Auto-scaling in Data Stream Processing Applications: A Model Based Reinforcement Learning Approach

Valeria Cardellini, Francesco Lo Presti,
Matteo Nardelli, and Gabriele Russo Russo

Department of Civil Engineering and Computer Science Engineering
University of Rome Tor Vergata, Italy
{cardellini,nardelli}@ing.uniroma2.it, lopresti@info.uniroma2.it,
gab.russorusso@gmail.com

Abstract By exploiting data-parallelism, Data Stream Processing (DSP) applications can adjust the number of parallel instances which process the incoming streams of data as to keep consistent level of QoS in face of varying input rate. Considering a single operator in isolation, in this paper we consider Reinforcement Learning based techniques for determining at run-time the number of instances. In particular, we propose two model based approaches and compare them to the baseline Q-learning algorithm. Our numerical investigations show how the proposed solutions provide better performance and faster convergence.

Keywords: Data Stream Processing, Elasticity, Reinforcement Learning

1 Introduction

Under several emerging application scenarios (e.g., Internet of Things and smart cities) Data Stream Processing (DSP) applications are required to process in near real-time fast data streams, often arriving at an unpredictable rate. In order to process these continuous data streams in an efficient and scalable manner, the deployment of DSP applications should be accordingly adapted at runtime.

A DSP application is represented as a directed acyclic graph, with data sources, operators, and final consumers as vertices, and streams as edges. Each *operator* can be seen as a black-box processing element, that continuously receives incoming streams, applies a transformation, and generates new outgoing streams. To deal with the fact that some operators in the application can become overloaded, a commonly adopted stream processing optimization technique is *data parallelism*, which consists of scaling-out or scaling-in the number of parallel instances for the operators, so that each instance can process a subset of the incoming data flow in parallel [7]. Due to the highly variable rate at which the sources may produce the data streams, a static or manual configuration of the operator *parallelization degree* does not provide an effective solution. Therefore, a key design choice in a DSP system is to enable it with *auto-scaling*, where the parallelization degree of each operator is self-configured at run-time. Since scaling-in/out decisions have an associated cost, not only monetary and related to number of operator instances, but also in terms of reconfiguration cost, the auto-scaling policy should also take the latter into account.

In this paper, we consider the auto-scaling problem for a single DSP operator considered in isolation, and focus on the adoption of Reinforcement Learning for determining at run-time the parallelization degree of the operator. *Reinforcement Learning* (RL) refers to a collection of trial-and-error methods by which an agent can learn to make good decisions through a sequence of interactions with a system or environment [12]. It can be considered a special method belonging to the machine learning branch. RL tries to learn at run-time the most suitable action, with a trial-and-error approach. The adaptability nature of RL makes it very appealing to devise auto-scaling policies; however, standard RL policies suffer from long learning phases, to the point that the time required to converge to a near-optimal policy can be unfeasible in a running system.

To improve the convergence to the optimal policy and the quality of the solution, we propose two model-based RL approaches and compare them to the baseline and model-free Q-learning algorithm, which is the most used RL approach adopted in literature to determine auto-scaling decisions. The first model-based approach integrates a partial knowledge of the system state into the learning algorithm; the second policy exploits a full backup approach, increasing the level of knowledge. We numerically evaluate the three RL policies using a real trace from the New York City taxis and show that the model-based solutions provide better performance and faster convergence than that achieved by Q-learning.

The rest of this paper is organized as follows. We review related work in Sec. 2. In Sec. 3 we describe the auto-scaling problem for an isolated DSP operator, before presenting in Sec. 4 the system model and problem formulation. In Sec. 5 we present three RL-based approaches for learning the auto-scaling strategy and discuss their numerical evaluation in Sec. 6. Finally, we conclude in Sec. 7.

2 Related Work

Elasticity is a key feature for DSP systems that is attracting many research efforts. Most approaches that enable elasticity in DSP systems, e.g., [3], exploit best-effort threshold-based policies based on the utilization of either the system nodes or the operator instances. Other works, e.g., [1,2,8], use more complex policies to determine the scaling decisions, exploiting optimization theory [1], control theory [2], or queueing theory [8].

In the context of auto-scaling policies for self-adaptive systems, RL-based policies learn from experience the adaptation policy, i.e., they learn the best scaling action to take with respect to the system state through a trial-and-error process. The system state can consider the amount of incoming workload, the current application deployment, its performance, or a combination thereof.

After executing an action, the policy gets a response or reward from the system (e.g., performance improvement), which indicates how good that action was. One of the challenges that arise in reinforcement learning is the trade-off between *exploration* and *exploitation*. To maximize the obtained reward, a RL agent must prefer actions that it has tried in the past and found to be effective in producing reward (exploitation). However, in order to discover such actions, it has to try actions that it has not selected before (exploration). The dilemma is that neither exploration nor exploitation can be pursued exclusively without failing at the task. The agent must try a variety of actions and progressively favor those that appear to be best [12]. To the best of our knowledge, only one work [5] has so far exploited RL techniques to drive the auto-scaling decisions in DSP systems. Heinze et al. [5] propose a simple RL approach that learns from experience when to acquire and release computing nodes so to efficiently process the incoming workload. The per-operator auto-scaler populates a lookup table that associates the utilization of the node on which the operator is executed with the action to perform (i.e., scale in, scale out, or do nothing). The adaptation goal is to keep the system utilization within a specific range; the SARSA learning algorithm [12] is used to update the lookup table.

A larger number of works has exploited RL techniques to drive elasticity in the Cloud computing context, as surveyed in [9]. Most of them use the simple Q-learning RL algorithm (described in Sec. 5), which suffers from slow convergence, as we also show in Section 6. Tesauro et al. [13] observe that RL approaches can suffer from poor scalability in systems with a large state space, because the lookup table has to store a separate value for every possible state-action pair. Moreover, the performance obtained during the on-line training may be unacceptably poor, due to the absence of domain knowledge or good heuristics. To overcome these issues, they combine RL with a model of the system, defined using queuing theory, which computes the initial deployment decisions and drives the exploration actions. They use the SARSA learning algorithm, which however suffers from slow convergence as Q-learning. Differently from [13], in this work we consider two model-aware learning approaches which do not require a queuing model of the system and are able to achieve faster convergence and good system performance.

3 Problem Description

In this paper, we consider the elasticity problem for a single DSP operator. As shown in Fig. 1, the system comprises an operator which is replicated into a number of instances, the number of which can be adjusted to adapt to the - possibly highly - variable input tuple rate. Arriving tuples are redirected to an instance for being processed. For simplicity, and without lack of generality, we consider ideal redirection with even distribution of the incoming data among the operator parallel instances.

A system component, named *Operator Manager*, monitors the operator input rate, the operator target response time, which we assume defined by a Service

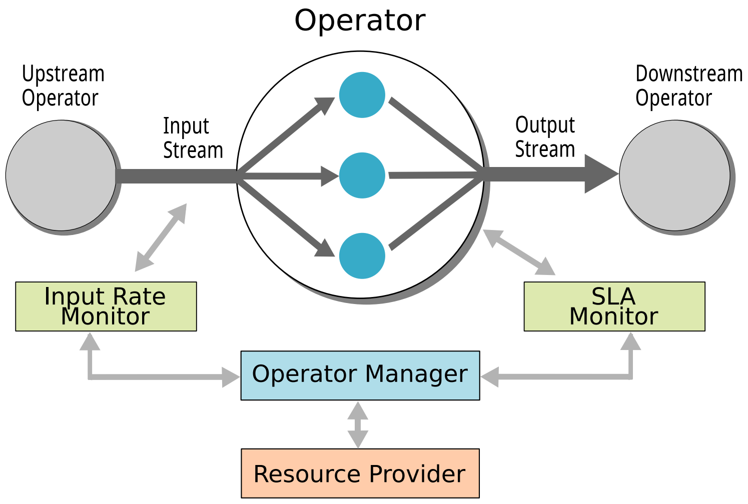


Figure 1: System architecture.

Level Agreement (SLA), and periodically adjusts the number of parallel instances used to run the operator. At each decision step, the Operator Manager can require to the system *Resource Provider* to add a new instance (scale-out), to terminate one of the running instances (scale-in), or to keep the current degree of parallelism (no change). Following a scaling decision, the operator is subject to a reconfiguration process in which the number of running instances is adjusted as requested. As the integrity of the stream and the operator internal state (if any) have to be preserved, the operator functionality is usually paused during the process [6], leading to *downtime*.

The goal of the Operator Manager is to take scaling decisions as to minimize a long-term cost function which accounts for the operator downtime and for the monetary cost to run the operator. The latter comprises: (i) the cost for running the number of instances during the next time slot, and (ii) possibly a penalty in case of SLA violation. In particular, we consider a constraint on the operator response time, so that a penalty is paid every time the response time exceeds a given threshold T_{SLA}.

4 System Model and Problem Formulation

Since decisions are taken periodically, we consider a slotted time system with fixed-length time intervals of length Δt, with the i-th time slot corresponding to the time interval $[i\Delta t, (i+1)\Delta t]$ (see Fig. 2). We denote by k_i the number of parallel instances at the beginning of slot i, and by λ_i the average tuple rate during slot $i-1$ (the previous slot). At the beginning of slot i the Operator-Manager makes the decision a_i on whether modify or keep the current instance configuration.

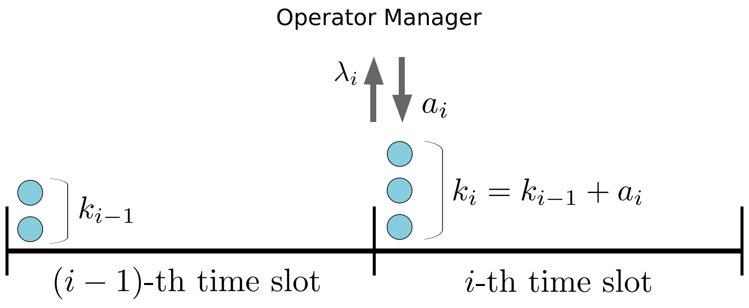


Figure 2: Relationship between λ_i, the average input rate measured over the previous time slot, the decision a_i made at the beginning of a time slot by the OperatorManager, and the resulting number of instances k_i.

We formulate the DSP Operator Elastic control problem as a discrete-time Markov Decision Process (MDP). A MDP is defined by a 5-tuple $\langle \mathcal{S}, A, p, c, \gamma \rangle$, where $\mathcal{S}$ is a finite set of states, $A(s)$ a finite set of actions for each state s, $p(s'|s,a)$ are the transition probabilities from state s to state s' given action $a \in A(s)$, $c(s,a)$ is the immediate cost when action a is executed in state s, and $\gamma \in [0,1]$ a discount factor that weights future costs.

In our setting, we define the state of the system as the pair $s_i = (k_i, \lambda_i)$, that is the number of operator instances and the tuple arrival rate. For the sake of analysis we consider a discrete state space, that is, we discretize the arrival rate λ_i by assuming that $\lambda_i \in \{0, \bar{\lambda}, \ldots, L\bar{\lambda}\}$ where $\bar{\lambda}$ is a suitable quantum (measured in *tuple/min*). We also assume that $k \in \{1, \ldots, K_{max}\}$.

For each state s, the action set is $\mathcal{A}(s) = \{+1, -1, 0\}$ except for those state with $k = 1$ where $\mathcal{A}(s) = \{+1, 0\}$ (at least one instance is always running), or $k = K_{max}$ where $\mathcal{A}(s) = \{-1, 0\}$ (we cannot add instances beyond the maximum allowed level).

System transitions occur as a consequence of auto-scaling decisions and tuple arrival rate variations. Let us denote by $p(s'|s,a)$ the transition probability from state s to state s' given action a. We readily obtain:

$$\begin{aligned} p(s'|s,a) = P[s_{i+1} = (k', \lambda') | s_i = (k, \lambda), a_i = a] &= \begin{cases} P[\lambda_{i+1} = \lambda' | \lambda_i = \lambda] & k' = k + a \\ 0 & \text{otherwhise} \end{cases} \\ &= \mathbb{1}_{\{k'=k+a\}} P[\lambda_{i+1} = \lambda' | \lambda_i = \lambda] \end{aligned} \tag{1}$$

where $\mathbb{1}_{\{\cdot\}}$ is the indicator function. It is easy to realize that the system dynamic comprises a stochastic component due to the tuple rate variation, which we assume exogenous, captured by the transition probabilities $P[\lambda_{i+1} = \lambda' | \lambda_i = \lambda]$, and a deterministic component due to the fact that, given action a, the number of instances k' is $k' = k + a$.

To each state pair (s, a) we associate a cost $c(s, a)$ which captures the cost of operating the system in state s and carrying out action a. In this paper we consider three different costs:

1. the instances cost $c_{res}(s,a)$, that is the cost of running $k+a$[1] instances of the operator. Assuming a fixed cost c_{res} for instance, we have $c_{res}(s,a) = (k+a)c_{res}$.
2. the reconfiguration cost c_{rcf}. Whenever the system carries out scale-out a or scale-in operation, the operator suffers a downtime period during which no tuple is processed. Since this downtime can be non-negligible especially for stateful operators [4,6], we need to account for the downtime by considering a reconfiguration penalty. For the sake of simplicity, we will assume c_{rcf} to be a constant.
3. a SLA violation cost c_{SLA} that captures a penalty incurred whenever the system response time violates a threshold.

We combine the different costs into a single cost function using the *Simple Additive Weighting* (SAW) technique [15]. According to SAW, we define the cost function $c(s,a)$ as the weighted sum of the normalized costs:

$$c(s,a) = w_{res}\frac{k+a}{K_{max}} + w_{rcf}\mathbb{1}_{\{a\neq 0\}} + w_{SLA}\mathbb{1}_{\{T(k+a,\lambda)>T_{SLA}\}} \tag{2}$$

where w_{res}, w_{rcf} and w_{SLA}, $w_{res}+w_{rcf}+w_{SLA}=1$, are non negative weights for the different costs. After normalization, the reconfiguration and SLA violation cost are binary functions which take value 0 when there is no reconfiguration/no violation, and take value 1 in case of reconfiguration/violation.

4.1 MDP Formulation

A policy is a function π that associates to each state s the action a to be adopted, which in our problem corresponds to a particular scaling decision. We are interested in determining the policy that minimizes the expected discounted cost with discounting factor $0 \leq \gamma < 1$. For a given policy π, let $V^{\pi}(s)$ be the value function, i.e., the expected infinite-horizon discounted cost given s as initial state, defined as $V^{\pi}(s) = E_s^{\pi}\left\{\sum_{i=0}^{\infty}\gamma^i c(s_i,a_i)\middle| s_0 = s\right\}$. The optimal policy π^* satisfies the Bellman optimality equation (see [11]):

$$V^{\pi^*}(s) = \min_{a\in\mathcal{A}(s)}\left\{c(s,a) + \gamma\sum_{s'\in\mathcal{S}} p(s'|s,a)V^{\pi^*}(s')\right\},\ \forall s\in\mathcal{S} \tag{3}$$

in which the first term represents the cost associated to the current state s and decision a; the second term represents the future expected discounted cost under the optimal policy.

It is also convenient to define the action-value function $Q^{\pi} : \mathcal{S}\times A \to \Re$ which is the expected infinite horizon discounted cost achieved by taking action a in state s and then following the policy π:

$$Q^{\pi}(s,a) = c(s,a) + \gamma\sum_{s'\in\mathcal{S}} p(s'|s,a)V^{\pi}(s'),\ \forall s\in\mathcal{S} \tag{4}$$

[1] Since we assume the action to be executed at the beginning of a time period, the number of instances during an interval is $k+a$

It is easy to realize that the value function V and the Q-function are closely related in that $V^{\pi}(s') = \min_{a \in A(s)} Q^{\pi}(s', a),\ \forall s \in \mathcal{S}$. More importantly, the knowledge of the Q function is fundamental in that it directly provides the associated policy: for a given function Q, the corresponding policy is $\pi(s) = \arg\min_{a \in A(s)} Q(s, a),\ \forall s \in \mathcal{S}$.

The optimal policy π^* can be obtained by solving the optimality equation (3) via standard techniques, e.g., value iteration, LP formulation, etc. However, computing the optimal policy requires a full knowledge of the system dynamics and parameters, e.g., the transition probabilities, that depend on the variable - and typically unknown - tuple rate, and the cost functions, e.g., the instance response time.

5 Reinforcement Learning

In this section we present three Reinforcement Learning-based approaches for learning the optimal auto-scaling strategy π^*. RL approaches are characterized by the basic principle of learning the optimal strategy π^* (and the optimal value functions V^* and Q^*) by direct interaction with the system.

Algorithm 1 illustrates the general RL scheme: the Q and or V functions are first initialized (setting all to 0 will often suffices) (line 1); then, by direct interaction with the system, the controller at each step t chooses an action a_t (based on current estimates of Q/V) (line 3), observes the incurred cost c_t and the next state s_{t+1} (line 4) and then updates the Q/V function based on what it just experienced during step t (line 5). The different solutions differ for the actual learning algorithm adopted and on the assumptions about the system.

In this paper we will consider the following three approaches, which differ on how to choose the action (line 3) and to how update the Q/V function (line 5). For its simplicity, we first consider the well-known *Q-Learning* algorithm. Q-Learning is a *model-free* learning algorithm which requires no knowledge of the system dynamics. We will then present two model-aware learning approaches. First, we consider the so called *post-decision state* (PDS), where we exploit the fact that part of the system dynamic, namely the impact of the auto-scaling decision on the number of instances, is known and let the learning only deal with the unknown dynamics. Then, we describe a *full backup model-based* approach, which basically estimates the unknown dynamic, that is, it estimates the arrival rate transition matrix and use these estimates to update the Q function.

5.1 Q-Learning

Q-Learning is an off-policy learning method that essentially estimates Q^* by its sample averages. Since it relies on estimates, at any decision step (line 3), Q-learning either: 1) *exploits* its knowledge about the system, that is, the current estimates Q_i, by selecting the *greedy* action $a_i = \arg\min_{a' \in A(s_i)} Q_i(s, a)$, *i.e.* the action minimizes the *estimated* future costs; or 2) *explores* by selecting a random

Algorithm 1 RL-based Operator Elastic Control Algorithm

```
1: Initialize $Q$ and/or $V$ functions
2: loop
3:     choose an action $a_i$ (based on current estimates of Q)
4:     observe the next state $s_{i+1}$ and the incurred cost $c_i$
5:     update $Q$ and/or $V$ functions based on experience
6: end loop
```

action to improve its knowledge of the system. Here we consider the simple ϵ-*greedy* action selection method which chooses a random action with probability ϵ or the greedy action with probability $1 - \epsilon$.

The algorithm performs simple one-step updates at the end of each time slot (line 5), as follows:

$$Q_{i+1}\left(s_i, a_i\right) \leftarrow (1-\alpha)Q_i\left(s_i, a_i\right) + \alpha\left[c_i + \gamma \min_{a' \in \mathcal{A}(s_{i+1})} Q_i(s_{i+1}, a')\right] \qquad (5)$$

where $\alpha \in [0, 1]$ is the *learning rate* parameter. Observe that (5) simply updates the old estimate Q_i with the just observed value (which comprises the just observed cost c_i plus the discounted cost of following the greedy policy onward, that is $\min_{a' \in \mathcal{A}} Q_i(s_{i+1}, a')$). It has been proven that, independently of the policy being followed and the initial values assigned to Q, the learned action-value function converges with probability 1 to Q^* [14], under the condition that every state-action pair continues to be sampled as $i \to \infty$.

5.2 Learning with Post-Decision States

Updating a single state-action pair per time slot and ignoring any known information about the system dynamics, Q-Learning may require a long time to converge to a near-optimal policy. Actually, as in many scenarios, the dynamics of the system we are considering are not completely unpredictable. In particular, the impact on the system state of the action performed is known and deterministic. We would like to provide the learner with this knowledge, so that it has only to learn about the unknown dynamics.

In order to integrate the partial knowledge of the system into a learning algorithm, we rely on the *post-decision state* (PDS) concept, exploiting the generalized definition given in [10]. A PDS (also known as *afterstate*) describes the state of the system *after* the known dynamics take place, but *before* the unknown dynamics take place. We denote a PDS as $\tilde{s} \in \mathcal{S}$. At any time i, we logically split the state transition $s_i \to s_{i+1}$ into two distinct transitions: $s_i \to \tilde{s}_i$ and $\tilde{s}_i \to s_{i+1}$. Given the current state $s_i = (k_i, \lambda_i)$ and the selected action a_i, we have:

$$\tilde{s}_i = (k_i + a_i, \lambda_i) = (k_{i+1}, \lambda_i) \qquad (6)$$

$$s_{i+1} = (k_{i+1}, \lambda_{i+1}) \qquad (7)$$

where $\tilde{s}_i$ fully reflects the consequences of the action a_i, and the next state s_{i+1} incorporates the unknown system dynamics (i.e., the input rate variation). The relationship between states, PDS, and actions is illustrated in Fig. 3.

In the same way we have logically split the state transitions, the cost associated to a state-action pair can be reformulated separating *known* and *unknown* components:

$$c(s, a) = c_k(s, a) + c_u(\tilde{s}) \tag{8}$$

where $c_k(s, a)$ accounts for the deterministic cost associated to the scaling action a in state s, and $c_u(\tilde{s})$ incorporates the unpredictable impact of the rate variation on the system performance when transitioning from a PDS.

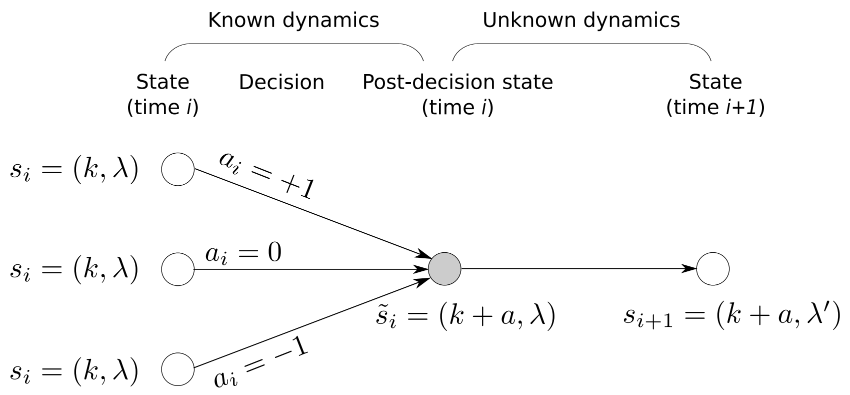


Figure 3: Relationships between current state, actions, PDS, and next state. (Adapted from the diagram reported in [10].)

We exploit the PDS concept to design a learning algorithm that aims at finding an optimal policy in less time than Q-Learning. To this end, we adapt the algorithm proposed in [10] to our problem. We integrate that solution into the generic Algorithm 1 by extending the update phase. In particular, the Q function has only to deal with the known system dynamics, since the unknown parts are hidden by the PDS, for which we introduce a PDS value function $\tilde{V}$ that is updated along with Q:

$$Q_i(s_{i+1}, a) \leftarrow c_k(s_{i+1}, a) + \tilde{V}_i(\tilde{s}_{i+1}) \quad \forall a \in \mathcal{A} \tag{9}$$

$$\tilde{V}_{i+1}(\tilde{s}_i) \leftarrow (1 - \alpha)\tilde{V}_i(\tilde{s}_i) + \alpha \left[c_{u,i} + \gamma \min_{a' \in A(s_{i+1})} Q_i(s_{i+1}, a') \right] \tag{10}$$

It is worth noting that, since the unknown system dynamics do not depend on the selected action, randomized exploration is not required any more, and a greedy policy can be followed during the learning phase.

5.3 Full Backup Model Based Reinforcement Learning

As third strategy we consider the *full backup model-based* reinforcement learning approach (see [12], Section 9). Here the idea is to directly use the MDP expression of the Q function (4) by replacing the unknown transition probabilities $p(s'|s,a)$, and the unknown cost function $c_u(s,a)$, $\forall s, s' \in \mathcal{S}$ and $a \in A(s)$ by their empirical estimates.

In order to estimate the transition probabilities $p(s'|s,a)$, from (1) it follows that it suffices to estimate the tuple arrival rate transition probabilities $P[\lambda_{i+1} = \lambda'|\lambda_i = \lambda]$. Hereafter, since λ takes value in a discrete set, we will write $P_{j,j'} = P[\lambda_{i+1} = j'\bar{\lambda}|\lambda_i = j\bar{\lambda}]$, $j, j' \in \{0, \ldots, L\}$ for short.

Let $n_{i,jj'}$ the number of times the arrival rate changes from state $j\bar{\lambda}$ to $j'\bar{\lambda}$, in the interval $\{1, \ldots, i\}$, $j, j' \in \{1, \ldots, L\}$ that is $n_{i,jj'} = \sum_{m=1,\ldots,i-1} \mathbb{1}_{\left\{\substack{\lambda_{i-1}=j\bar{\lambda}, \\ \lambda_i = j'\bar{\lambda}}\right\}}$. At time i the transition probabilities estimates are

$$\widehat{P_{j,j'}} = \frac{n_{i,jj'}}{\sum_{l=0}^{L} n_{i,jl}}$$

from which we derive the estimates $\hat{p}(s'|s,a)$ via (1).

For the estimates of the unknown cost, that in our case corresponds to the SLA violation cost, we use a simple exponential weighted average:

$$\hat{c}_{i,u}(s_i, a_i) \leftarrow (1-\alpha)\hat{c}_{i-1,u}(s_i, a_i) + \alpha c_{i,u} \tag{11}$$

where $c_i, u = w_{SLA}$ if a violation occur a time i (remember that we consider it normalized and weighted) and 0 otherwise. The Q estimates updating rules are then:

$$Q_i(s,a) \leftarrow \hat{c}_i(s,a) + \gamma \sum_{s' \in \mathcal{S}} \hat{p}(s'|s,a) \min_{a' \in \mathcal{A}(s')} Q_{i-1}(s', a') \quad \substack{\forall s \in \mathcal{S}, \\ a \in \mathcal{A}(s)} \tag{12}$$

The Q update step is summarized in Algorithm 2.

Algorithm 2 Full Backup Model Based Learning Update Algorithm

```
1: Update estimates $\widehat{P_{j,j'}}$ and $\hat{c}_{i,u}(s_i, a_i)$
2: for all $s \in \mathcal{S}$ do
3:     for all $a \in A(s)$ do
4:         $Q_i(s,a) \leftarrow \hat{c}_i(s,a) + \gamma \sum_{s' \in \mathcal{S}} \hat{p}(s'|s,a) \min_{a' \in A(s')} Q_{i-1}(s',a')$
5:     end for
6: end for
```

5.4 Complexity

Having presented the three RL-based algorithms, in this section we briefly discuss the complexity of the considered solutions, as summarized in Table 1. In both Q-Learning and the PDS-based solution, assuming that a lookup table is used for storing the Q values, both the action selection and the learning update at each time step have complexity $O(|\mathcal{A}|)$. Since we store an entry for each state-action pair, the space requirement is $O(|\mathcal{S}||\mathcal{A}|)$. In the model-based approach, the complexity of the update phase defined by Algorithm 2 is $O(|\mathcal{S}|^2|\mathcal{A}|^2)$ instead. Moreover, storing the input rate transition probability estimates increases the space complexity when the number of quantization levels L is large (i.e., $L > K_{max}|\mathcal{A}|$).

Table 1: Time and space complexity of the three considered algorithms.

Complexity	**Action selection**	**Update step**	**Space**
Q-Learning	$O(\|\mathcal{A}\|)$	$O(\|\mathcal{A}\|)$	$O(\|\mathcal{S}\|\|\mathcal{A}\|)$
PDS	$O(\|\mathcal{A}\|)$	$O(\|\mathcal{A}\|)$	$O(\|\mathcal{S}\|\|\mathcal{A}\|)$
Full Backup Model Based	$O(\|\mathcal{A}\|)$	$O(\|\mathcal{S}\|^2\|\mathcal{A}\|^2)$	$O(\|\mathcal{S}\|\|\mathcal{A}\| + L^2)$

6 Evaluation

We evaluate the three learning algorithms presented above simulating their behavior on a realistic application workload, using MATLAB®. We consider a dataset made available by Chris Whong[2] that contains information about the activity of the New York City taxis throughout one year. Each entry in the dataset corresponds to a taxi trip, reporting time and location for both the departure and the arrival. Figure 4 shows the number of events per minute throughout the first two weeks in the dataset. The taxi service utilization is clearly characterized by a daily pattern, thus requiring elastic capabilities for a system in charge of analyzing the generated data in real-time.

We consider one minute time slots, thus setting $\Delta_t = 1$ min. In order to produce a sequence of input rate values λ_i, we aggregate the events in the dataset over one minute windows. We assume each instance behaves as a M/D/1 queue with service rate $\mu = 3.3$ tuple/s. We compare the average cost achieved by each of the presented algorithms, setting different values for the quantum $\bar{\lambda}$ used to discretize the arrival rate: 20, 40, 60 tuples/min. Using larger values for discretizing the input rate, we reduce the number of system states, thus simplifying the learning process. However, coarse-grained input quantization makes the operator controller less precise, possibly leading to worse policies. The constant parameters used in the experiments are reported in Table 2.

Figure 5 shows the average cost achieved by the different algorithms in our simulations. In Fig. 5a we report the results for the experiment with $\bar{\lambda} = 20$ tuples/min. The full backup model based algorithm achieves the best performance,

[2] `http://chriswhong.com/open-data/foil_nyc_taxi/`

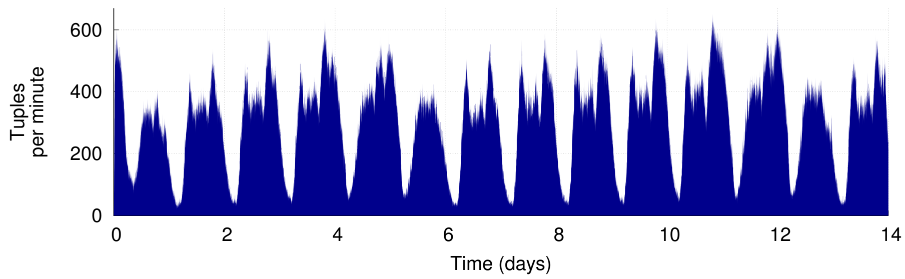


Figure 4: Events per minute in the first part of the dataset we used in our simulations. The data rate follows a clear daily pattern.

Table 2: Parameters used in the experiments.

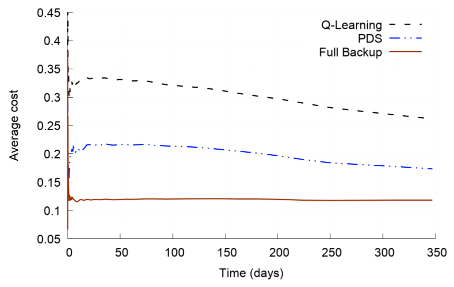
Discount factor γ	0.99	‖	Learning rate α	0.1
K_{max}	10	‖	$\varDelta_t$	1 min
w_{res},w_{rcf},w_{SLA}	$\frac{1}{3}$	‖	$\bar{\lambda}$	20,40,80 tuples/min
Service rate	3.33 tuple/s	‖	T_{SLA}	650 ms

with an average cost that converges quickly to less than 0.15. The PDS-based algorithm is slower to converge, and achieves a slightly higher average cost at the end of the simulation. The conventional Q-Learning is even slower to converge, and achieves the highest average cost at the end of the simulation.

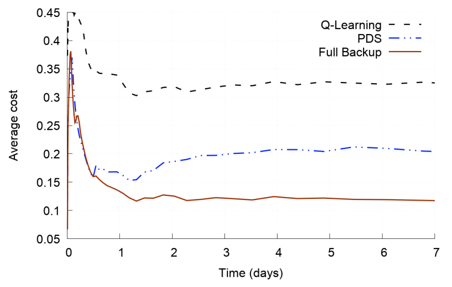
Table 3 reports the number of reconfigurations and violations observed during the simulation, along with the average number of instances allocated. These results reflect the average cost behavior described above. The full backup approach performs a dramatically smaller number of reconfigurations, incurring in less SLA violations, and running less instances on average. The PDS-based algorithm gets slightly worse results, with Q-Learning being the worst solution.

Figure 5b highlights the behavior of the algorithms at the beginning of the simulation (i.e., for the first simulated week). All the proposed solutions incur high costs at the beginning, when they still have to learn a good policy. After the first simulated day, their behavior gets much more stable, especially for the full backup model based algorithm.

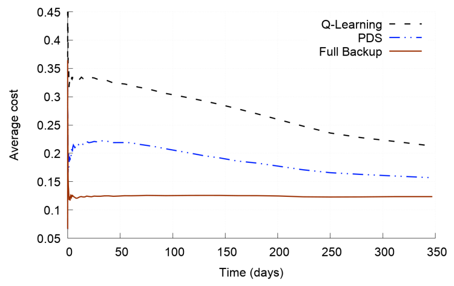
It is interesting to compare the results achieved with different values for the *quantum* $\bar{\lambda}$. When we increase it, as reported in Figs. 5c-5d, the average cost achieved by the full backup model based algorithm is slightly higher, as we could expect. The benefits on the convergence time are much more evident for the other two algorithms. With the reduced state space, they achieve an average cost at the end of the simulation that is closer to the model based solution.



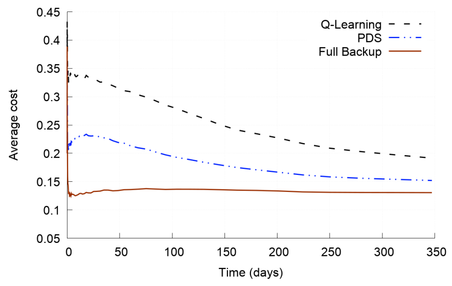
(a) $\bar{\lambda} = 20$ tuples/min



(b) First week, $\bar{\lambda} = 20$ tuples/min



(c) $\bar{\lambda} = 40$ tuples/min



(d) $\bar{\lambda} = 80$ tuples/min

Figure 5: Average cost achieved by the different algorithms with different values for the input rate quantization step $\bar{\lambda}$.

7 Conclusions

In this paper we have studied the auto-scaling problem for DSP applications. Focusing on a single operator, we have proposed and evaluated two model based Reinforcement Learning algorithms. Our numerical evaluation reveals that by exploiting our knowledge of (part of) the system under study, we are capable to achieve faster convergence and good system performance compared to the baseline Q-learning algorithm which is often adopted in the literature.

As future work, our goal is to extend these results to address the auto-scaling of a DSP application, which typically consists of many interconnected operators. To tackle the inherent complexity and the state space explosion of these systems, we plan to investigate the use of more refined Reinforcement Learning techniques, e.g., Function Approximation and Bayesian Reinforcement Learning [12].

References

1. Cardellini, V., Lo Presti, F., Russo Russo, G., Nardelli, M.: Optimal operator deployment and replication for elastic distributed data stream processing. Concurr. Comput. (2017), to appear

Table 3: Number of reconfigurations, SLA violations, and allocated instances (on average) in the experiments.

Algorithm	**Reconfigurations**	**Violations**	**Avg. instances**
Q-learning	115296	47942	4.58
PDS	13674	36337	4.20
Full Backup Model Based	2772	1430	3.46

2. De Matteis, T., Mencagli, G.: Elastic scaling for distributed latency-sensitive data stream operators. In: Proc. PDP '17. pp. 61–68 (2017)
3. Fernandez, R.C., Migliavacca, M., Kalyvianaki, E., Pietzuch, P.: Integrating scale out and fault tolerance in stream processing using operator state management. In: Proc. ACM SIGMOD '13 (2013)
4. Gedik, B., Schneider, S., Hirzel, M., Wu, K.L.: Elastic scaling for data stream processing. IEEE Trans. Parallel Distrib. Syst. 25(6), 1447–1463 (2014)
5. Heinze, T., Pappalardo, V., Jerzak, Z., Fetzer, C.: Auto-scaling techniques for elastic data stream processing. In: Proc. IEEE ICDEW '14. pp. 296–302 (2014)
6. Heinze, T., Aniello, L., Querzoni, L., Jerzak, Z.: Cloud-based data stream processing. In: Proc. ACM DEBS '14. pp. 238–245 (2014)
7. Hirzel, M., Soulé, R., Schneider, S., Gedik, B., Grimm, R.: A catalog of stream processing optimizations. ACM Comput. Surv. 46(4) (Mar 2014)
8. Lohrmann, B., Janacik, P., Kao, O.: Elastic stream processing with latency guarantees. In: Proc. IEEE ICDCS '15. pp. 399–410 (2015)
9. Lorido-Botran, T., Miguel-Alonso, J., Lozano, J.A.: A review of auto-scaling techniques for elastic applications in cloud environments. J. Grid Comput. 12(4) (2014)
10. Mastronarde, N., van der Schaar, M.: Fast reinforcement learning for energy-efficient wireless communication. IEEE Trans. Signal Process. 59(12) (2011)
11. Puterman, M.L.: Markov Decision Processes: Discrete Stochastic Dynamic Programming. John Wiley & Sons (2014)
12. Sutton, R.S., Barto, A.G.: Reinforcement learning: An introduction. MIT Press, Cambridge (1998)
13. Tesauro, G., Jong, N.K., Das, R., Bennani, M.N.: On the use of hybrid reinforcement learning for autonomic resource allocation. Cluster Comput. 10(3) (2007)
14. Watkins, C.J., Dayan, P.: Q-learning. Machine learning 8(3-4), 279–292 (1992)
15. Yoon, K.P., Hwang, C.L.: Multiple Attribute Decision Making: An Introduction, vol. 104. Sage Publications (1995)