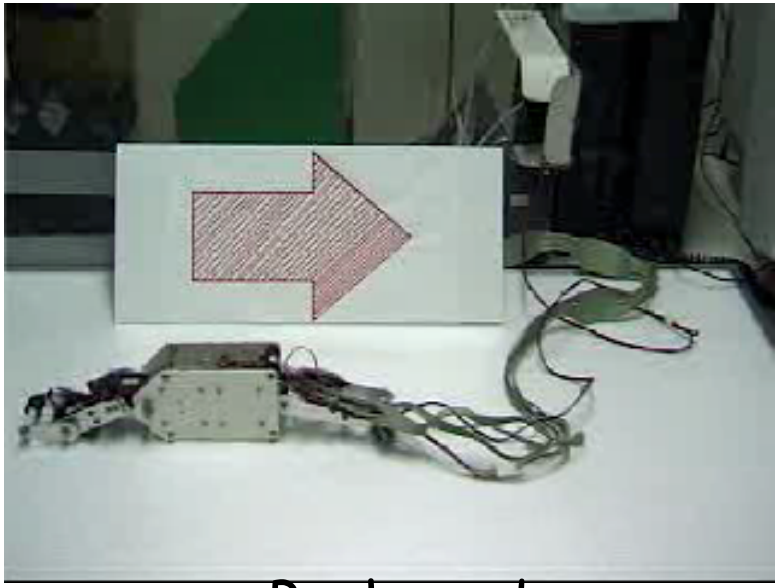
Example: Hajime Kimura's RL Robots



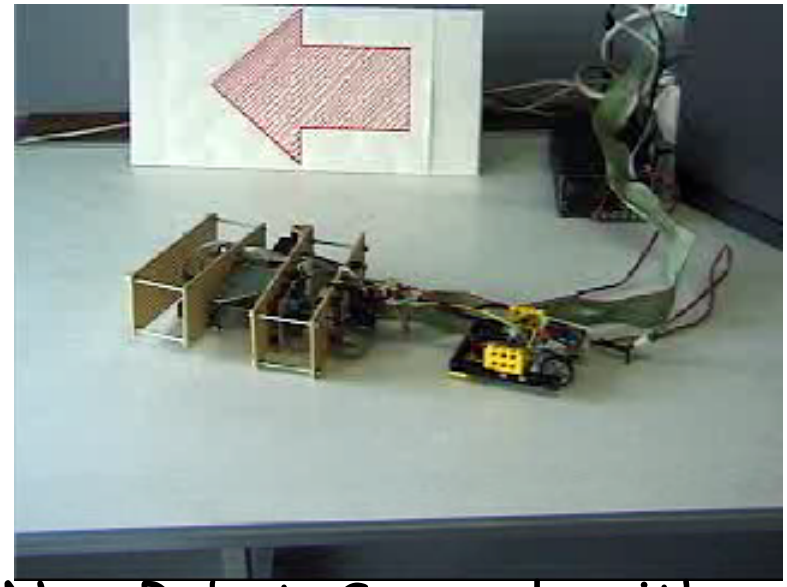
Before



After

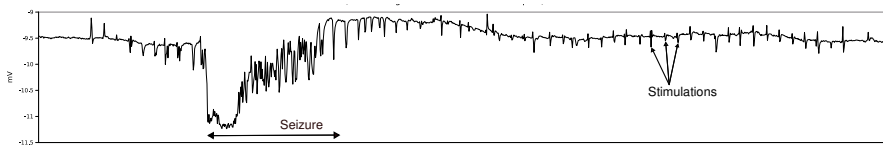


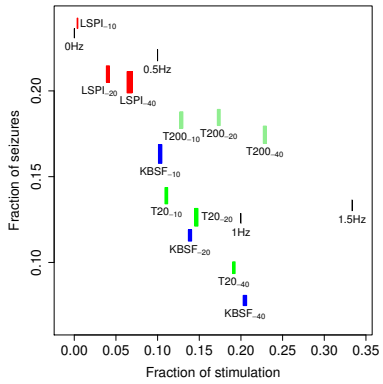
Backward



New Robot, Same algorithm

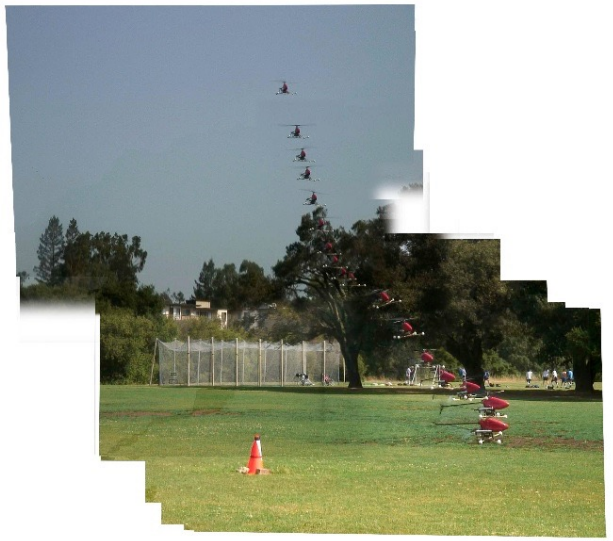
Applications: Few trajectories



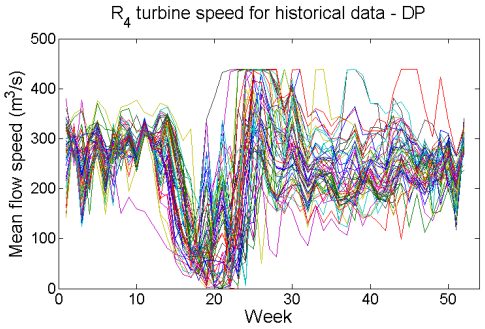


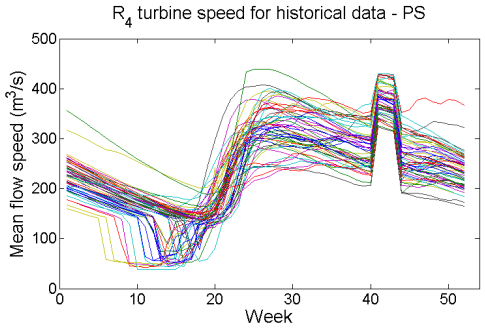
Epileptic seizure control

Our lab: Guez et al, 2008
Barreto et al, 2011, 2012



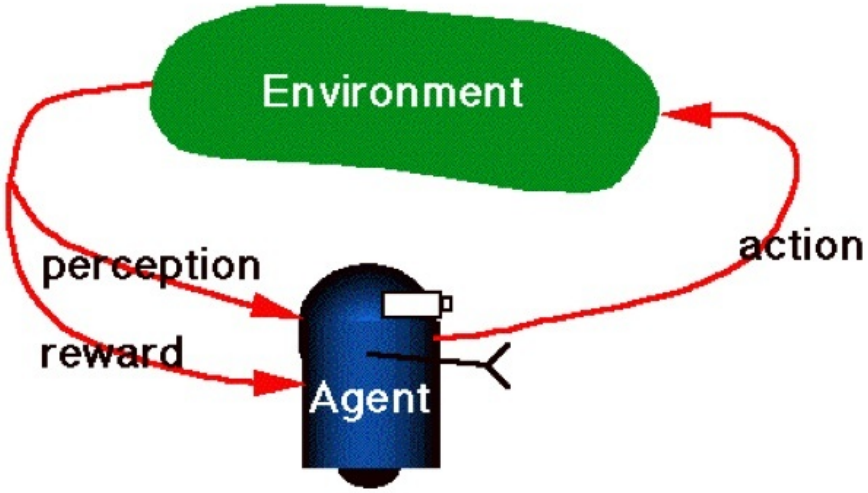
Helicopter control





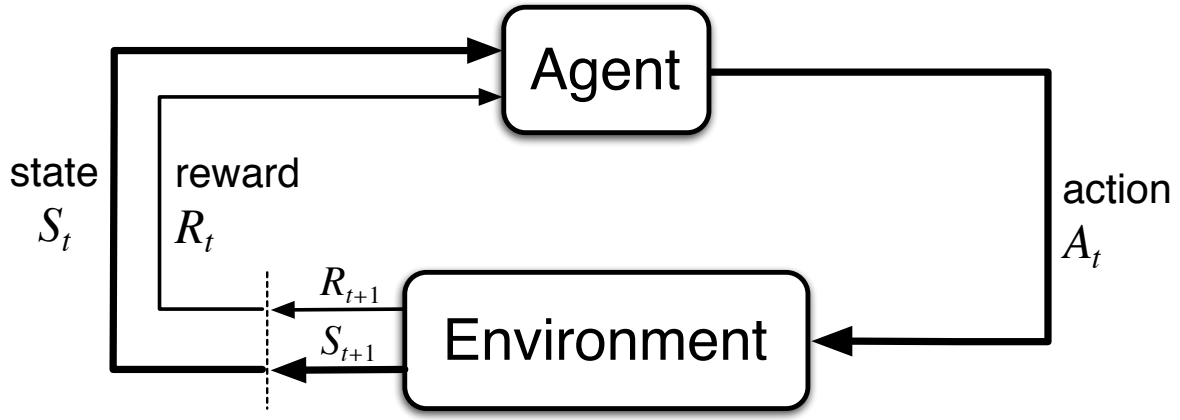
Power plant optimization
Our lab: Grinberg et al, 2014

Computational framework



- At every time step t, the agent perceives the *state* of the environment
- Based on this perception, it chooses an *action*
- The action causes the agent to receive a *numerical reward*
- Find a way of choosing actions, called a *policy* which *maximizes the agent's long-term expected return*

The Agent-Environment Interface



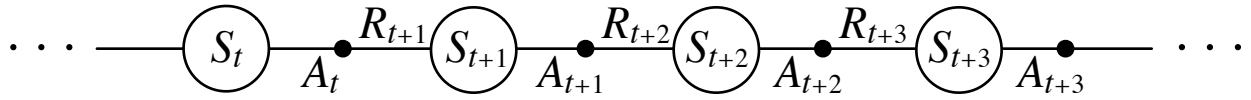
Agent and environment interact at discrete time steps: $t = 0, 1, 2, 3, \ldots$

Agent observes state at step t: $S_t \in \mathcal{S}$

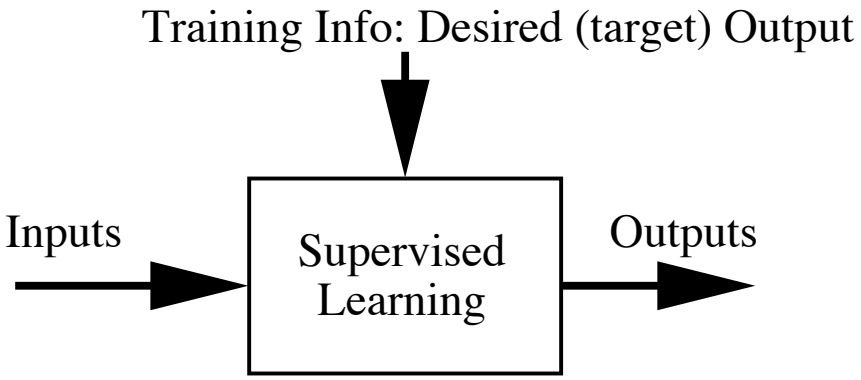
produces action at step t : $A_t \in \mathcal{A}(S_t)$

gets resulting reward: $R_{t+1} \in \mathcal{R} \subset \mathbb{R}$

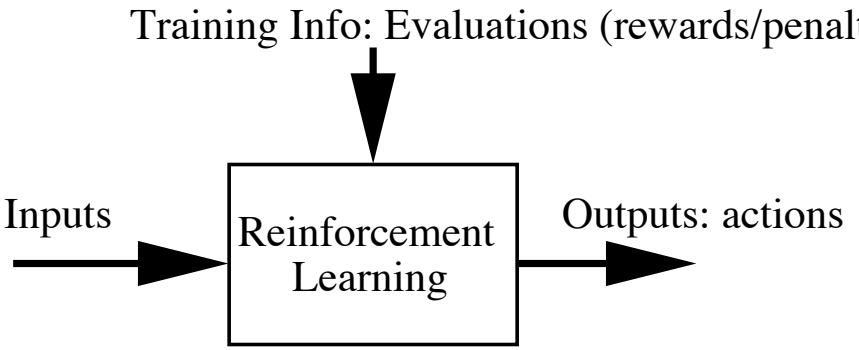
and resulting next state: $S_{t+1} \in \mathcal{S}^+$



Supervised vs Reinforcement Learning



Error = (target output - actual output)



Objective: Get as much total reward as possible

Agent's learning task

- Execute actions in environment, observe results, and learn *policy* (strategy, way of behaving) $\pi : \mathcal{S} \times \mathcal{A} \rightarrow [0, 1]$,

$$\pi(s, a) = P\left(A_t = a | S_t = s\right)$$

- The policy can be deterministic, $\pi : \mathcal{S} \rightarrow \mathcal{A}$, with $\pi(s) = a$ giving the action chosen in state s.
- Note that the target function is $\pi : \mathcal{S} \rightarrow \mathcal{A}$ but we have *no training examples* of form $\langle s, a \rangle$

  (That would be behavior cloning or imitation learning)
- In RL training examples are of form $\langle \langle s, a \rangle, r, s', \ldots \rangle$
- RL methods specify how the agent should change the policy π as a function of the rewards received over time

Return

Suppose the sequence of rewards after step t is:

$$R_{t+1}, R_{t+2}, R_{t+3}, \ldots$$

What do we want to maximize?

At least three cases, but in all of them,

we seek to maximize the **expected return**, $E\left\{G_t\right\}$, on each step t.

- Total reward, G_t = sum of all future reward in the episode
- Discounted reward, G_t = sum of all future *discounted* reward
- Average reward, G_t = average reward per time step

Episodic Tasks

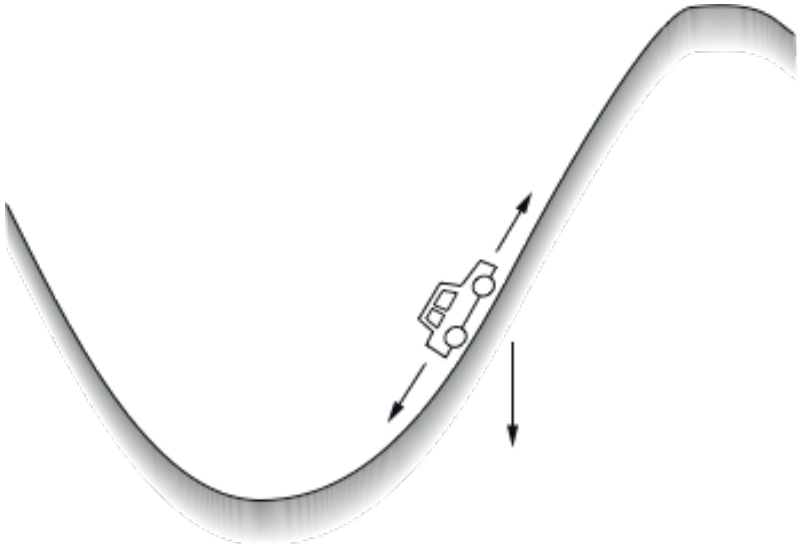
Episodic tasks: interaction breaks naturally into episodes, e.g., plays of a game, trips through a maze

In episodic tasks, we almost always use simple *total reward*:

$$G_t = R_{t+1} + R_{t+2} + \cdots + R_T ,$$

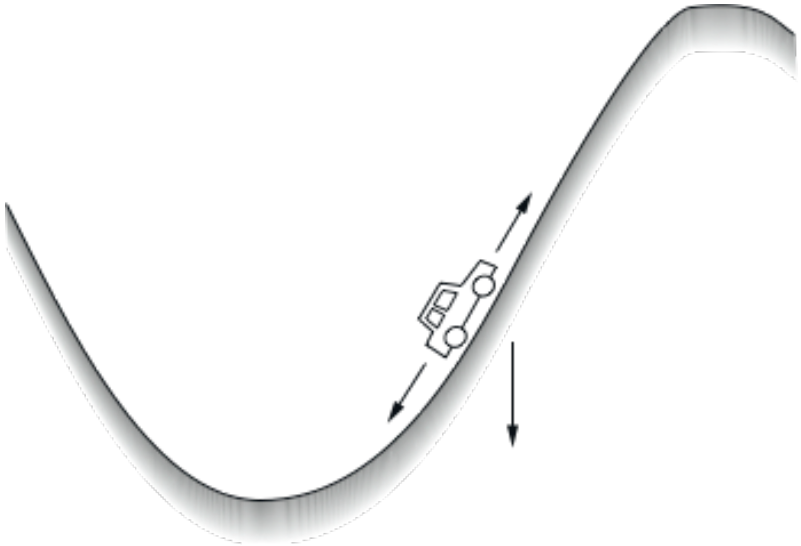
where T is a final time step at which a **terminal state** is reached, ending an episode.

Example: Mountain Car



Get to the top of the hill as quickly as possible.

Example: Mountain Car



Get to the top of the hill
as quickly as possible.

reward = –1 for each step where **not** at top of hill

$\Rightarrow$ return = – number of steps before reaching top of hill

Return is maximized by minimizing
number of steps to reach the top of the hill.

Continuing Tasks

Continuing tasks: interaction does not have natural episodes, but just goes on and on...

In this class, for continuing tasks we will always use *discounted return*:

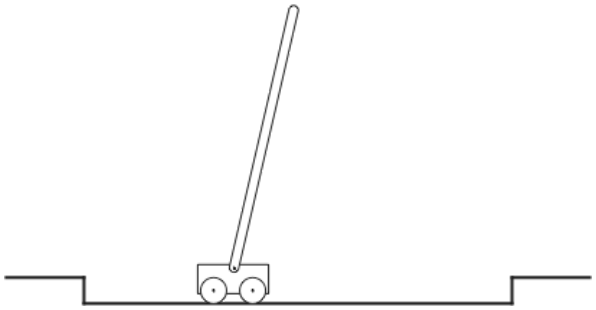
$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1},$$

where γ, $0 \leq \gamma \leq 1$, is the **discount rate**.

shortsighted $0 \leftarrow \gamma \rightarrow 1$ farsighted

Typically, $\gamma = 0.9$

Example: Pole Balancing



Avoid **failure:** the pole falling beyond a critical angle or the cart hitting end of track

As an **episodic task** where episode ends upon failure:

reward = +1 for each step before failure

$\Rightarrow$ return = number of steps before failure

As a **continuing task** with discounted return:

reward = −1 upon failure; 0 otherwise

$\Rightarrow$ return = $-\gamma^k$, for k steps before failure

In either case, return is maximized by avoiding failure for as long as possible.

4 value functions

	state values	action values
prediction	v_π	q_π
control	v_*	q_*

- All theoretical objects, mathematical ideals (expected values)

- Distinct from their estimates:

$$V_t(s) \qquad Q_t(s,a)$$

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 :

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

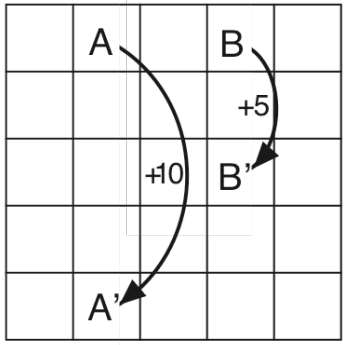
- The **value of an action (in a state)** is the expected return starting after taking that action from that state; depends on the agent's policy:

Action - value function for policy π :

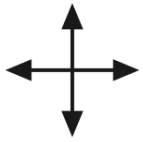
$$q_\pi(s,a) = E_\pi\left\{G_t \mid S_t = s, A_t = a\right\} = E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \mid S_t = s, A_t = a\right\}$$

Gridworld

- Actions: `north`, `south`, `east`, `west`; deterministic.
- If would take agent off the grid: no move but reward = –1
- Other actions produce reward = 0, except actions that move agent out of special states A and B as shown.



(a)



Actions

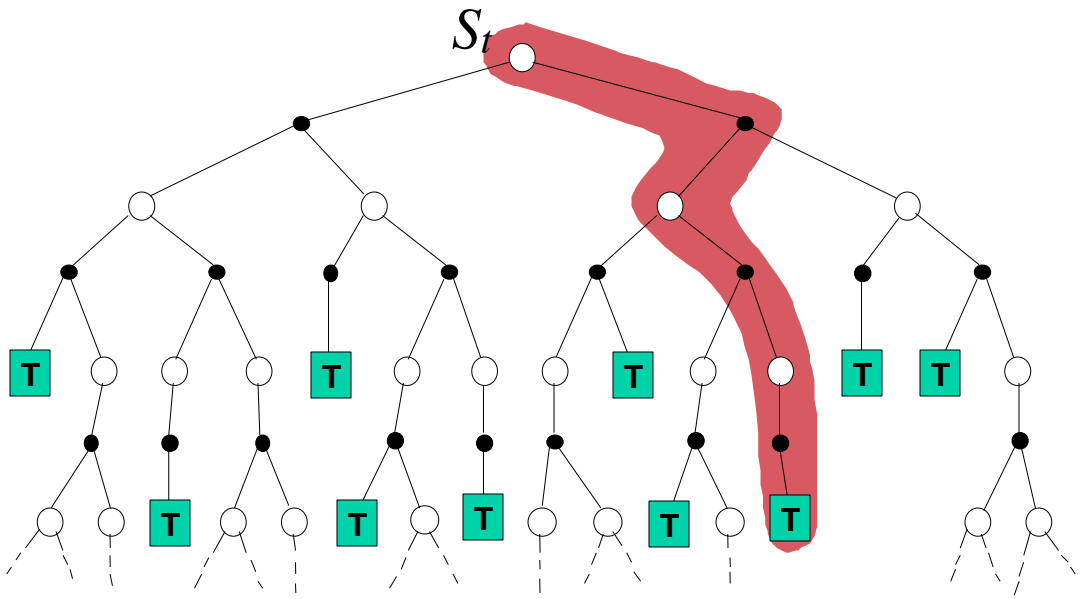
3.3	8.8	4.4	5.3	1.5
1.5	3.0	2.3	1.9	0.5
0.1	0.7	0.7	0.4	-0.4
-1.0	-0.4	-0.4	-0.6	-1.2
-1.9	-1.3	-1.2	-1.4	-2.0

(b)

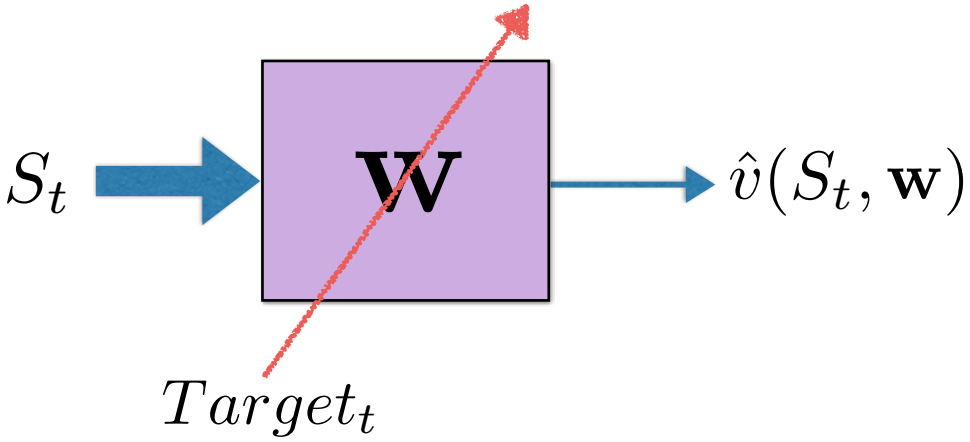
State-value function for equiprobable random policy; $\gamma = 0.9$

Simple Monte Carlo

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



Value function approximation



A natural objective in VFA
is to minimize the Mean Square Value Error

$$\text{MSVE}(\mathbf{w}) \doteq \sum_{s \in \mathcal{S}} \mu(s) \Big[v_\pi(s) - \hat{v}(s,\mathbf{w}) \Big]^2$$

where $\mu(s)$ is the fraction of time steps spent in state s

True SGD will converge to a local minimum of the error objective

In *linear* VFA, there is only one minimum: local=global

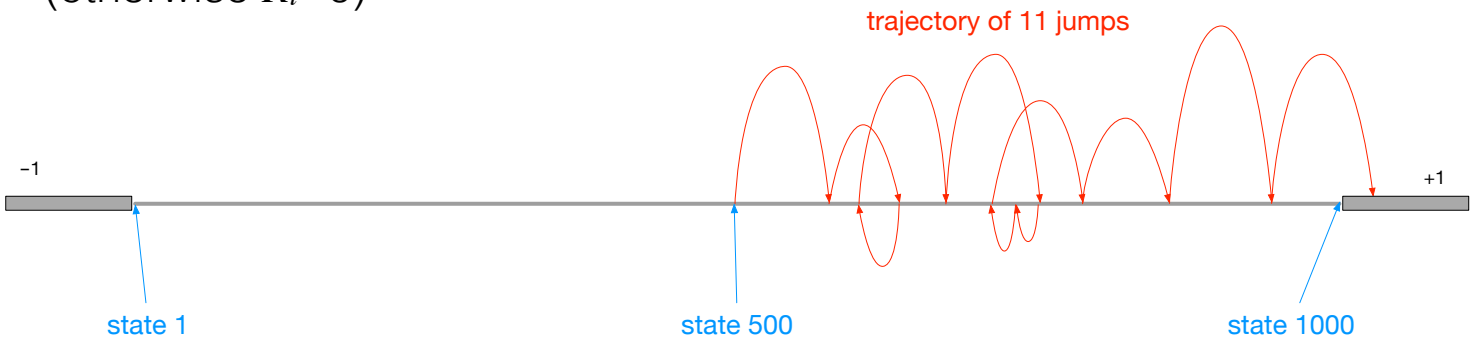
Gradient Monte Carlo Algorithm for Estimating $\hat{v} \approx v_\pi$

Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S} \times \mathbb{R}^d \to \mathbb{R}$

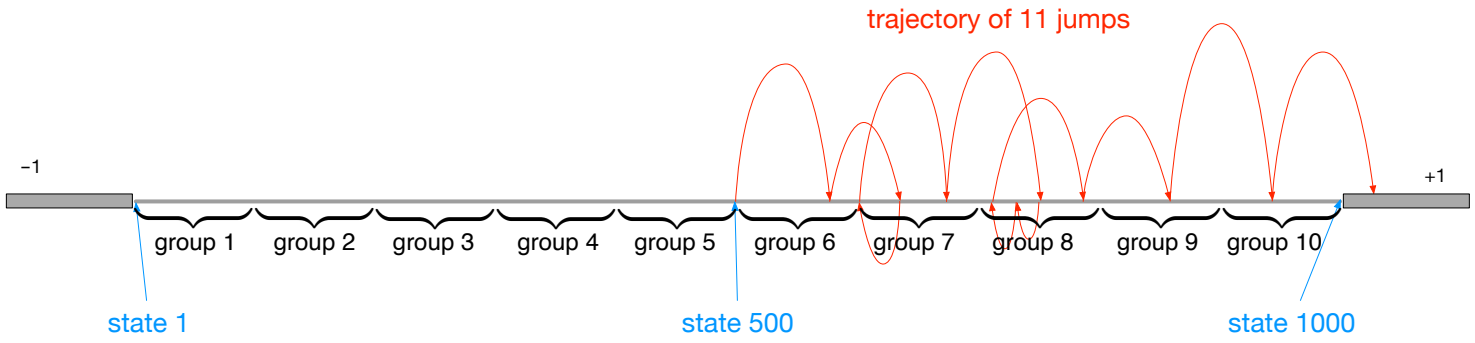
Initialize value-function weights $\mathbf{w} \in \mathbb{R}^d$ as appropriate (e.g., $\mathbf{w} = \mathbf{0}$)
Repeat forever:
    Generate an episode $S_0, A_0, R_1, S_1, A_1, \ldots, R_T, S_T$ using π
    For $t = 0, 1, \ldots, T-1$:
        $\mathbf{w} \leftarrow \mathbf{w} + \alpha\big[G_t - \hat{v}(S_t,\mathbf{w})\big]\nabla\hat{v}(S_t,\mathbf{w})$

The 1000-state random walk example

- States are numbered 1 to 1000
- Walks start in the near middle, at state 500 $S_0 = 500$
- At each step, *jump* to one of the 100 states to the right, or to one of the 100 states to the left $S_1 \in \{400..499\} \cup \{501..600\}$
- If the jump goes beyond 1 or 1000, terminates with a reward of −1 or +1 (otherwise R_t=0)



State aggregation into 10 groups of 100



The whole value function over 1000 states will be approximated with 10 numbers!

State aggregation is the simplest kind of VFA

- States are partitioned into disjoint subsets (groups)
- One component of $\mathbf{w}$ is allocated to each group

$$\hat{v}(s, \mathbf{w}) \doteq w_{group(s)}$$

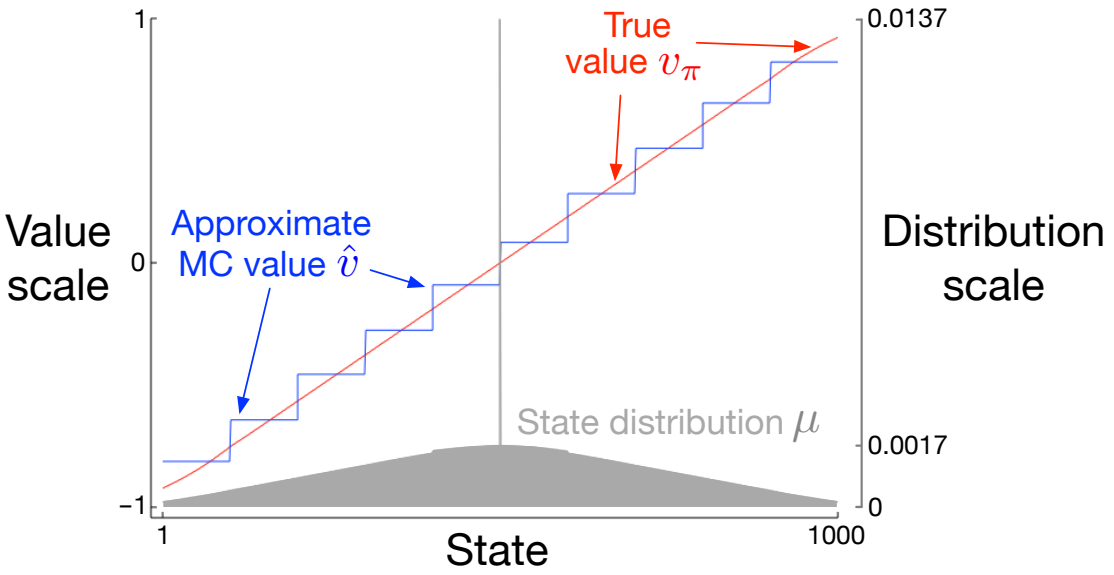
$$\nabla_{\mathbf{w}} \hat{v}(s, \mathbf{w}) \doteq [0, 0, \ldots, 0, 1, 0, 0, \ldots, 0]$$

Recall: $\mathbf{w} \leftarrow \mathbf{w} + \alpha \left[Target_t - \hat{v}(S_t, \mathbf{w}) \right] \nabla_{\mathbf{w}} \hat{v}(S_t, \mathbf{w})$

Gradient MC works well on the 1000-state random walk using *state aggregation*

- 10 groups of 100 states
- after 100,000 episodes
- α = 2 x 10^{-5}

- state distribution affects accuracy



Markov Decision Processes

- If a reinforcement learning task has the Markov Property, it is a **Markov Decision Process (MDP)**.
- If state and action sets are finite, it is a **finite MDP**.
- To define a finite MDP, you need to give:

$$p(s', r|s, a) = \mathbf{Pr}\{S_{t+1}=s', R_{t+1}=r \mid S_t=s, A_t=a\}$$

$$p(s'|s, a) \doteq \Pr\{S_{t+1}=s' \mid S_t=s, A_t=a\} = \sum_{r\in\mathcal{R}} p(s', r|s, a)$$

$$r(s, a) \doteq \mathbb{E}[R_{t+1} \mid S_t=s, A_t=a] = \sum_{r\in\mathcal{R}} r \sum_{s'\in\mathcal{S}} p(s', r|s, a)$$

Optimal Value Functions

- For finite MDPs, policies can be **partially ordered**:

$$\pi \geq \pi' \quad \text{if and only if } v_\pi(s) \geq v_{\pi'}(s) \text{ for all } s \in \mathcal{S}$$

- There are always one or more policies that are better than or equal to all the others. These are the **optimal policies**. We denote them all π_*.
- Optimal policies share the same **optimal state-value function**:

$$v_*(s) = \max_\pi v_\pi(s) \quad \text{for all } s \in \mathcal{S}$$

- Optimal policies also share the same **optimal action-value function**: $q_*(s,a) = \max_\pi q_\pi(s,a)$ for all $s \in \mathcal{S}$ and $a \in \mathcal{A}$

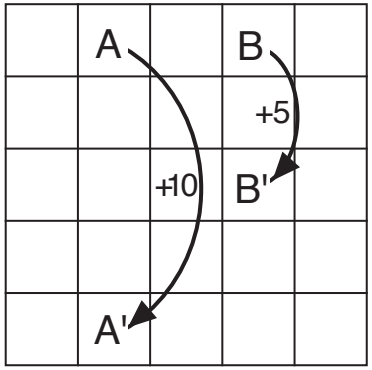
  This is the expected return for taking action a in state s and thereafter following an optimal policy.

Why Optimal State-Value Functions are Useful

Any policy that is greedy with respect to v_* is an optimal policy.

Therefore, given v_*, one-step-ahead search produces the long-term optimal actions.

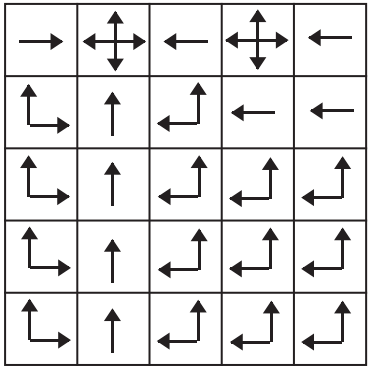
E.g., back to the gridworld:



a) gridworld

22.0	24.4	22.0	19.4	17.5
19.8	22.0	19.8	17.8	16.0
17.8	19.8	17.8	16.0	14.4
16.0	17.8	16.0	14.4	13.0
14.4	16.0	14.4	13.0	11.7

b) v_*



c) π_*

What About Optimal Action-Value Functions?

Given q_*, the agent does not even have to do a one-step-ahead search:

$$\pi_*(s) = \arg\max_a q_*(s,a)$$

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:

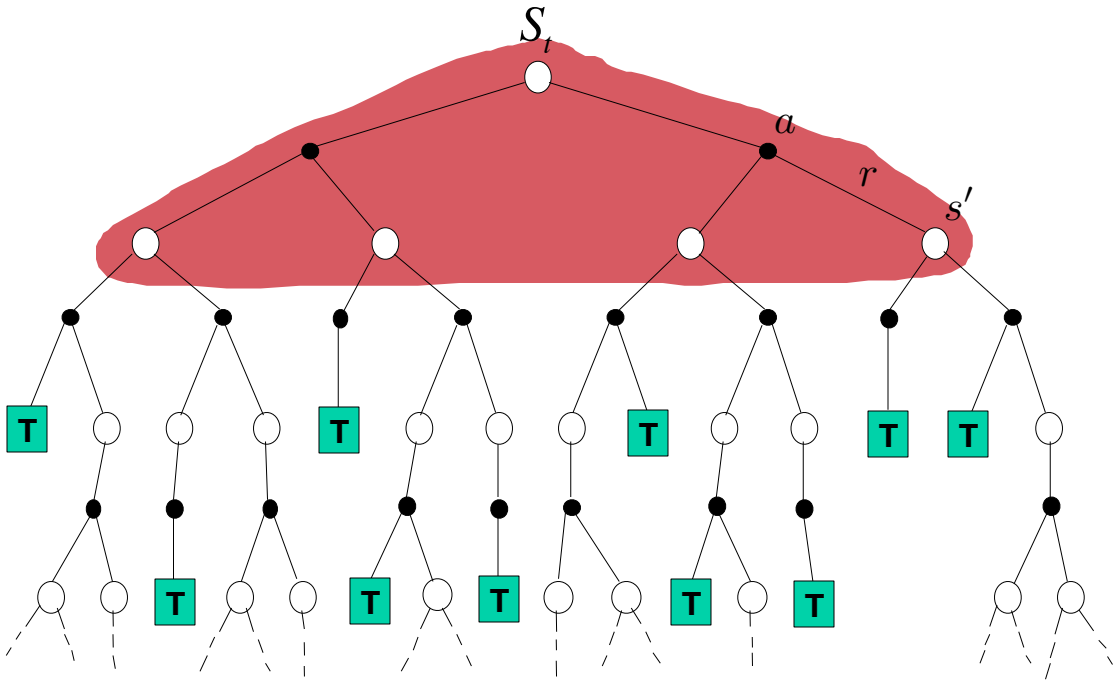
$$\begin{aligned} v_\pi(s) &= E_\pi \left\{ G_t \,\middle|\, S_t = s \right\} \\ &= E_\pi \left\{ R_{t+1} + \gamma v_\pi \left(S_{t+1} \right) \middle|\, S_t = s \right\} \end{aligned}$$

Or, without the expectation operator:

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

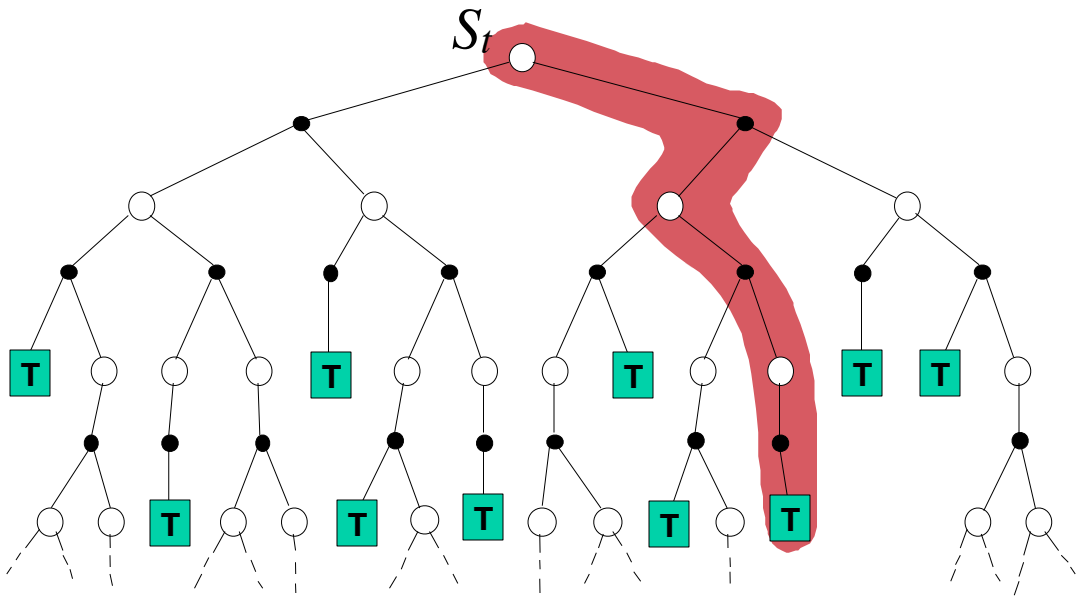
cf. Dynamic Programming

$$V(S_t) \leftarrow E_\pi\left[R_{t+1} + \gamma V(S_{t+1})\right] = \sum_a \pi(a|S_t) \sum_{s',r} p(s', r|S_t, a)[r + \gamma V(s')]$$



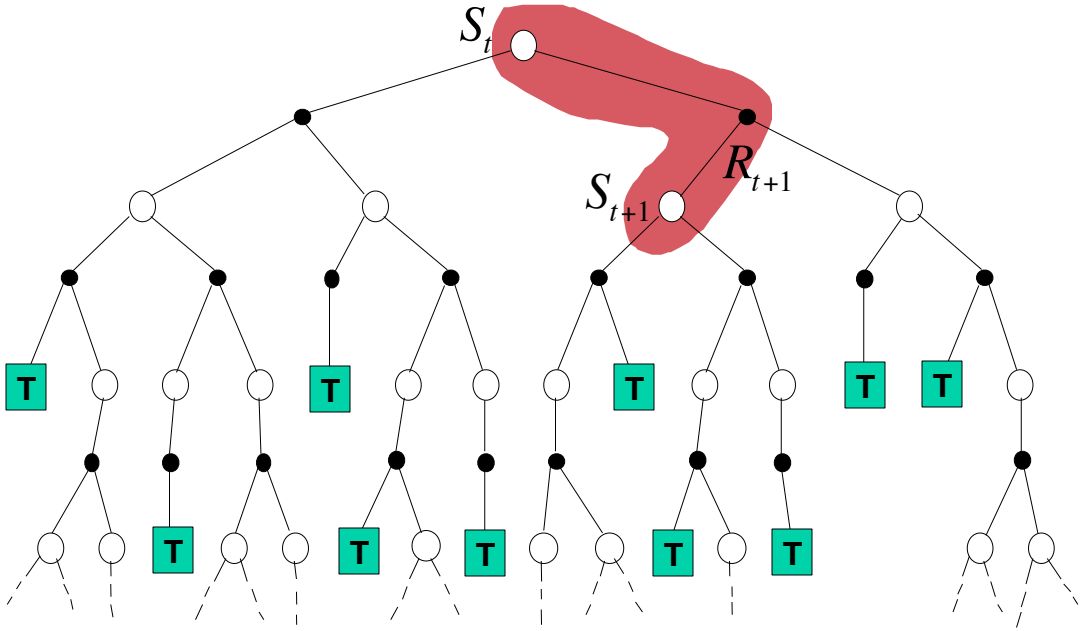
Recall: Monte Carlo

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



Simplest TD Method

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



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

You are the Predictor

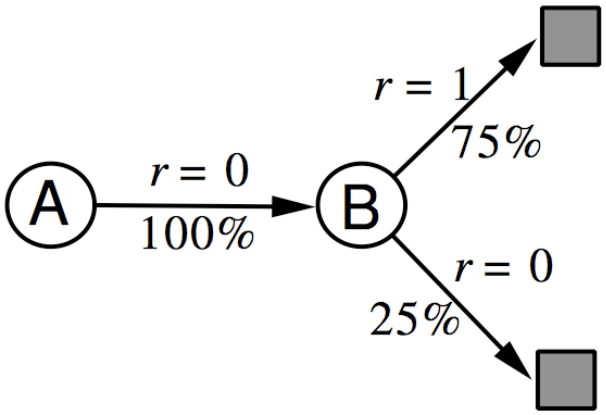
Suppose you observe the following 8 episodes:

A, 0, B, 0
B, 1
B, 1
B, 1
B, 1
B, 1
B, 1
B, 0

V(B)?
V(A)?

Assume Markov states, no discounting ($\gamma = 1$)

You are the Predictor



$V(\mathrm{A})?$

TD vs MC

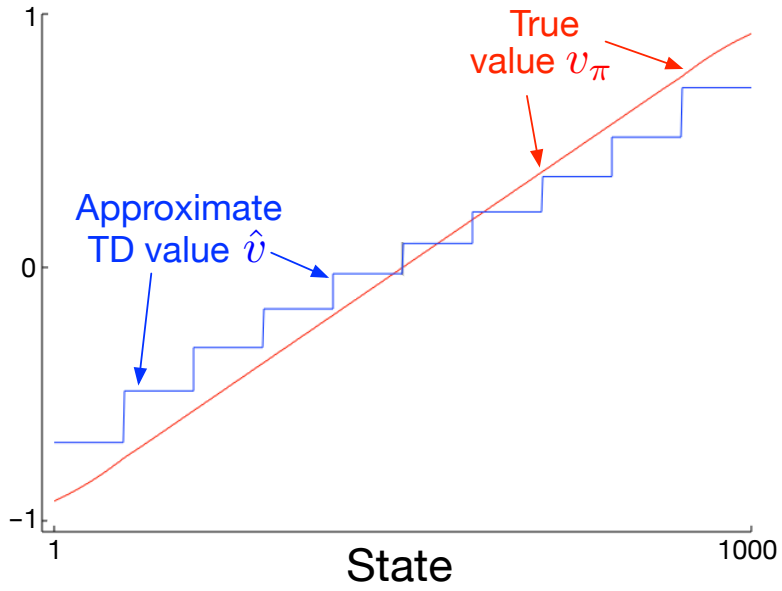
- Monte Carlo is gradient-based, converges to a local optimum of the weighted

Mean-squared error (MSE) (weighted by state visitation density)

- Temporal-Difference Learning (TD) fits (conceptually) a Markovian model
- Value function approximated the one that would be computed from the model
- TD introduces further bias in order to reduce variance
- Bias comes from Markov assumption, use of current (incorrect) value estimates

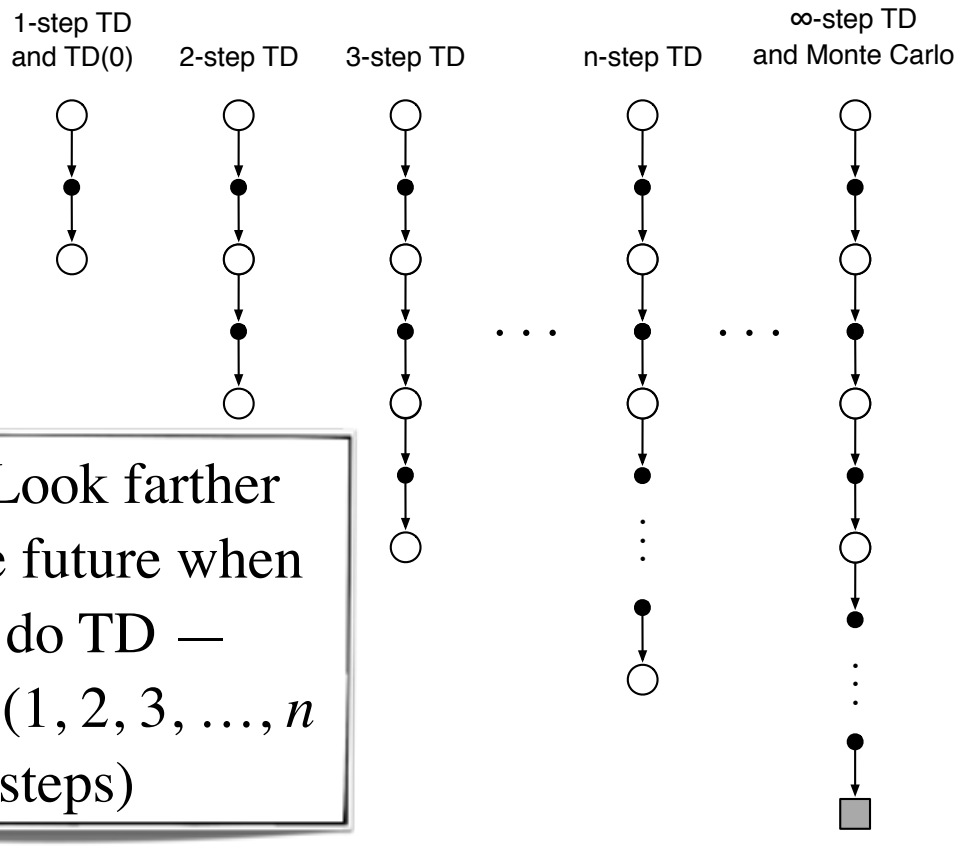
Semi-gradient TD is less accurate than MC on the 1000-state random walk using state aggregation

- 10 groups of 100 states
- after 100,000 episodes
- α = 2 x 10^{-5}



Relative values are still pretty accurate

n-step TD Prediction



Idea: Look farther into the future when you do TD — backup $(1, 2, 3, \ldots, n$ steps)

Mathematics of *n*-step TD 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:
  - Use V_t to estimate remaining return

$$G_t^{(1)} \doteq R_{t+1} + \gamma V_t(S_{t+1})$$

- *n*-step TD:
  - 2 step return:

$$G_t^{(2)} \doteq R_{t+1} + \gamma R_{t+2} + \gamma^2 V_t(S_{t+2})$$

  - *n*-step return:

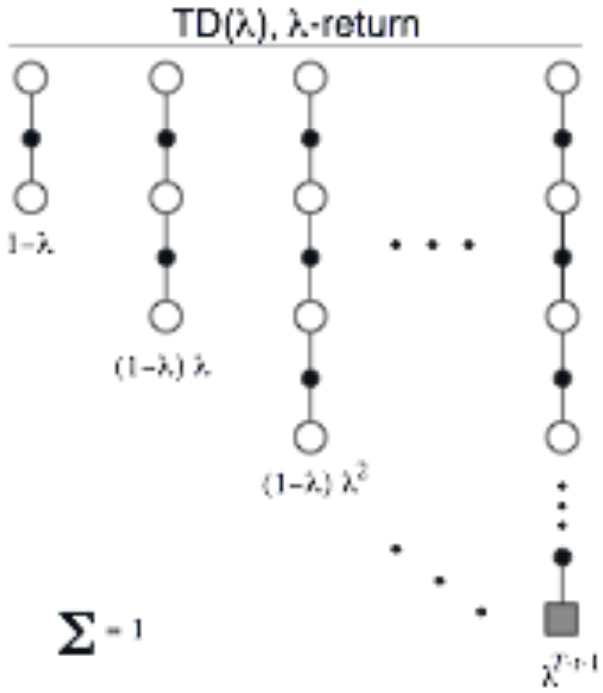
with

$$G_t^{(n)} \doteq G_t \text{ if } t + n \geq T)$$

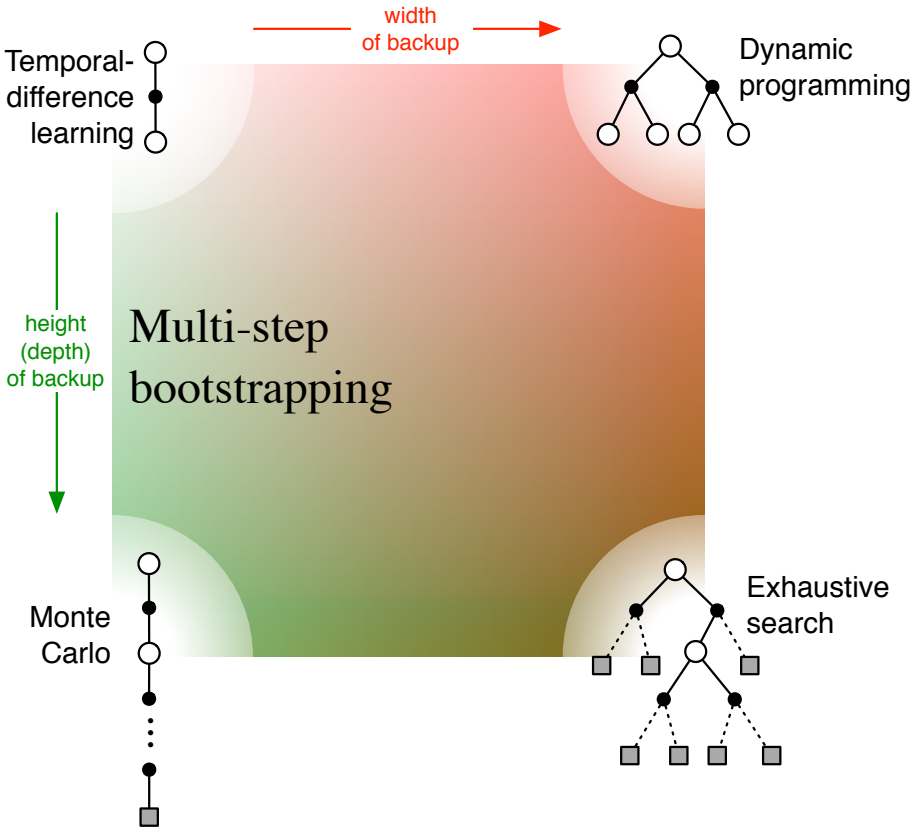
The λ-return is a compound update target

- The λ-return a target that averages all n-step targets
  - each weighted by λ^{n-1}

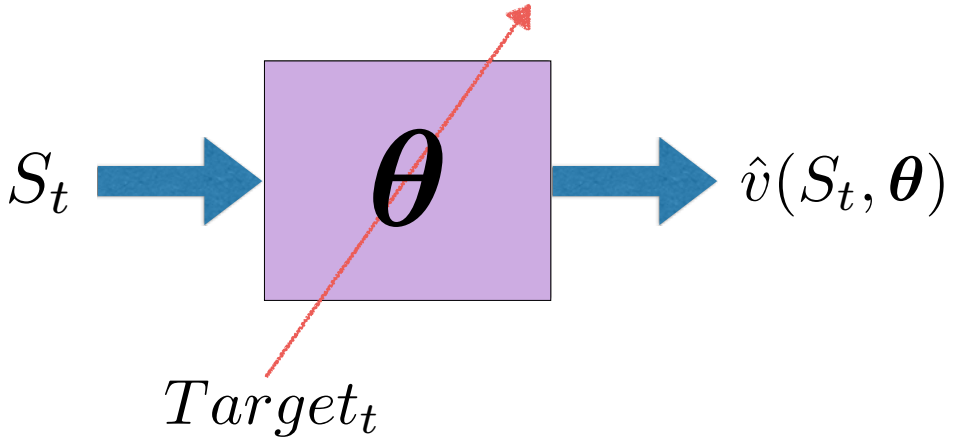
$$G_t^\lambda \doteq (1-\lambda)\sum_{n=1}^{\infty}\lambda^{n-1}G_t^{(n)}$$



Unified View



Value function approximation (VFA) replaces the table with a general parameterized form



Target depends on the agent's behavior, and in TD, also on its current estimates!

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

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

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

Semi-gradient TD(0) for estimating $\hat{v} \approx v_\pi$

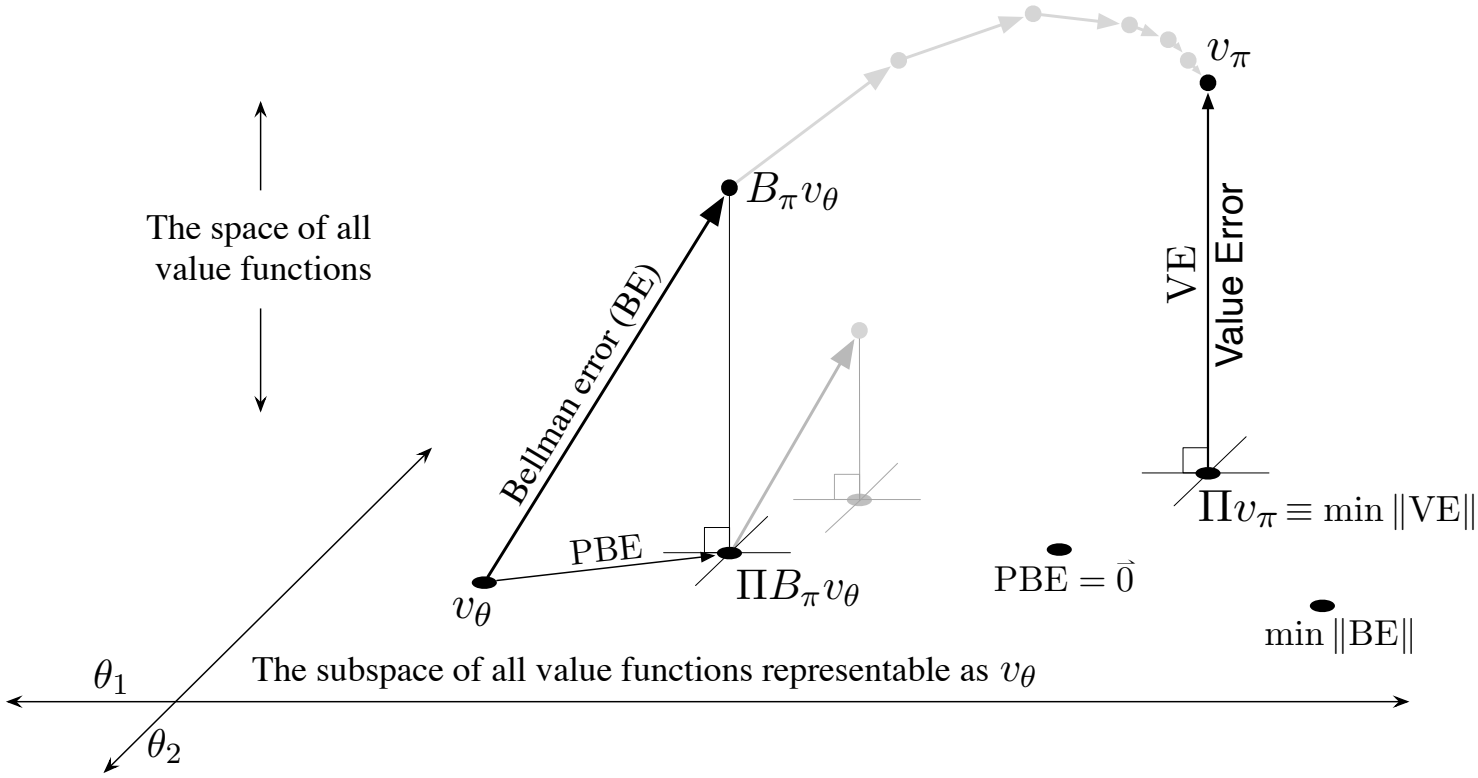
Input: the policy π to be evaluated
Input: a differentiable function $\hat{v} : \mathcal{S}^+ \times \mathbb{R}^n \to \mathbb{R}$ such that $\hat{v}(\text{terminal},\cdot) = 0$

Initialize value-function weights $\boldsymbol{\theta}$ arbitrarily (e.g., $\boldsymbol{\theta} = \mathbf{0}$)
Repeat (for each episode):
    Initialize S
    Repeat (for each step of episode):
        Choose $A \sim \pi(\cdot|S)$
        Take action A, observe R, S'
        $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\big[R + \gamma\hat{v}(S',\boldsymbol{\theta}) - \hat{v}(S,\boldsymbol{\theta})\big]\nabla\hat{v}(S,\boldsymbol{\theta})$
        $S \leftarrow S'$
    until S' is terminal

Geometric intuition

$v_{\boldsymbol{\theta}} \doteq \hat{v}(\cdot, \boldsymbol{\theta})$ as a giant vector $\in \mathbb{R}^{|\mathcal{S}|}$

$$(B_\pi v)(s) \doteq \sum_{a \in \mathcal{A}} \pi(s,a) \left[r(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) v(s') \right]$$



TD converges to the TD fixedpoint, $\boldsymbol{\theta}_{TD}$, a biased but interesting answer

TD(0) update:

$$\begin{aligned}\boldsymbol{\theta}_{t+1} &\doteq \boldsymbol{\theta}_t + \alpha\Big(R_{t+1} + \gamma\boldsymbol{\theta}_t^\top\boldsymbol{\phi}_{t+1} - \boldsymbol{\theta}_t^\top\boldsymbol{\phi}_t\Big)\boldsymbol{\phi}_t \\ &= \boldsymbol{\theta}_t + \alpha\Big(R_{t+1}\boldsymbol{\phi}_t - \boldsymbol{\phi}_t\big(\boldsymbol{\phi}_t - \gamma\boldsymbol{\phi}_{t+1}\big)^\top\boldsymbol{\theta}_t\Big)\end{aligned}$$

Fixed-point analysis:

$$\begin{aligned}&\mathbf{b} - \mathbf{A}\boldsymbol{\theta}_{TD} = \mathbf{0} \\ \Rightarrow\quad &\mathbf{b} = \mathbf{A}\boldsymbol{\theta}_{TD} \\ \Rightarrow\quad &\boldsymbol{\theta}_{TD} \doteq \mathbf{A}^{-1}\mathbf{b}\end{aligned}$$

In expectation:

$$\mathbb{E}[\boldsymbol{\theta}_{t+1}|\boldsymbol{\theta}_t] = \boldsymbol{\theta}_t + \alpha(\mathbf{b} - \mathbf{A}\boldsymbol{\theta}_t),$$

where

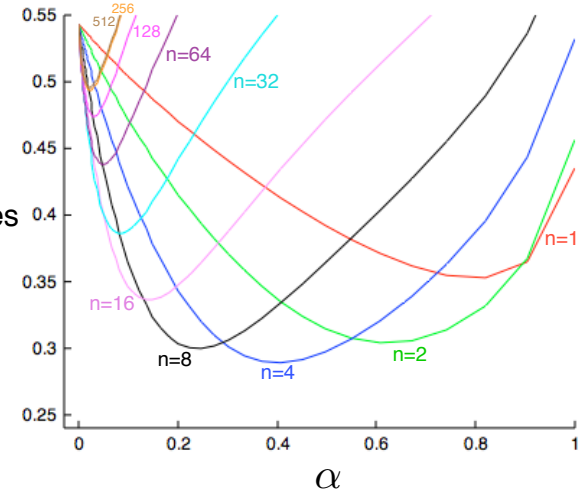
$$\mathbf{b} \doteq \mathbb{E}[R_{t+1}\boldsymbol{\phi}_t] \in \mathbb{R}^n \quad \text{and} \quad \mathbf{A} \doteq \mathbb{E}\Big[\boldsymbol{\phi}_t\big(\boldsymbol{\phi}_t - \gamma\boldsymbol{\phi}_{t+1}\big)^\top\Big] \in \mathbb{R}^n \times \mathbb{R}^n$$

Guarantee:

$$\text{MSVE}(\boldsymbol{\theta}_{TD}) \leq \frac{1}{1-\gamma}\min_{\boldsymbol{\theta}} \text{MSVE}(\boldsymbol{\theta})$$

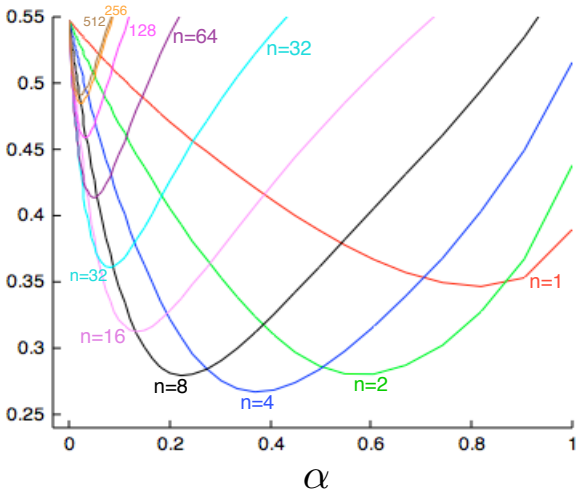
Bootstrapping greatly speeds learning

Average RMS error over 1000 states and first 10 episodes



1000 states aggregated into 20 groups of 50

Average RMS error over 19 states and first 10 episodes



19 states tabular

Eligibility traces (mechanism)

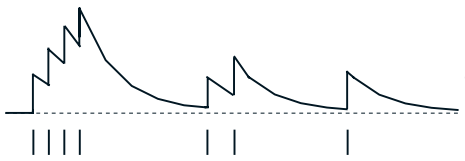
- The forward view was for theory
- The backward view is for *mechanism*
- New memory vector called *eligibility trace*

same shape as $\boldsymbol{\theta}$

$$\mathbf{e}_t \in \mathbb{R}^n \geq \mathbf{0}$$

  - On each step, decay each component by $\boldsymbol{\gamma\lambda}$ and increment the trace for the current state by 1
  - *Accumulating trace*

$$\mathbf{e}_0 \doteq \mathbf{0},$$
$$\mathbf{e}_t \doteq \nabla \hat{v}(S_t, \boldsymbol{\theta}_t) + \gamma\lambda \mathbf{e}_{t-1}$$



The Semi-gradient TD(λ) algorithm

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha \delta_t \mathbf{e}_t$$

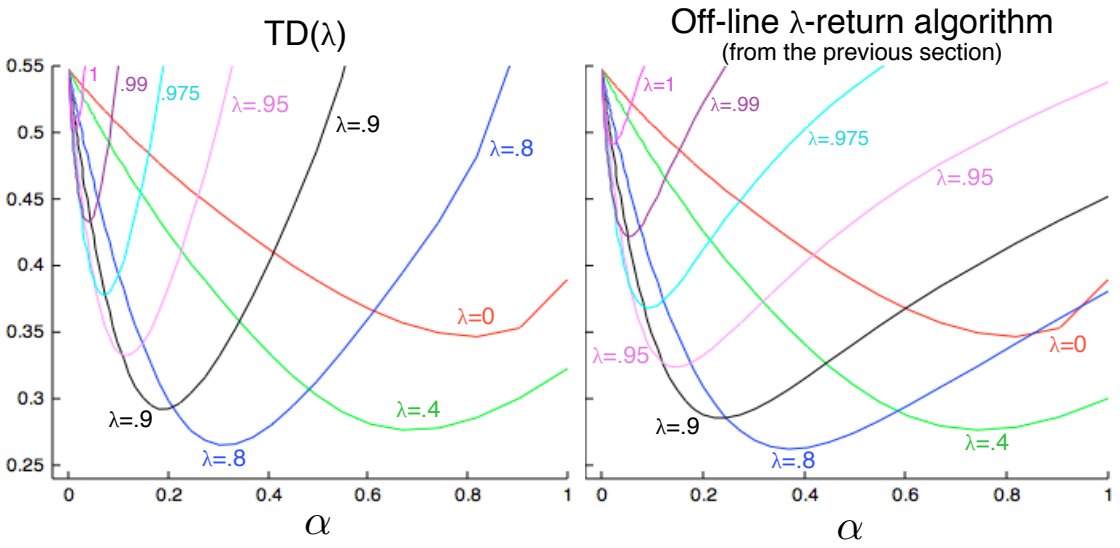
$$\delta_t \doteq R_{t+1} + \gamma \hat{v}(S_{t+1}, \boldsymbol{\theta}_t) - \hat{v}(S_t, \boldsymbol{\theta}_t)$$

$$\mathbf{e}_0 \doteq \mathbf{0},$$
$$\mathbf{e}_t \doteq \nabla \hat{v}(S_t, \boldsymbol{\theta}_t) + \gamma \lambda \mathbf{e}_{t-1}$$

TD(λ) performance with α

Tabular 19-state random walk task

RMS error at the end of the episode over the first 10 episodes



Summing up policy evaluation

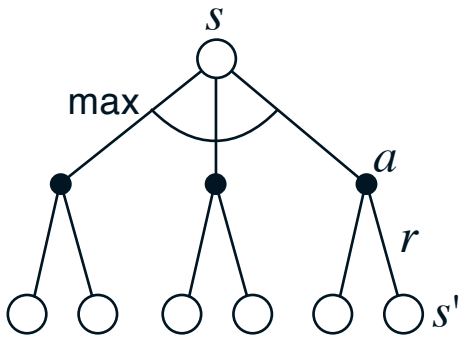
- Value-function approximation by stochastic gradient descent enables RL to be applied to arbitrarily large state spaces
- Most algorithms just carry over the targets from the tabular case
- With bootstrapping (TD), we don't get true gradient descent methods
    - this complicates the analysis
    - but the linear, on-policy case is still guaranteed convergent
    - and learning is still *much faster*

Bellman Optimality Equation for v_*

The value of a state under an optimal policy must equal the expected return for the best action from that state:

$$
\begin{aligned}
v_*(s) &= \max_a q_{\pi_*}(s,a) \\
&= \max_a \mathbb{E}[R_{t+1} + \gamma v_*(S_{t+1}) \mid S_t = s, A_t = a] \\
&= \max_a \sum_{s',r} p(s',r|s,a)\big[r + \gamma v_*(s')\big].
\end{aligned}
$$

The relevant backup diagram:

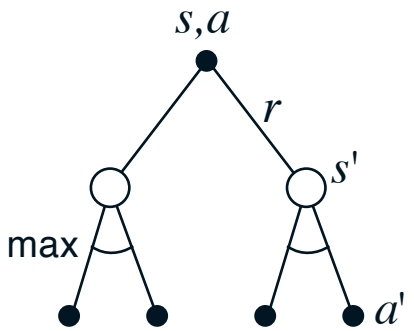


v_* is the unique solution of this system of nonlinear equations.

Bellman Optimality Equation for q_*

$$
\begin{aligned}
q_*(s,a) &= \mathbb{E}\left[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a') \;\middle|\; S_t = s, A_t = a\right] \\
&= \sum_{s',r} p(s', r|s, a)\left[r + \gamma \max_{a'} q_*(s', a')\right].
\end{aligned}
$$

The relevant backup diagram:



q_* is the unique solution of this system of nonlinear equations.

Value Iteration

Recall the **full policy-evaluation backup**:

$$v_{k+1}(s) = \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a) \Big[r + \gamma v_k(s') \Big] \qquad \forall s \in \mathcal{S}$$

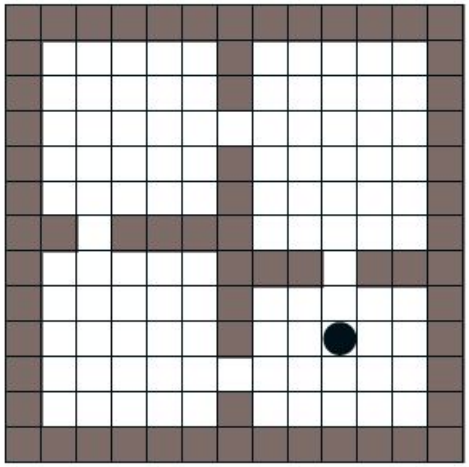
Here is the **full value-iteration backup**:

$$v_{k+1}(s) = \max_a \sum_{s',r} p(s', r|s, a) \Big[r + \gamma v_k(s') \Big] \qquad \forall s \in \mathcal{S}$$

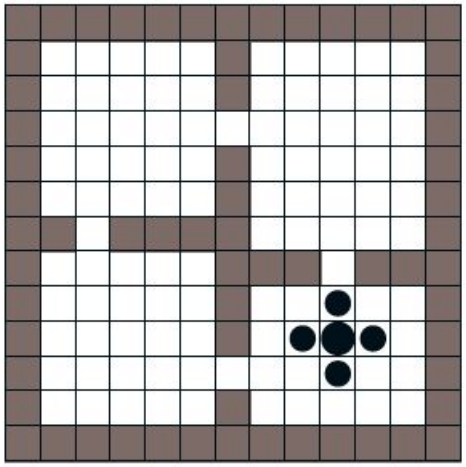
Illustration: Rooms Example

Four actions, fail 30% of the time

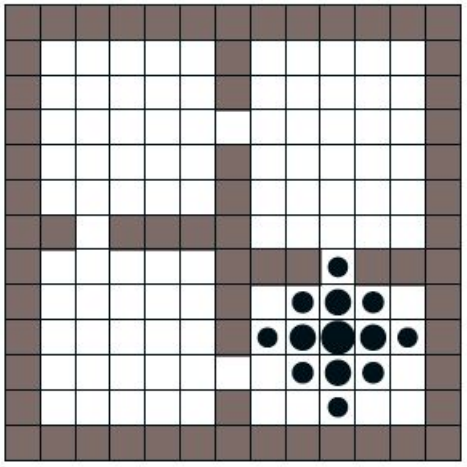
No rewards until the goal is reached, $\gamma = 0.9$.



Iteration #1

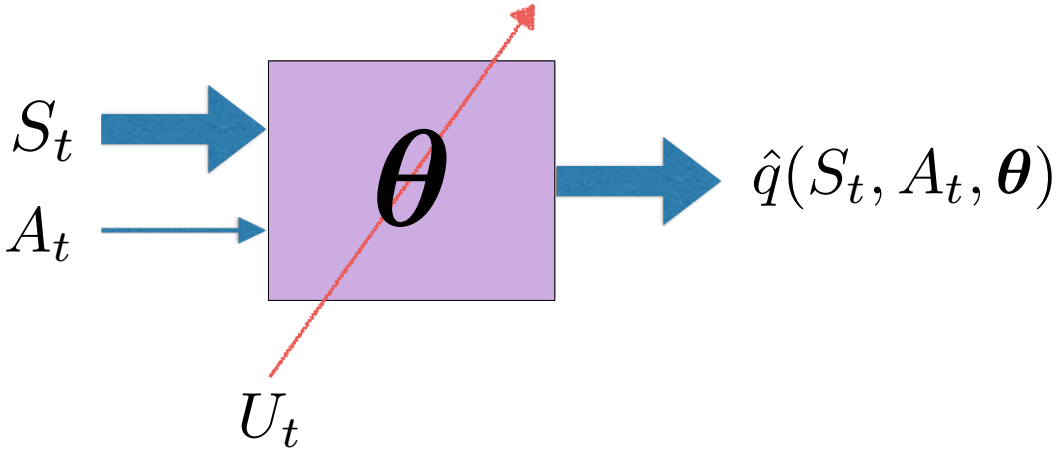


Iteration #2



Iteration #3

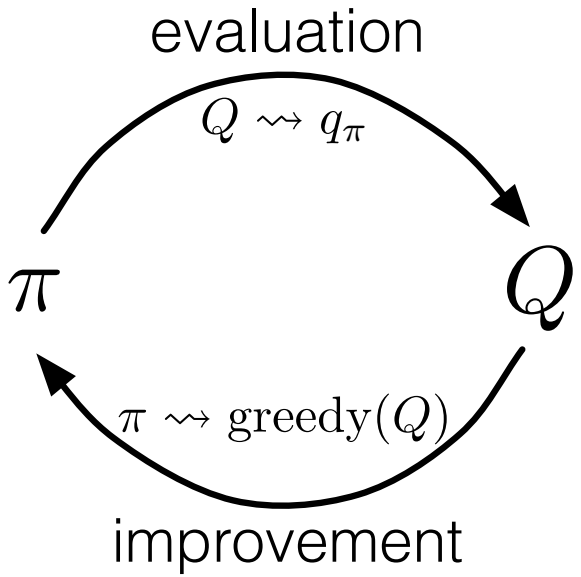
Value function approximation (VFA) for control



Monte Carlo Estimation of Action Values (Q)

- Monte Carlo is most useful when a model is not available
- We want to learn q_*
- $q_\pi(s,a)$ - average return starting from state s and action a following π
- Converges asymptotically *if* every state-action pair is visited
- *Exploring starts:* Every state-action pair has a non-zero probability of being the starting pair

Monte Carlo Control



- MC policy iteration: Policy evaluation using MC methods followed by policy improvement
- Policy improvement step: greedify with respect to value (or action-value) function

Convergence of MC Control

$$
\begin{aligned}
q_{\pi_k}(s, \pi_{k+1}(s)) &= q_{\pi_k}(s, \arg\max_a q_{\pi_k}(s, a)) \\
&= \max_a q_{\pi_k}(s, a) \\
&\geq q_{\pi_k}(s, \pi_k(s)) \\
&= v_{\pi_k}(s).
\end{aligned}
$$

- Greedified policy meets t conditions for policy improvement:

- And thus must be $\geq \pi_k$ by the policy improvement theorem
- This assumes exploring starts and infinite number of episodes for MC policy evaluation
- To solve the latter:
  - update only to a given level of performance
  - alternate between evaluation and improvement per episode

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

Fixed point is optimal policy π^*

Now proven (almost)

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:
        $\pi(s) \leftarrow \arg\max_a Q(s,a)$

On-policy Monte Carlo Control

- *On-policy:* learn about policy currently executing
- How do we get rid of exploring starts?
  - The policy must be eternally *soft*:
    - $\pi(a|s) > 0$ for all s and a
  - e.g. ε-soft policy:
    - probability of an action = $\frac{\epsilon}{|\mathcal{A}(s)|}$ or $1 - \epsilon + \frac{\epsilon}{|\mathcal{A}(s)|}$

non-max max (greedy)

- Similar to GPI: move policy *towards* greedy policy (e.g., ε-greedy)
- Converges to best ε-soft policy

The Exploration/Exploitation Dilemma

- Suppose you form estimates

$$Q_t(a) \approx q_*(a), \quad \forall a$$

action-value estimates

- Define the *greedy action* at time *t* as

$$A_t^* \doteq \arg\max_a Q_t(a)$$

- Exploring: choose a non-greedy action

On-policy MC Control

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

$Q(s,a) \leftarrow$ arbitrary

$Returns(s,a) \leftarrow$ empty list

$\pi(a|s) \leftarrow$ an arbitrary ε-soft policy

Repeat forever:

(a) Generate an episode using π

(b) For each pair s, a appearing in the episode:

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

Append G to $Returns(s,a)$

$Q(s,a) \leftarrow \text{average}(Returns(s,a))$

(c) For each s in the episode:

$A^* \leftarrow \arg\max_a Q(s,a)$

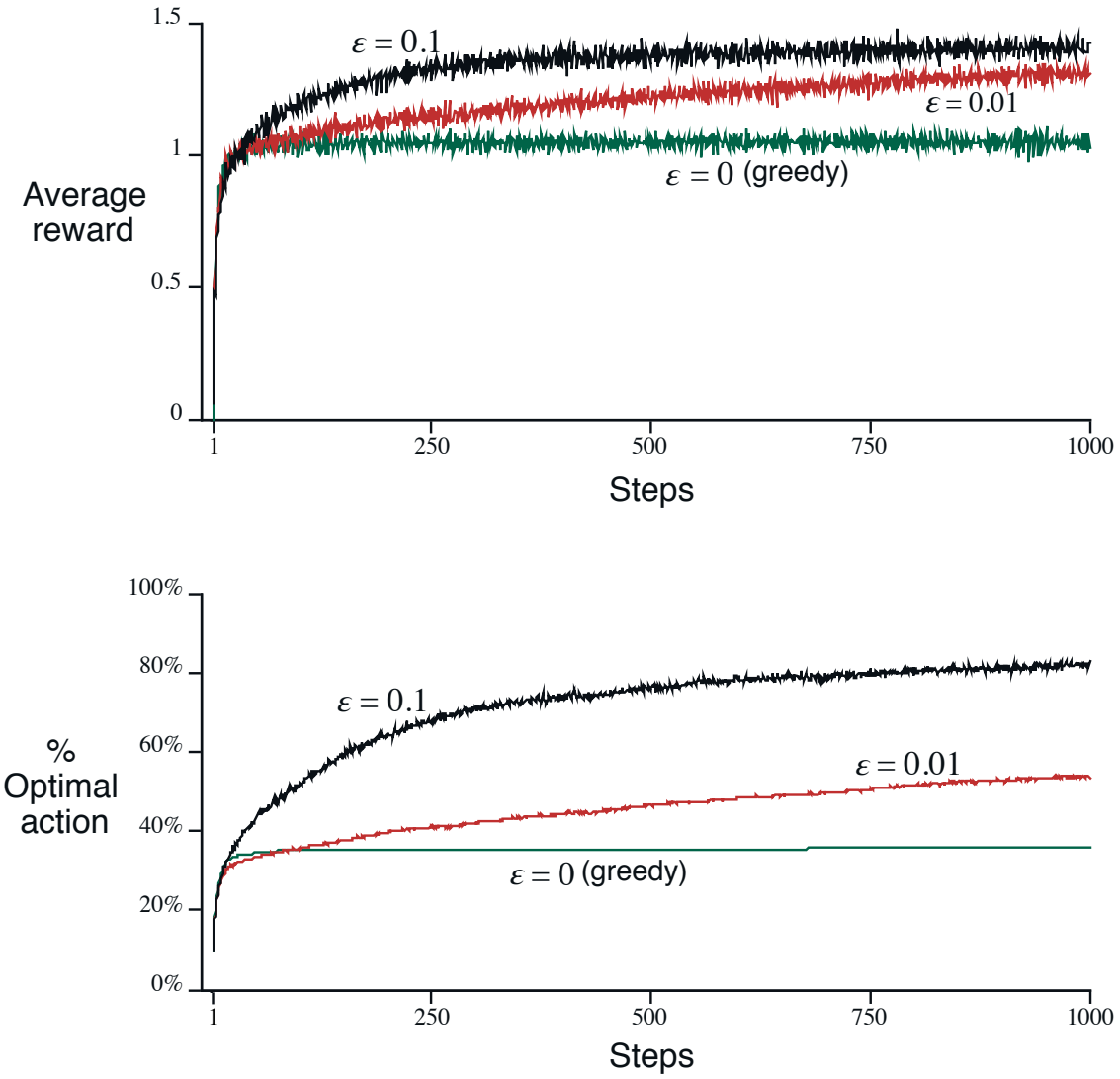
For all $a \in \mathcal{A}(s)$:

$$\pi(a|s) \leftarrow \begin{cases} 1 - \varepsilon + \varepsilon/|\mathcal{A}(s)| & \text{if } a = A^* \\ \varepsilon/|\mathcal{A}(s)| & \text{if } a \neq A^* \end{cases}$$

ε-Greedy Action Selection

- In greedy action selection, you always exploit
- In ε-greedy, you are usually greedy, but with probability ε you instead pick an action at random (possibly the greedy action again)
- This is perhaps the simplest way to balance exploration and exploitation

ε-Greedy Methods on the 10-Armed Testbed



(Semi-)gradient methods carry over to control in the usual on-policy GPI way

- Always learn the action-value function of the current policy
- Always act near-greedily wrt the current action-value estimates
- The learning rule is:

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha \Big[U_t - \hat{q}(S_t, A_t, \boldsymbol{\theta}_t) \Big] \nabla \hat{q}(S_t, A_t, \boldsymbol{\theta}_t)$$

update target, e.g.,

$U_t = G_t$ (MC)

$U_t = R_{t+1} + \gamma \hat{q}(S_{t+1}, A_{t+1}, \boldsymbol{\theta}_t)$ (Sarsa)

$U_t = R_{t+1} + \gamma \sum_a \pi(a|S_{t+1}) \hat{q}(S_{t+1}, a, \boldsymbol{\theta}_t)$ (Expected Sarsa)

$U_t = \sum_{s',r} p(s', r|S_t, A_t) \Big[r + \gamma \sum_{a'} \pi(a'|s') \hat{q}(s', a', \boldsymbol{\theta}_t) \Big]$ (DP)

(Semi-)gradient methods carry over to control

$$\boldsymbol{\theta}_{t+1} \doteq \boldsymbol{\theta}_t + \alpha\Big[U_t - \hat{q}(S_t, A_t, \boldsymbol{\theta}_t)\Big]\nabla\hat{q}(S_t, A_t, \boldsymbol{\theta}_t)$$

Episodic Semi-gradient Sarsa for Estimating $\hat{q} \approx q_*$

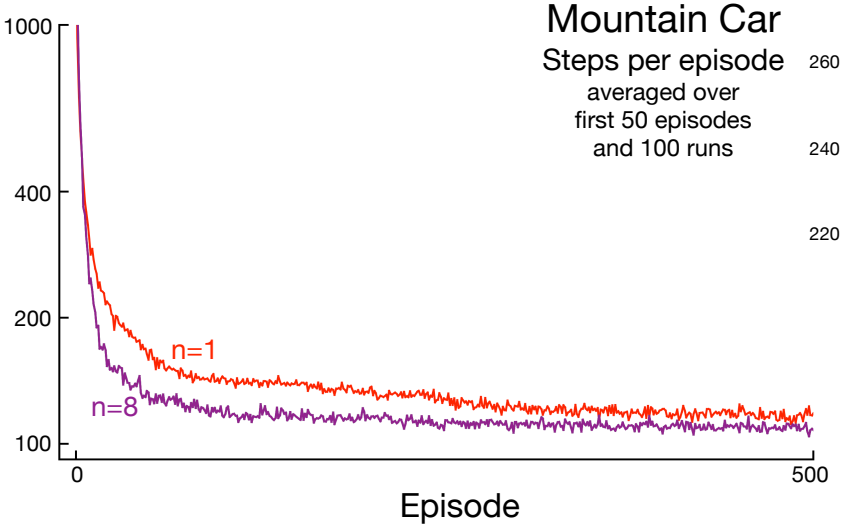
Input: a differentiable function $\hat{q} : \mathcal{S} \times \mathcal{A} \times \mathbb{R}^n \to \mathbb{R}$

Initialize value-function weights $\boldsymbol{\theta} \in \mathbb{R}^n$ arbitrarily (e.g., $\boldsymbol{\theta} = \mathbf{0}$)
Repeat (for each episode):
  $S, A \leftarrow$ initial state and action of episode (e.g., ε-greedy)
  Repeat (for each step of episode):
    Take action A, observe R, S'
    If S' is terminal:
      $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\big[R - \hat{q}(S, A, \boldsymbol{\theta})\big]\nabla\hat{q}(S, A, \boldsymbol{\theta})$
      Go to next episode
    Choose A' as a function of $\hat{q}(S', \cdot, \boldsymbol{\theta})$ (e.g., ε-greedy)
    $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha\big[R + \gamma\hat{q}(S', A', \boldsymbol{\theta}) - \hat{q}(S, A, \boldsymbol{\theta})\big]\nabla\hat{q}(S, A, \boldsymbol{\theta})$
    $S \leftarrow S'$
    $A \leftarrow A'$

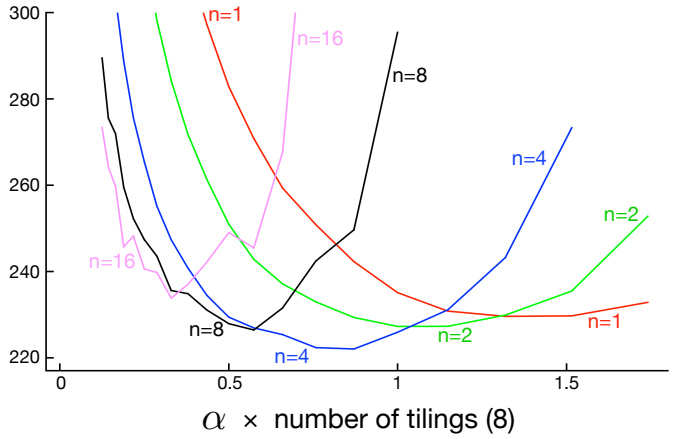
n-step semi-gradient Sarsa is better for *n*>1

$$\boldsymbol{\theta}_{t+n} \doteq \boldsymbol{\theta}_{t+n-1} + \alpha \left[G_t^{(n)} - \hat{q}(S_t, A_t, \boldsymbol{\theta}_{t+n-1}) \right] \nabla \hat{q}(S_t, A_t, \boldsymbol{\theta}_{t+n-1}), \qquad 0 \leq t < T$$

Mountain Car
Steps per episode
log scale
averaged over 100 runs



Mountain Car
Steps per episode
averaged over
first 50 episodes
and 100 runs



Summing up on-policy control

- Control is straightforward in the on-policy case
- Formal results (bounds) exist for the linear, on-policy case (eg. Gordon, 2000, Perkins & Precup, 2003 and follow-up work)
    - we get chattering near a good solution, not convergence

Q-learning

- Value iteration for action values:

$$q_*(s,a) = r(s,a) + \gamma \sum_{s'} P(s'|s,a) \max_{a'} q_*(s',a')$$

- This is *off-policy*: No matter what the policy does, we imagine the best possible course of action
- Q-learning: uses the sampled, bootstrapped corresponding target

$$U_t = R_{t+1} + \gamma \max_a Q_t(S_{t+1}, a)$$

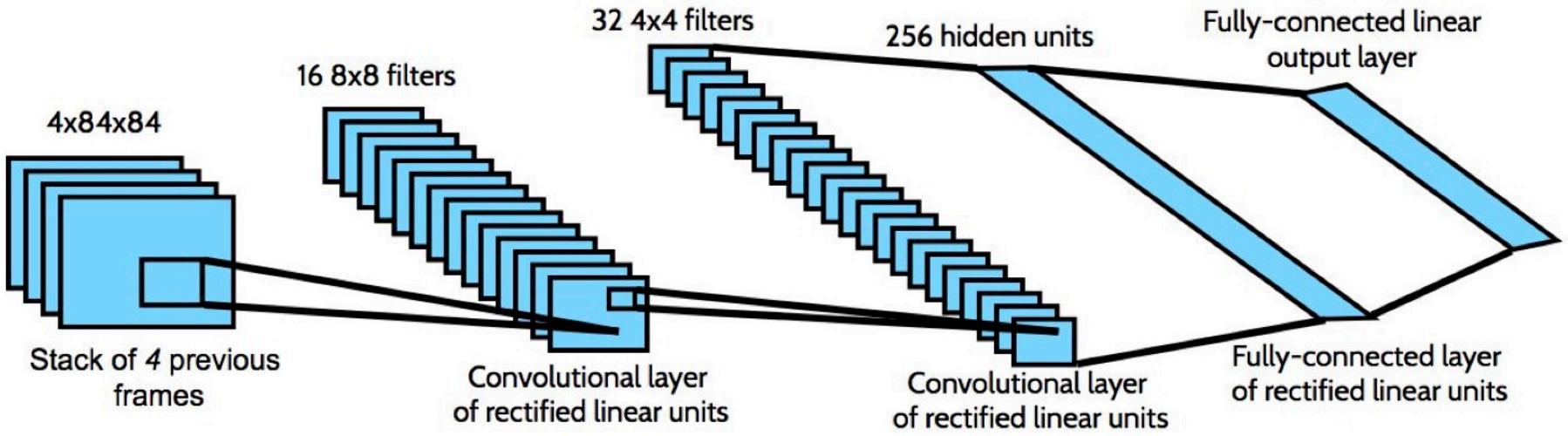
- In the tabular case, converges to q_* *even if the behavior policy is random*!
- Theoretical properties with function approximation are very bad but empirical results are very strong

DQN

(Mnih, Kavukcuoglu, Silver, et al., Nature 2015)

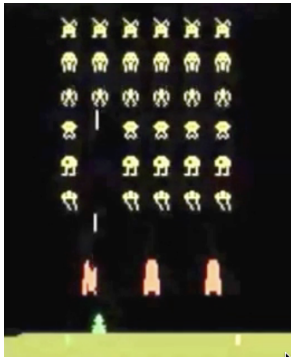
- Learns to play video games **from raw pixels**, simply by playing
- Can learn Q function by Q-learning

$$\Delta \boldsymbol{w} = \alpha \left(R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \boldsymbol{w}) - Q(S_t, A_t; \boldsymbol{w}) \right) \nabla_{\boldsymbol{w}} Q(S_t, A_t; \boldsymbol{w})$$



RL + Deep Learning, applied to Classic Atari Games

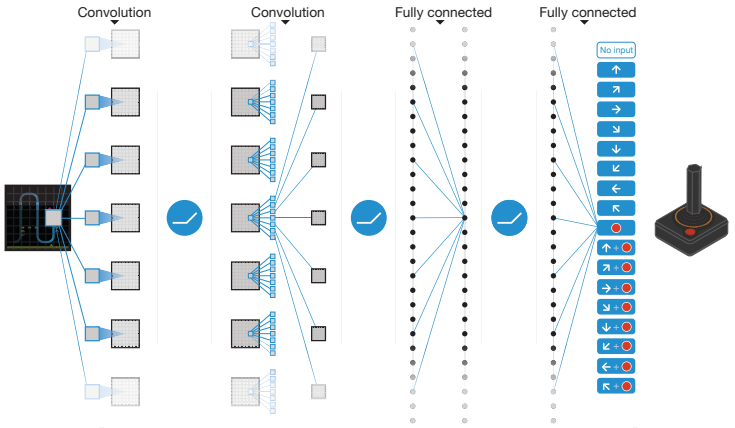
Google Deepmind 2015, Bowling et al. 2012





- Learned to play 49 games for the Atari 2600 game console, without labels or human input, from self-play and the score alone

mapping raw screen pixels



to predictions of final score for each of 18 joystick actions

- Learned to play better than all previous algorithms and at human level for more than half the games

Same learning algorithm applied to all 49 games!
w/o human tuning

DQN

(Mnih, Kavukcuoglu, Silver, et al., Nature 2015)

- Learns to play video games **from raw pixels**, simply by playing
- Can learn Q function by Q-learning

$$\Delta \boldsymbol{w} = \alpha \left(R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \boldsymbol{w}) - Q(S_t, A_t; \boldsymbol{w}) \right) \nabla_{\boldsymbol{w}} Q(S_t, A_t; \boldsymbol{w})$$

- Core components of DQN include:
  - Target networks (Mnih et al. 2015)

$$\Delta \boldsymbol{w} = \alpha \left(R_{t+1} + \gamma \max_a Q(S_{t+1}, a; \boldsymbol{w}^-) - Q(S_t, A_t; \boldsymbol{w}) \right) \nabla_{\boldsymbol{w}} Q(S_t, A_t; \boldsymbol{w})$$

  - Experience replay (Lin 1992): replay previous tuples (s, a, r, s')

Target Network Intuition

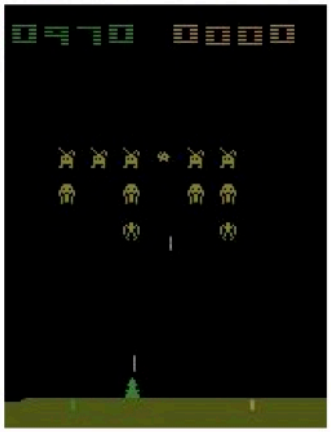
(Slide credit: Vlad Mnih)

- Changing the value of one action will change the value of other actions and similar states.
- The network can end up chasing its own tail because of bootstrapping.
- Somewhat surprising fact - bigger networks are less prone to this because they alias less.

$$L_i(\theta_i) = \mathbb{E}_{s,a,s',r\sim D}\left(\underbrace{r + \gamma \max_{a'} Q(s', a'; \theta_i^-)}_{\text{target}} - Q(s, a; \theta_i)\right)^2$$

s

s'



DQN

(Mnih, Kavukcuoglu, Silver, et al., Nature 2015)

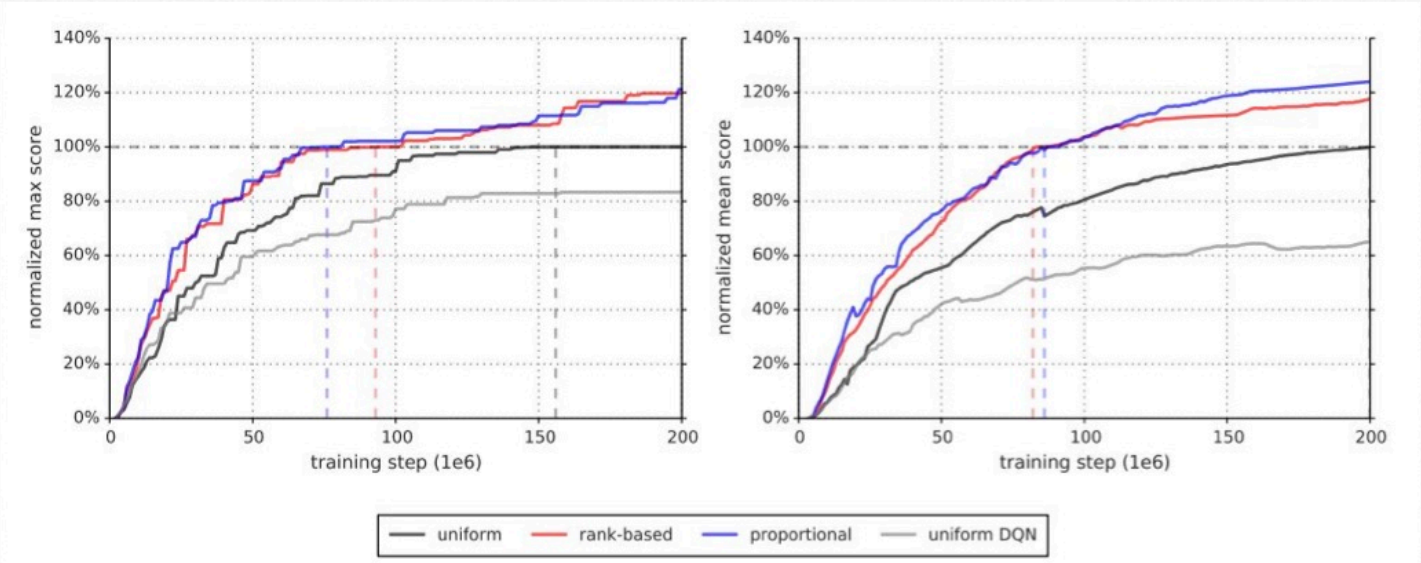
- Many later improvements to DQN
  - Double Q-learning (van Hasselt 2010, van Hasselt et al. 2015)
  - Prioritized replay (Schaul et al. 2016)
  - Dueling networks (Wang et al. 2016)
  - Asynchronous learning (Mnih et al. 2016)
  - Adaptive normalization of values (van Hasselt et al. 2016)
  - Better exploration (Bellemare et al. 2016, Ostrovski et al., 2017, Fortunato, Azar, Piot et al. 2017)
  - Distributional losses (Bellemare et al. 2017)
  - Multi-step returns (Mnih et al. 2016, Hessel et al. 2017)
  - … many more …

Prioritized Experience Replay

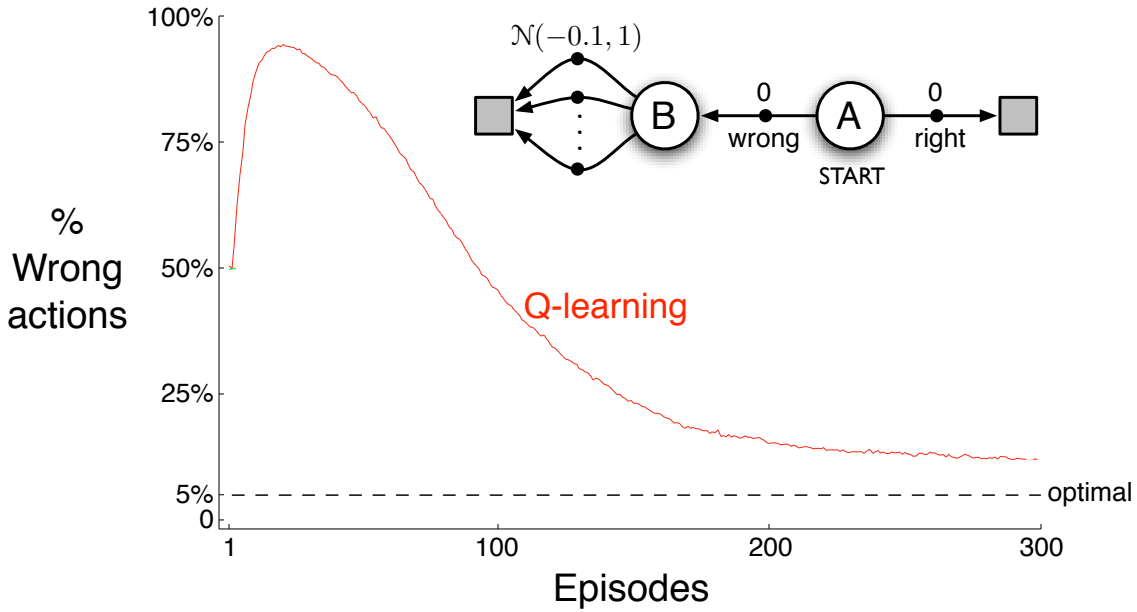
"Prioritized Experience Replay", Schaul et al. (2016)

- **Idea**: Replay transitions in proportion to TD error:

$$\left| r + \gamma \max_{a'} Q(s', a'; \theta^-) - Q(s, a; \theta) \right|$$



Maximization Bias Example



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

Double Q-Learning

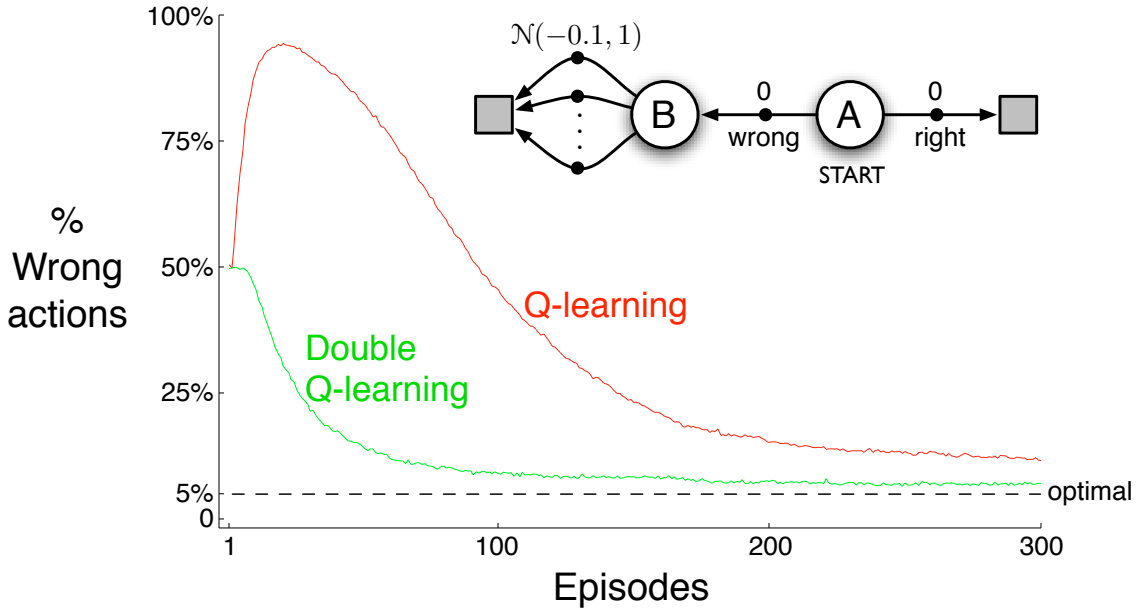
Hado van Hasselt 2010

- Train 2 action-value functions, Q_1 and Q_2
- Do Q-learning on both, but
  - never on the same time steps (Q_1 and Q_2 are indep.)
  - pick Q_1 or Q_2 at random to be updated on each step
- If updating Q_1, use Q_2 for the value of the next state and vice versa

$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha\Big(R_{t+1} + Q_2\big(S_{t+1}, \arg\max_a Q_1(S_{t+1}, a)\big) - Q_1(S_t, A_t)\Big)$$

- Action selections are (say) ε-greedy with respect to the sum of Q_1 and Q_2

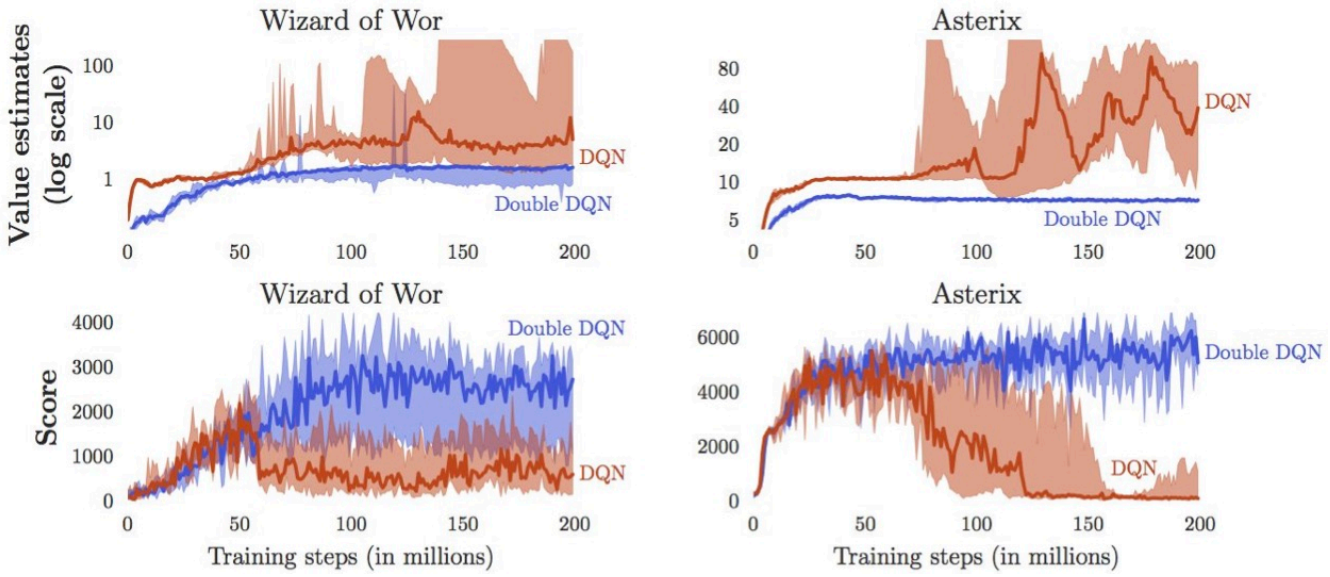
Example of Maximization Bias



Double Q-learning:

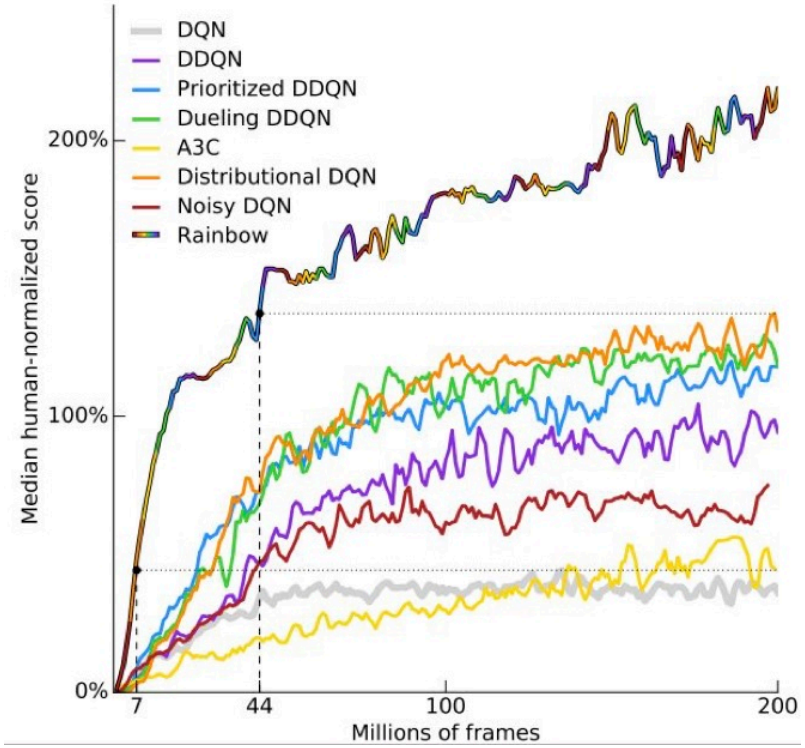
$$Q_1(S_t, A_t) \leftarrow Q_1(S_t, A_t) + \alpha \Big[R_{t+1} + \gamma Q_2\big(S_{t+1}, \arg\max_a Q_1(S_{t+1}, a)\big) - Q_1(S_t, A_t) \Big]$$

Double DQN



cf. van Hasselt et al, 2015)

DQN improvements



Rainbow model, Hessel et al, 2017)