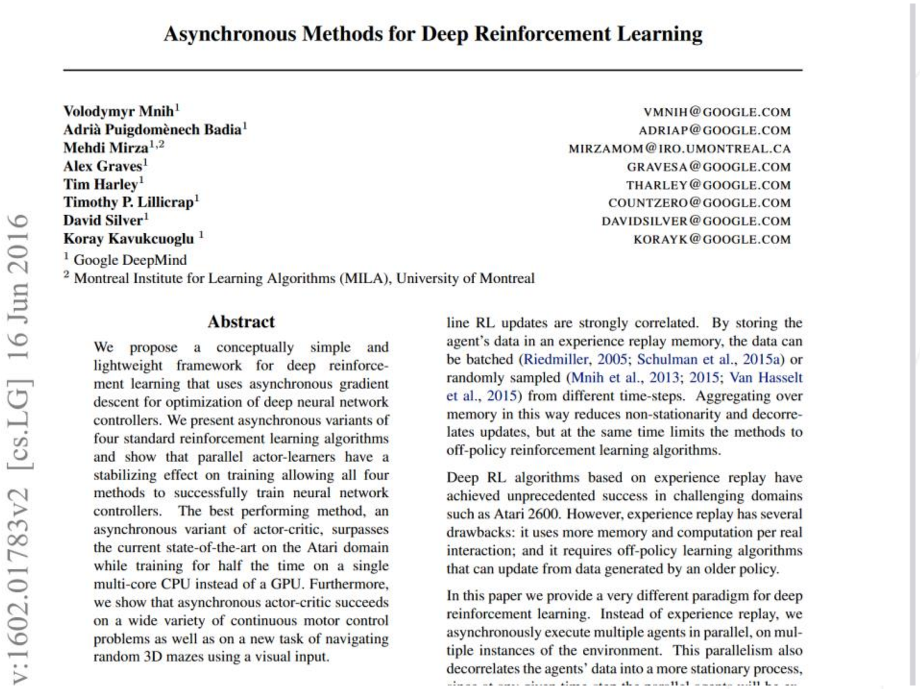
Asynchronous RL Methods

Anne Elster
Lain Mustafaoglu
Keshav Pingali

What this talk about



Asynchronous Methods for Deep Reinforcement Learning

Volodymyr Mnih[1] VMNIH@GOOGLE.COM
Adrià Puigdomènech Badia[1] ADRIAP@GOOGLE.COM
Mehdi Mirza[1,2] MIRZAMOM@IRO.UMONTREAL.CA
Alex Graves[1] GRAVESA@GOOGLE.COM
Tim Harley[1] THARLEY@GOOGLE.COM
Timothy P. Lillicrap[1] COUNTZERO@GOOGLE.COM
David Silver[1] DAVIDSILVER@GOOGLE.COM
Koray Kavukcuoglu [1] KORAYK@GOOGLE.COM
[1] Google DeepMind
[2] Montreal Institute for Learning Algorithms (MILA), University of Montreal

v:1602.01783v2 [cs.LG] 16 Jun 2016

Abstract

We propose a conceptually simple and lightweight framework for deep reinforcement learning that uses asynchronous gradient descent for optimization of deep neural network controllers. We present asynchronous variants of four standard reinforcement learning algorithms and show that parallel actor-learners have a stabilizing effect on training allowing all four methods to successfully train neural network controllers. The best performing method, an asynchronous variant of actor-critic, surpasses the current state-of-the-art on the Atari domain while training for half the time on a single multi-core CPU instead of a GPU. Furthermore, we show that asynchronous actor-critic succeeds on a wide variety of continuous motor control problems as well as on a new task of navigating random 3D mazes using a visual input.

line RL updates are strongly correlated. By storing the agent's data in an experience replay memory, the data can be batched (Riedmiller, 2005; Schulman et al., 2015a) or randomly sampled (Mnih et al., 2013; 2015; Van Hasselt et al., 2015) from different time-steps. Aggregating over memory in this way reduces non-stationarity and decorrelates updates, but at the same time limits the methods to off-policy reinforcement learning algorithms.

Deep RL algorithms based on experience replay have achieved unprecedented success in challenging domains such as Atari 2600. However, experience replay has several drawbacks: it uses more memory and computation per real interaction; and it requires off-policy learning algorithms that can update from data generated by an older policy.

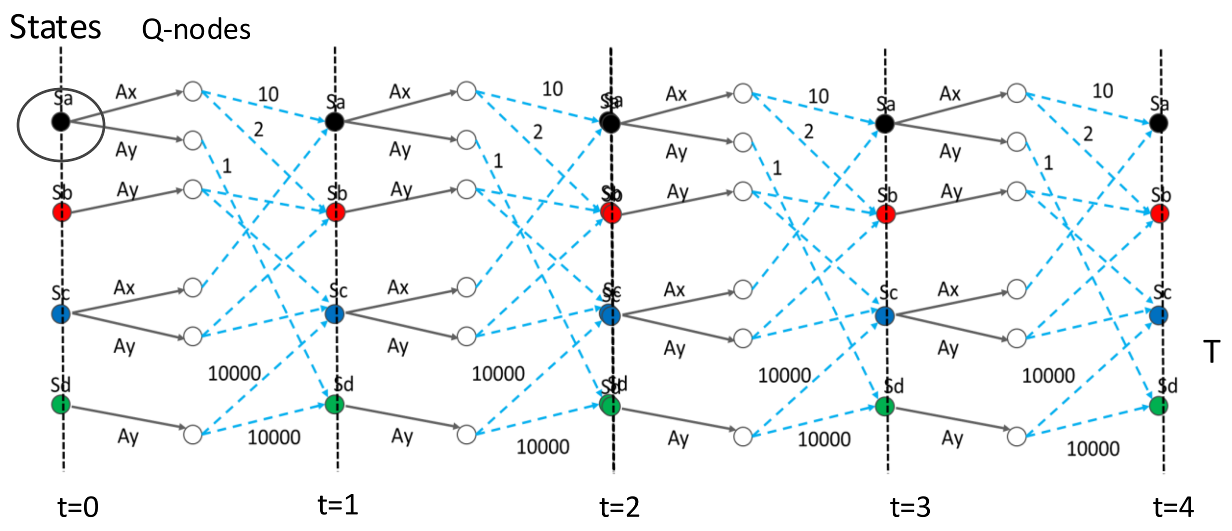
In this paper we provide a very different paradigm for deep reinforcement learning. Instead of experience replay, we asynchronously execute multiple agents in parallel, on multiple instances of the environment. This parallelism also decorrelates the agents' data into a more stationary process,

- Revisit sampling methods like TD(0), SARSA, Q-learning
- Paper in ICML 2016: lightweight shared-memory framework for deep RL
  - Based on asynchronous gradient descent (Hogwild)
  - Implementations for SARSA, Q-learning, actor-critic methods,....
  - Results for Atari, TORCS car racing simulator, continuous action control in MuJoCo,...

"Hogwild!: A Lock-Free Approach to Parallelizing Stochastic Gradient Descent" Niu, Re et al (2011)

Review of MDPs, TD(0), SARSA, Q-learning

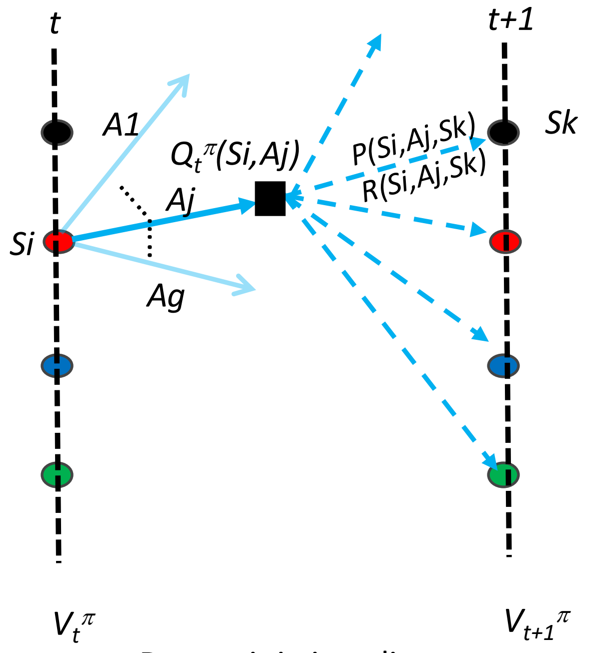
Discrete-time finite-state MDP



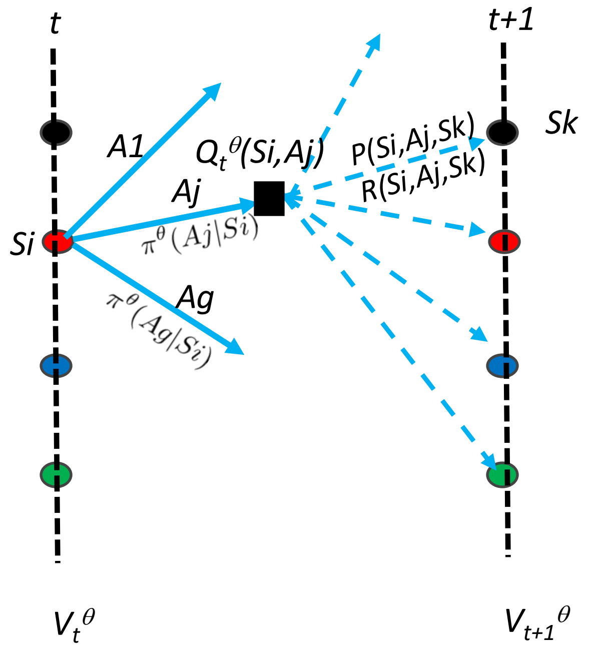
Probabilities not shown on transition edges

- S: set of states (possibly infinite but in this presentation, finite)
  - Initial state specified
- A: set of actions (possibly infinite but in this presentation, finite)
- Step: <State, Action, Reward, State>
  - Agent is in some state, takes action, observes reward and transitions to next state
  - Reward and next state are probabilistic because of environment
- Two kinds of tasks
  - Episodic task: task ends after T steps
  - Continuous task no notion of end

Policy: what action should be taken at each state?



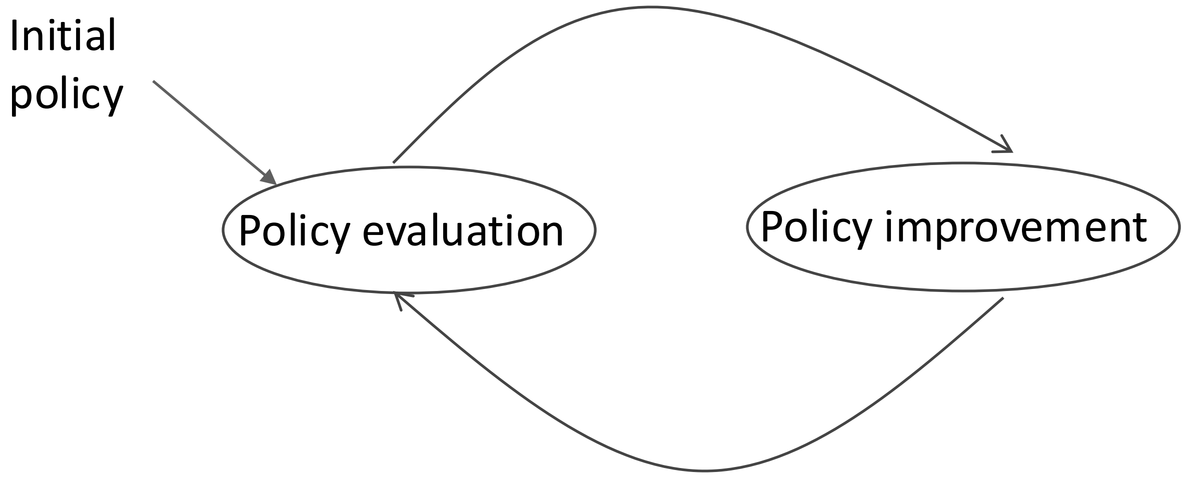
Deterministic policy



Stochastic policy

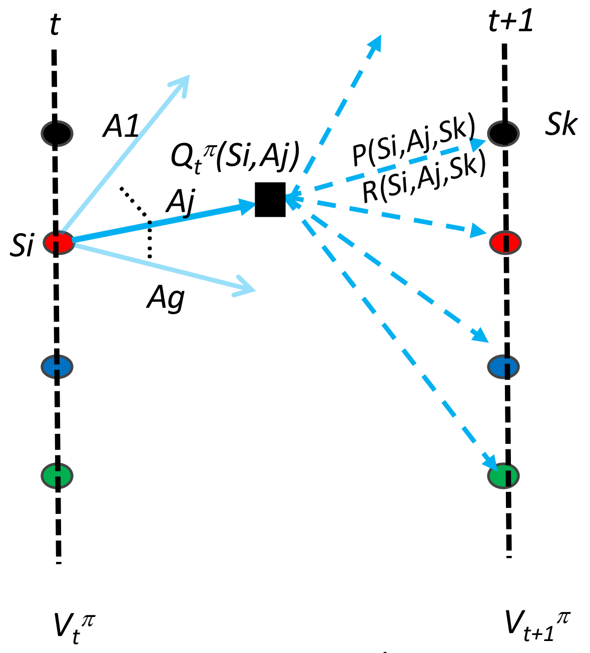
- Deterministic policy $\pi : S \to A$
  - Policy: State → Action
- Stochastic policy $\pi : S \to \mathbb{P}(A)(\text{written as } \pi(Aj|Si))$
  - Action is picked probabilistically from distribution by a sampler
  - Examples: random selection, ∊-greedy, policy network,….,
- Expected return from following policy π
  - $V^\pi(S)$: expected return at state S when following policy π
  - $Q^\pi(S,A)$: expected return at state S if you take action A and then following policy π

Policy optimization

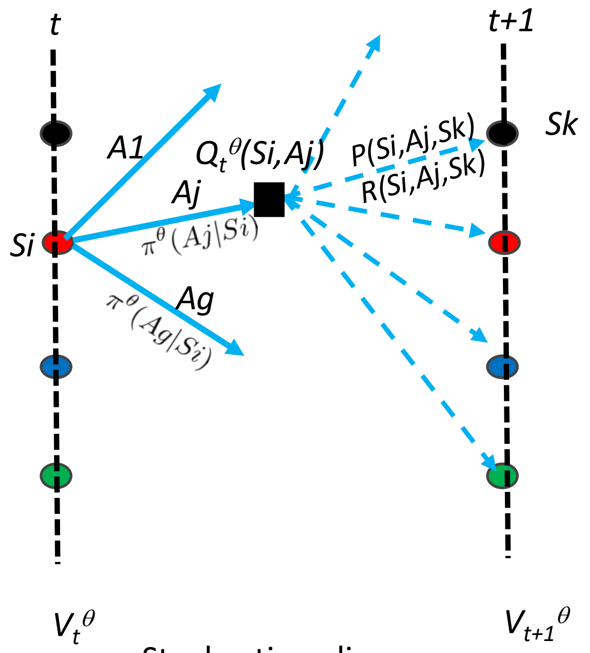


- Expected return from following policy π
  - $V^{\pi}(S)$: expected return at state S when following policy π
  - $Q^{\pi}(S,A)$: expected return at state S if you take action A and then following policy π
- What policy maximizes the expected return at starting state?
- Find iteratively
  - rounds of policy evaluation and policy improvement

Policy evaluation: episodic tasks



Deterministic policy



Stochastic policy

Policy π: $Si \rightarrow Aj$

$Q_t^\pi(Si,Aj) = \sum_{k=0}^{|S|-1}[V_{t+1}^\pi(Sk) + R(Si,Aj,Sk)] * P(Si,Aj,Sk)$

$V_t^\pi(Si) = Q_t^\pi(Si,Aj)$

$V_T^\pi(Si) = 0$

Policy improvement: switch to action with higher Q-value

Policy distribution: $\pi^\theta(Aj|Si)$

$Q_t^\theta(Si,Aj) = \sum_{k=0}^{|S|-1}[V_{t+1}^\theta(Sk)+R(Si,Aj,Sk)] * P(Si,Aj,Sk)$

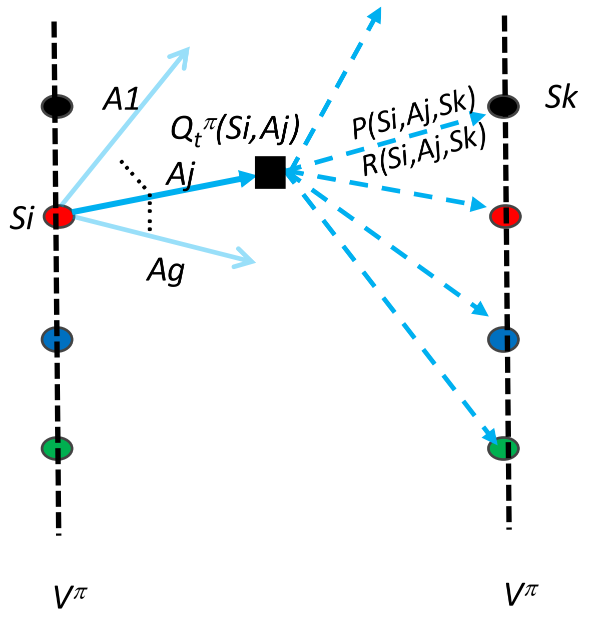
$V_t^\theta(Si) = \sum_{j=0}^{|A|-1}\pi^\theta(Aj|Si)Q_t^\theta(Si,Aj)$

$V_T^\theta(Si) = 0$

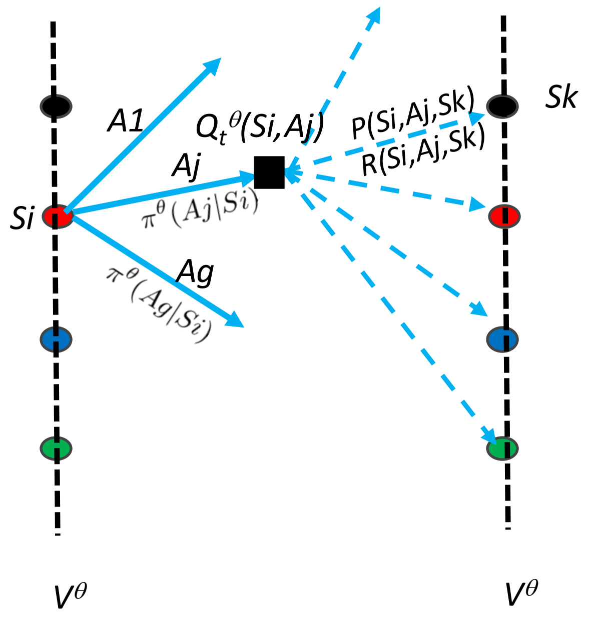
Policy improvement: change θ to promote actions with higher Q-values

Bellman iteration

Policy evaluation: continuous tasks



Deterministic policy



Stochastic policy

Systems of linear equations

Policy π: $Si \rightarrow Aj$

$$Q^\pi(Si,Aj) = \sum_{k=0}^{|S|-1}[\gamma * V^\pi(Sk) + R(Si,Aj,Sk)] * P(Si,Aj,Sk)$$

$$V^\pi(Si) = Q^\pi(Si,Aj)$$

Policy improvement: switch to action with higher Q-value

Policy distribution: $\pi^\theta(Aj|Si)$

$$Q^\theta(Si,Aj) = \sum_{k=0}^{|S|-1}[\gamma * V^\theta(Sk) + R(Si,Aj,Sk)] * P(Si,Aj,Sk)$$

$$V^\theta(Si) = \sum_{j=0}^{|A|-1}\pi^\theta(Aj|Si)Q^\theta(Si,Aj)$$

Policy improvement: change θ to promote actions with higher Q-values

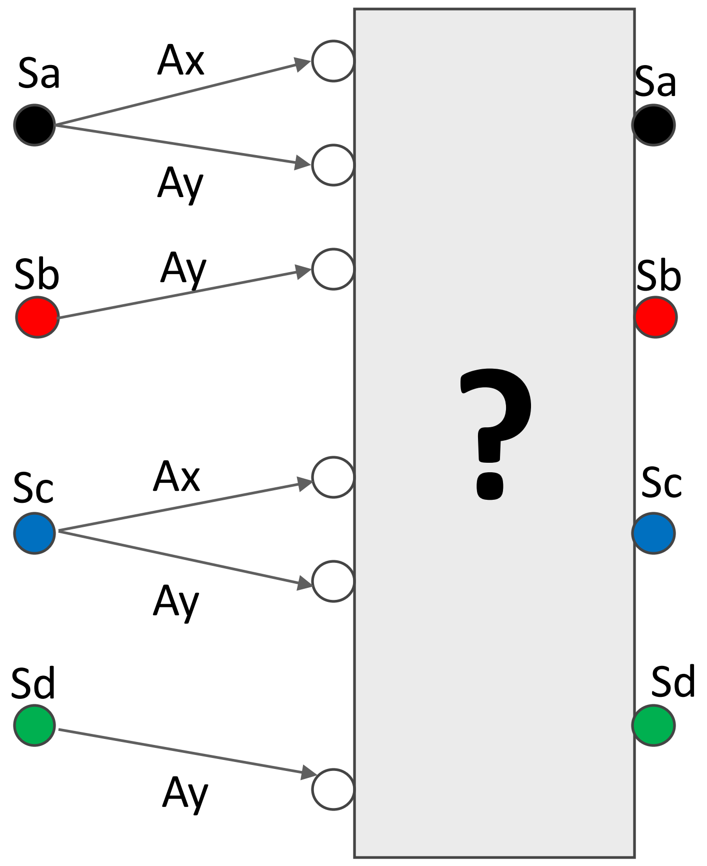
Iterative solver

$$V_0^\pi(Si) = arbitrary\ value \quad (\forall Si \in S)$$

$$V_i^\pi(Si) = \Sigma_{k=0}^{|S|-1}(V_{i-1}^\pi(Sk) + R(Si,\pi(Si),Sk)) * P(s,\pi(Si),Sk)$$

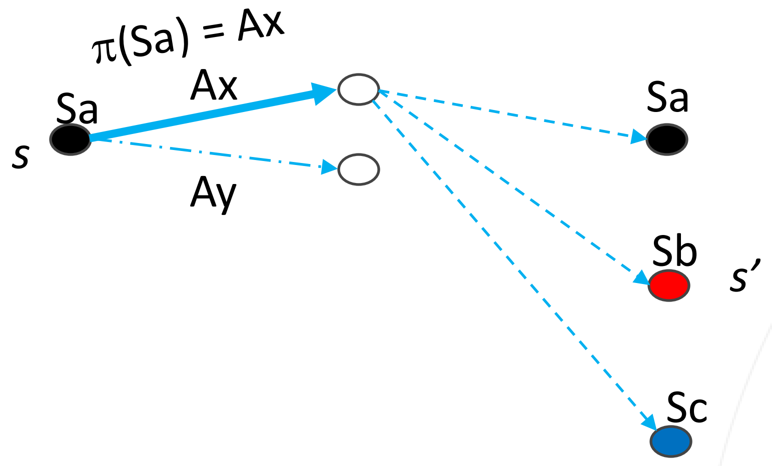
Sampling-based methods: TD(0),SARSA,Q-learning

Problem



- In practice, we usually have incomplete knowledge of environment
  - Rewards?
  - Probabilities?
- Obvious solution: **sampling**
  - Learn **estimates** for rewards and probabilities by experimentation
  - With enough samples, estimates will converge to values of underlying MDP
- Issues
  - What are samples?
  - How do we take samples efficiently?
  - What do we learn from a sample?
  - When do we learn from samples?
    - Offline: after samples are collected
    - Online: learn as samples are collected
  - Storage and computation cost
  - Sample efficiency: accurate estimates with as few samples as possible

High-level idea



- Focus on deterministic policy evaluation for continuous tasks : need to solve linear system

$$V^{\pi}(s) = \Sigma_{s'} P(s, \pi(s), s') * (R(s, \pi(s), s') + \gamma * V^{\pi}(s'))$$

  - Since π is fixed, simplify notation by dropping π from equations

$$V(s) = \Sigma_{s'} P(s, s') * (R(s, s') + \gamma * V(s'))$$

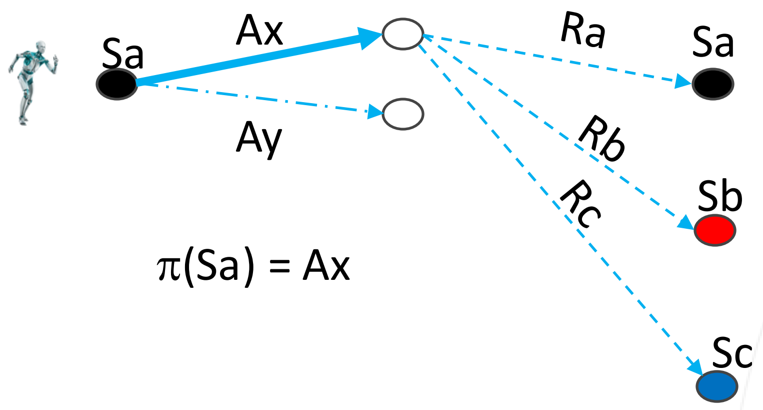
- We do not know *P* and *R*, but assume
  - $\hat{P}$: estimate for *P* and row-stochastic matrix
  - $\hat{R}$: estimate for *R*

- Compute $\hat{V}$, an estimate for state valuations, by solving linear system

$$\hat{V}(s) = \Sigma_{s'} \hat{P}(s, s') * (\hat{R}(s, s') + \gamma * \hat{V}(s'))$$

- If $\hat{P} \approx$ P and $\hat{R} \approx$ R, then $\hat{V} \approx$ V

First attempt: offline method

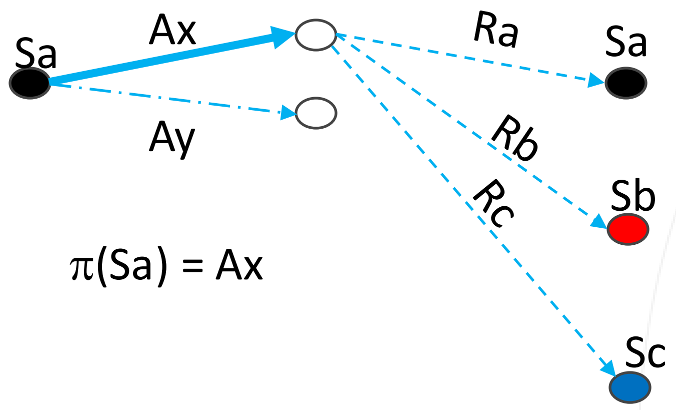


- Collect a sequence of samples
  - At state Sa, agent takes action π(Sa) and sees what happens
    - > Observation SARS = <Start**S**tate, **A**ction, Observed **R**eward, Observed **S**tate>
  - Repeat to obtain sequence of observations starting at various states
    - > O = [$SARS_1$, $SARS_2$, $SARS_3$,.....,$SARS_n$]

- Estimate V by offline processing of observations
  - n(Sx) = number of times agent started in state Sx in O (assume > 0)
  - n(Sx,Sy) = number of times agent ended up in state Sy when it started in state Sx
  - $\hat{P}$(Sx,Sy) = $\frac{n(Sx,Sy)}{n(Sx)}$ ($\hat{P}$ is a row-stochastic matrix)
  - $\hat{R}$(Sx,Sy) = observed reward in any Sx →Sy sample
  - Solve linear system to find $\hat{V}$

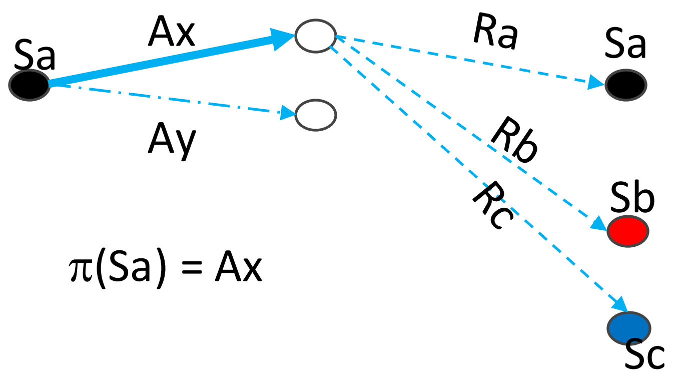
$$\boxed{\hat{V}(s) = \frac{1}{n(s)}\Sigma_{s'}n(s,s') * (\hat{R}(s,s') + \gamma * \hat{V}(s'))}$$

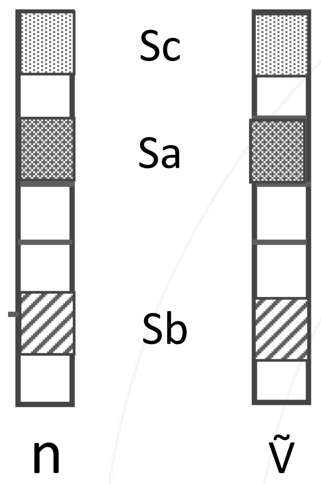
Disadvantages of offline method



- May need to store billions of samples before processing
- No information about state valuations or policies until all samples have been collected
- Does not permit “adaptive” sampling that improves sample efficiency by focusing on promising states/actions

TD(0): simplest online method





- Recall: offline processing

$$\hat{V}(s) = \frac{1}{n(s)} \sum_{s'} n(s,s') * (\hat{R}(s,s') + \gamma * \hat{V}(s'))$$

- Online updates: maintain two arrays while processing samples
  - n(s) = number of times state s has been visited (initialized to 0)
  - Ṽ(s)= current estimate of V (initialized arbitrarily)

- New sample <Sa, Ax, R(Sa,S'), S'> comes in

$$\tilde{V}(Sa) \leftarrow \frac{n(Sa)*\tilde{V}(Sa) + (R(Sa,S')+\gamma*\tilde{V}(S'))}{n(Sa)+1} \text{ which is equivalent to}$$

$$\boxed{\tilde{V}(Sa) \leftarrow \tilde{V}(Sa) + \frac{1}{\alpha}\boxed{\boxed{R(Sa,S')+\gamma*\tilde{V}(S')} - \tilde{V}(Sa)} \text{ where } \alpha{=}n(Sa){+}1}$$

$$n(Sa) \leftarrow n(Sa)+1$$

TD-target
TD-error

Comments

- Convergence (Chatterjee et al.): Given a sequence of one-hop samples O, the $\tilde{V}$ values computed by the TD(0) program will converge to the $\hat{V}$ values in the fixpoint equation

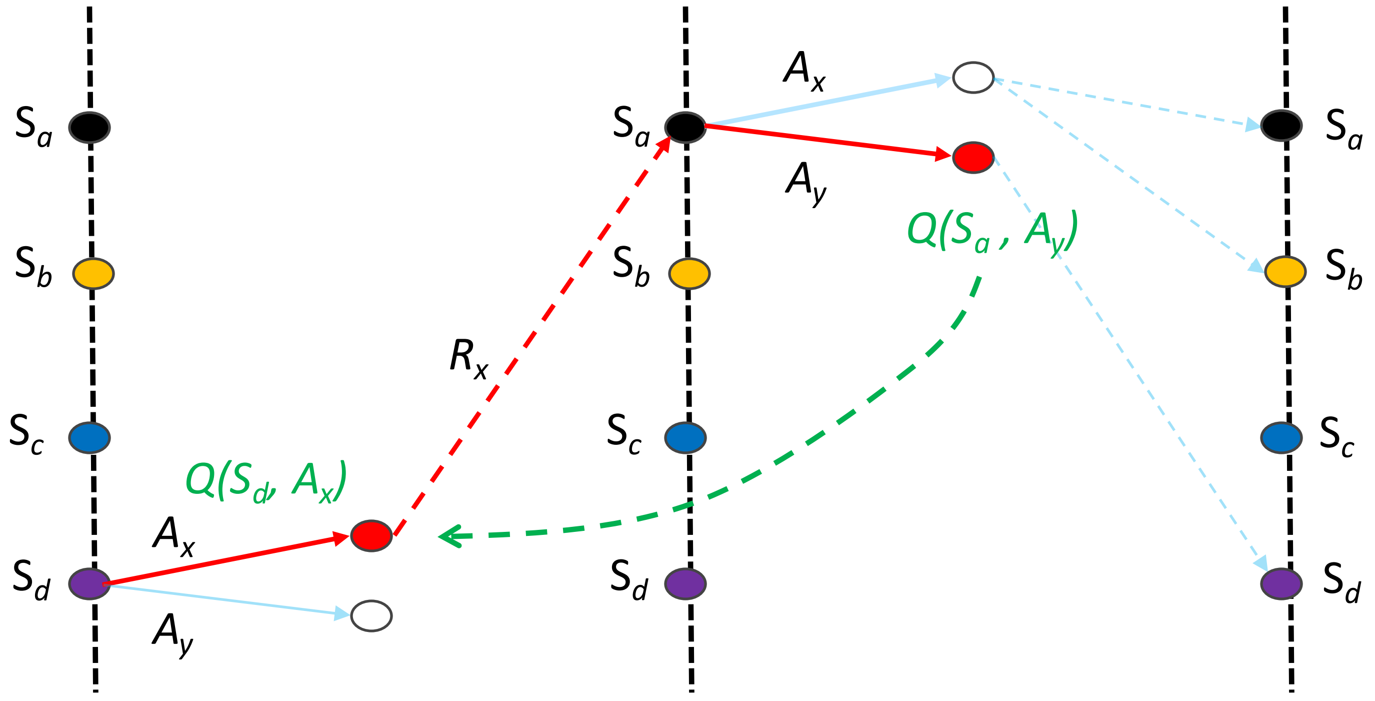
$$\hat{V}(s) = \frac{1}{n(s)} \sum_{s'} n(s, s') * (\hat{R}(s, s') + \gamma * \hat{V}(s'))$$

- Law of large numbers: if you sample every state unboundedly many times, $\tilde{V}$ values will converge to the MDP values (V^{π})

- Practice: non-stationary problems
  - Instead of harmonic weights, use constant α to give higher weight to more recent samples

- On-policy method

SARSA



$$Q(Sd,\pi(Sd)) \leftarrow Q(Sd,\pi(Sd)) + \alpha * [Rx + \gamma * Q(Sa,\pi(Sa)) - Q(Sd,\pi(Sd))]$$

- On-policy method for estimating Q-values by sampling

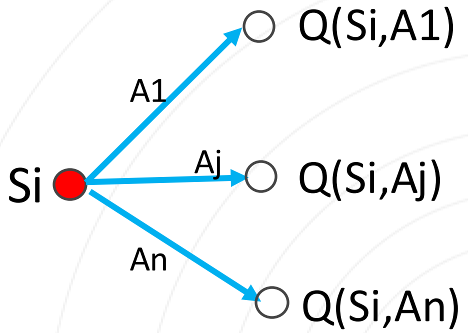
- Method
  - Pick a state (S_d)
  - **Behavior**: Use policy to determine action $\pi(S_d)$: A_x
  - Perform transition to new state and observe reward (R_x, S_a)
  - **Target**: Use policy to determine action $\pi(S_a)$ but do not take it: A_y
  - Use Q(S_a, A_y) and R_x to update Q(S_d, A_x)

Disadvantages of on-policy methods

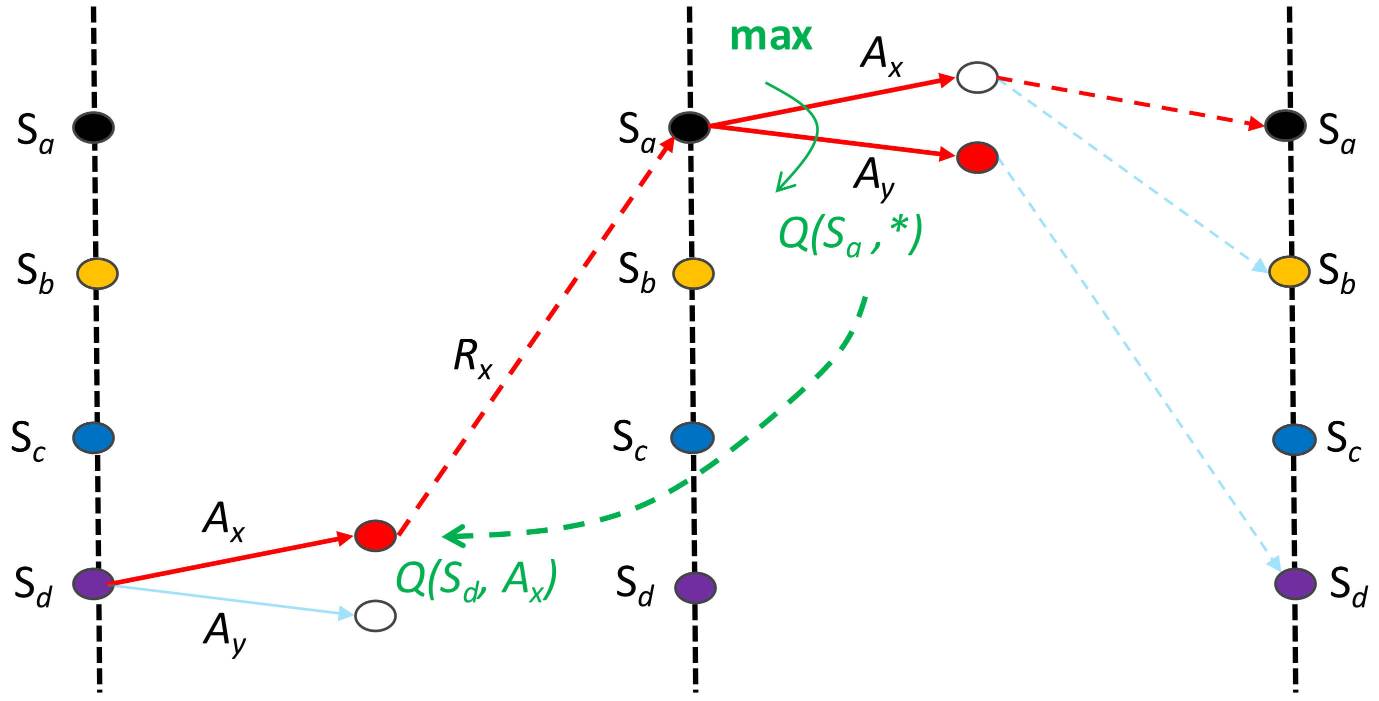
- Why complete evaluation of current policy before improving it?
  - We may find opportunities for policy improvement while performing evaluation of current policy
- Generalized policy iteration
  - Improve as you evaluate policy
  - One approach: stochastic policies
- Off-policy methods
  - Q-learning [Watkins '89]
  - Uses stochastic policies

Stochastic policies: exploration vs. exploitation dilemma

- Naïve exploration: pick action at random
  - Disadvantage: not using Q-values (what we have discovered so far)
- Pure exploitation: pick action with highest Q-value
  - Disadvantage: may get stuck in local minima
  - Bad idea particularly at start of training
- ∊-greedy: parameter ∊ to trade-off exploration and exploitation
  - Probability (1-∊): select action with highest Q value (exploitation)
  - Probability ∊: select random action (exploration)
  - Probably most popular stochastic policy
- Boltzmann exploration
  - like ∊-greedy but biases random action towards actions with higher Q-values
  - $P(Aj|Si) = \frac{e^{\rho.Q(S1,Aj)}}{\sum_{Ai \in A} e^{\rho.Q(Si,Aj)}}$
- Policy network
  - DNN that maps state to action probabilities



Q-learning

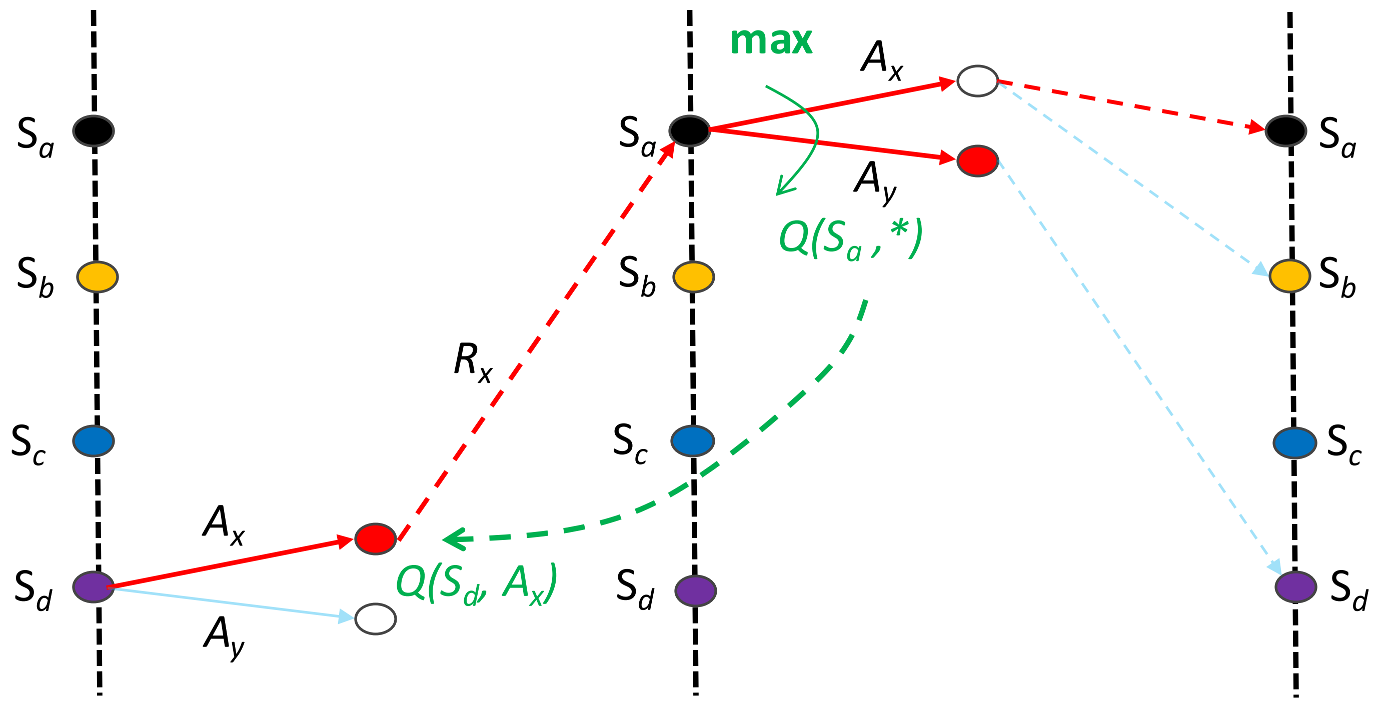


$$Q(Sd, Ax) \quad \leftarrow \quad Q(Sd, Ax) + \alpha * [Rx + \gamma * max_{As} Q(Sa, As) - Q(Sd, Ax)]$$

- Off-policy method

- Method:
  - Pick a state (S_d)
  - **Behavior**: Use stochastic policy π to determine action (A_x)
  - Perform transition to new state and observe reward (R_x, S_a)
  - **Target**: max(Q(S_a,*))
  - Use max(Q(S_a,*)) and R_x to update Q(S_d, A_x)

Q-learning: convergence to optimal Q-values

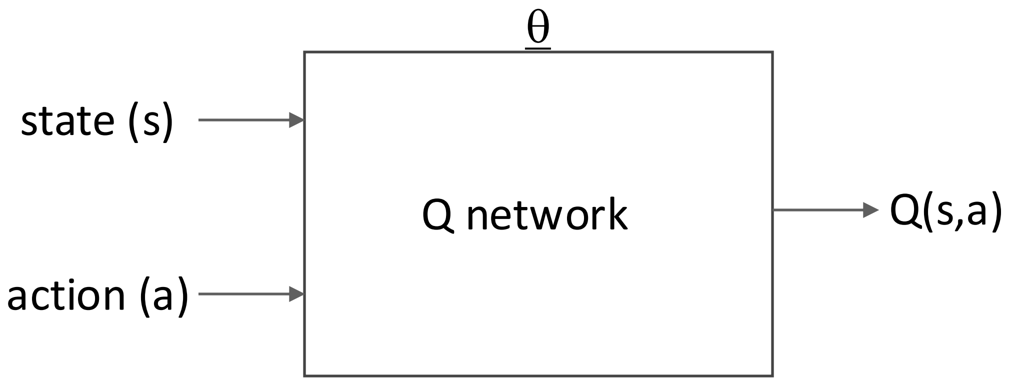


- Q-learning makes sense only with stochastic policies
  - Max is meaningless otherwise
- Convergence: if behavior policy ensures that every (state,action) is visited unboundedly often, Q-learning converges to optimal Q-values (Melo 2007)
  - Intuition: need sufficient exploration
- No need for explicit policy improvement
  - Only place where policy is used is in selecting (state,action) pairs to be explored (behavior)

Shared-memory parallel implementation of SARSA,Q-learning etc. (Mnih et al.)

Implementation of Q-tables using NNs

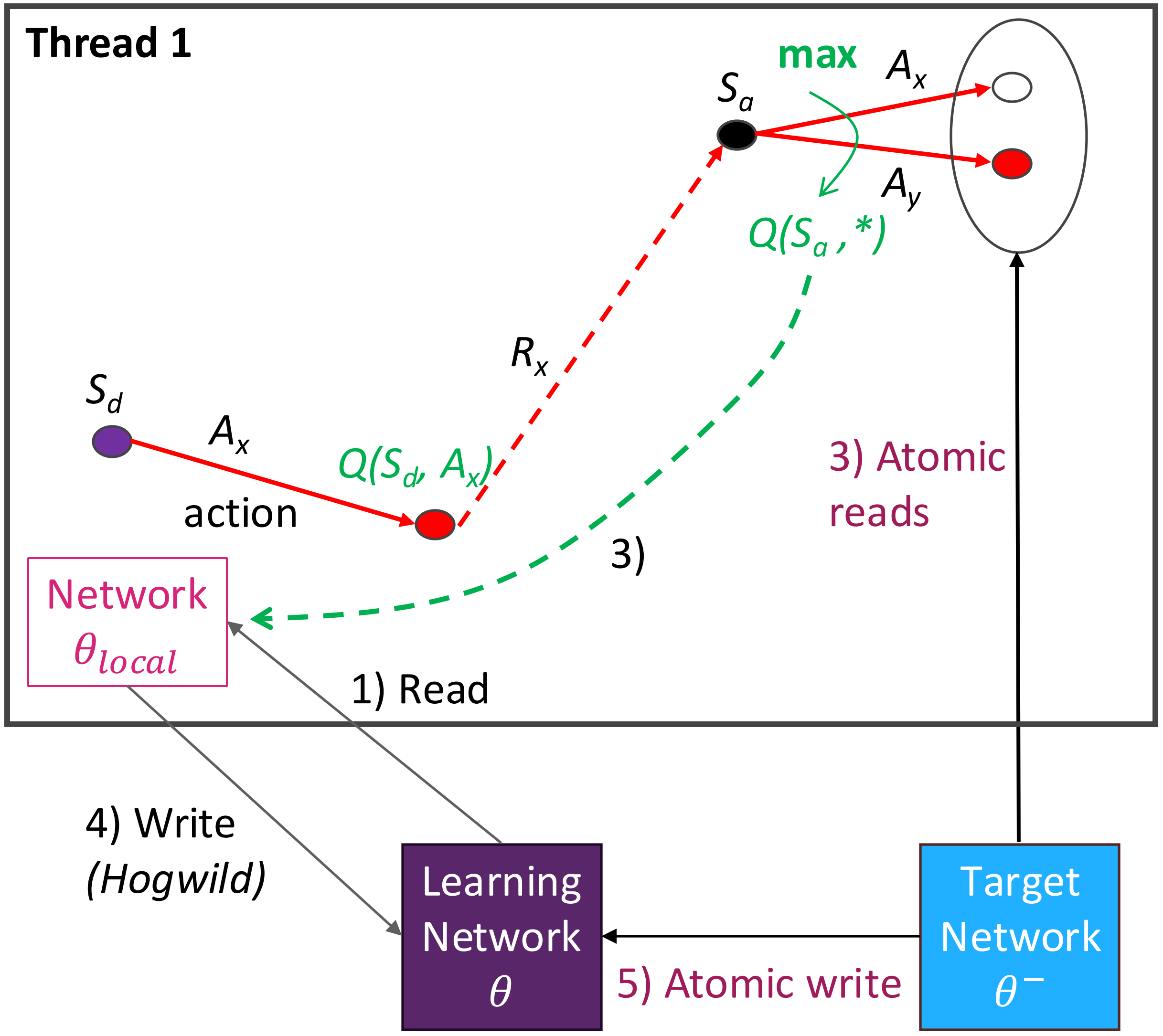
Explicit Q-tables not practical for very large state/action spaces
Solution: use a neural network to approximate Q-table (function approximation)



To "read" value of Q(s,a)
- Forward propagation with (s,a) as input

To "write" v to Q(s,a)
- Forward propagation with (s,a) as input
- Loss = v – Q(s,a)
- Backprop to update $\underline{\theta}$

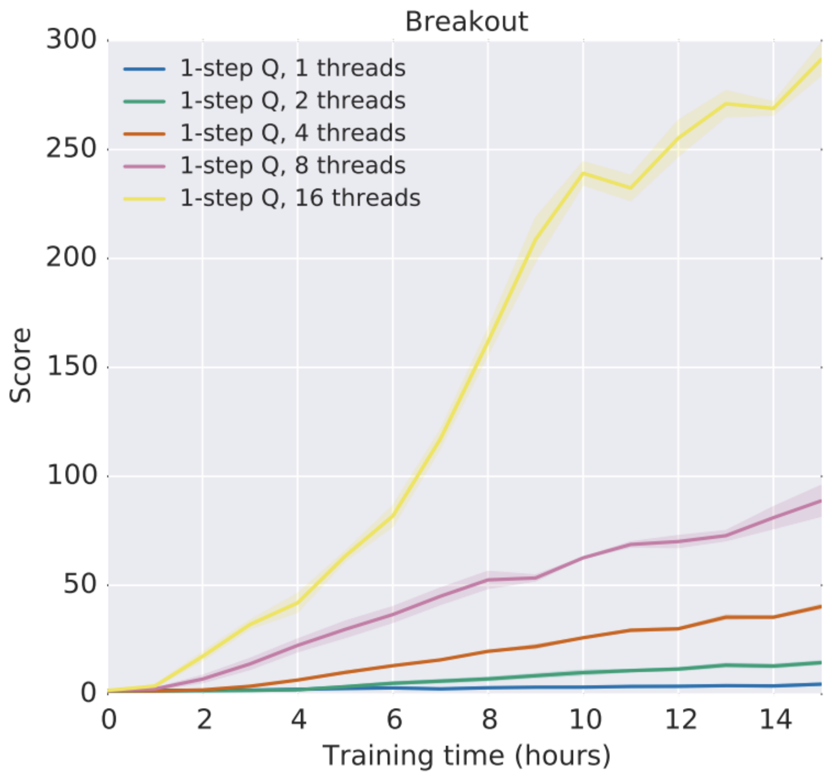
Q-learning

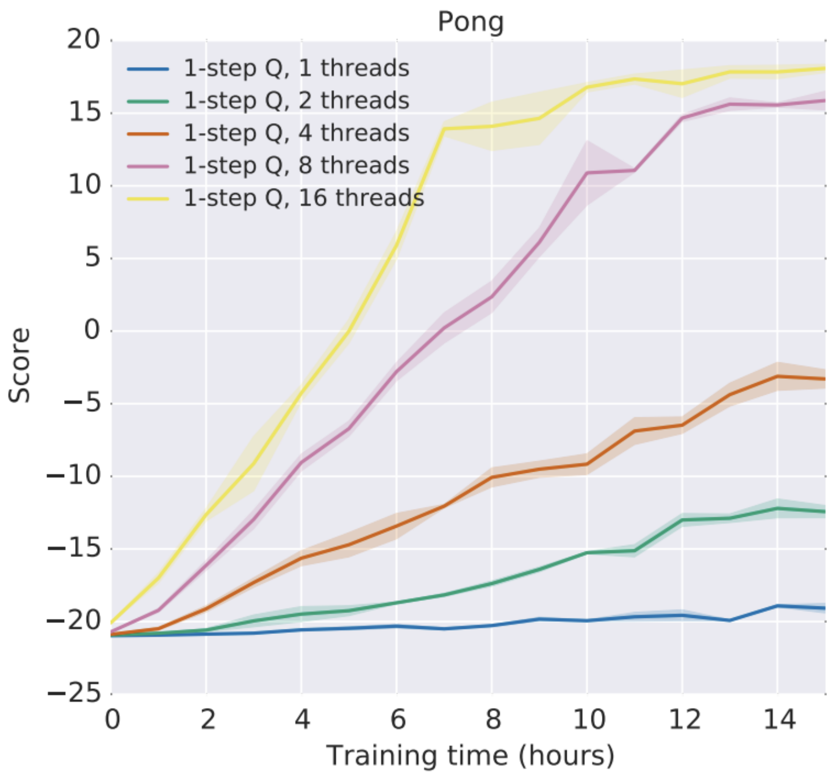


1) $\theta_{local} \leftarrow \theta$
2) Start at random state and collect transitions using θ_{local} to pick action
3) Update θ_{local} using $\max Q(s, a, \theta^-)$
4) $\theta \leftarrow \theta_{local}$
5) $\theta^- \leftarrow \theta$

$$Q(Sd, Ax) \leftarrow Q(Sd, Ax) + \alpha * [Rx + \gamma * max_{As} Q(Sa, As) - Q(Sd, Ax)]$$

Results


Breakout
Score
Training time (hours)
1-step Q, 1 threads
1-step Q, 2 threads
1-step Q, 4 threads
1-step Q, 8 threads
1-step Q, 16 threads
0
50
100
150
200
250
300
0
2
4
6
8
10
12
14


Pong
Score
Training time (hours)
1-step Q, 1 threads
1-step Q, 2 threads
1-step Q, 4 threads
1-step Q, 8 threads
1-step Q, 16 threads
20
15
10
5
0
−5
−10
−15
−20
−25
0
2
4
6
8
10
12
14