Model-free and Model-based Auto-Scaling Techniques for Distributed Applications

Model-free and Model-based Policies

Gabriele Russo Russo

University of Rome Tor Vergata, Italy

June 2023 – Roma Tre University

Auto-scaling Policy

- A **policy** determines which scaling action (if any) should be performed based on the analysis of current (and future) system state.
- We focus on the Analysis and Plan phases of the MAPE loop

Threshold-based Policies

- Resource allocation varies according to a set of rules
- Rule = condition + action
- Conditions defined in terms of thresholds
- Model-free approach

if $x_1 > H_1$ and/or $x_2 > H_2$ and/or ... for D_H seconds
    scale-out(N)
if $x_1 < L_1$ and/or $x_2 < L_2$ and/or ... for D_L seconds
    scale-in(N)

Threshold-based Policies (2)

- Conditions may involve one or more metrics
  - e.g., CPU utilization, used memory
- For each metric, multiple conditions (hence, thresholds) can be defined
- Usually upper (or, high) and lower (or, low) thresholds are used
  - e.g., for CPU utilization, upper threshold 75% and lower 20%
- Conditions may also require thresholds to be exceeded for a certain amount of time before triggering an action

Example of rules

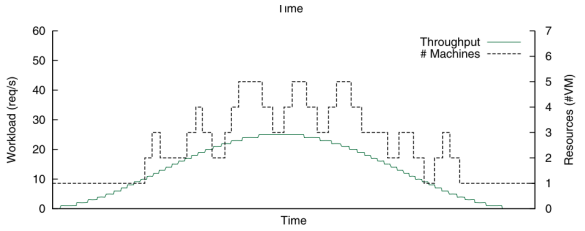
- If CPU utilization > 70% for at least 1 minute, add one VM
- If CPU utilization < 30% for at least 5 minutes, terminate one VM
- If avg. response time > 100ms for at least 30s, increase CPU frequency by 10%

Issues with thresholds

- Threshold-based policies are easy to implement and execute
- But defining suitable rules is not trivial!
- Which metrics? System vs application-oriented
  - System metrics may work across different apps
  - Application metrics directly mapped onto QoS requirements
- Which thresholds? Tuning required!

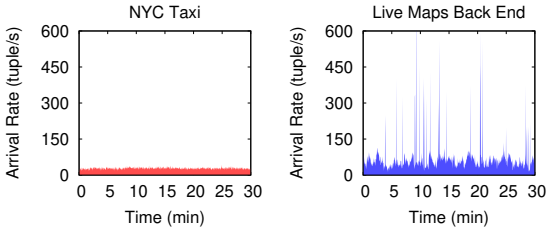
Issues with Thresholds: Oscillation

- If thresholds are too close, frequent oscillations may occur
- Oscillations negatively impact performance
  - System oscillates between under- and over-provisioning
  - Scaling may introduce additional overhead too
- Possible workaround: cooldown (or, calm) period
  - Auto-scaler inhibited for a short period after every scaling action



Issues with Thresholds: Bursts

- Scaling conditions are usually evaluated over short-medium time periods (e.g., seconds, minutes)
- Not all workloads can be characterized looking at average metrics over such time windows
- These workloads lead to similar average utilization over 1-minute windows, but they are profoundly different due to bursts [Russo Russo, V. Cardellini, Casale, et al. 2021]



Automatic Tuning

- Difficult to set static thresholds that work all the time (and possibly for multiple applications)
- Various approaches to automatically tune/adjust thresholds
- Dynamic thresholds [Lorido-Botrán, Miguel-Alonso, and Lozano 2013]
  - Start with 90%-10% thresholds (upper and lower)
  - If SLO violations occur for a certain time (e.g. 5 minutes), threshold range widened (e.g., from 60-40% to 80-20). This makes the system less reactive to workload changes.
  - When no SLO violations during the last period of time, threshold range is narrowed.
- Reinforcement learning-based (e.g., Fabiana Rossi, Valeria Cardellini, and Francesco Lo Presti 2020, Lombardi et al. 2018)

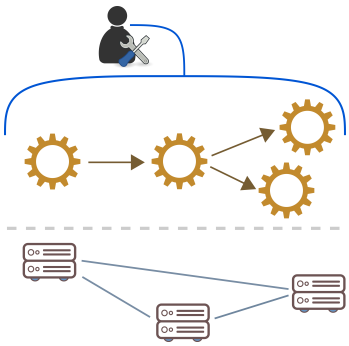
Proactive Scaling

- Threshold-based policies are usually reactive
- If some predictive analysis is performed on monitoring data, it can work in a proactive manner
- e.g., using past utilization measures to predict future CPU utilization
- See, e.g., [Khatua, Ghosh, and Mukherjee 2010]

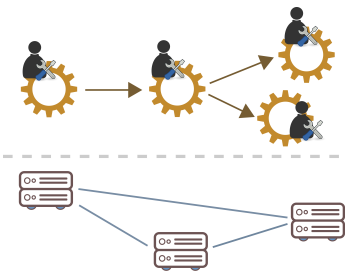
Introduction to Markov Decision Processes and Reinforcement Learning

Controller Architectural Alternatives

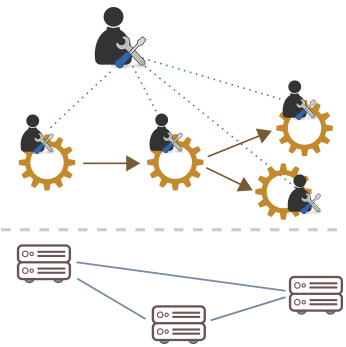
Centralized Controller



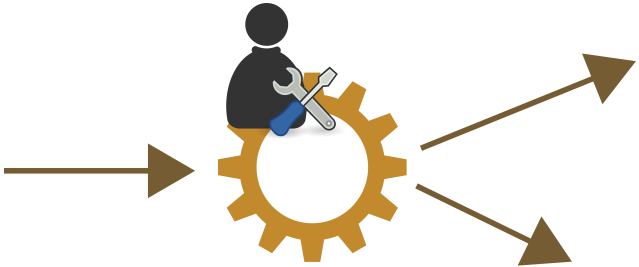
Distributed Controllers



Hierarchical Solution

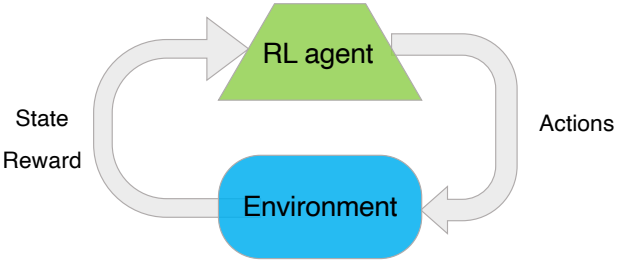


Component Elasticity



- Traditional solution: threshold-based policies
- More general approach: Scaling Manager trade-offs key performance indices
  - Allocated resources
  - Performance
  - Reconfiguration costs
- We tackle this problem using Reinforcement Learning

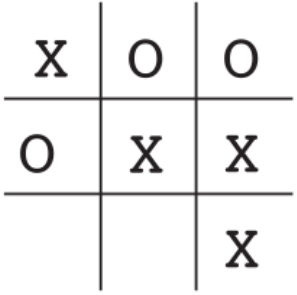
Reinforcement Learning



- A branch of ML dealing with sequential decision-making
- Agent interacts with environment through actions and receives feedback in the form of reward (or paid cost)
- Goal: learning to act as to maximize long-term reward
- Trial-and-error experience (no complete knowledge of environment a priori)

Example: Tic-Tac-Toe

- State: representation of the board (3x3 matrix)
- Actions: available cells to mark
- Reward: 1 for a winning move, 0 otherwise



Example: AlphaZero by DeepMind

- Software able to play Go, Chess and Shogi [David Silver et al. 2018]
  - Board games with huge number of legal positions (i.e., state space)
- Trained via self-play and advanced deep RL techniques
- Superhuman level of play with 24-hour training
- First presented in 2017; in 2019 MuZero, generalization to play Atari games and other board games without prior rule knowledge

Example: AlphaDev by DeepMind

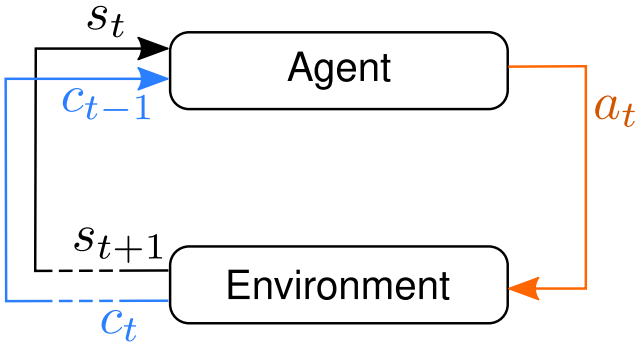
- Announced a few days ago[1]
- RL used to develop new C++ sorting algorithm, now accepted in the standard library
- 70% faster on short sequences (2-3 items), 1.7% faster on long sequences
- State: instructions generated so far and state of the CPU
- Actions: assembly instructions to add
- Reward: based on sorting correctness and efficiency

[1] https://www.deepmind.com/blog/alphadev-discovers-faster-sorting-algorithms

Other Examples

- Self-driving cars
- Videogames
- Trading agents
- Cluster schedulers
- …

RL for System Adaptation

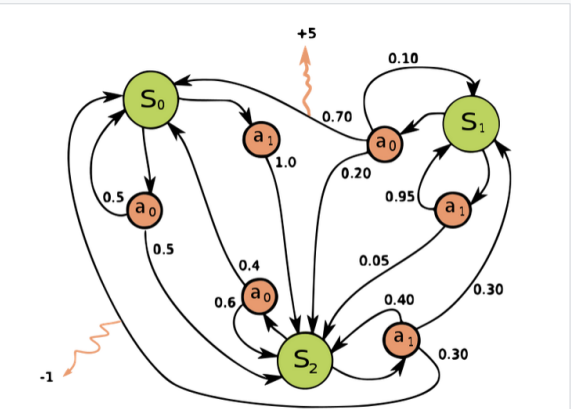


- States?, Actions?, Costs?
  - Defined according to the scenario
  - Actually, a Markov Decision Process (MDP)

Markov Decision Process

- Markov decision process (MDP) is a stochastic control process
- A framework to model decision making in situations where outcomes are partly random
- Extension of Markov Chains
- At each time step, the process is in some state s, and the decision maker (the agent) chooses an action a that is available in state s
  - e.g., a robot observes its current position and decides direction to move
- The environment responds at the next time step by (randomly) moving into a new state s', and possibly assigning the agent a corresponding reward (or a cost)
  - e.g., the robot may get a reward when it reaches its final destination

Example



Example of a simple MDP with three states (green circles) and two actions (orange circles), with two rewards (orange arrows).

Markov Decision Process (2)

A Markov Decision Process is a tuple $\langle \mathcal{S}, \mathcal{A}, p, c, \gamma \rangle$

- $\mathcal{S}$ is a finite set of states
- $\mathcal{A}$ is a finite set of actions
- p is a state transition probability function

$$p(s'|s,a) = P[S_{t+1} = s' | S_t = s, A_t = a]$$

- c is a cost function
  1. $c(s,a) = E[C_t | S_t = s, A_t = a]$
  2. $c(s,a,s') = E[C_t | S_t = s, A_t = a, S_{t+1} = s'] \longrightarrow c(s,a) = \sum_{s'} p(s'|s,a) c(s,a,s')$
- γ is a discount factor, $\gamma \in [0,1)$

Markov Property

"The future is independent of the past given the present"

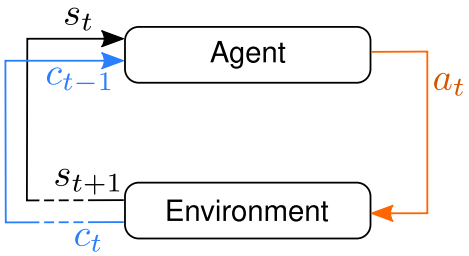
Definition

A state S_t is Markov if and only if

$$P[S_{t+1}|S_1, \ldots, S_t] = P[S_{t+1}|S_t]$$

- The state captures all relevant information from the history
- i.e., the state is a sufficient statistic of the future

MDP: Objective



The overall objective is to minimize the expected cumulative discounted cost

$$G_t = C_t + \gamma C_{t+1} + \ldots = \sum_{k=0}^{\infty} \gamma^k C_{t+k}$$

- Discount factor $\gamma \in [0, 1)$ for future costs
- e.g., $\gamma \approx 0 \rightarrow$ agent only cares about immediate costs

Policy

Definition

A policy π is a distribution over actions given state,

$$\pi(a|s) = p(A_t = a|S_t = s)$$

- Deterministic policy: $\pi : \mathcal{S} \to \mathcal{A}$

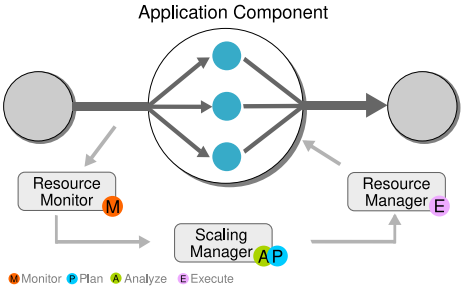
- A policy fully defines the behaviour of an agent
- MDP policies depend on the current state

Example: Deterministic Policy

- Most of the time we are only interested in deterministic policies

State	*Action*
s_1	a_1
s_2	a_1
s_3	a_2
s_4	a_1

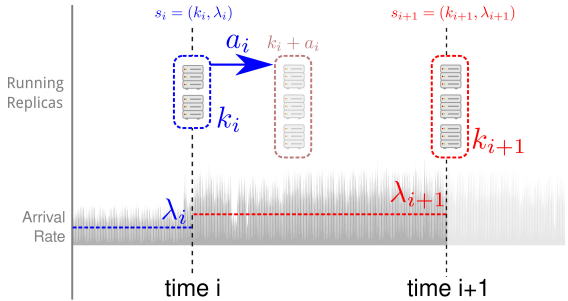
Horizontal Auto-scaling: MDP formulation



- We periodically make a decision about scaling in/out an app component
- We are concerned with 3 objectives:
  - Monetary resource cost (or, resource usage in general)
  - Performance req. satisfaction (e.g., max response time)
  - Scaling overhead

Horizontal Auto-scaling: MDP formulation

- We model the problem as a Markov Decision Process (MDP)



- State at time slot i: $s_i = (k_i, \lambda_i)$
  - k_i component parallelism
  - λ_i avg. arrival rate (of requests, jobs, data, ...)
- Action at time slot i: $a_i \in \{0, +1, -1\}$

MDP Model: Transition Probabilities

- State of the system $s = (k, \lambda)$
  - $1 \leq k \leq K^{max}$ Component parallelism
  - λ avg. input rate
    - λ is discretized , i.e., $\lambda_i \in \{0, \Delta\lambda, 2\Delta\lambda, (L-1)\Delta\lambda\}$
    - $\Delta\lambda$ quantization step size, L number of discrete values
- Available actions $\mathcal{A} = \{-1, 0, +1\}$
- Transition probabilities $p(s'|s, a) = p\left((k', \lambda')|(k, \lambda), a\right)$

$$
\begin{aligned}
p(s'|s,a) &= P[s_{t+1} = (k', \lambda') | s_t = (k, \lambda), a_t = a] = \\
&= \begin{cases} P[\lambda_{t+1} = \lambda' | \lambda_t = \lambda] & k' = k + a \\ 0 & \text{otherwise} \end{cases} = \\
&= \mathbb{1}_{\{k' = k+a\}} P[\lambda_{t+1} = \lambda' | \lambda_t = \lambda]
\end{aligned}
$$

MDP Model: Cost Function

- Simple Additive Weighting method
- Cost associated with action execution and state transition $(s, a) \rightarrow s'$

$$c(s, a, s') = w_{res} \frac{k + a}{K^{max}} + w_{perf} \mathbb{1}_{\{R(s,a,s') > R^{max}\}} + w_{rcf} \mathbb{1}_{\{a \neq 0\}}$$

Resource Cost | Performance | Reconfiguration

- $w_{res} + w_{perf} + w_{rcf} = 1$, $w_x \geq 0, x \in \{res, perf, rcf\}$
- $R(s, a, s')$ is the component performance index, e.g, response time,
- R^{max} is the component reference performance value

We want to minimize $\sum_{t=0}^{\infty} \gamma^t c(s_t, a_t, s_{t+1}), \quad \gamma \in [0, 1)$

Value Function

Value function is a prediction of future costs

- can be used to evaluate how good/bad states and/or actions are
- and therefore to select actions e.g.

State	a_1	a_2	a_3
s_1	10	5	3
s_2	8	6	4
s_3	6	5	6
s_4	5	4	6
s_5	4	3	7
s_6	1	5	9
s_7	0	9	15

s	$\pi(s)$
s_1	a_3
s_2	a_3
s_3	a_2
s_4	a_2
s_5	a_2
s_6	a_1
s_7	a_1

Value Functions

Action value function (or, Q function)

Expected cost starting from state s, taking action a and then following policy π

$$Q_\pi(s, a) = E_\pi[G_t | S_t = s, A_t = a]$$

State value function

Expected cost starting from state s and then following policy π

$$V_\pi(s) = E_\pi[G_t | S_t = s]$$

Action Value Functions

The action value function can be decomposed into two parts:

- immediate cost
- discounted costs from successor state S_{t+1}

$$
\begin{aligned}
Q_\pi(s,a) &= E_\pi[G_t | S_t = s, A_t = a] \\
&= E_\pi[C_t + \gamma C_{t+1} + \gamma^2 C_{t+2} \ldots | S_t = s, A_t = a] \\
&= E_\pi[C_t + \gamma \left(C_{t+1} + \gamma C_{t+2} \ldots\right) | S_t = s, A_t = a] \\
&= E_\pi[C_t + \gamma G_{t+1} | S_t = s, A_t = a]
\end{aligned}
$$

Bellman equation:

$$
Q_\pi(s,a) = c(s,a) + \gamma \sum_{s' \in S} p(s'|s,a) Q_\pi(s', \pi(s'))
$$

State Value Functions

The value function can be similarly decomposed into two parts:

- immediate cost C_t
- discounted cost from successor state $V(S_{t+1})$

$$\begin{aligned} V_\pi(s) &= E_\pi[G_t|S_t = s] \\ &= E_\pi[C_t + \gamma C_{t+1} + \gamma^2 C_{t+2} \ldots |S_t = s] \\ &= E_\pi[C_t + \gamma \left(C_{t+1} + \gamma C_{t+2} \ldots\right)|S_t = s] \\ &= E_\pi[C_t + \gamma G_{t+1}|S_t = s] \end{aligned}$$

Bellman equation:

$$V_\pi(s) = c(s, \pi(s)) + \gamma \sum_{s'} p(s'|s, \pi(s)) V_\pi(s')$$

Optimal Value Function

Optimal action value function

$Q^*(s; a)$ is the minimum action-value function over all policies

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

Optimal state value function

$V^*(s)$ is the minimum value function over all policies

$$V^*(s) = min_\pi V_\pi(s)$$

Bellman Optimality Equations

$$Q_\pi(s,a) = c(s,a) + \gamma \sum_{s' \in S} p(s'|s,a) Q_\pi(s', \pi(a'))$$

$$\Downarrow$$

$$Q^*(s,a) = c(s,a) + \gamma \sum_{s' \in S} p(s'|s,a) \min_{a'} Q^*(s',a') \quad (1)$$

$$V^*(s) = \min_a Q^*(s,a) \quad (2)$$

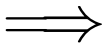
$$Q^*(s,a) = c(s,a) + \gamma \sum_{s' \in S} p(s'|s,a) V^*(s') \quad (3)$$

Optimal Policy

Given $Q^*(s, a)$ the optimal action when the system is in state s is:

$$\pi^*(s) = a^*(s) = \arg\min_{a \in \mathcal{A}} Q^*(s, a)$$

State	a_1	a_2	a_3
s_1	10	5	3
s_2	8	6	4
s_3	6	5	6
s_4	5	4	6
s_5	4	3	7
s_6	1	5	9
s_7	0	9	15



Optimal Action
a_3
a_3
a_2
a_2
a_2
a_1
a_1

How to compute V^*?

- If we know the optimal value function, we readily have an optimal policy!
- But… how do we compute the optimal value function??

Value Iteration

Bellman Equation

$$Q^*(s,a) = c(s,a) + \gamma \sum_{s' \in S} p(s'|s,a) \min_{a'} Q^*(s',a')$$

- Suppose we know the solution to subproblems $Q^*(s',a')$
- $Q^*(s,a)$ can be computed by one-step lookahead

$$Q^*(s,a) \leftarrow c(s,a) + \gamma \sum_{s' \in S} p(s'|s,a) \min_{a'} Q^*(s',a')$$

- The idea is to apply these updates iteratively
- Convergence is ensured by the Contraction Mapping Theorem [Sutton and A. Barto 2018]

Value Iteration: the Algorithm

Value Iteration

$i \leftarrow 0$;
$Q_i(s,a) \leftarrow 0, \forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s)$;
repeat
  forall $s \in \mathcal{S}$ **do**
    forall $a \in \mathcal{A}(s)$ **do**
      $Q_{i+1}(s,a) \leftarrow c(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) \min_{a' \in \mathcal{A}(s')} Q_i(s',a')$;
    end
  end
  $i \leftarrow i+1$;
until $\max_{s,a} |Q_i(s,a) - Q_{i-1}(s,a)| < \epsilon$;
$\pi^*(s) = \arg\min_a Q_i(s,a), \forall s \in \mathcal{S}$

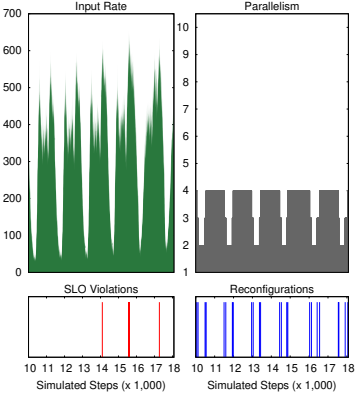
Value Iteration: Alternative Algorithm

Value Iteration - Alternative

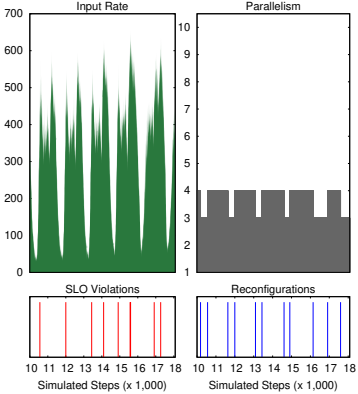
$i \leftarrow 0$;
$Q_i(s,a) \leftarrow 0, \forall s \in \mathcal{S}, \forall a \in \mathcal{A}(s)$;
$V_i(s) \leftarrow 0, \forall s \in \mathcal{S}$;
repeat
  forall $s \in \mathcal{S}$ **do**
    forall $a \in \mathcal{A}(s)$ **do**
      $Q_{i+1}(s,a) \leftarrow c(s,a) + \gamma \sum_{s' \in \mathcal{S}} p(s'|s,a) V_i(s)$;
    end
    $V_{i+1}(s) = \min_{a' \in \mathcal{A}(s)} Q_{i+1}(s,a')$;
  end
  $i \leftarrow i+1$;
until $\max_{s,a} |Q_i(s,a) - Q_{i-1}(s,a)| < \epsilon$;
$\pi^*(s) = \arg\min_a Q_i(s,a), \forall s \in \mathcal{S}$

Trading-off Objectives

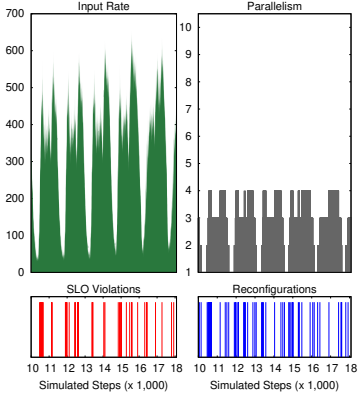
VI (perf=.6, res=rcf=.2)



VI (rcf=.6, perf=res=.2)

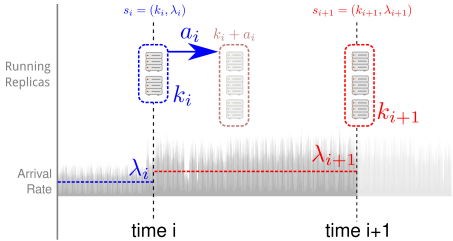


VI (res=.6, perf=rcf=.2)



Vertical/Horizontal Scaling: an MDP formulation

- We can extend previous model to account for Vertical Scaling [F. Rossi, Nardelli, and V. Cardellini 2019]

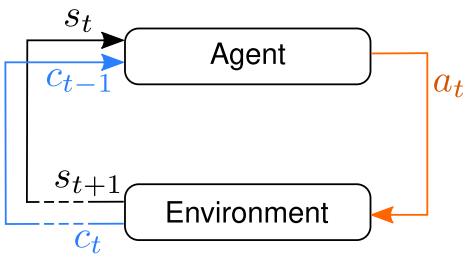


- State at time slot i: $s_i = (k_i, r_i, \lambda_i)$
  - r_i amount of resources allocated to each component replica
- r_i is discretized
  - Δr quantisation step size
- Action at time slot i: $a_i \in \{0, +1, -1, \Delta r, -\Delta r\}$ or even more general $a_i \in \{0, +1, -1\} \times \{\Delta r, 0, -\Delta r\}$

MDP Resolution

- We can use the Value Iteration algorithm to solve the MDP
- i.e., finding the optimal policy
- Is this enough?
- Unfortunately, solving the MDP requires exact and complete knowledge of the underlying model
  - state transition probabilities
  - cost function
- In practice, we don't have such information!

Reinforcement Learning



- Learn the optimal policy through interaction and evaluative feedback

General Reinforcement Learning Algorithm

RL-based Auto-Scaling

1 $t \leftarrow 0$
2 Initialize the Q functions
3 **Loop**
4 choose a scaling action a_t (based on current estimates of Q)
5 observe the next state s_{t+1} and the incurred cost c_t
6 update the $Q(s_t, a_t)$ functions based on the experience
7 $t \leftarrow t + 1$
8 **EndLoop**

Q-learning: Choosing an action

3 **Loop**
4 choose a scaling action a_t (based on current estimates of Q)
5 observe the next state s_{t+1} and the incurred cost c_t

Exploration vs Exploitation dilemma

- Exploitation exploits known information to minimize cost
  - choose the "best" action, i.e., $a_t = \arg\min_a Q(s_t, a)$
- Exploration finds more information about the environment
  - choose other actions to learn more about the system behavior

Convergence requires all state-pairs to be visited an infinite number of times

- you can't exploit all the time
- you can't explore all the time

ϵ-Greedy Exploration

3 **Loop**
4 choose a scaling action a_t (based on current estimates of Q)
5 observe the next state s_{t+1} and the incurred cost c_t

- All m actions are tried with non-zero probability
- With probability $1-\epsilon$ choose the greedy action $a^* = \arg\max_{a \in \mathcal{A}} Q(s,a)$
- With probability ϵ choose an action at random

$$\pi(a|s) = \begin{cases} \epsilon/m + 1 - \epsilon & a^* = \arg\max_{a \in \mathcal{A}} Q(s,a) \\ \epsilon/m & \text{otherwise} \end{cases}$$

Q-learning: Q updates

4 choose a scaling action a_t (based on current estimates of Q)
5 observe the next state s_{t+1} and the incurred cost c_t
6 update the $Q(s_t, a_t)$ functions based on the experience
7 $t \leftarrow t + 1$

With known model we can compute Q iteratively, e.g., using value iteration:

$$Q(s, a) \leftarrow c(s, a) + \gamma \sum_{s' \in S} p(s'|s, a) \min_{a'} Q(s', a')$$

With unknown model, we can update Q using *point estimates* on experience $\{s_t, a_t, c_t, s_{t+1}\}$:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t \left(c_t + \gamma \min_{a' \in \mathcal{A}} Q(s_{t+1}, a') - Q(s_t, a_t) \right)$$

Learning Rate — Target

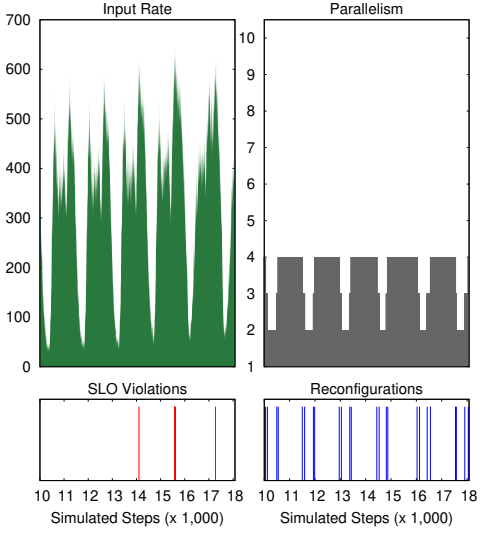
Q-learning

Q-learning Auto-Scaling

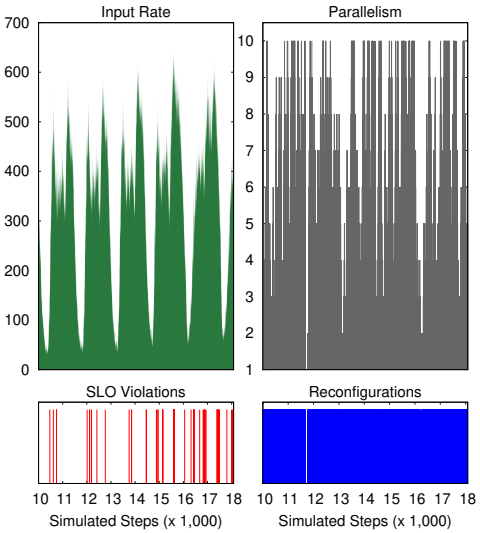
1 $t \rightarrow 0$
2 Initialize the Q functions
3 **Loop**
4 choose a scaling action a_i based on current estimates of Q, e.g., ϵ-greedy
5 observe the next state s_{i+1} and the incurred cost c_t
6 $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t \left[c_t + \gamma \min_{a' \in \mathcal{A}} Q(s_{t+1}, a') - Q(s_t, a_t)\right]$
7 $t \leftarrow t + 1$
8 **EndLoop**

MDP (VI) vs Q-learning

VI



Q-learning



VI vs Q-Learning

- Transition probability

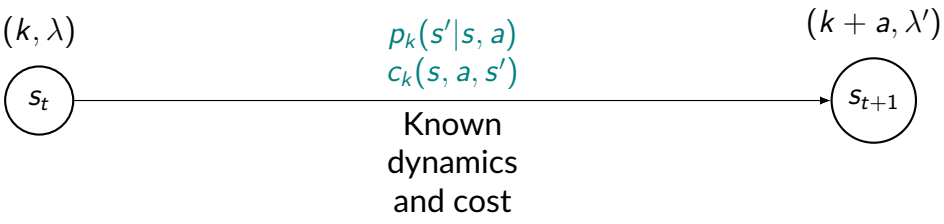
$$p(s'|s,a) = \mathbb{1}_{\{k'=k+a\}} P[\lambda_{t+} = \lambda'|\lambda_i = \lambda]$$

- Cost associated with action execution and state transition $(s,a) \rightarrow s'$

$$c(s,a,s') = w_{res}\frac{k+a}{K^{max}} + w_{perf}\mathbb{1}_{\{R(s,a,s')>R^{max}\}} + w_{rcf}\mathbb{1}_{\{a\neq 0\}}$$

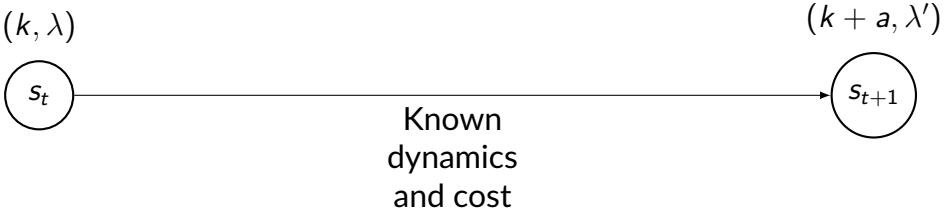
VI vs Q-Learning

VI assumes knowledge of system dynamics and costs



VI vs Q-Learning

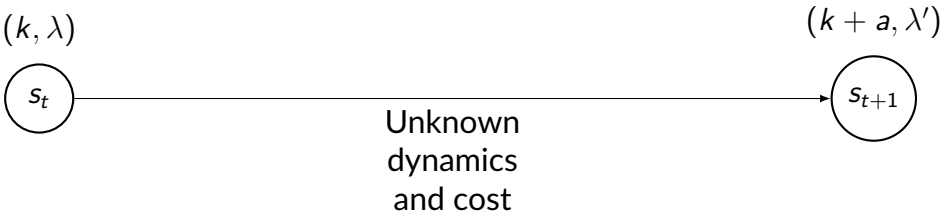
VI assumes knowledge of system dynamics and costs



$$Q^*(s,a) \leftarrow c_k(s,a) + \gamma \sum_{s' \in S} p_k(s'|s,a) V^*(s')$$
$$V^*(s) \leftarrow \min_a Q^*(s,a)$$

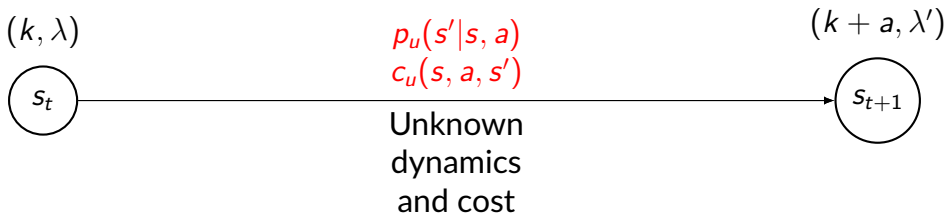
VI vs Q-Learning (2)

Q-learning does not require prior knowledge of system dynamics and costs



Vi vs Q-Learning (2)

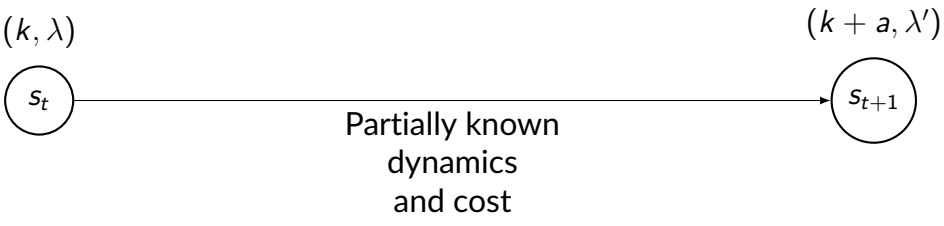
Q-learning does not require prior knowledge of system dynamics and costs



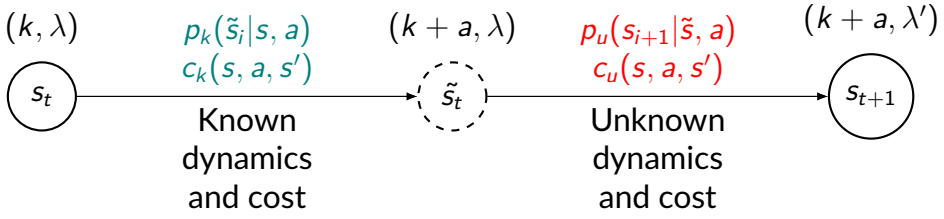
$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t \left(c_t + \gamma \min_{a' \in \mathcal{A}} Q(s_{t+1}, a') - Q(s_t, a_t) \right)$$

VI vs Q-Learning (3)

Sometimes we know something but not everything!



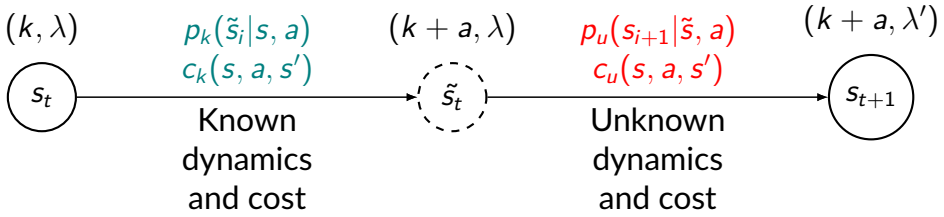
Separating the Known from the Unknown



Post-Decision state $\tilde{s}_t$

- $\tilde{s}_t = (k + a, \lambda)$ is the system state after the known dynamics and the action have taken place but before the unknown dynamics has occurred
- $\tilde{V}(\tilde{s}_t)$ Post-decision state value function

Separating the Known from the Unknown



$$Q(s_t, a) \leftarrow c_k(s_t, a) + \sum_{\tilde{s}} p_k(\tilde{s}|s_t, a)\tilde{V}(\tilde{s})$$

$$V(s_t) \leftarrow \min_{a \in \mathcal{A}} Q(s_t, a)$$

$$\tilde{V}(\tilde{s}_t) \leftarrow (1 - \alpha_t)\tilde{V}(\tilde{s}_t) + \alpha_t(c_u + \gamma V(s_{t+1}))$$

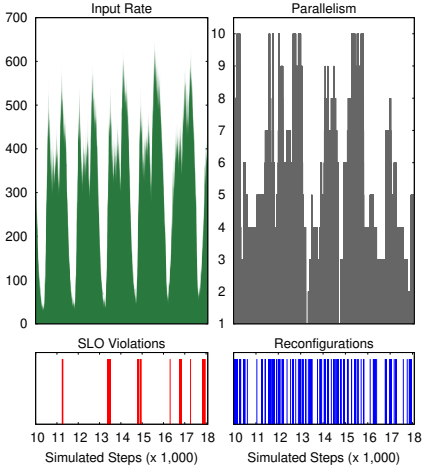
PDS-learning

PDS-learning PDS-learning Auto-Scaling

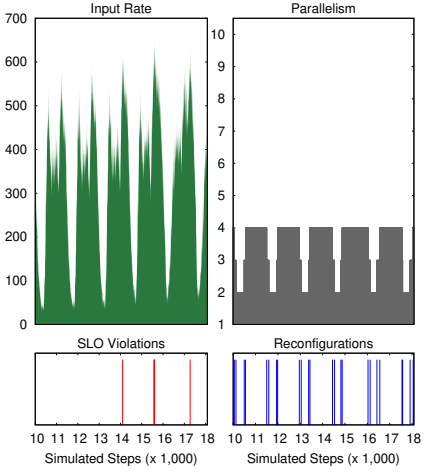
1 $t \rightarrow 0$
2 Initialize the $\tilde{V}(\tilde{s})$ functions
3 **Loop**
4 $a_t \leftarrow min_{a \in \mathcal{A}} \left\{ c_k(s_t, a) + \sum_{\tilde{s}} p_k(\tilde{s}|s_t, a) \tilde{V}(\tilde{s}) \right\}$; /* a_t=greedy action */
5 observe the post decision-state $\tilde{s}_t$ and the incurred cost c_k
6 $Q(s_{t+1}, a) \leftarrow c_k(s_{t+1}, a) + \sum_{\tilde{s}} p_k(\tilde{s}|s_{t+1}, a) \tilde{V}(\tilde{s}), \forall a \in \mathcal{A}$
7 $V(s_{t+1}) \leftarrow \min_{a \in \mathcal{A}} Q(s_{t+1}, a)$
8 $\tilde{V}(\tilde{s}_t) \leftarrow (1 - \alpha_t) \tilde{V}(\tilde{s}_t) + \alpha_t (c_u + \gamma V(s_{t+1}))$
9 $t \leftarrow t + 1$
10 **EndLoop**

PDS vs MDP (VI)

QL-PDS

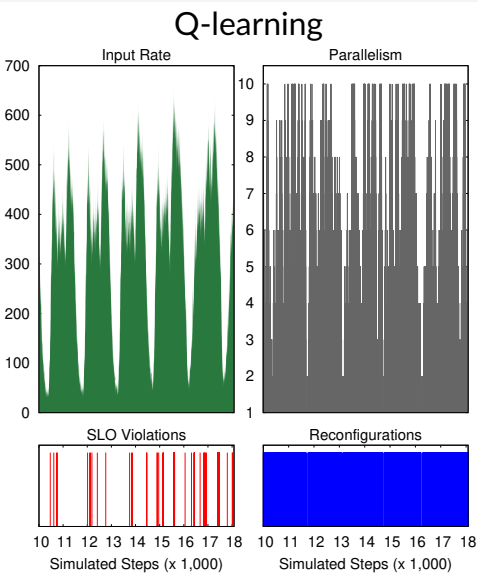


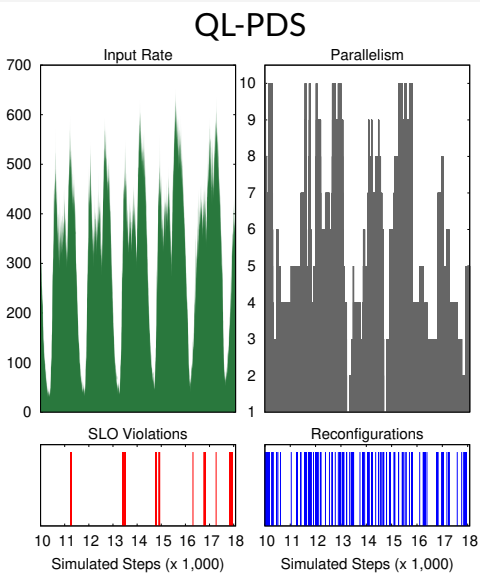
MDP (VI)



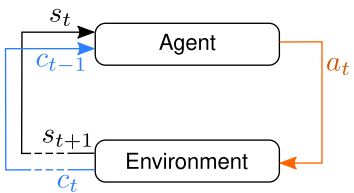
Note: Q-learning converges to the optimal MDP policy, but it needs more time

PDS vs Q-learning





Model Based Reinforcement Learning



- Idea: Do not learn the optimal behaviour directly
- Instead, learn a model of the environment by taking actions and observing the outcomes that include the next state and the immediate cost

Parameters Estimation

1. Estimate probabilities $p(s'|s,a)$ with empirical frequencies
   - In our setting, this requires estimating $P[\lambda'|\lambda]$
   - Observe arrival rate $\lambda_1, \lambda_2, \ldots$, and compute estimates
     $$\hat{p}(\lambda', \lambda) = \frac{\sum_{h=0}^{t-1} \mathbb{1}_{\{\lambda_{h+1}=\lambda', \lambda_h=\lambda\}}}{\sum_{h=0}^{t-1} \mathbb{1}_{\{\lambda_h=\lambda\}}}$$
2. Estimate costs $c(s,a,s')$ with time averages
   - In our setting, this requires estimating $\mathbb{1}_{\{R(s,a,s')>R^{max}\}}$ by their empirical frequencies
3. Use estimates $\hat{p}(s'|s,a)$ and $\hat{c}(s,a,s')$ in the Value Iteration algorithm

$$Q(s,a) \leftarrow \sum_{s' \in \mathcal{S}} \hat{p}(s'|s,a) \left[\hat{c}(s,a,s') + \gamma \min_{a' \in \mathcal{A}} Q(s',a') \right]$$

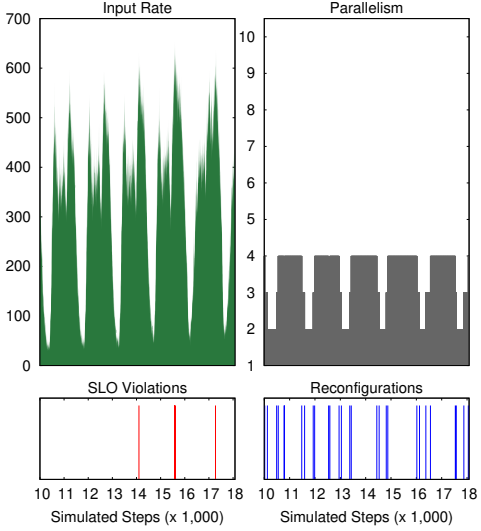
Model Based RL

Model Based Auto-Scaling

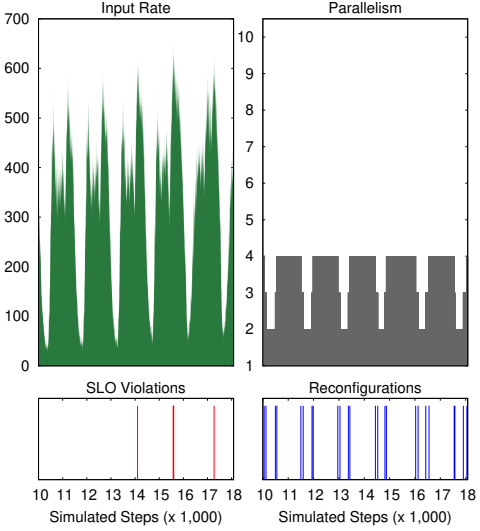
1 $t \rightarrow 0$
2 Initialize the Q functions
3 **Loop**
4 choose the greedy action $a_i = \arg\max_{a \in \mathcal{A}} Q(s_i, a)$
5 observe the next state s_{i+1} and the incurred cost c_i
6 update the estimates $\hat{p}(s'|s, a)$ and $\hat{c}(s, a, s')$
7 **forall** $s \in \mathcal{S}$ **do**
8 **forall** $a \in \mathcal{A}(s)$ **do**
9 $Q(s, a) \leftarrow \sum_{s' \in \mathcal{S}} \hat{p}(s'|s, a) \left[\hat{c}(s, a, s') + \gamma \min_{a' \in \mathcal{A}} Q(s', a')\right]$
10 **end**
11 **end**
12 $t \leftarrow t + 1$
13 **EndLoop**

Model Based vs MDP

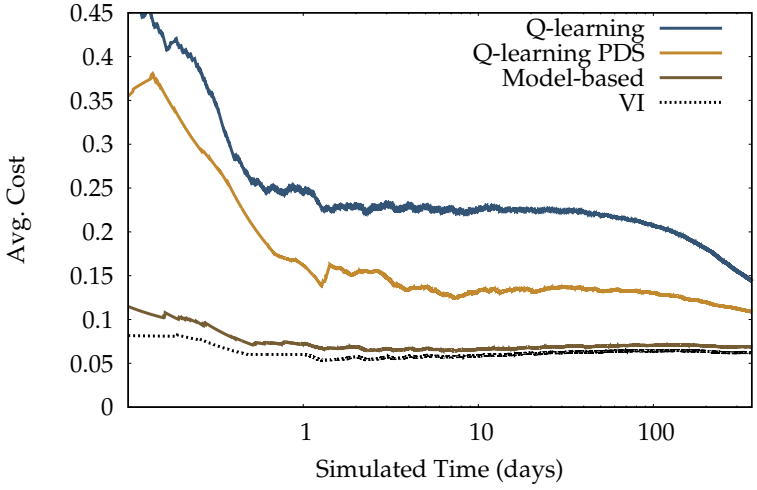
Model-based Full Backup



MDP (VI)



Performance Comparison



Dealing with Large State Spaces: Value Function Approximation

Issues with Tabular RL

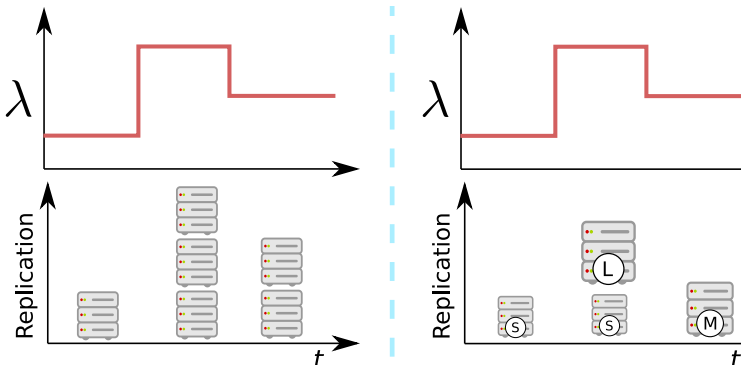
- So far, we have considered tabular representations of the value function

State/Action	a_1	a_2	...
s_1	$Q(s_1, a_1)$	$Q(s_1, a_2)$	...
s_2	$Q(s_2, a_1)$	$Q(s_2, a_2)$	...
...	...		
s_n	$Q(s_n, a_1)$	$Q(s_n, a_2)$	...

- Not ideal as the state space grows...
- Memory demand: $\mathcal{O}(|\mathcal{S}||\mathcal{A}|)$
- No generalization
- How to handle continuous state spaces?

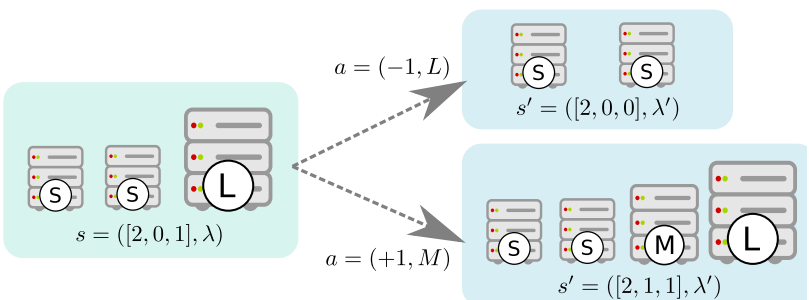
Example: Auto-Scaling on Heterogeneous Nodes

- Let's consider a heterogeneous computing infrastructure
  - Nodes with different types/amounts of resources
  - Trade-offs between cost, capacity, energy consumption, …
- We must decide how many replicas to run + which types of nodes to host them



Example: Updated MDP Formulation

- N resource types: $T_{res} = \{$  $\}$
- State $s = (\mathbf{k}, \lambda)$
  - k_i = # replicas on nodes of type i
  - λ = input rate
- Actions = $\mathcal{A}(s) = \{(\delta, \tau) : \delta \in \{-1, +1\}, \tau \in T_{res}\} \cup \{\text{“do} - \text{nothing”}\}$



Example: Updated MDP Formulation (2)

We also need to slightly update the cost function $c(s, a, s')$

- Resources cost

$$c_{res}(s, a, s') = \sum_{\tau \in T_{res}} k'_\tau c_\tau$$

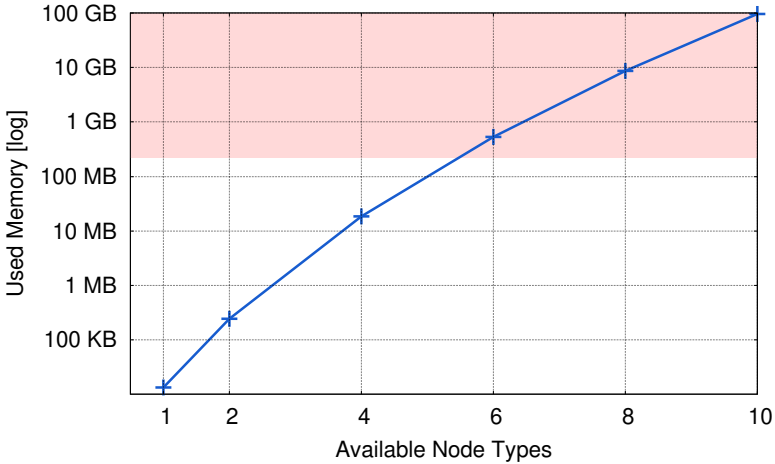
- Reconfiguration cost

$$c_{rcf}(s, a, s') = \mathbb{1}_{\{a \neq \text{do-nothing}\}}$$

- Performance violation penalty

$$c_{perf}(s, a, s') = \mathbb{1}_{\{R(s') > R_{max}\}}$$

Example: How Much Memory for the Q Table?



As each operator/microservice has its own policy, we would need a Q table for each of them!

Value Function Approximation

We can resort to parametric function approximation:

$$V_\pi(s) \approx \hat{V}(s, \mathbf{w})$$
$$Q_\pi(s, a) \approx \hat{Q}(s, a, \mathbf{w}')$$

- $\mathbf{w} \in \mathbb{R}^d$ is a vector of parameters
- $d \ll |\mathcal{S}|$ ✓
- No need to store the Q table ✓
- Potential generalization ✓

- How to pick a function $\hat{Q}$ (or $\hat{V}$)?
- How to compute $\mathbf{w}$?

Value Function Approximation (2)

- We search for a vector $\mathbf{w}$ so as to approximate V "well"
- A natural choice is to minimize the mean squared error:

$$J(\mathbf{w}) = \sum_{s \in \mathcal{S}} \mu(s) \left[V_\pi(s) - \hat{V}(s, \mathbf{w}) \right]^2$$

where $\mu(s) \geq 0$ is a distribution over states

- Weights updated through gradient descent

$$\begin{aligned} \mathbf{w}_{t+1} &= \mathbf{w}_t - \frac{1}{2} \alpha \nabla_{\mathbf{w}} J(\mathbf{w_t}) = \\ &= \mathbf{w}_t + \alpha \sum_{s \in \mathcal{S}} \mu(s) \left[V_\pi(s) - \hat{V}(s, \mathbf{w}) \right] \nabla_{\mathbf{w}} \hat{V}(s, \mathbf{w}) \end{aligned}$$

Updating Weights

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \sum_{s \in \mathcal{S}} \mu(s) \left[V_\pi(s) - \hat{V}(s, \mathbf{w}) \right] \nabla_{\mathbf{w}} \hat{V}(s, \mathbf{w})$$

How to get exact values?

Gradient computed over all states…

1) Stochastic gradient descent

one (or few) samples $(s_t, V_\pi(s_t))$ at each step

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[V_\pi(s_t) - \hat{V}(s_t, \mathbf{w}) \right] \nabla_{\mathbf{w}} \hat{V}(s_t, \mathbf{w})$$

2) Stochastic semi-gradient descent

we replace $V_\pi(s_t)$ with a noisy approximation U_t

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[U_t - \hat{V}(s_t, \mathbf{w}) \right] \nabla_{\mathbf{w}} \hat{V}(s_t, \mathbf{w})$$

Linear Function Approximation

$$\hat{V}(s, \boldsymbol{w}) = \boldsymbol{w}^T \boldsymbol{\phi}(s) = \sum_{i=1}^{d} w_i \phi_i(s)$$

Weights
$\boldsymbol{w} \in \mathbb{R}^d$

Features
$\boldsymbol{\phi} : \mathcal{S} \to \mathbb{R}^d$

Example: $\phi(s) = \frac{k}{K^{max}}$

Update rule becomes very simple:

$$\nabla_{\mathbf{w}} \hat{V}(s, \mathbf{w}) = \boldsymbol{\phi}(s)$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[U_t - \hat{V}(s_t, \mathbf{w_t}) \right] \boldsymbol{\phi}(s_t)$$

See also [Geramifard et al. 2013]

Linear Function Approximation (2)

We have equivalent formulas for Q:

$$\hat{Q}(s, a, \boldsymbol{w}) = \boldsymbol{w}^T \phi(s, a) = \sum_{i=1}^{d} w_i \phi_i(s, a)$$

Weights
$\boldsymbol{w} \in \mathbb{R}^d$

Features
$\phi : \mathcal{S} \times \mathcal{A} \to \mathbb{R}^d$

$$\nabla_{\mathbf{w}} \hat{Q}(s, a, \mathbf{w}) = \phi(s, a)$$

$$\mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[U_t - \hat{Q}(s_t, a_t, \mathbf{w_t}) \right] \phi(s_t, a_t)$$

??

Q-learning + Linear FA

- Q-learning: $Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha_t \left[c_t + \gamma \min_{a' \in \mathcal{A}} Q(s_{t+1}, a') - Q(s_t, a_t)\right]$

1 $t \rightarrow 0$
2 Initialize $\boldsymbol{w}$
3 **Loop**
4 $\quad a_i \leftarrow \epsilon$-greedy action selection
5 $\quad$ gather experience $\langle s_i, a_i, c_i, s_{i+1} \rangle$
6 $\quad U_t \leftarrow c_i + \gamma \min_{a' \in \mathcal{A}} \hat{Q}(s_{t+1}, a', \boldsymbol{w_t})$
7 $\quad \mathbf{w}_{t+1} = \mathbf{w}_t + \alpha \left[U_t - \hat{Q}(s_t, a_t, \mathbf{w_t})\right] \phi(s_t, a_t)$
8 $\quad t \leftarrow t + 1$
9 **EndLoop**

Feature Construction

- Accuracy depends on how states (and actions) are represented
- Features should capture (relevant) properties
- Domain-specific
- Note: we need $d \ll |\mathcal{S}|$

Tabular RL as a special case of linear FA

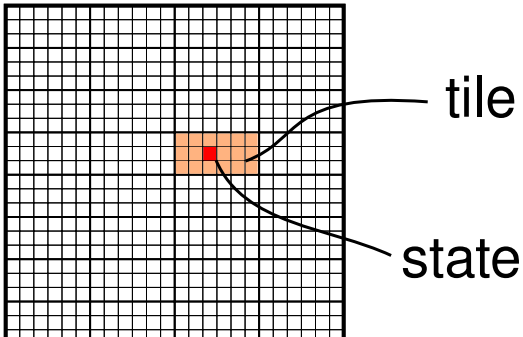
$$\phi(s) = \begin{bmatrix} \mathbb{1}_{\{s=s_1\}} \\ \mathbb{1}_{\{s=s_2\}} \\ \cdots \end{bmatrix}$$

After learning, we'd expect:
$w_i = V_\pi(s_i)$ and, thus, $\hat{V}(s) = V_\pi(s)$

Of course, no memory savings!

Tile Coding

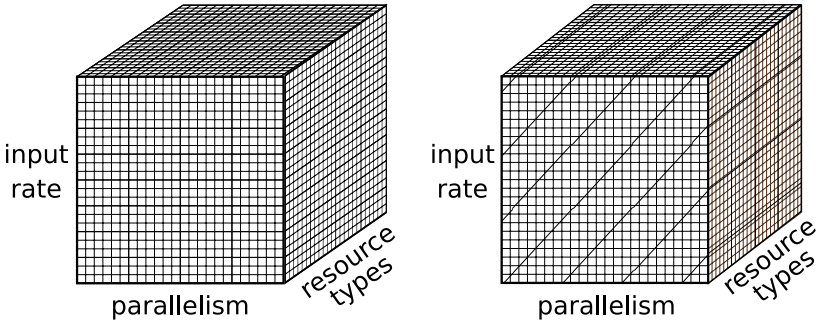
- Handcrafting features for non-trivial problems seems unfeasible
- We can exploit tile coding
  - Partition the state space with “tiles”
  - 1 tile $\rightarrow$ 1 binary feature
  - Generalization over contiguous regions
  - Multiple shapes, sizes and layers



Example: Tile Coding for Auto-Scaling

We aggregate “similar” states along 3 dimensions

- input rate
- parallelism
- set of used resource types



[Russo Russo, V. Cardellini, and F. Lo Presti 2019]

Problem: Representing States and Actions

- To approximate $Q(s, a)$, we need to represent state **and actions**!

$$\hat{Q}(s, a, \boldsymbol{w}) = \sum_i \phi_i(s, a)\, w_i$$

- **Simple approach**: replicate state features for every action

$$\boldsymbol{\phi}^S(s) = \begin{bmatrix} \phi_1(s) \\ \cdots \\ \phi_n(s) \end{bmatrix}$$

$$\boldsymbol{\phi}(s, a) = \begin{bmatrix} \phi_1(s)\mathbb{1}_{\{a=a_1\}} \\ \cdots \\ \phi_n(s)\mathbb{1}_{\{a=a_1\}} \\ \hline \cdots \\ \hline \phi_1(s)\mathbb{1}_{\{a=a_j\}} \\ \cdots \\ \phi_n(s)\mathbb{1}_{\{a=a_j\}} \\ \hline \cdots \end{bmatrix}$$

Post-decision States (again)

We aim to approximate:

$$Q_\pi(s,a) = E_\pi[G_t | S_t = s, A_t = a] =$$
$$= E_\pi[\sum_{k=0}^{\infty} \gamma^k \; c_{t+k} \; | S_t = s, A_t = a]$$

$$c_t = w_{res} \; c_{res}(s_t, a_t, s_{t+1}) + w_{perf} \; c_{perf}(s_t, a_t, s_{t+1}) + w_{rcf} \; c_{rcf}(s_t, a_t, s_{t+1})$$

$$\downarrow \qquad \downarrow \qquad \downarrow$$

$$\tilde{c}_{res}(\tilde{s}_t, s_{t+1}) \qquad \tilde{c}_{perf}(\tilde{s}_t, s_{t+1}) \qquad \tilde{c}_{rcf}(a_t)$$

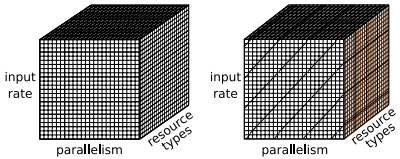
Different features to capture PDS-related and action-related costs:

$$\hat{Q}(s,a,\boldsymbol{w}) = \sum_{i=1}^{d'} \phi_i^S(\tilde{s}) w_i + \sum_{i=d'+1}^{d} \phi_i^A(a) w_i$$

Example: Features for Auto-Scaling

$$\hat{Q}(s, a, \boldsymbol{w}) = \sum_{i=1}^{d'} \phi_i^S(\tilde{s}) w_i \; + \sum_{i=d'+1}^{d} \phi_i^A(a) w_i$$

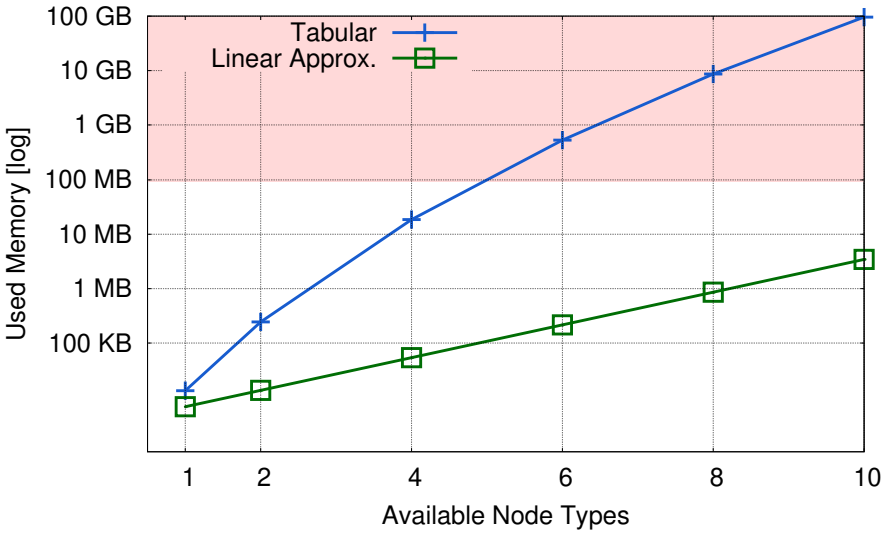
Tilings



A single feature is enough!

$$\phi^A(a) = \begin{cases} 0 & a = \text{do} - \text{nothing} \\ 1 & \textit{otherwise} \end{cases}$$

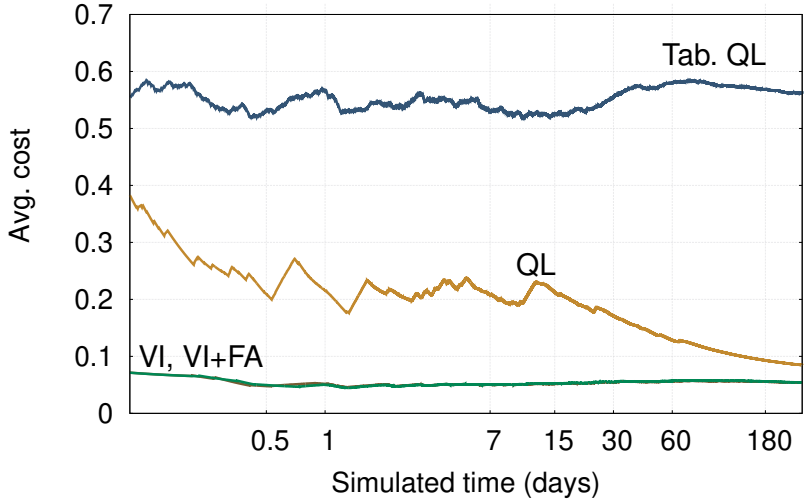
Example: How Much Memory?



Example: Results

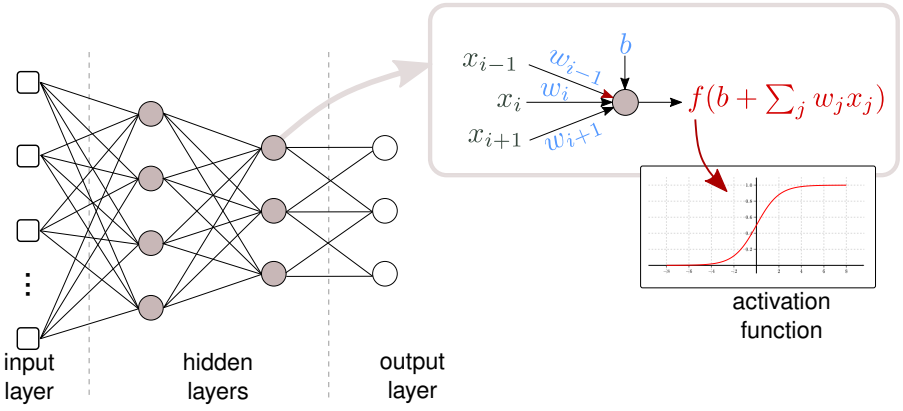
Results with 3 types of nodes

- **Dyn. Prog.**: VI, VI+FA
- **Q-learning**: Tabular QL, QL



Beyond Linear Approximation

- Natural upgrade: nonlinear approximation
- Artificial Neural Networks

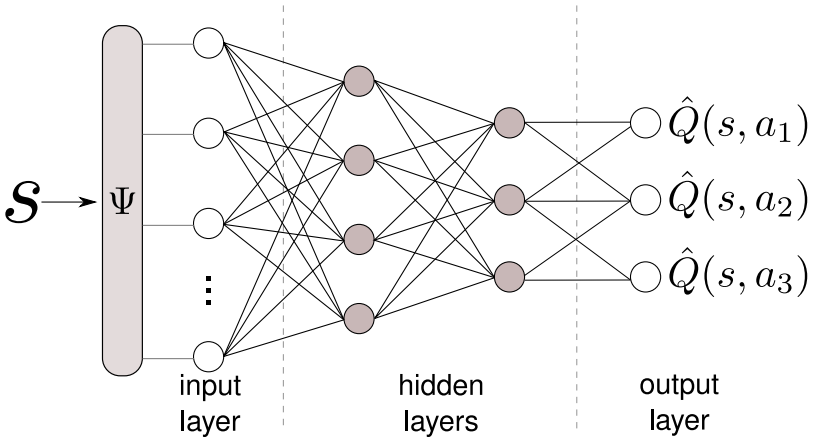


Deep Neural Networks and RL

- We aim to exploit approximation ability of ANNs
- …and avoid the burden of handcrafting features
- Deep neural networks (DNNs)
- Each layer possibly builds a new representation of the input

Deep Q Network

- **Input**: a state s (with proper encoding)
- **Output**: $\hat{Q}(s, a)$, for every action a



See also [Mnih et al. 2015]

Training

Classic approach for network training

- Training data: (large) collection of examples $(\boldsymbol{x}_i, \boldsymbol{y}_i)$ to learn from
  - we would need pairs $(s_i, [Q(s_i, a_1) \cdots Q(s_i, a_n)]^T)$
- Suitable algorithms for gradient computation (e.g., backpropagation) and optimization (e.g., SGD)
- Possibly specialized hardware for training (e.g., GPUs, TPUs)
- …and many more issues (e.g., overfitting, vanishing gradients)

- Problem: we don't have true examples of $Q(s, a)$ to use for training
- We can estimate Q on-line based on experience (as usual in RL)

Training: Experience Replay

- Idea: performing training iteration using mini-batches of experience:

$$\langle s_t, a_t, s_{t-1}, c_t \rangle \leftarrow \langle s_{t-1}, a_{t-1}, s_{t-2}, c_{t-1} \rangle \leftarrow \langle s_{t-2}, a_{t-2}, s_{t-3}, c_{t-1} \rangle \leftarrow \ldots$$

- Experience tuples stored in a FIFO buffer (experience buffer) and replayed as needed
- Naive approach: mini-batch of size b, finite buffer of size b
  - i.e., at each iteration, train on most recent experience
  - sequential observations likely correlated X
  - less recent experience possibly forgotten X
- Smarter approach: mini-batch of size b, finite buffer of size $B > b$
  - at each iteration, b tuples drawn randomly from the buffer
  - correlation between observations reduced/removed ✓
  - if B is large, old observations are "seen" more than once ✓

Deep Q-learning (DQL)

1 Initialize $\boldsymbol{w}$
2 Initialize empty buffer $\mathcal{B}$
3 $i \leftarrow 0$
4 **Loop**
5 $a_i \leftarrow \epsilon$-greedy action selection
6 gather experience $e_i = \langle s_i, a_i, c_i, s_{i+1} \rangle$
7 store e_i in $\mathcal{B}$
8 sample minibatch of b $\langle s_j, a_j, s_{j+1}, c_j \rangle$ tuples from $\mathcal{B}$
9 $y_j \leftarrow c_j + \gamma \min_{a'} \hat{Q}(s_{i+1}, a', \mathbf{w}), j = 1, \ldots, b$ `/* compute targets */`
10 update $\boldsymbol{w}$ using samples $(y_j - \hat{Q}(s_j, a_j, \boldsymbol{w}))^2$ `/* e.g., gradient descent */`
11 $i \leftarrow i + 1$
12 **EndLoop**

Target Network

DQL may suffer from instability during training:

- As for linear FA, we can't compute true gradient values because we don't know the true value function $Q(s, a)$!
- We overcome this issue by using the approximated $\hat{Q}$ in the update target value:

$$y_j \leftarrow c_j + \gamma \min_{a'} \hat{Q}(s_{i+1}, a', \mathbf{w}), j = 1, \ldots, b$$

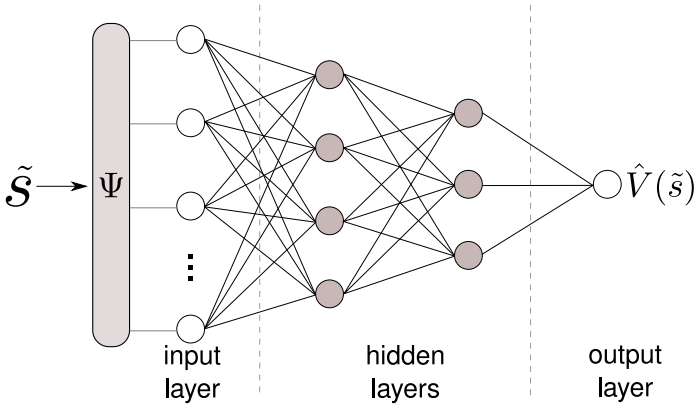
- ……but we keep changing $\boldsymbol{w}$ at each iteration
- Let's use a second neural network to stabilize the targets

Deep Q-learning with Target Network

1 Initialize $\boldsymbol{w}$ and $\boldsymbol{w}^- = \boldsymbol{w}$
2 Initialize empty buffer $\mathcal{B}$
3 $i \leftarrow 0$
4 **Loop**
5 $a_i \leftarrow \epsilon$-greedy action selection
6 gather experience $e_i = \langle s_i, a_i, c_i, s_{i+1} \rangle$
7 store e_i in $\mathcal{B}$
8 sample minibatch of b $\langle s_j, a_j, s_{j+1}, c_j \rangle$ tuples from $\mathcal{B}$
9 $y_j \leftarrow c_j + \gamma \min_{a'} \hat{Q}(s_{i+1}, a', \mathbf{w}^-), j = 1, \ldots, b$ `/* compute targets */`
10 update $\boldsymbol{w}$ using samples $(y_j - \hat{Q}(s_j, a_j, \boldsymbol{w}))^2$ `/* e.g., gradient descent */`
11 every C steps reset $\boldsymbol{w}^- = \boldsymbol{w}$
12 $i \leftarrow i + 1$

Deep Q-learning + PDS

- Can we (again) exploit post-decision states to get faster convergence?
- We need to revise our network architecture:

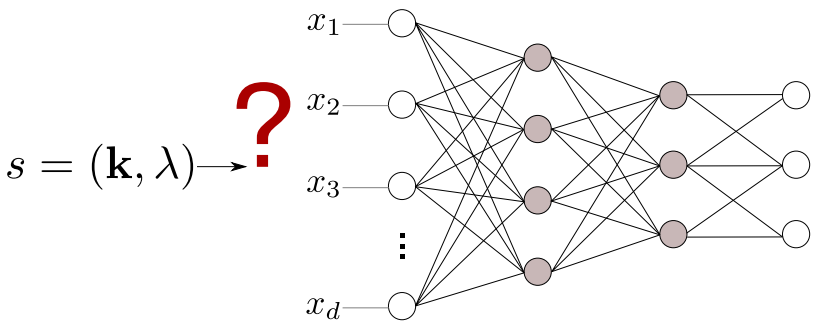


Deep Q-learning + PDS (2)

1 Initialize $\boldsymbol{w}$
2 Initialize empty buffer $\mathcal{B}$
3 $i \leftarrow 0$
4 Let $\hat{Q}(s, a, \boldsymbol{w}) := c_k(s, a) + \hat{V}(\tilde{s}, \boldsymbol{w})$
5 **Loop**
6 $a_i \leftarrow \arg\min_{a \in \mathcal{A}(s_i)} \hat{Q}(s_i, a, \boldsymbol{w})$
7 gather experience $e_i = \langle s_i, a_i, c_i - c_k(s_i, a_i), s_{i+1} \rangle$
8 store e_i in $\mathcal{B}$
9 sample minibatch of b $\langle s_j, a_j, s_{j+1}, c_{u,j} \rangle$ tuples from $\mathcal{B}$
10 $y_j \leftarrow c_{u,j} + \gamma \min_{a'} \hat{Q}(s_{i+1}, a', \mathbf{w}), j = 1, \ldots, b$ `/* compute targets */`
11 update $\boldsymbol{w}$ using samples $(y_j - \hat{V}(\tilde{s}_j, \boldsymbol{w}))^2$ `/* e.g., gradient descent */`
12 $i \leftarrow i + 1$

State Representation

- DNNs are able to learn representations of the input data at each layer
- So, we can forget about feature engineering, right?



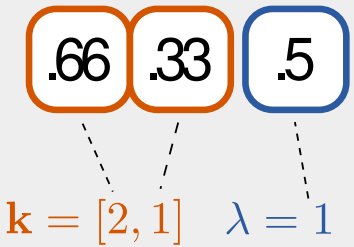
- Some features $\mathbf{\Psi} : \mathcal{S} \rightarrow \mathbb{R}^d$ are still useful

State Representation: Minimal

- We just normalize each state component
- Size of the input layer: $N_{res} + 1$

Example

Let's assume $N_{res} = 2$, $K^{max} = 3$ and $\lambda \in \{0, 1, 2\}$.

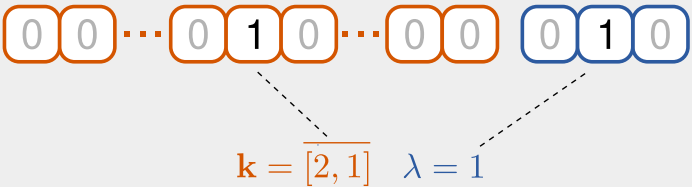


State Representation: Large

- We use one-hot coding both for $\boldsymbol{k}$ and λ
  - 1 input for each possible value of $\boldsymbol{k}$
  - 1 input for each possible value of λ

Example

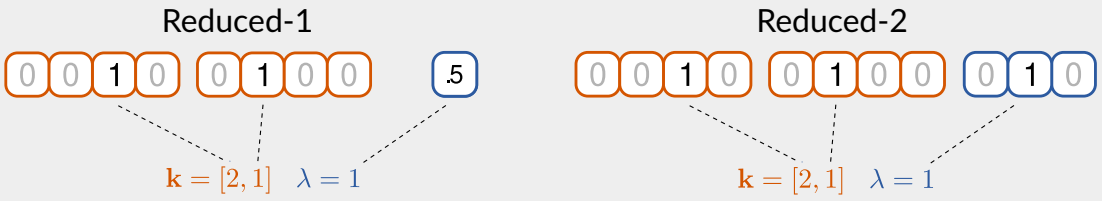
Let's assume $N_{res} = 2$, $K^{max} = 3$ and $\lambda \in \{0, 1, 2\}$.



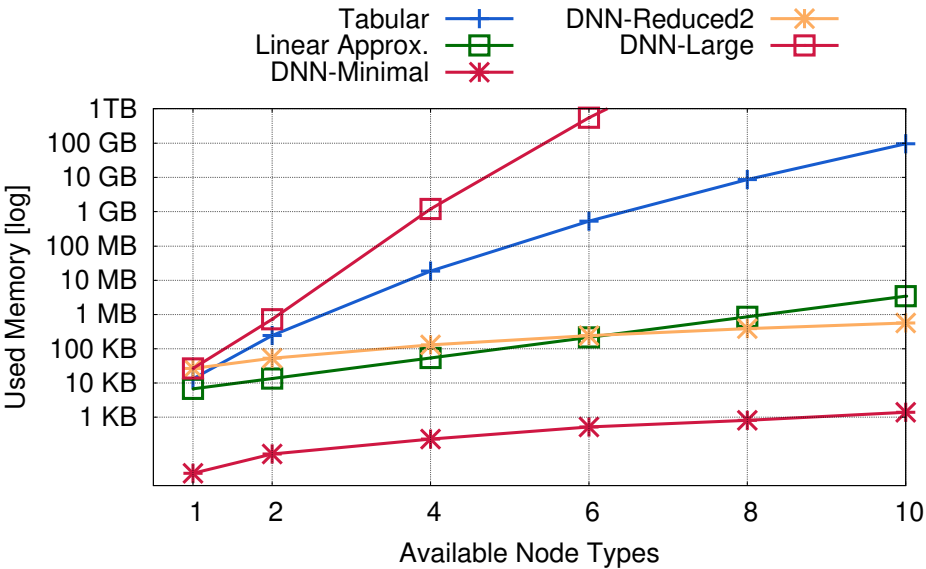
State Representation: Reduced

- We use one-hot coding for each component of $\boldsymbol{k}$
- We normalize λ (**Reduced-1**) or rely on a one-hot vector (**Reduced-2**)
- At most, $K^{max} \times N_{res} + \Lambda^{max}$ inputs

Example - $N_{res} = 2$, $K^{max} = 3$ and $\lambda \in \{0, 1, 2\}$

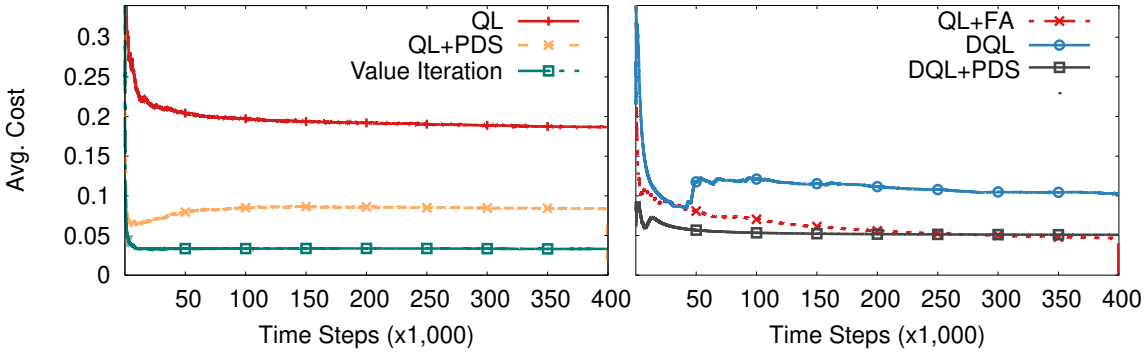


Memory Demand

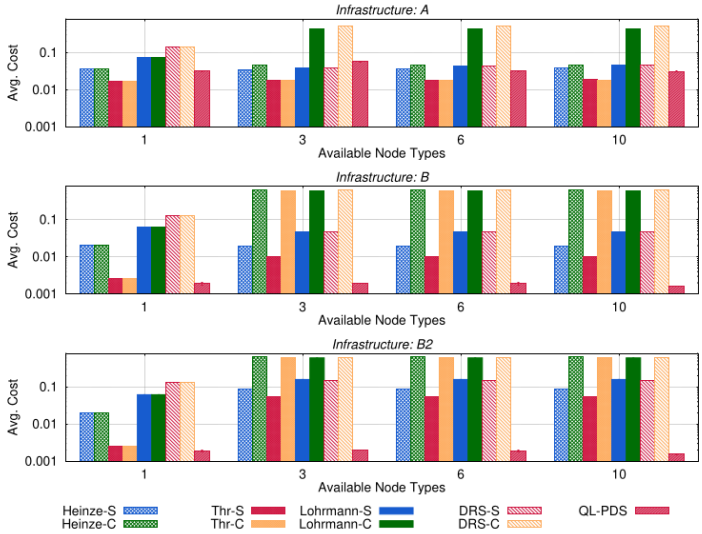


Results

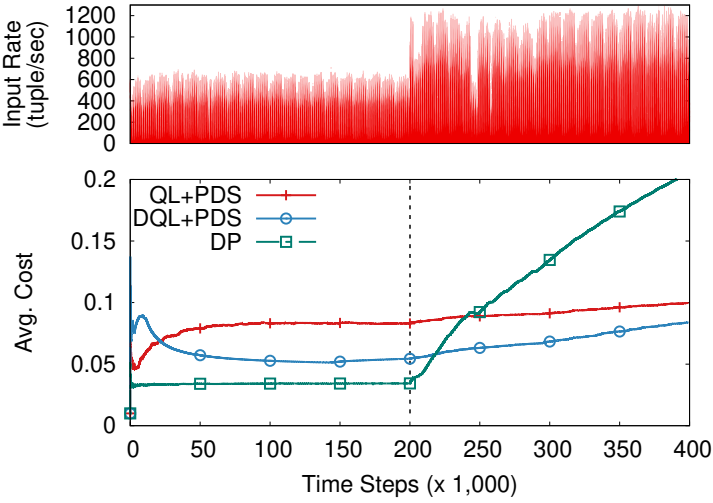
- 3 types of nodes; $K^{max} = 10$



Results: against baselines



Results: Non-Stationary Environment

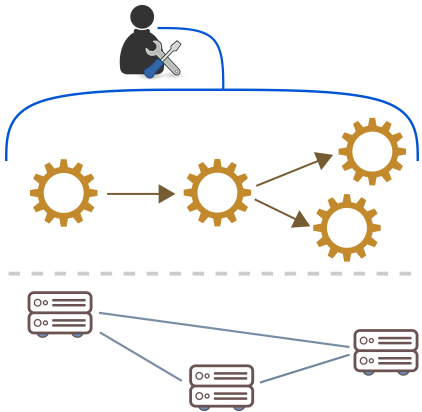


More results: Russo Russo, V. Cardellini, and Lo Presti 2023

Application-level Control using Bayesian Optimization

Application Auto-Scaling

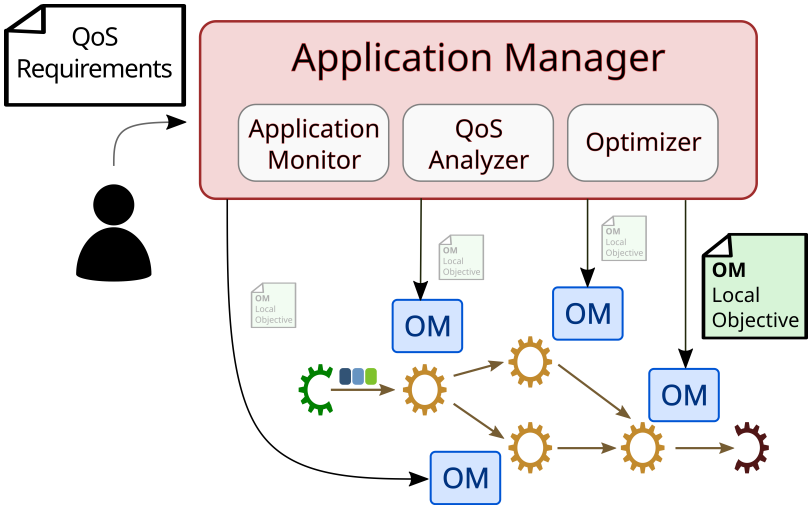
- We have apps, not just single components...
- Performance requirements (e.g., max response time) expressed at application-level
- First idea: Design a global RL controller
- Problem Action/State Space Explosion
  - $\mathcal{S} = (\mathbf{s}, \boldsymbol{\lambda})$, $\mathbf{s} = \{s^1, \ldots, s^M\}$, $\boldsymbol{\lambda} = \{\lambda^1, \ldots, \lambda^M\}$
  - $\mathcal{A} = \times_{i=1}^{M} \mathcal{A}_i$, $\mathcal{A}_i$ component i action space
- We exploit the hierarchical architecture



Application Manager

- Per-application controller at the top layer
- Goal: tune (and adapt) the local objective configuration of component controllers w.r.t. application-level requirements
- Local objectives in our model depends on:
  - Cost weights
  - Local performance requirement (e.g., response time)

Solution Overview



Application Manager - Problem Definition

- Objective: minimize cost due to resource usage
- Constraint: performance reqs. satisfied at least X% of the time
- Constraint: app reconfigured no more than Y% of the time

$$\min_{\mathbf{w}, R_u^{max}} \quad \sum_{t=0}^{T} C_{res}(t)$$

$$\text{s.t.} \quad \frac{1}{T}\sum_{t=0}^{T} Viol(t) \leq \eta_V$$

$$\frac{1}{T}\sum_{t=0}^{T} Rcf(t) \leq \eta_R$$

$$w_{perf}, w_{rcf}, w_{res} \in (0,1)$$

$$R_u^{max} \geq 0, \forall u \in V_{dsp}$$

Problem Resolution

- The objective depends on locally learned policies!
- Cannot formulate it
- Black-box optimization

$$\min_{\mathbf{w}, R_u^{max}} \quad \sum_{t=0}^{T} C_{res}(t)$$

$$\text{s.t.} \quad \frac{1}{T}\sum_{t=0}^{T} Viol(t) \leq \eta_V$$

$$\frac{1}{T}\sum_{t=0}^{T} Rcf(t) \leq \eta_R$$

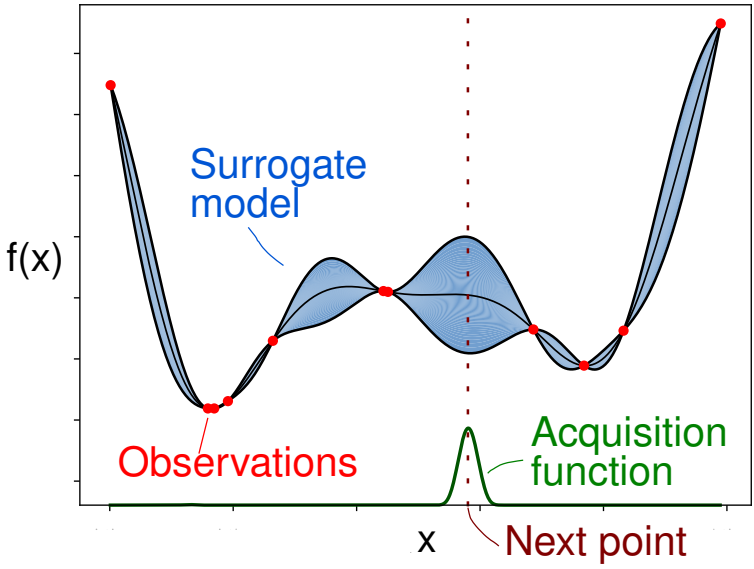
$$w_{perf}, w_{rcf}, w_{res} \in (0, 1)$$

$$R_u^{max} \geq 0, \forall u \in V_{dsp}$$

Bayesian Optimization (BO)

- Search for the global optimum of an objective f
- f can only be evaluated at arbitrary points
  - Limited number of evaluations
- BO builds a surrogate model to capture prior belief about f
  - Usually based on Gaussian Processes
- Surrogate model helps choosing promising points to sample
- How to evaluate the function in our case?
  - System observation (possibly slow)
  - Co-simulation

Bayesian Optimization (2)



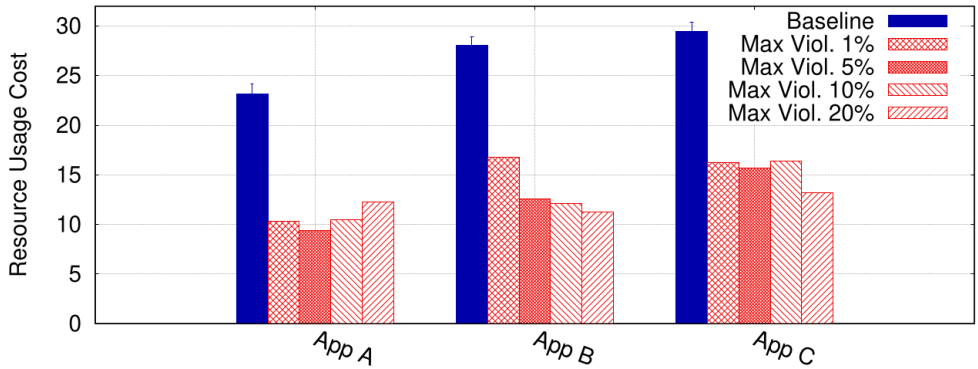
Bayesian Optimization (3)

Input: Black-box objective f, domain D, allowed iterations N_{BO}

choose arbitrarily $\mathbf{x_1}, \mathbf{x_2}, \ldots, \mathbf{x_{n_0}} \in D$, $n_0 < N_{BO}$
evaluate $f(\mathbf{x_1}), f(\mathbf{x_2}), \ldots, f(\mathbf{x_{n_0}})$
build a surrogate model of f using the available samples
$n \leftarrow n_0$
while $n < N_{BO}$ **do**
    pick $\bar{\mathbf{x}} \in D$ that maximizes the acquisition function
    evaluate $f(\bar{\mathbf{x}})$
    update the surrogate model using the new sample
    $n \leftarrow n + 1$
end while
return $\mathbf{x}$ that either is the point evaluated with largest $f(\mathbf{x})$ or maximizes the surrogate model

Results

Baseline: same response time requirement for every component, same weight for every objective term (cost/perf/overhead)



Hands-on

Our Implementation

- We have developed a simple auto-scaling simulator
- Implementation related to a specific use-case (DSP auto-scaling)
- RL algorithms implemented (almost) from scratch
- `https://github.com/grussorusso/dsp-elasticity-sim/`
- Test it following `docs/examples.md`

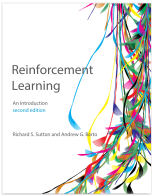
RL Libraries

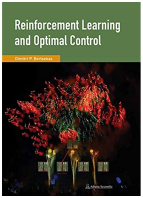
- If you want to use RL in your own research project, you should consider one of the (many!) RL libraries available nowadays, e.g.:
  - Keras RL
  - pyqlearning
  - Tensorforce
  - RLLib
  - …
- Example: `https://pytorch.org/tutorials/intermediate/reinforcement_q_learning.html`
- Try increasing the number of episodes!

Conclusion

Essential References

- MDPs and RL (including FA)
  Sutton and A. Barto 2018, Bertsekas 2019, David Silver 2015
- Linear Function Approximation for RL
  Geramifard et al. 2013
- Deep RL
  Mnih et al. 2015





Conclusion

- Uncertainty can play an important role in run-time system management
- RL provides a rich toolbox to learn control policies for complex tasks
- Blending together models and learning agents to learn from fewer data
- Deep RL for very large / continuous state spaces

Advanced Topics

There are many topics we haven't discussed, including:

- Policy-gradient methods and the Actor-Critic framework
  Sutton and A. Barto 2018; Grondman et al. 2012
- Multi-agent RL (cooperative/competitive)
  Busoniu, Babuska, and Schutter 2006
- Hierarchical RL
  A. G. Barto and Mahadevan 2003
- Combining RL and tree search techniques
  Sutton and A. Barto 2018; D. Silver et al. 2016
- Transfer Learning for RL
  Taylor and Stone 2009

References I

- Barto, A. G. and S. Mahadevan (2003). "Recent Advances in Hierarchical Reinforcement Learning". In: *Discret. Event Dyn. Syst.* 13.4, pp. 341–379. DOI: 10.1023/A:1025696116075. URL: https://doi.org/10.1023/A:1025696116075.
- Bertsekas, Dimitri (2019). *Reinforcement Learning and Optimal Control*. Athena Scientific.
- Busoniu, L., R. Babuska, and B. De Schutter (2006). "Multi-Agent Reinforcement Learning: A Survey". In: *Proc. of Int'l Conference on Control, Automation, Robotics and Vision*. IEEE, pp. 1–6. DOI: 10.1109/ICARCV.2006.345353. URL: https://doi.org/10.1109/ICARCV.2006.345353.
- Geramifard, A. et al. (2013). "A Tutorial on Linear Function Approximators for Dynamic Programming and Reinforcement Learning". In: *Found. Trends in Mach. Learn.* 6.4, pp. 375–451.

References II

- Grondman, I. et al. (2012). “A Survey of Actor-Critic Reinforcement Learning: Standard and Natural Policy Gradients”. In: *IEEE Trans. Syst. Man Cybern. Part C* 42.6, pp. 1291–1307. DOI: 10.1109/TSMCC.2012.2218595. URL: https://doi.org/10.1109/TSMCC.2012.2218595.
- Khatua, Sunirmal, Anirban Ghosh, and Nandini Mukherjee (2010). “Optimizing the utilization of virtual resources in Cloud environment”. In: *2010 IEEE International Conference on Virtual Environments, Human-Computer Interfaces and Measurement Systems*, pp. 82–87. DOI: 10.1109/VECIMS.2010.5609349.
- Lombardi, Federico et al. (2018). “Elastic Symbiotic Scaling of Operators and Resources in Stream Processing Systems”. In: *IEEE Trans. Parallel Distributed Syst.* 29.3, pp. 572–585. DOI: 10.1109/TPDS.2017.2762683. URL: https://doi.org/10.1109/TPDS.2017.2762683.

References III

- Lorido-Botrán, Tania, Jose Miguel-Alonso, and Jose Lozano (Jan. 2013). "Comparison of Auto-scaling Techniques for Cloud Environments". In.
- Mnih, V. et al. (2015). "Human-level control through deep reinforcement learning". In: *Nat.* 518.7540, pp. 529–533. DOI: 10.1038/nature14236. URL: https://doi.org/10.1038/nature14236.
- Rossi, F., M. Nardelli, and V. Cardellini (2019). "Horizontal and Vertical Scaling of Container-Based Applications Using Reinforcement Learning". In: *Proc. of 12th IEEE CLOUD 2019*. IEEE, pp. 329–338. DOI: 10.1109/CLOUD.2019.00061. URL: https://doi.org/10.1109/CLOUD.2019.00061.

References IV

- Rossi, Fabiana, Valeria Cardellini, and Francesco Lo Presti (2020). "Self-adaptive Threshold-based Policy for Microservices Elasticity". In: *28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems, MASCOTS 2020, Nice, France, November 17-19, 2020*. IEEE, pp. 1–8. DOI: 10.1109/MASCOTS50786.2020.9285951. URL: https://doi.org/10.1109/MASCOTS50786.2020.9285951.
- Russo Russo, G., V. Cardellini, G. Casale, et al. (2021). "MEAD: Model-Based Vertical Auto-Scaling for Data Stream Processing". In: *Proceedings of 21th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing, CCGRID '21, Virtual Event, May 10-13, 2021*, pp. 314–323. DOI: 10.1109/CCGrid51090.2021.00041.

References V

- Russo Russo, G., V. Cardellini, and F. Lo Presti (2019). "Reinforcement Learning Based Policies for Elastic Stream Processing on Heterogeneous Resources". In: *Proc. of 13th ACM Int'l Conference on Distributed and Event-based Systems, DEBS '19*, pp. 31–42.
- — (2023). "Hierarchical Auto-Scaling Policies for Data Stream Processing on Heterogeneous Resources". In: *ACM Transactions on Autonomous and Adaptive Systems*. DOI: 10.1145/3597435. URL: https://doi.org/10.1145/3597435.
- Silver, D. et al. (2016). "Mastering the Game of Go with Deep Neural Networks and Tree Search". In: *Nat.* 529.7587, pp. 484–489. DOI: 10.1038/nature16961. URL: https://doi.org/10.1038/nature16961.
- Silver, David (2015). *Introduction to Reinforcement Learning*. URL: https://deepmind.com/learning-resources/-introduction-reinforcement-learning-david-silver.

References VI

- Silver, David et al. (2018). “A general reinforcement learning algorithm that masters chess, shogi, and Go through self-play”. In: *Science* 362.6419, pp. 1140–1144. DOI: 10.1126/science.aar6404. URL: https://www.science.org/doi/abs/10.1126/science.aar6404.
- Sutton, R. and A Barto (2018). *Reinforcement Learning: An Introduction*. second. MIT Press.
- Taylor, Matthew E. and Peter Stone (2009). “Transfer Learning for Reinforcement Learning Domains: A Survey”. In: *J. Mach. Learn. Res.* 10, pp. 1633–1685. URL: https://dl.acm.org/citation.cfm?id=1755839.