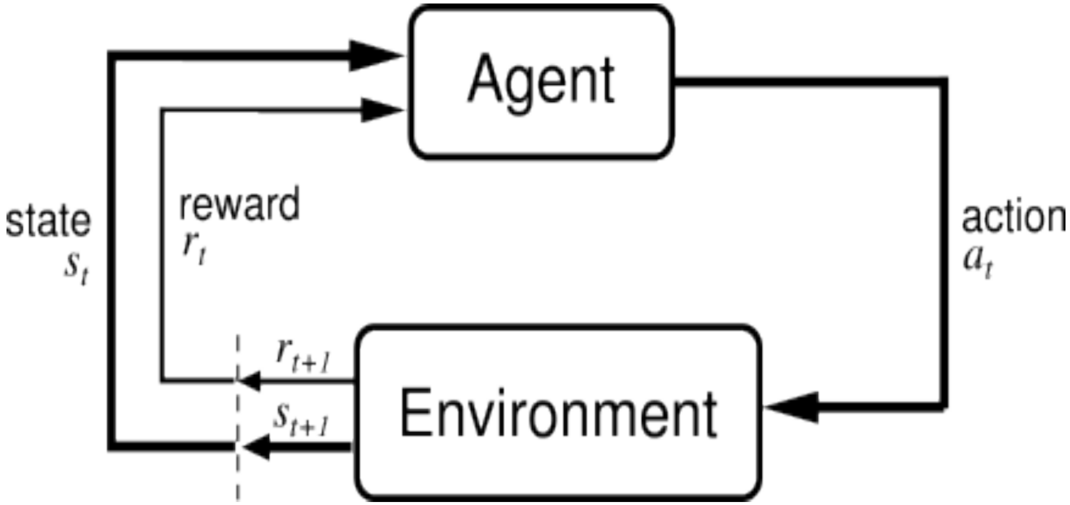
Introduction to Reinforcement Learning

MAL Seminar 2014-2015

RL Background

- Learning by interacting with the environment
- Reward good behavior, punish bad behavior
- Trial & Error
- Combines ideas from psychology and control theory

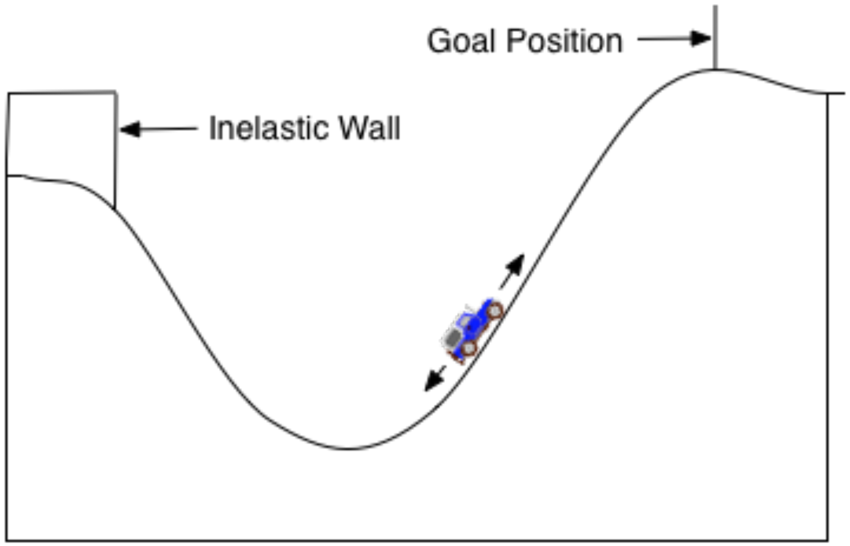
The Problem



Reinforcement learning is learning what to do--how to ***map situations to actions****--so as to* ***maximize a numerical reward signal****. The learner is not told which actions to take, as in most forms of machine learning, but instead must* ***discover*** *which actions yield the most reward by* ***trying*** *them. In the most interesting and challenging cases, actions may affect not only the immediate reward but also the next situation and, through that, all subsequent rewards.*

Sutton & Barto

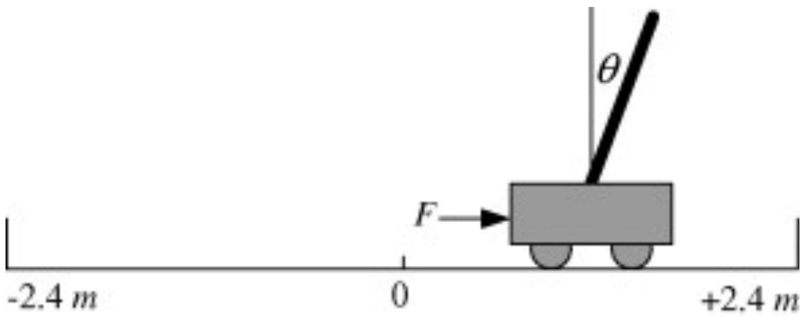
Some Examples



Mountain Car:

- **Goal**: Accelerate (underpowered) car to top of hill
- **state observations**: position (1d), velocity (1d)
- **actions**: apply force -40N,0,+40N

Some Examples



Pole balancing:

- **Goal**: keep pole in upright position on moving cart
- **state observations**: pole angle, angular velocity
- **actions**: apply force to cart

Some Examples

Helicopter hovering:

- **Goal**: stable hovering in the presence of wind
- **observed states**: posities (3d), velocities (3d), angular rates (3d)
- **actions**: pitches (4d)

Formal Problem Definition: Markov Decision Process

a Markov Decision Process consists of:

- set of **States** S= {s1,...,sn} (for now: finite & discrete)
- set of **Actions** A = {a1,..,am} (for now: finite & discrete)
- **Transition function** T:

  $T(s,a,s') = P(s(t+1)=s' \mid s(t) =s, a(t)=a)$
- **Reward function** r:

  $r(s,a,s') = E[r(t+1) \mid s(t) =s, a(t)=a, s(t+1)=s']$

Formal definition of reinforcement learning problem.

Note: assumes the Markov property (next state / reward are independent of history, given the current state)

Goal

- Goal of RL is to maximize the **expected** long term **future return R_t**
- Usually the **discounted** sum of rewards is used:

$$\sum_{t=0}^{\infty} \gamma^t r_{t+1} \quad \gamma \in [0, 1)$$

- Note: this is not the same as maximizing immediate rewards r(s,a,s'), **R_t takes into account the future**
- Other measures exist (e.g. total or average reward over time)

Note on reward functions

- RL considers the reward function as an **unknown part of the environment**, external to the learning agent.
- In practice, reward functions are typically **chosen by the system designer** and known
- Knowing the reward function, however, does not mean we know how to maximize long term rewards. This also depends on the system dynamics (T), which are unknown
- Typical reward function (keep it simple!):

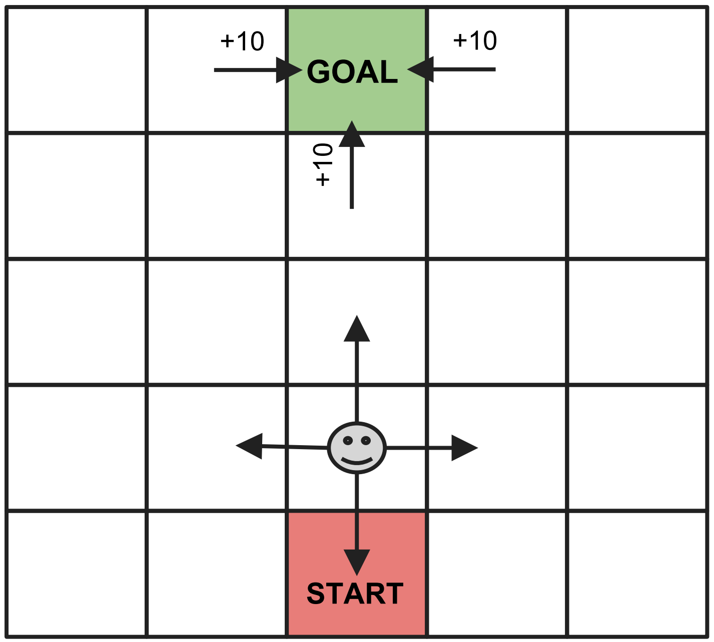
$$\begin{cases} 0, & \text{if goal is reached} \\ -50, & \text{if system goes out of bounds} \\ -1, & \text{else} \end{cases}$$

Policies

The agent's goal is to learn a policy π, which determines the **probability of selecting each action in a given state** in order to maximize future rewards

- $\pi(s,a)$ gives the probability of selecting action a in state s under policy π
- For deterministic policies we use $\pi(s)$ to denote the action a for which $\pi(s,a)=1$
- In finite MDPs it can be shown that a deterministic optimal policy always exists

Example



- States: Location 1 ... 25
- Actions: Move N,E,S,W
- Transitions: move 1 step in selected direction (except at borders)
- Rewards: +10 if next loc == goal, 0 else

- find shortest path to goal
- Rewards can be delayed: only receive reward when reaching goal
- unknown environment
- Consequences of an action can only be discovered by trying it and observing the result (new state s', reward r)

Value Functions

State Values (V-values):

$$\begin{aligned}
V^\pi(s) &= E_\pi\{R_t \mid s_t = s\} \\
&= E_\pi\{r_{r+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \ldots \mid s_t = s\} \\
&= E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s\right\} \\
&= \sum_a \pi(s,a) \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^\pi(s')]
\end{aligned}$$

Expected future (discounted) reward when starting from state s and following policy π.

Optimal values

A policy π is better than π' (π≥π') iff:

$$V^{\pi}(s) \geq V^{\pi'}(s) \quad \forall s \in S$$

A **policy π*** is **optimal** iff it is better or equal to all other policies. The associated **optimal value function**, denoted V*, is defined as:

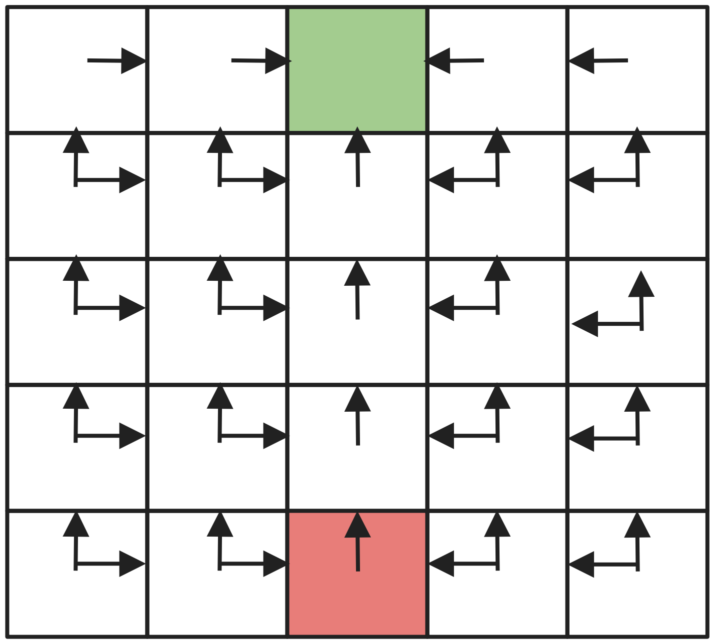
$$V^{*}(s) = max_{\pi} V^{\pi}(s) \quad \forall s \in S$$

Multiple optimal policies can exist, but they all share the same value function V*

Optimal values example

9	10	0	10	9
8.1	9	10	9	8.1
7.2	8.1	9	8.1	7.2
6.3	7.2	8.1	7.2	6.3
5.4	6.3	7.2	6.3	5.4

$V^*(s)$



$\pi^*(s)$

Q-values

Often it is easier to use state-action values (Q-values) rather than state values:

$$
\begin{aligned}
Q^\pi(s,a) &= E_\pi\{R_t \mid s_t = s, a_t = a\} \\
&= E_\pi\left\{\sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s, a_t = a\right\} \\
&= \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma V^\pi(s')]
\end{aligned}
$$

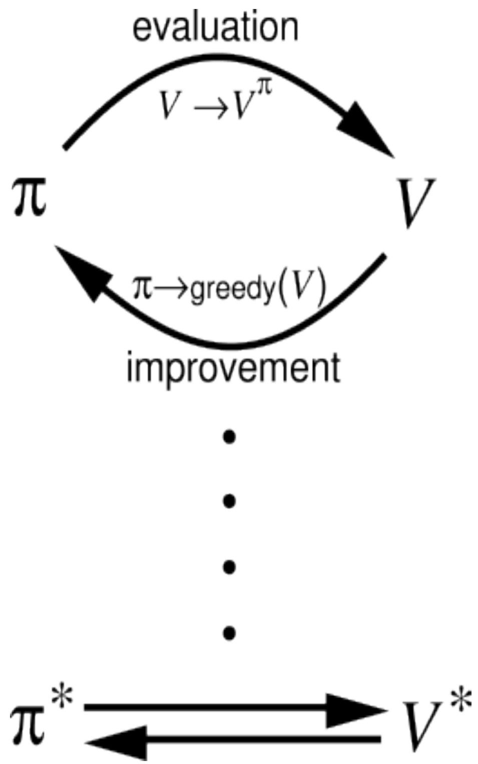
The optimal Q-values can be expressed as:

$$Q^*(s,a) = \sum_{s'} T(s,a,s')[R(s,a,s') + \gamma max_{a'} Q^*(s',a')]$$

Given Q*, the optimal policy can be obtained as follows:

$$\pi^*(s) = argmax_a Q^*(s,a)$$

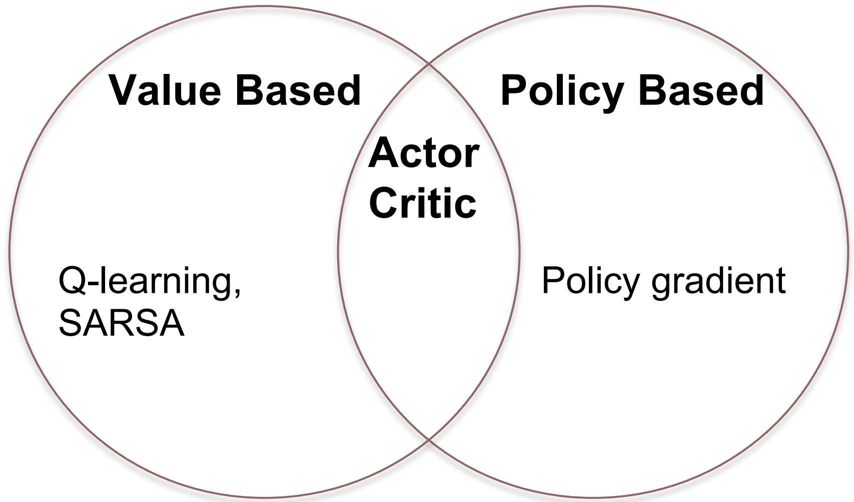
Policy iteration vs Value iteration



- Policy Iteration algorithms iterate **policy evaluation** and **policy improvement**.
- Value iteration algorithms directly construct a series of **estimates** in order to immediately learn the **optimal value function**.

Model-free RL Taxonomy

- **Value Based (Critic only):**
  - Learn Value Function
  - Policy is implicit (e.g. Greedy)
- **Policy Based (Actor only):**
  - Explicitly store Policy
  - Directly update Policy (e.g. using gradient, evolution, …)
- **Actor-Critic:**
  - Learn Policy
  - Learn Value function
  - Update policy using Value Function



Learning Values

- Goal: learn $V(s)$ / $Q(s,a)$ for some policy π from experience
- Recall that the (discounted) return is:

$$R_t = r_{t+1} + \gamma\, r_{t+2} + \gamma^2 r_{t+3} + \ldots. + \gamma^n r_T$$

- $V(s)$ is the expected value of this return over possible trajectories sampled by applying π

Learning Values

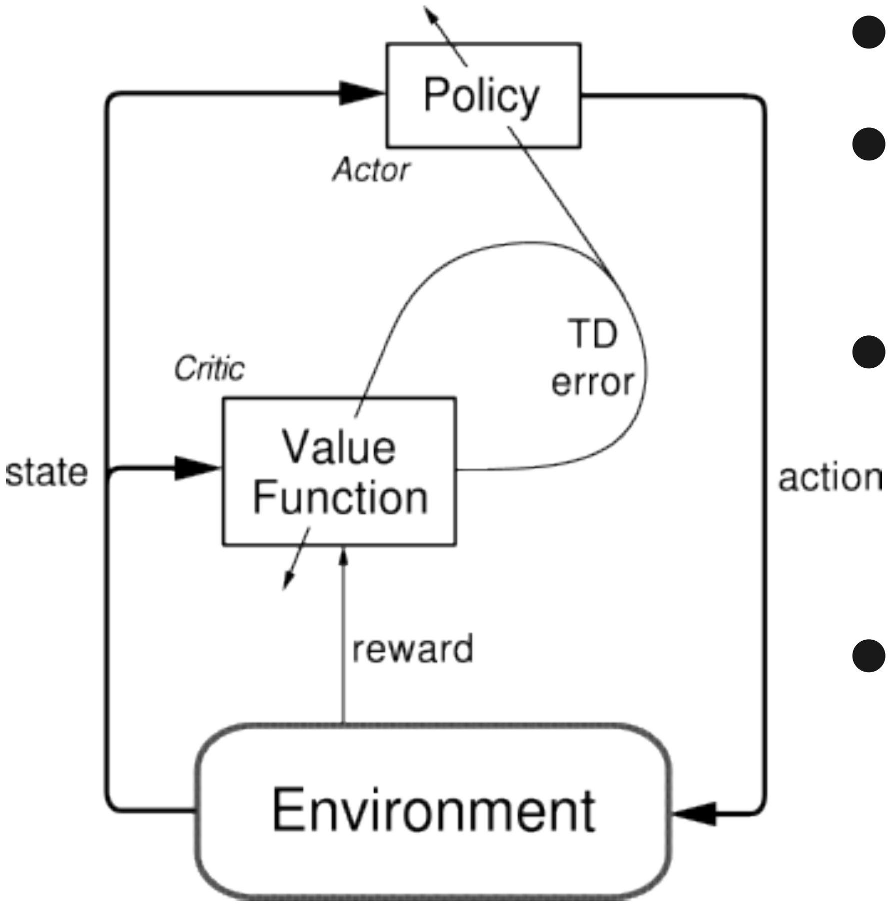
- Basic value learning can be described as:
  - Start in state s_t
  - Apply policy π
  - Observe new sample of return R_t
  - Update value estimate $\tilde{V}$ for s_t under π as:
    - $\tilde{V}(s_t) = \tilde{V}(s_t) + \alpha (R_t - \tilde{V}(s_t))$
    - Or: $Q(s_t,a_t) = Q(s_t,a_t) + \alpha (R_t - Q(s_t,a_t))$
  - Multiple possibilities to get sample R_t

Dimensions of RL

Some design decisions when selecting RL algorithms:

- Exploration vs Exploitation
- Monte carlo vs Bootstrapping
- On-policy vs Off-policy learning

Actor-Critic



- Policy iteration method
- Consists of 2 learners: actor and critic
- Critic learns evaluation (Values) for current policy
- Actor updates policy based on critic feedback

Actor-critic

Initialize $P_0(s, a)$, for all s, a

Initialize $V_0(s)$, for all s

Select s_0

For each step $t = 0, 1, 2, \ldots$:

  Derive a policy π_t from P_t (i.e. with exploration)

  Select a_t according to π_t (i.e. probability of selecting a is $\pi_t(s_t, a)$)

  Perform a_t, observe r_t, s_{t+1}

  If s_{t+1} is terminal:

$$P_{t+1}(s_t, a_t) \xleftarrow{\alpha_t} r_t - V_t(s_t) + P_t(s_t, a_t)$$

$$V_{t+1}(s_t) \xleftarrow{\beta_t} r_t$$

    Select new s_{t+1} (starting point for next episode)

  else:

$$P_{t+1}(s_t, a_t) \xleftarrow{\alpha_t} r_t + \gamma V_t(s_{t+1}) - V_t(s_t) + P_t(s_t, a_t)$$

$$V_{t+1}(s_t) \xleftarrow{\beta_t} r_t + \gamma V_t(s_{t+1})$$

Actor-critic

Initialize $P_0(s, a)$, for all s, a

Initialize $V_0(s)$, for all s

Select s_0

For each step $t = 0, 1, 2, \ldots$:

Derive a policy π_t from P_t (i.e. with exploration)

Select a_t according to π_t (i.e. probability of selecting a is $\pi_t(s_t, a)$)

Perform a_t, observe r_t, s_{t+1}

If s_{t+1} is terminal:

$$P_{t+1}(s_t, a_t) \xleftarrow{\alpha_t} r_t - V_t(s_t) + P_t(s_t, a_t)$$

$$V_{t+1}(s_t) \xleftarrow{\beta_t} r_t$$

Select new s_{t+1} (starting point for next episode)

else:

Actor: update using critic estimate

$$P_{t+1}(s_t, a_t) \xleftarrow{\alpha_t} \boxed{r_t + \gamma V_t(s_{t+1}) - V_t(s_t) + P_t(s_t, a_t)}$$

$$V_{t+1}(s_t) \xleftarrow{\beta_t} \boxed{r_t + \gamma V_t(s_{t+1})}$$

Critic: On-policy TD update

Exploration Vs. Exploitation

In online learning, where the system is actively controlled during learning, it is important to balance **exploration** and **exploitation**

- **Exploration** means **trying new actions** in order to observe their results. It is needed to learn and discover good actions
- **Exploitation** means using what was **already learnt**: select actions known to be good in order to obtain high rewards.
- Common choices: greedy, e-greedy, softmax

Greedy Action selection

- always select action with highest Q-value

$$a = \text{argmax}_{a}\, Q(s,a)$$

- Pure exploitation, no exploration
- Will immediately converge to action if observed value is higher than initial Q-values
- Can be made to explore by initializing Q-values optimistically

ε-greedy

- With probability ε select random action, else select greedy
- Fixed rate of exploration for fixed ε
- ε can be reduced over time to reduce amount of exploration

Softmax

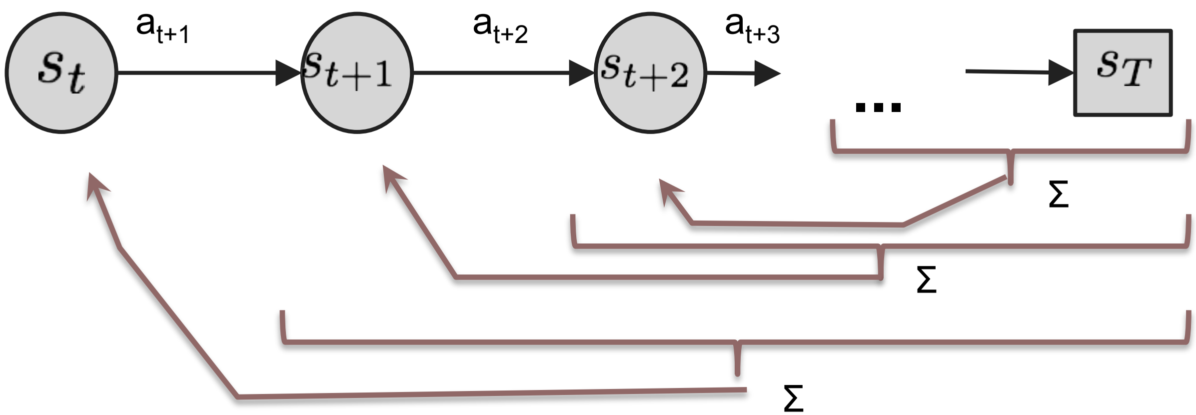
- Assign each action a probability, based on Q-value:

$$P(s,a) = \frac{e^{\frac{Q(s,a)}{T}}}{\sum_b e^{\frac{Q(s,b)}{T}}}$$

- Parameter T determines amount of exploration. Large T: play more randomly, small T: play greedily (T can also be reduced over time)

Sampling returns

- Apply policy, observe complete return, update estimate
- Sample the actual returns and calculate the empirical mean
- This is called **Monte Carlo** estimation



Repeat this for multiple episodes and average

Monte Carlo

- Monte Carlo sampling gives an unbiased estimate of the values
- Estimates converge to true value, but:
  - Only updates at the end of an episode
    - (continuous problems? -> see later)
  - High variance (noisy, many samples needed)
  - Typically provides slow learning

Bootstrapping

- **Monte Carlo** updates use the complete return over the remainder of the episode:

$$R_t = r_{t+1} + \gamma \underbrace{r_{t+2} + \gamma^2 r_{t+3} + \ldots + \gamma^n r_T}_{R_{t+1}\text{: sample for } V(s_{t+1})}$$

- **Bootstrapping** updates update after single step:

$$R_t = r_{t+1} + \gamma \tilde{V}(s_{t+1})$$

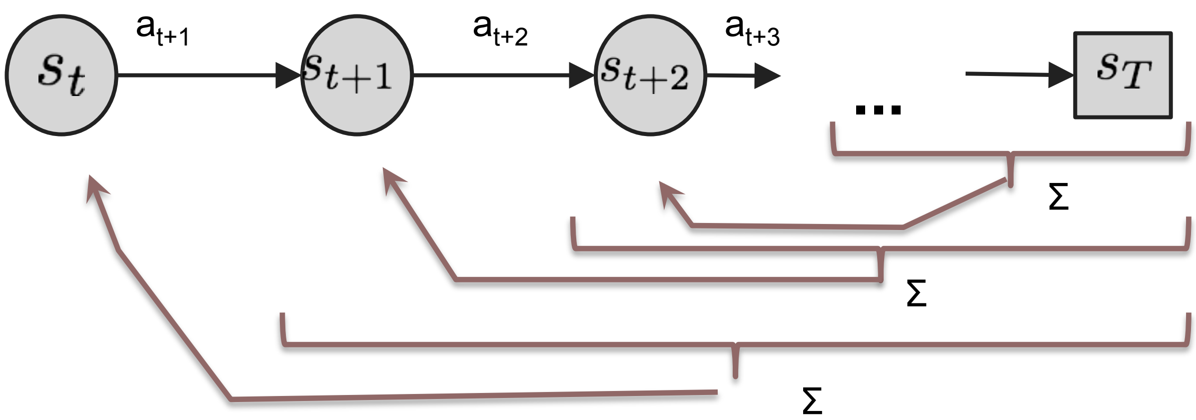
Future returns are approximated using the estimated value of the next state.

Bootstrapping

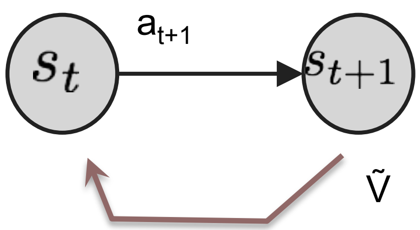
- Using bootstrapping updates:
  - Lower variance than Monte Carlo (typically learns faster)
  - Biased estimate of the Return
  - Convergences to true values (in finite discrete case)
  - Can be sensitive to initializations

Bootstrapping Vs. Monte Carlo

- Monte Carlo: Complete episode, then update s_t using rewards over remainder of episode



- Bootstrapping: Take 1 step, then update s_t using estimate of $V(s_{t+1})$



On-policy vs Off-Policy

- On-policy learning estimates values for behaviour policy
- Off-policy learning can learn values for any target policy:
  - More flexible
  - No need to execute target policy
  - Can reuse samples
  - Learn rom demonstrations
  - Allows for multiple target policies
  - Can lead to problems when used with approximation

SARSA & Q-learning

- 2 algorithms for on-line Temporal Difference (TD) control
- Learn Q-values while actively controlling system
- Both use **TD error** to update value function estimates:

$$\delta = [r_{t+1} + \gamma V(s_{t+1}) - V(s_t)]$$

- Both algorithms use **bootstrapping**: Q-value estimates are updated using using estimates for the next state
- Use different estimates for the next state value $V(s_{t+1})$
- SARSA is **on-policy**: learns value Q^{π} for active control policy π
- Q-learning is **off-policy**: learns Q*, regardless of control policy that is used

SARSA

Initialize $Q(s, a)$ arbitrarily
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)
        $Q(s, a) \leftarrow Q(s, a) + \alpha\big[r + \gamma Q(s', a') - Q(s, a)\big]$
        $s \leftarrow s'$; $a \leftarrow a'$;
    until s is terminal

SARSA

exploration

Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Choose a from s using $\boxed{\text{policy derived from } Q \text{ (e.g., } \varepsilon\text{-greedy)}}$
  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)
    $Q(s,a) \leftarrow Q(s,a) + \alpha\big[r + \gamma \boxed{Q(s',a')} - Q(s,a)\big]$
    $s \leftarrow s'$; $a \leftarrow a'$;
  until s is terminal

On-policy: $V(s_{t+1}) = Q(s_{t+1}, a_{t+1})$

bootstrapping

Q-Learning

Initialize $Q(s,a)$ arbitrarily
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'
        $Q(s,a) \leftarrow Q(s,a) + \alpha\big[r + \gamma \max_{a'} Q(s',a') - Q(s,a)\big]$
        $s \leftarrow s'$;
    until s is terminal

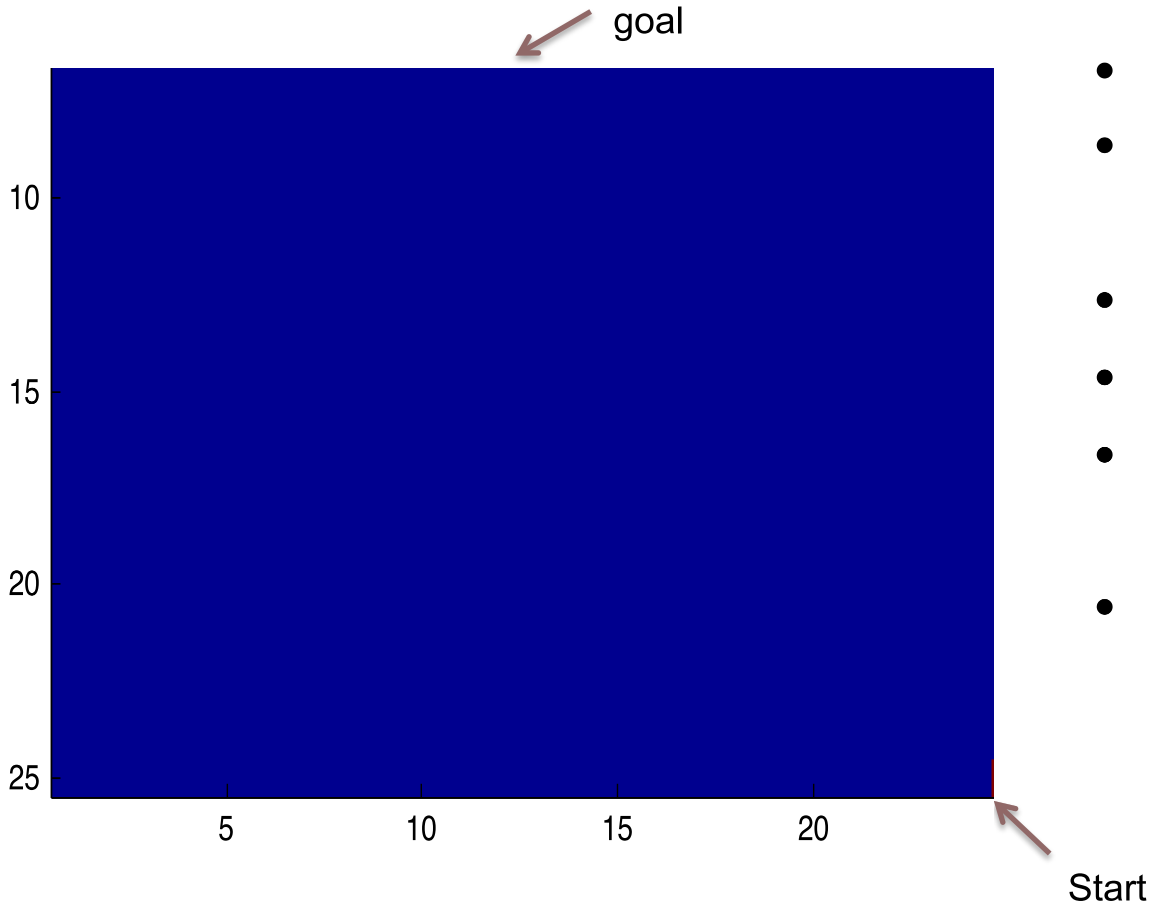
Q-Learning

Initialize $Q(s,a)$ arbitrarily
Repeat (for each episode):
  Initialize s
  Repeat (for each step of episode):
    Choose a from s using $\boxed{\text{policy derived from } Q \text{ (e.g., } \varepsilon\text{-greedy)}}$
    Take action a, observe r, s'
    $Q(s,a) \leftarrow Q(s,a) + \alpha\left[r + \gamma \boxed{\max_{a'} Q(s',a')} - Q(s,a)\right]$
    $s \leftarrow s'$;
  until s is terminal

Q-learning: policy does not have to depend on Q

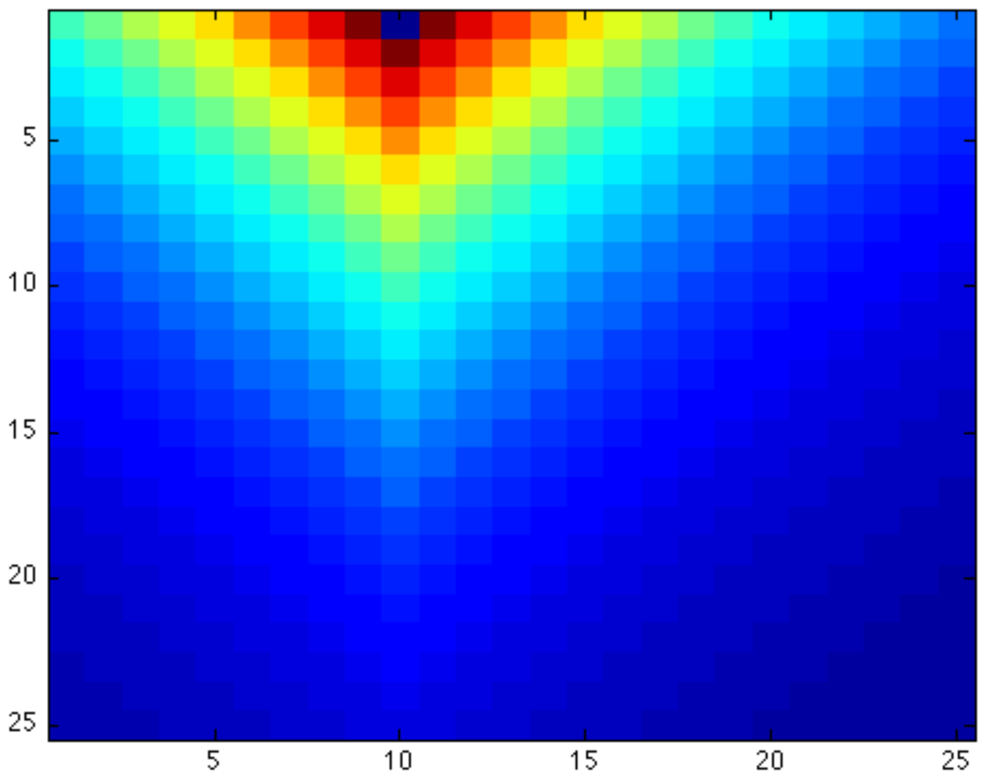
$\boxed{\text{Off-policy: } V(s_{t+1}) = \max_{a'} Q(s_{t+1}, a')}$

Monte Carlo vs Bootstrapping

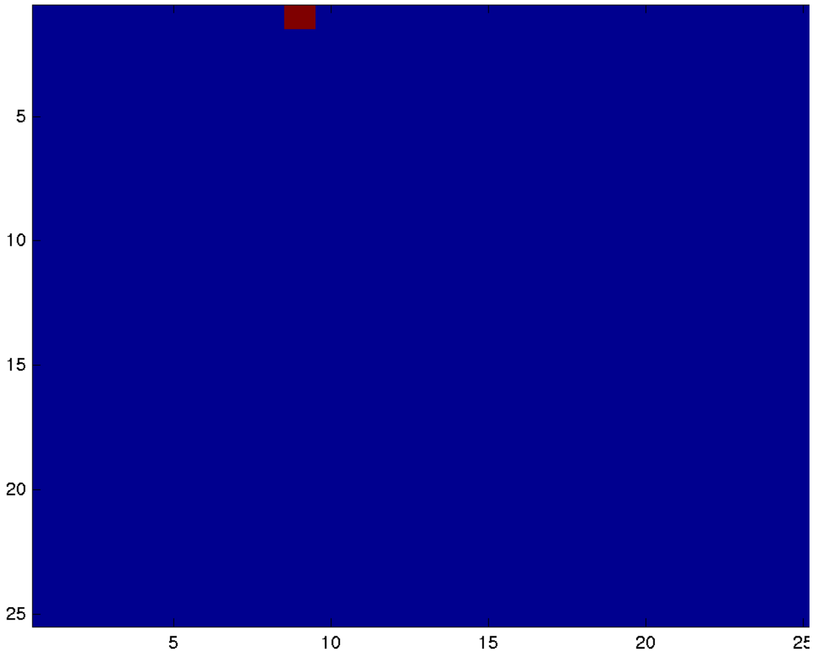


- 25 x 25 grid world
- +100 reward for reaching goal
- 0 reward else
- discount = 0.9
- Q-learning with 0.9 learning rate
- Monte carlo updates vs bootstrapping

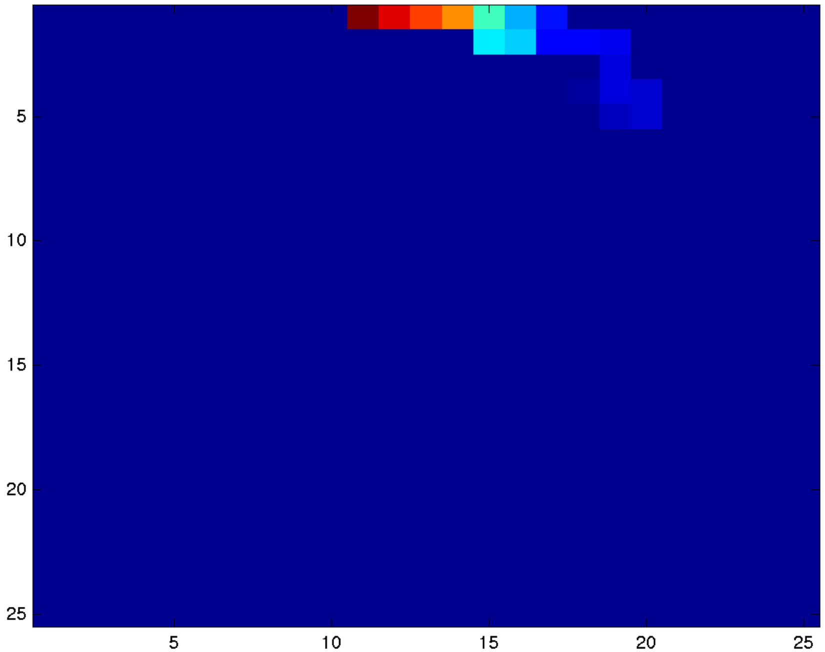
Optimal Value function



Monte Carlo vs Bootstrapping



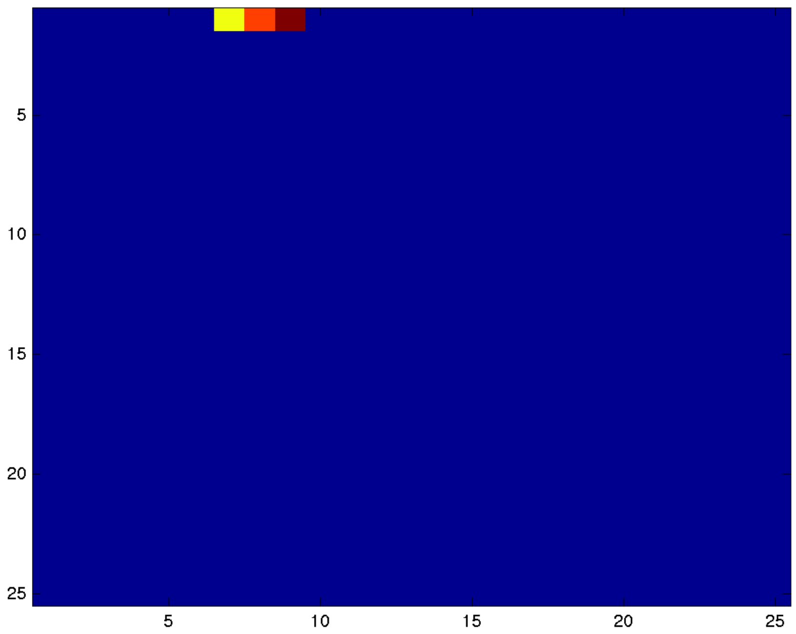
Bootstrapping



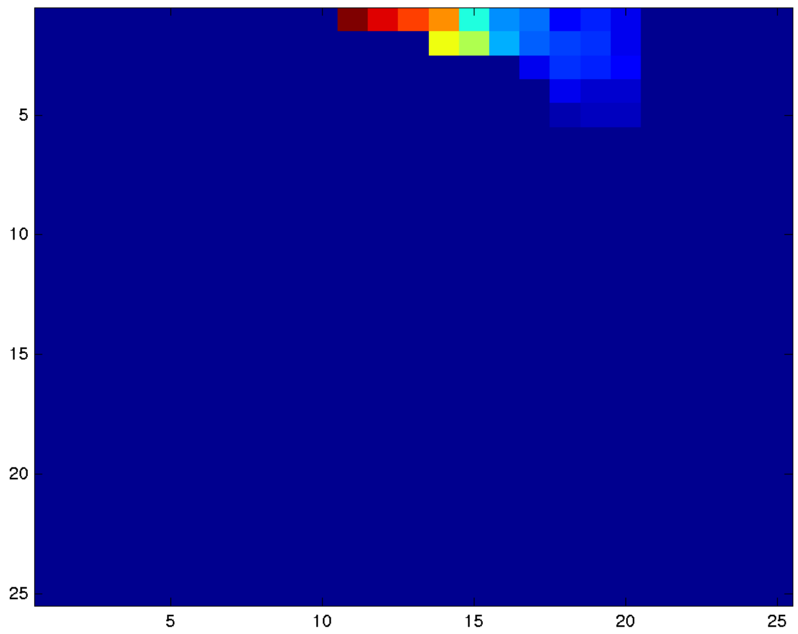
Monte Carlo

Episode 1

Monte Carlo vs Bootstrapping



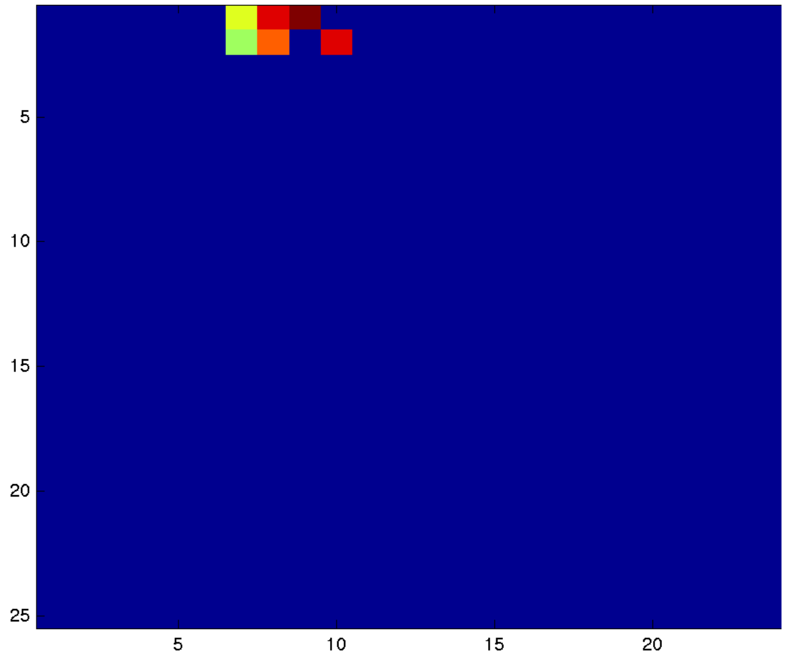
Bootstrapping



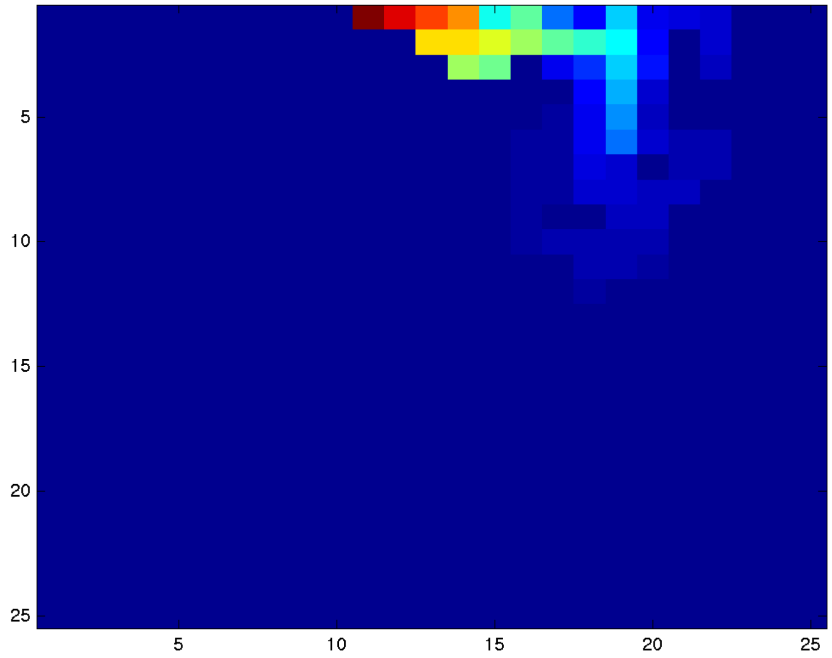
Monte Carlo

Episode 2

Monte Carlo vs Bootstrapping



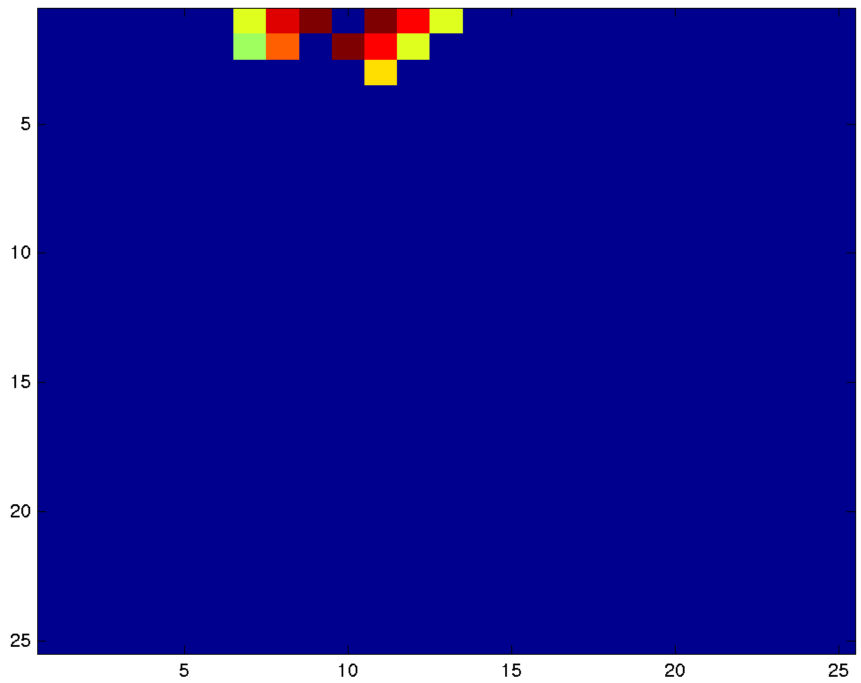
Bootstrapping



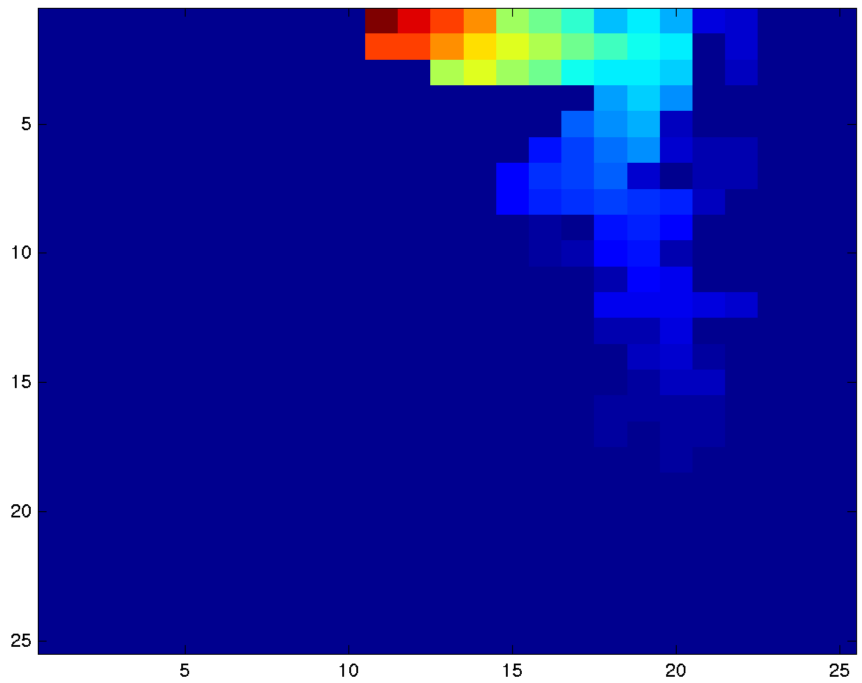
Monte Carlo

Episode 5

Monte Carlo vs Bootstrapping



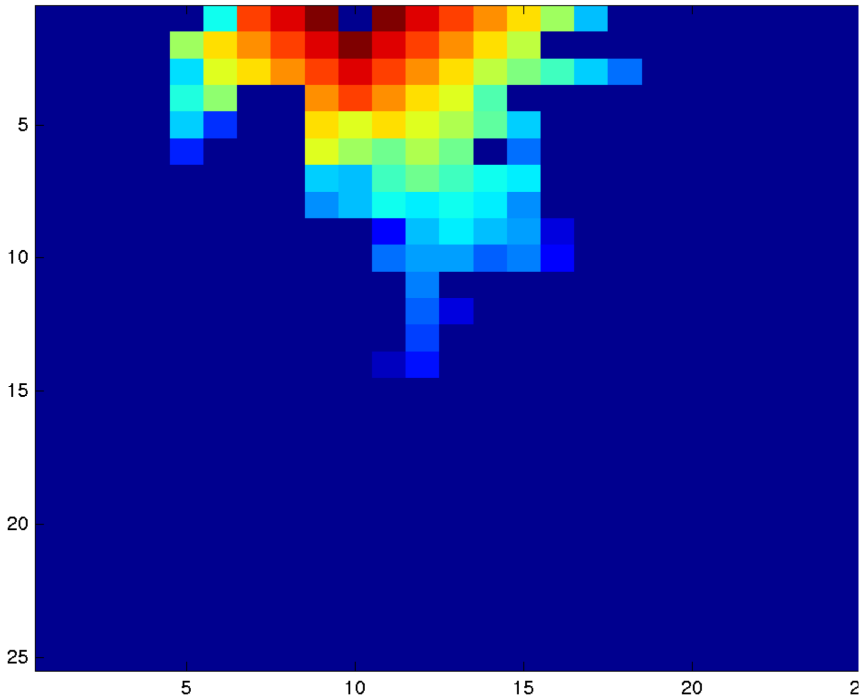
Bootstrapping



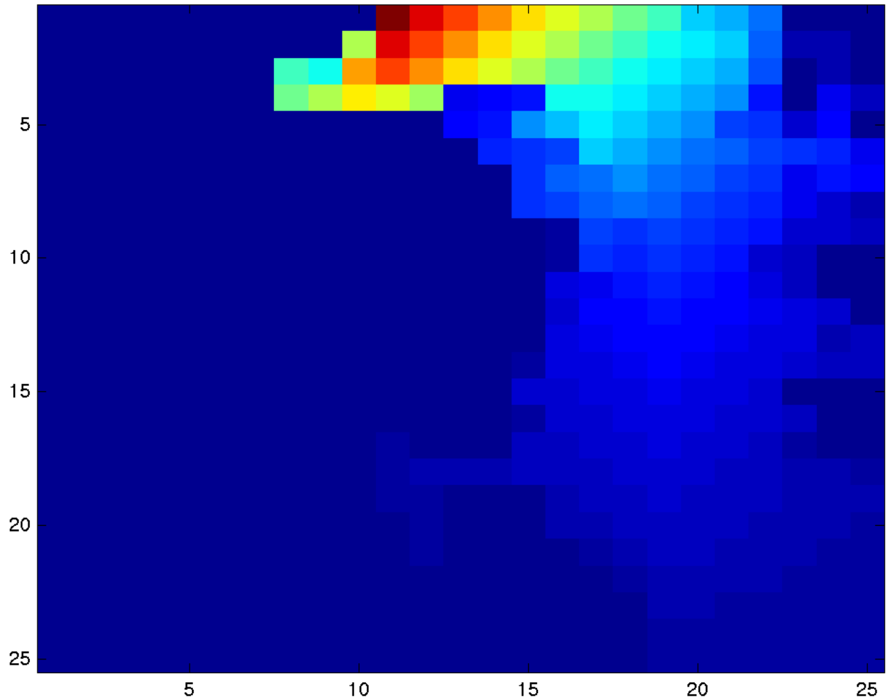
Monte Carlo

Episode 10

Monte Carlo vs Bootstrapping



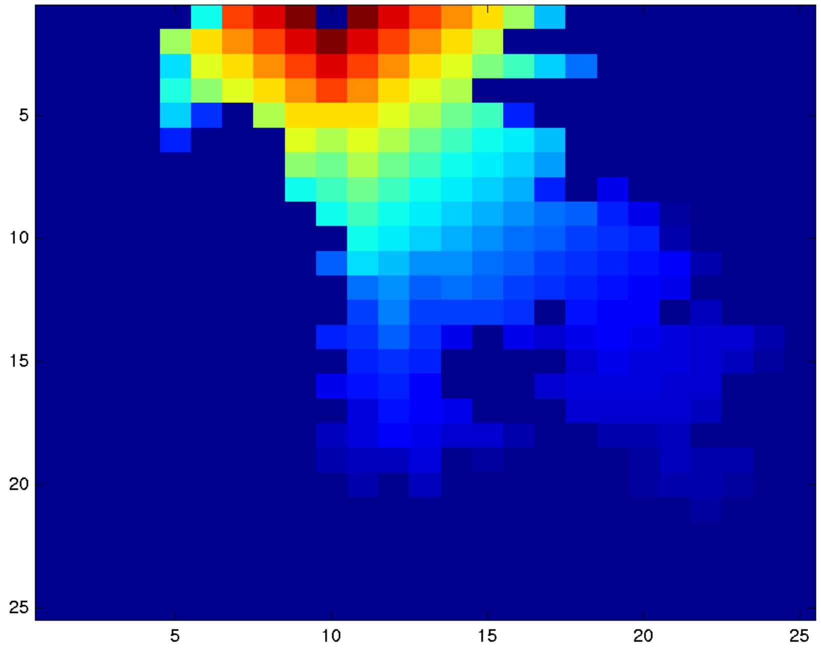
Bootstrapping



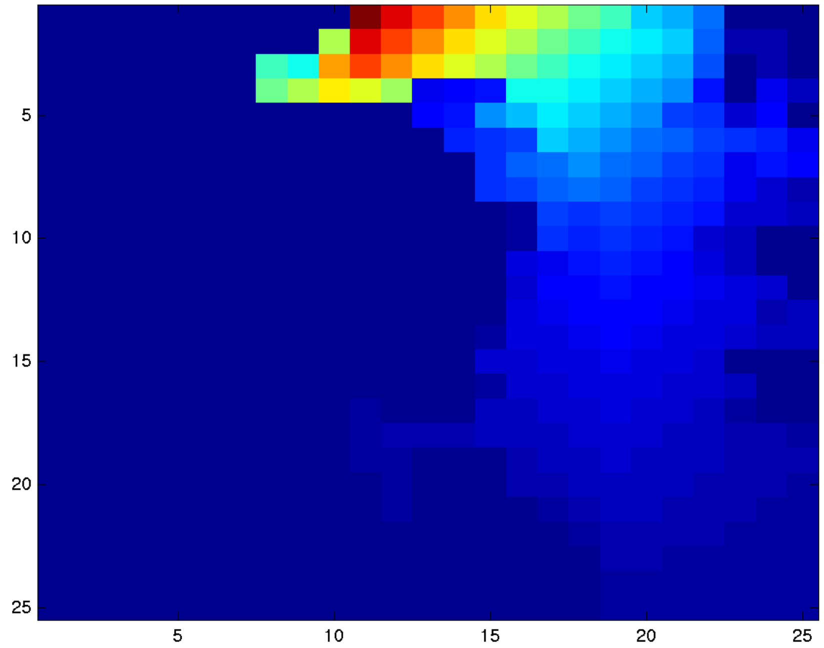
Monte Carlo

Episode 50

Monte Carlo vs Bootstrapping



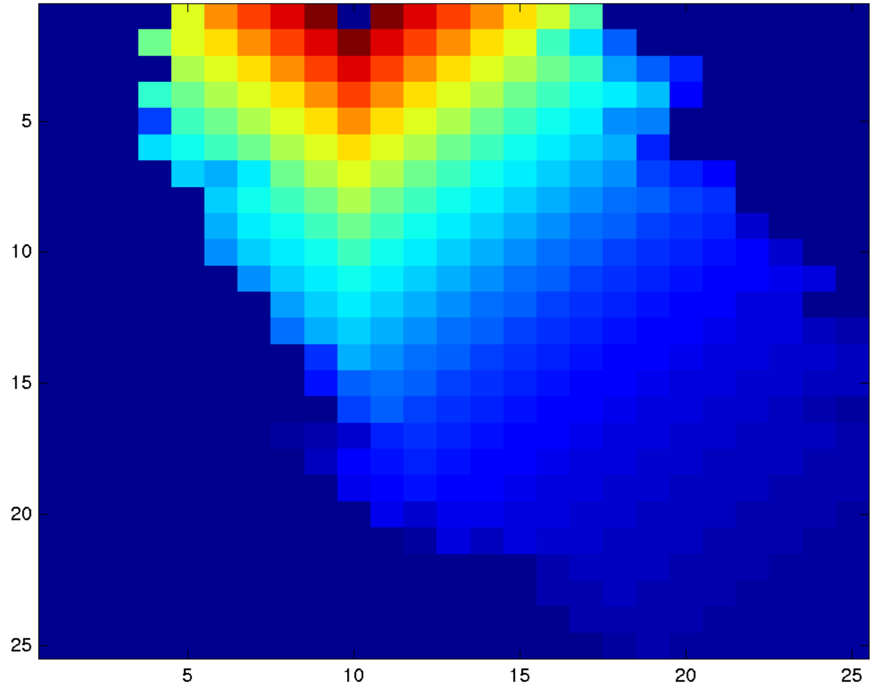
Bootstrapping



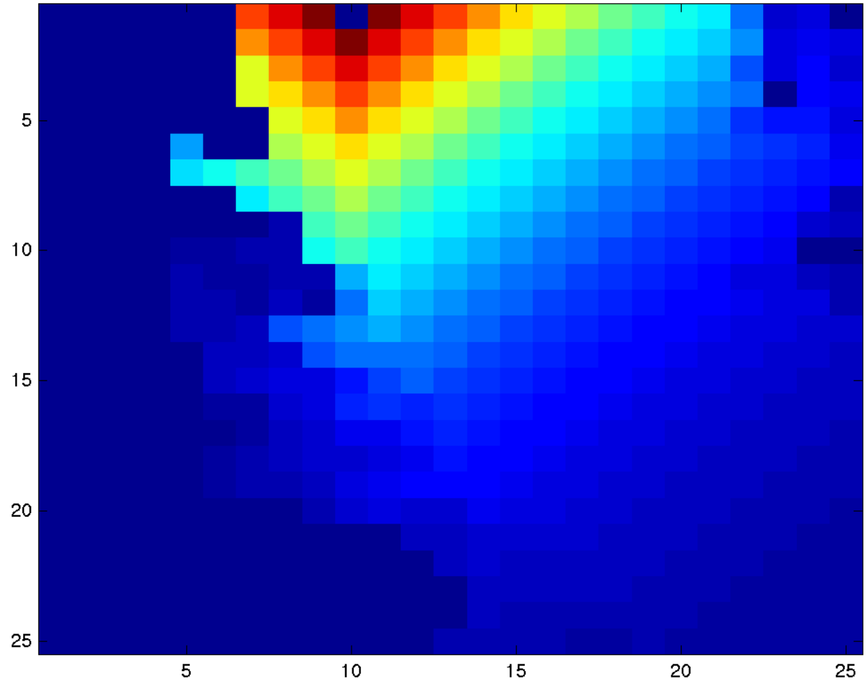
Monte Carlo

Episode 100

Monte Carlo vs Bootstrapping



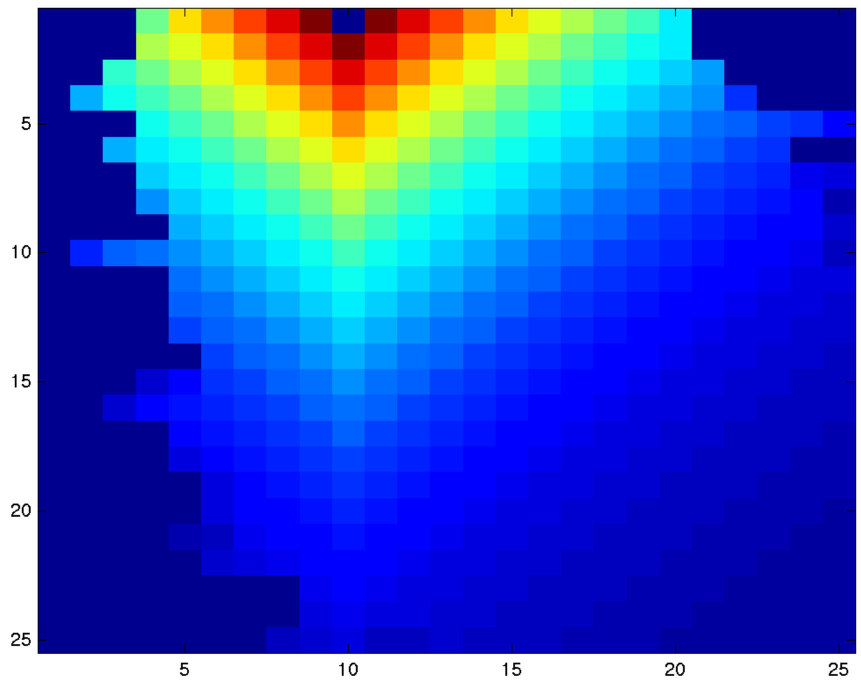
Bootstrapping



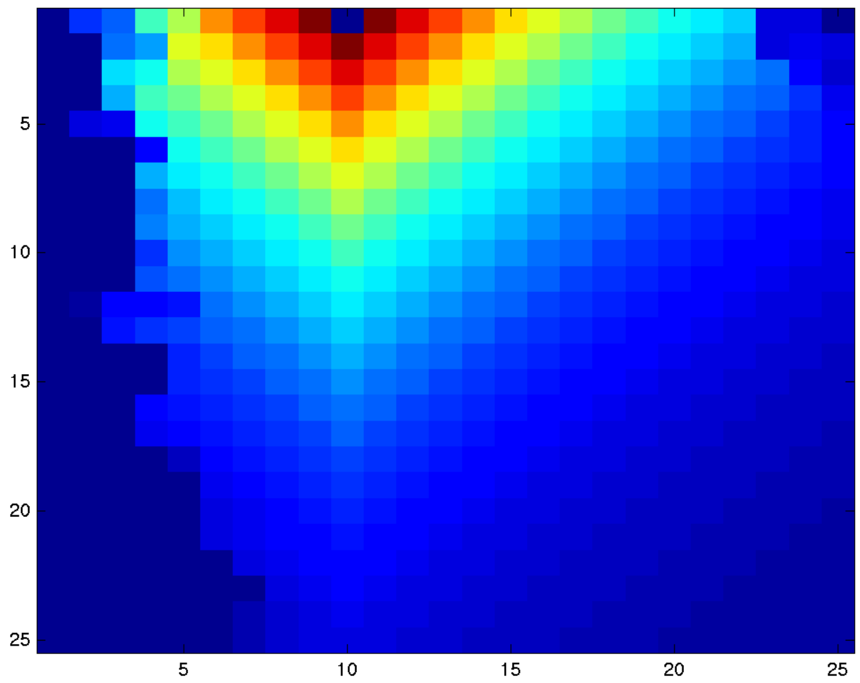
Monte Carlo

Episode 1000

Monte Carlo vs Bootstrapping



Bootstrapping



Monte Carlo

Episode 10000

N-step returns

$$R_t^{(1)} = r_{t+1} + \gamma V_t(s_{t+1}).$$

Bootstrapping

$$R_t^{(2)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 V_t(s_{t+2}),$$

...

$$R_t^{(n)} = r_{t+1} + \gamma r_{t+2} + \gamma^2 + \cdots + \gamma^{n-1} r_{t+n} + \gamma^n V_t(s_{t+n}).$$

...

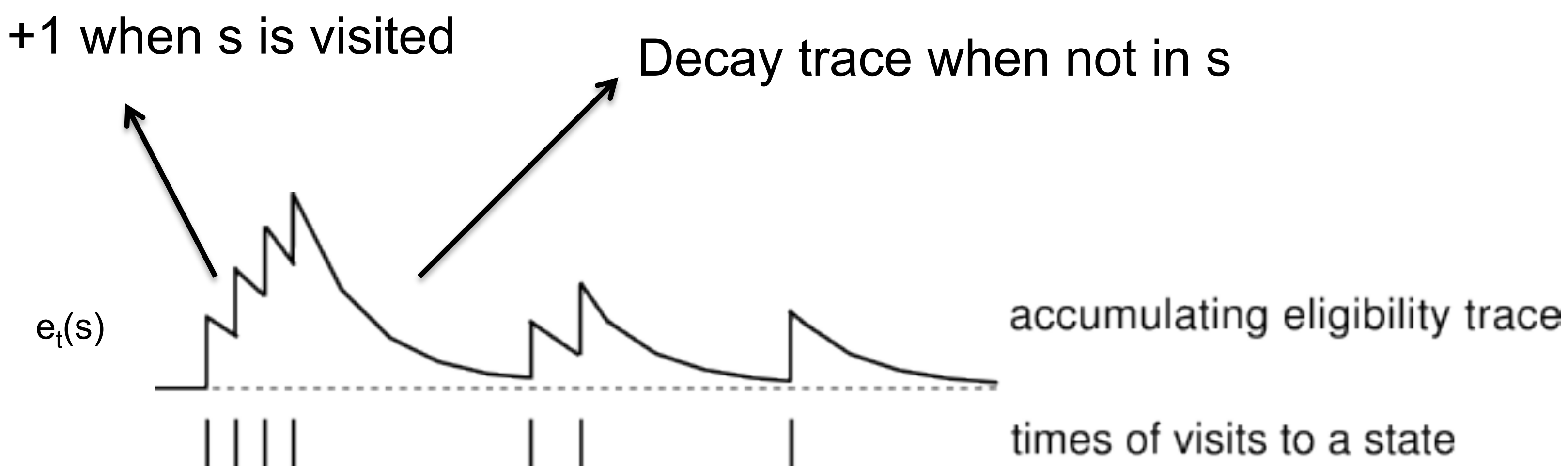
$$R_t = r_{t+1} + \gamma r_{t+2} + \gamma^2 r_{t+3} + \cdots + \gamma^{T-t-1} r_T,$$

Monte Carlo

Eligibility Traces

- Idea: after receiving a reward states (or state action pairs) are updated depending on how recently they were visited
- A trace value e(s,a) is kept for each (s,a) pair. This value is increased when (s,a) is visited and decayed else.
- The TD update for a state is weighted by e(s,a)
- (Almost) equivalent to using n-step return

Eligibility traces (2)



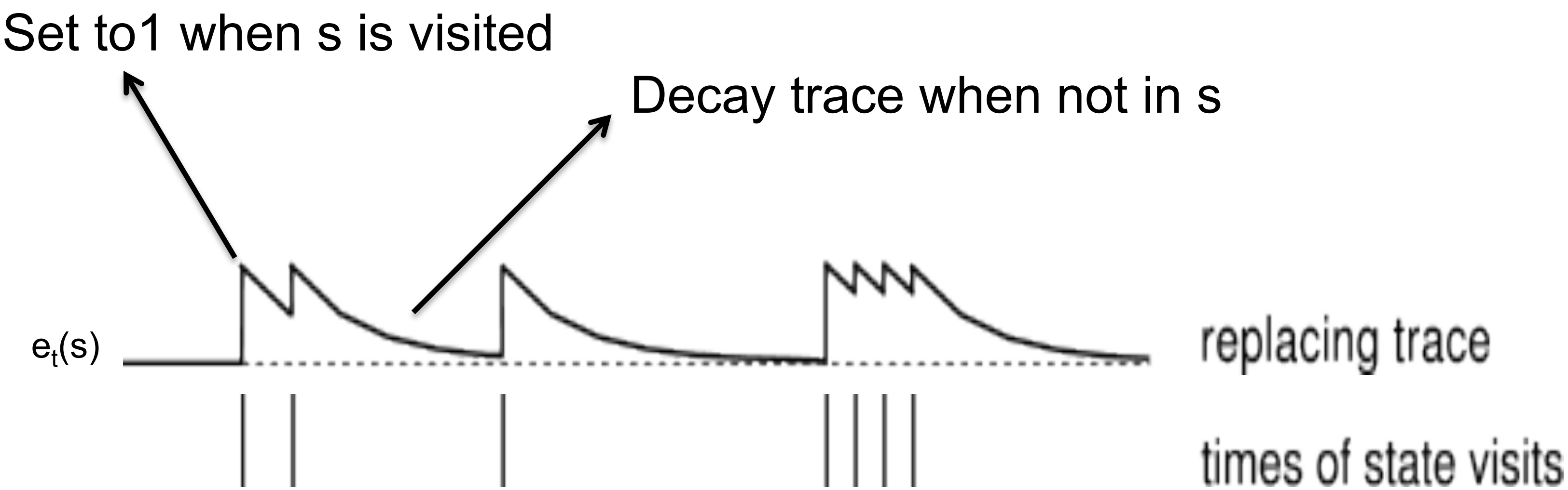
$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & \text{if } s \neq s_t; \\ \gamma\lambda e_{t-1}(s) + 1 & \text{if } s = s_t, \end{cases}$$

λ determines trace decay:

λ = 0: bootstrapping

λ = 1: Monte Carlo

Replacing Traces



$$e_t(s) = \begin{cases} \gamma\lambda e_{t-1}(s) & \text{if } s \neq s_t; \\ 1 & \text{if } s = s_t. \end{cases}$$

Typically more stable than accumulating traces

SARSA(λ)

Initialize $Q(s, a)$ arbitrarily and $e(s, a) = 0$, for all s, a
Repeat (for each episode):
  Initialize s, a
  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)
    $\delta \leftarrow r + \gamma Q(s', a') - Q(s, a)$
    $e(s, a) \leftarrow e(s, a) + 1$
    For all s, a:
      $Q(s, a) \leftarrow Q(s, a) + \alpha \delta e(s, a)$
      $e(s, a) \leftarrow \gamma \lambda e(s, a)$
    $s \leftarrow s'$; $a \leftarrow a'$
  until s is terminal

Q(λ)

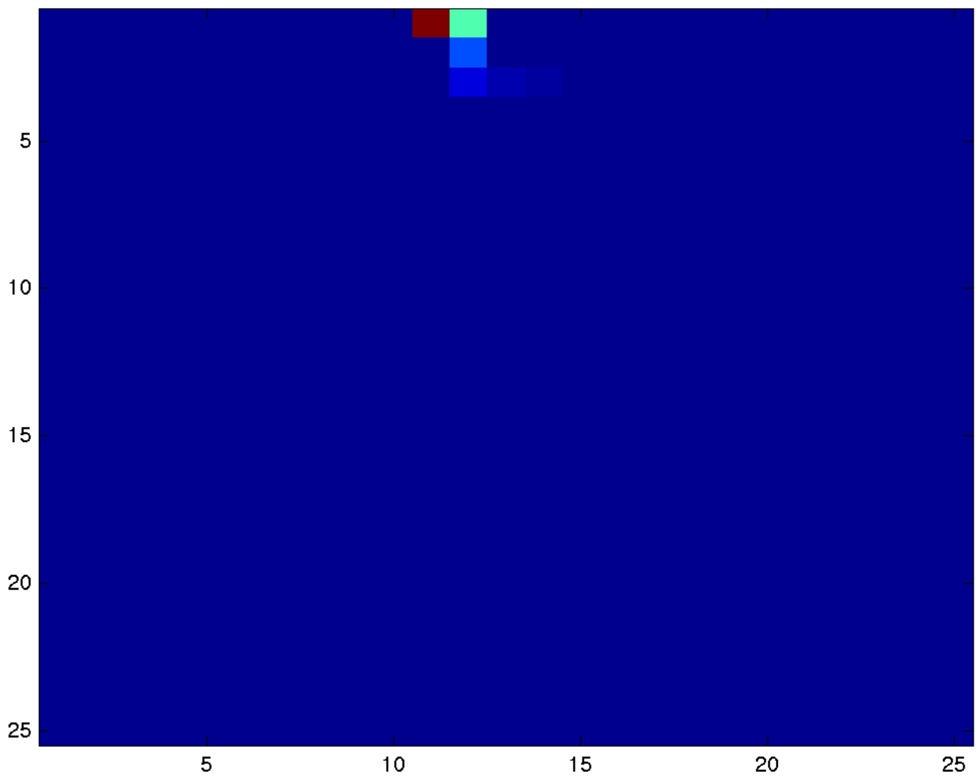
Initialize $Q(s,a)$ arbitrarily and $e(s,a) = 0$, for all s, a
Repeat (for each episode):
    Initialize s, a
    Repeat (for each step of episode):
        Take action a, observe r, s'
        Choose a' from s' using policy derived from Q (e.g., ε-greedy)
        $a^* \leftarrow \arg\max_b Q(s', b)$ (if a' ties for the max, then $a^* \leftarrow a'$)
        $\delta \leftarrow r + \gamma Q(s', a^*) - Q(s,a)$
        $e(s,a) \leftarrow e(s,a) + 1$
        For all s, a:
            $Q(s,a) \leftarrow Q(s,a) + \alpha\delta e(s,a)$
            If $a' = a^*$, then $e(s,a) \leftarrow \gamma\lambda e(s,a)$
                else $e(s,a) \leftarrow 0$
        $s \leftarrow s'$; $a \leftarrow a'$
    until s is terminal

Q(λ)

Initialize $Q(s,a)$ arbitrarily and $e(s,a) = 0$, for all s, a
Repeat (for each episode):
    Initialize s, a
    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)
        $a^* \leftarrow \arg\max_b Q(s', b)$ (if a' ties for the max, then $a^* \leftarrow a'$)
        $\delta \leftarrow r + \gamma Q(s', a^*) - Q(s,a)$
        $e(s,a) \leftarrow e(s,a) + 1$
        For all s, a:
            $Q(s,a) \leftarrow Q(s,a) + \alpha \delta e(s,a)$
            If $a' = a^*$, then $e(s,a) \leftarrow \gamma \lambda e(s,a)$
                else $e(s,a) \leftarrow 0$
        $s \leftarrow s'$; $a \leftarrow a'$
    until s is terminal

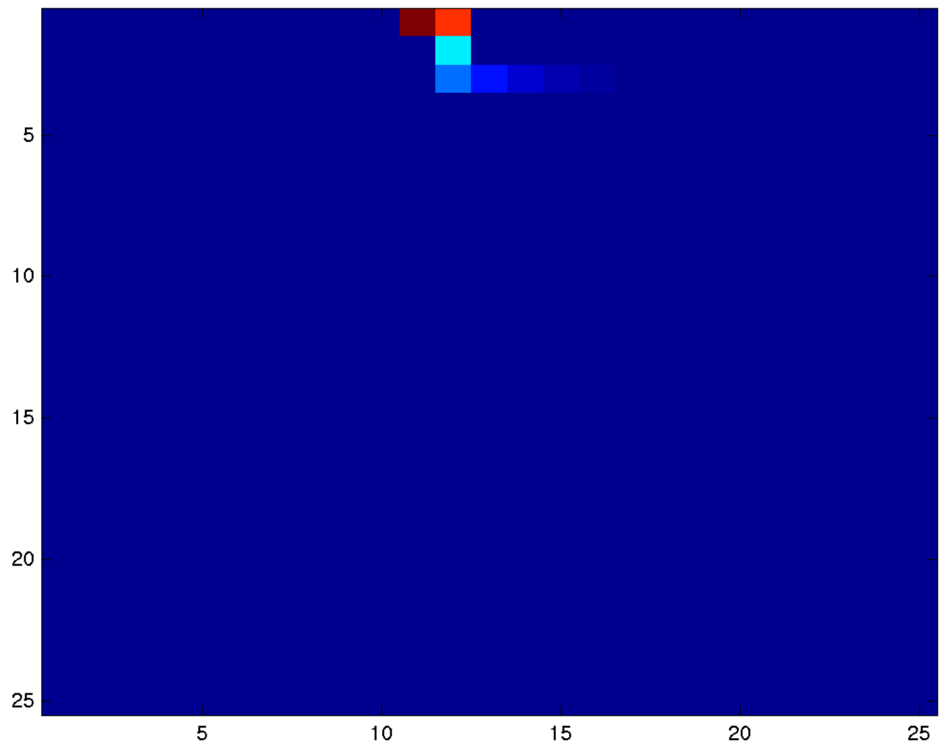
Reset trace when non-greedy action is selected

Q(0.5)



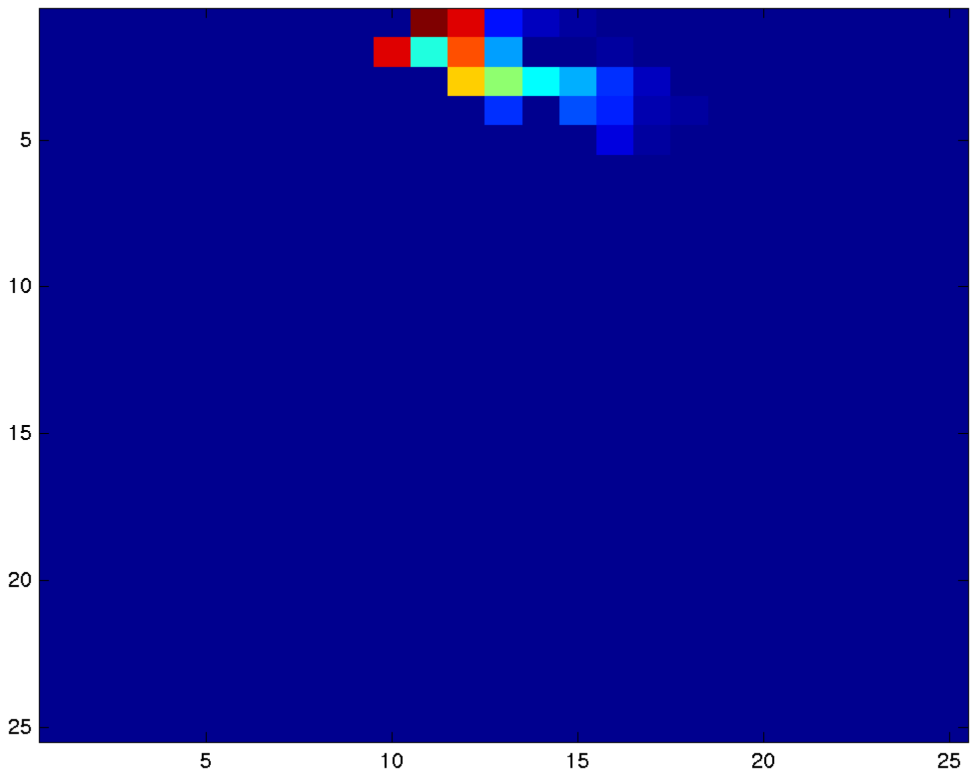
Episode 1

Q(0.5)



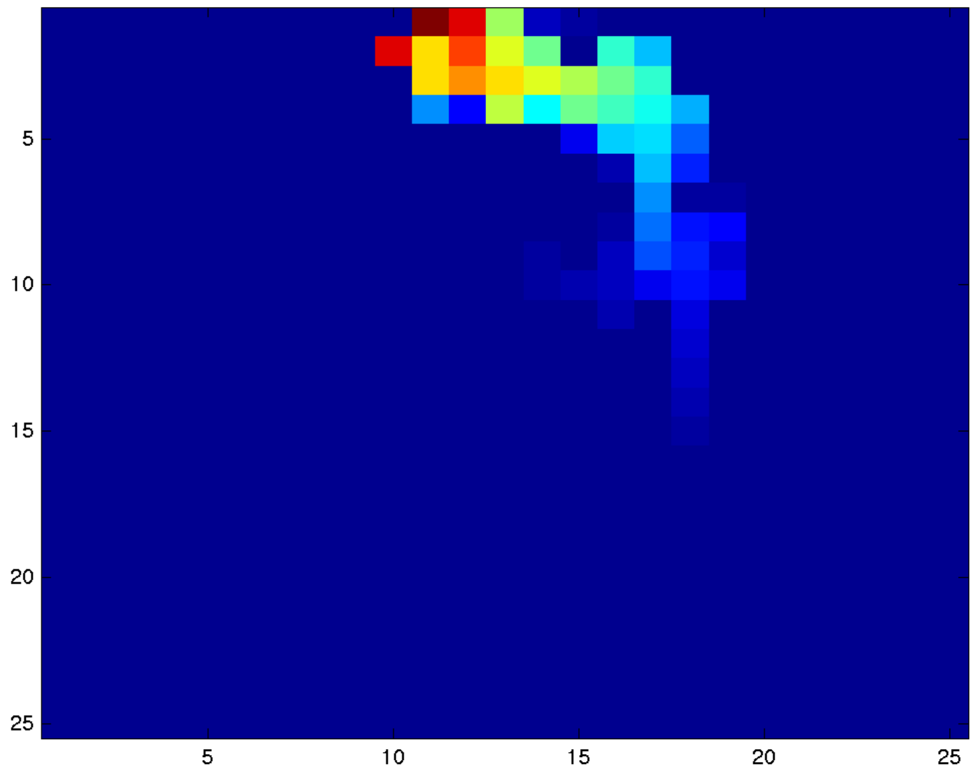
Episode 2

Q(0.5)



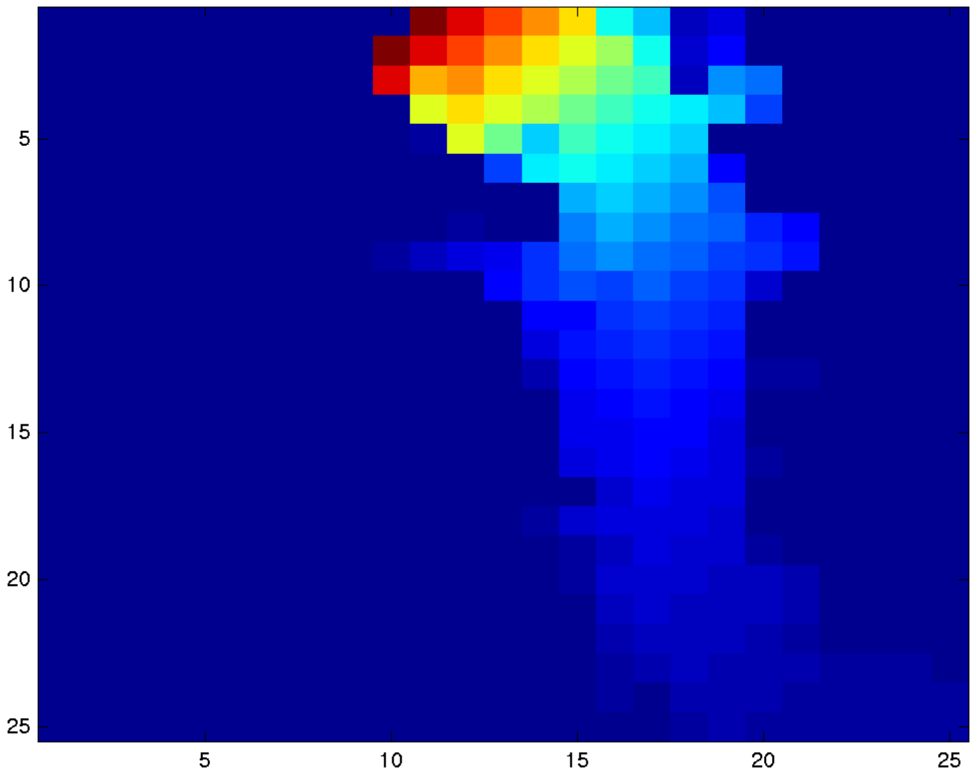
Episode 5

Q(0.5)



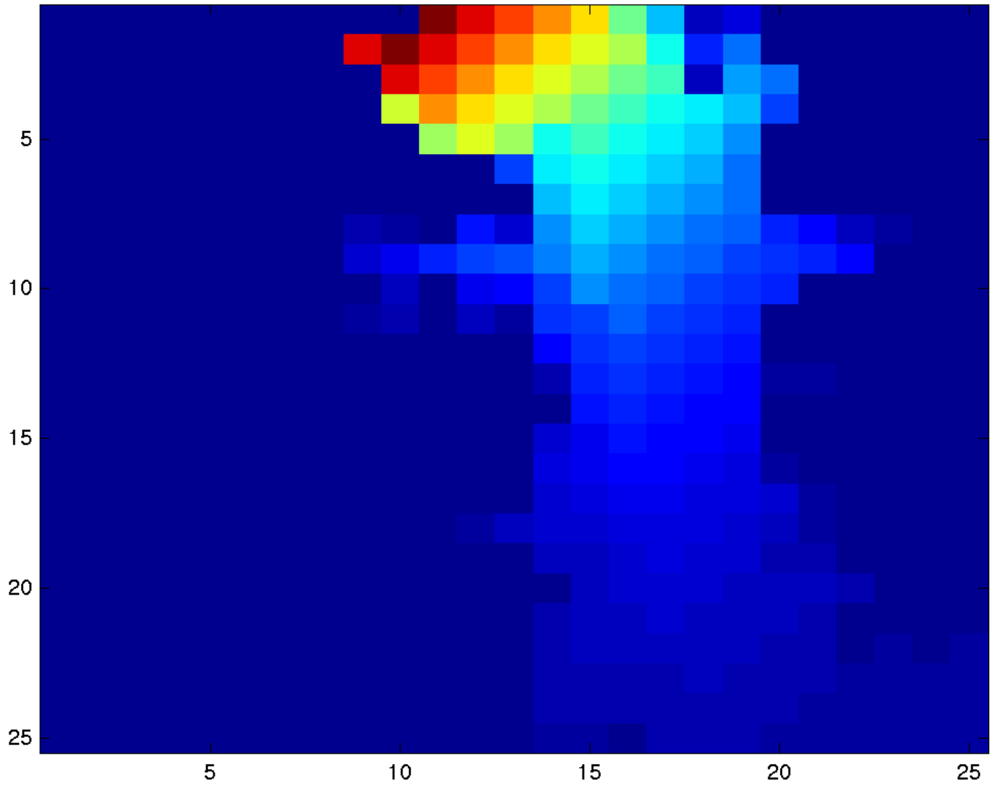
Episode 10

Q(0.5)



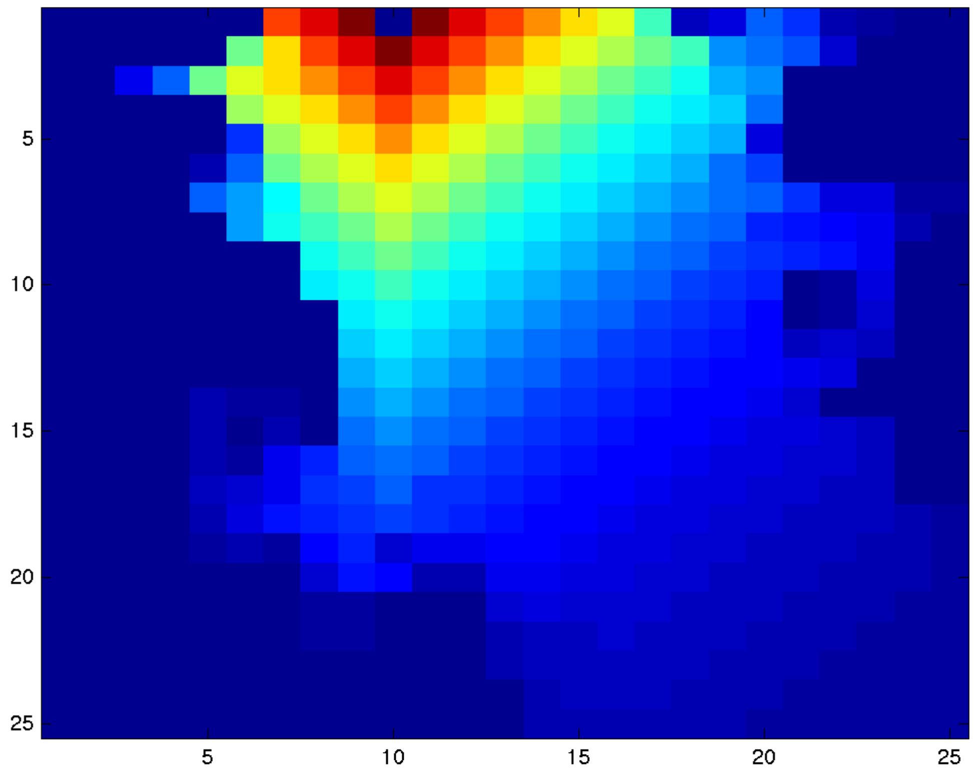
Episode 50

Q(0.5)



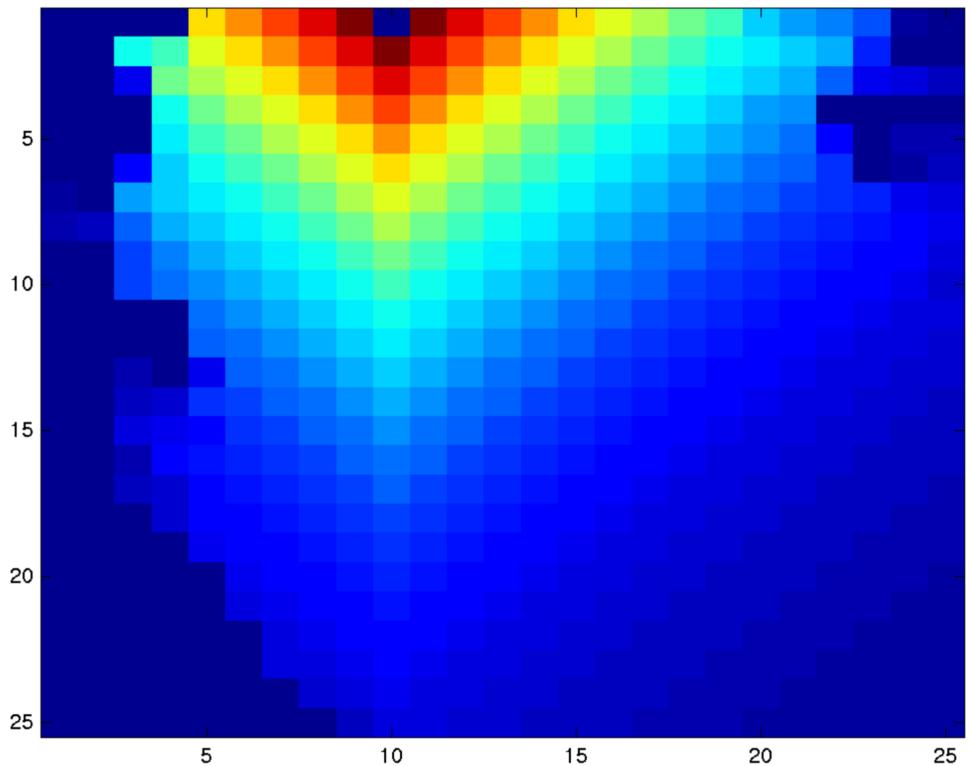
Episode 100

Q(0.5)



Episode 1000

Q(0.5)



Episode 10000

Using traces

- Setting λ allows full range of backups from monte carlo (λ=1) to bootstrapping (λ=0)
- Intermediate approaches often more efficient than extreme λs (1 or 0)
- Often easier to reason about #steps trace will last:

$$\tau = \frac{1}{1-\lambda}$$

- Offer a method to apply Monte Carlo methods in non-episodic tasks

Optimal λ values

Steps per episode
averaged over
first 20 trials
and 30 runs