Cloud Management with Reinforcement Learning

QIZHEN WENG, Department of Computer Science and Engineering, the Hong Kong University of Science and Technology, supervised by Prof. Wei WANG

Cloud Computing, hiding the complexity of managing clusters from the developers, poses the challenges to cloud service providers to improve the quality of services while maintaining low operating cost. Through the years, many researchers have been dealing with these issues from different aspects, including network optimization, resource management, workload scheduling, etc. But the difficulty also arises with the growing scale of clusters and the heterogeneity of applications.

Inspired by the recent advances in artificial intelligence problems, the idea of applying reinforcement learning techniques to assisting the management of cloud systems is increasingly attractive. Instead of manually exploring vast configuration space in response to various workloads, cloud service providers can deploy learning agents to collect the data, interact with complex environments automatically, and improve the system's efficiency to the human expert-level or even beyond.

This survey serves a review of selected cloud management topics addressed by reinforcement learning approaches. We first give a brief introduction of fundamental concepts and advanced models of reinforcement learning. Then we present several series of applications in particular fields, i.e., network optimization, virtual machine configuration, dynamic power management, and cluster scheduling. After showing how researchers formulate system optimizations in different settings as reinforcement learning problems and design learning agents tackling real-world issues, we conclude the survey by discussing the common characteristics and challenges of these applications, motivating further research and industrial-oriented solutions.

Additional Key Words and Phrases: cloud computing, reinforcement learning, resource management, cluster scheduling

Author's address: Qizhen WENG, qwengaa@cse.ust.hk, Department of Computer Science and Engineering, the Hong Kong University of Science and Technology, supervised by Prof. Wei WANG.

Contents

1 INTRODUCTION

Cloud computing [11] refers to both applications delivered as services over the Internet, termed Software-as-a-Service *(SaaS)*, and the infrastructure hardware and systems software in datacenters, called *Cloud*, that provide these services. From the users' perspective, cloud computing spares them from the trouble of provisioning, operating, maintaining, and updating the cluster. To the service providers, who made Cloud available to the general public in a pay-as-you-go manner, cloud computing leverages the very large economies of scale to reduce operating cost and multiplexes statistical to increase utilization, thus providing good profit. Cloud providers, e.g., Amazon Web Services (AWS), Google Cloud Platform, Microsoft Azure, Alibaba Cloud, etc., keep expanding their business and competing for this billion dollar market.

The key to providing profitable cloud service lies in the efficient management of extremely large-scale, commodity-computer datacenters at low operating cost. However, with the growing scale of clusters, the heterogeneity of machines increases in terms of system software versions, hardware configurations, geographical locations, etc., the difficulties of management also arises; service providers should tackle technical challenges in different fields, such as network optimization, resource management, workload scheduling, etc. to maintain high performance and low costs. Also, to satisfy a variety of users' requirements, e.g., service availability, data confidentiality, scalability, and even further simplicity (e.g., serverless computing), the heterogeneity of services also increases; service providers should further care about different users' Quality of Service (QoS) requirements, e.g., auto-scaling on demand, fast data transfer, quick launching of instances, while still limiting the complexity of maintenance and human-labor required in the large-scale system.

Inspired by the recent advance in artificial intelligence [24, 25, 34, 35] and profound theoretical foundation of machine learning, the idea of applying reinforcement learning (RL) techniques to assisting the management of cloud systems is increasingly attractive. Characterized with trial-and-error search and delayed reward [36], RL offers a more general and flexible framework for a remarkable variety of problems; it enables an agent to interact with complex environments autonomously, learn certain strategies to manage the cloud, and improve the system's efficiency to human expert-level or even beyond.

While RL has achieved many success in a variety of domains with some powerful learning methods, the general scheme provided by RL still requires different domain experts to tailor the problem formulation, state representation, and complex reward structures in the first step in order to make a learning agent work under different scenarios achieving designed purposes; the task becomes more demanding when the scope of the problem, usually comparable with the scale of cluster and applications, grows over thousands of dimensions with complex dynamics, termed *curse of dimensionality* [26]. Also, applying deep learning techniques to RL as a modern approach, though dramatically improves its power in feature representation, also confronts some unique problems like instability, divergence, and data-hungry training paradigm. Addressing these challenges requires the confluence of both machine learning and system expertise.

Nevertheless, every challenge is an opportunity. Cloud management with reinforcement learning approach is an increasingly exciting area of focus for both machine learning and system community; the resulting wave of technical advances gives us a chance to re-examine the traditional designs, heuristics, or hand-tuned parameters used in the cloud systems, and probably replace them with learning agents of better performance, adaptability, profitability, and so forth.

This survey serves a review of selected cloud management topics addressed by reinforcement learning approaches. We first give a brief introduction of fundamental concepts and advanced models of reinforcement learning. Then we present several series of applications in particular fields, including *network optimization*, *virtual machine configuration*, *dynamic power management*, and

cluster scheduling. After showing how researchers formulate system optimization problems of different settings into reinforcement learning framework and design learning agents solving real-world issues, we conclude the survey by discussing the common characteristics and challenges of these applications, motivating further research and industrial-oriented solutions. Hopefully, this work could help readers better understand the recent progress and select the most suitable techniques for their own applications.

2 REINFORCEMENT LEARNING

In this section, we first introduce fundamental knowledge of Markov Decision Process (MDP) and Reinforcement Learning (RL), then present the recent progress in Deep Learning (DL) techniques and how their combination, i.e., Deep Reinforcement Learning (DRL), leads to advanced model design and achieves great success.

2.1 Markov Decision Processes

Markov Decision Process (MDP), as a discrete time stochastic control process, formally describes an environment for sequential decision making with uncertain outcomes and introduces classical notations for dynamic programming and reinforcement learning.

Considering a decision maker or an *agent* interacting with an *environment* that provides feedback, including new states and numerical rewards at each of a sequence of discrete time steps, $t = 0, 1, 2, \cdots$. More specifically, an MDP is defined by a 4-tuple $(\mathcal{S}, \mathcal{A}, p, \mathcal{R})$ where $\mathcal{S}$ is a set of all possible *states* $S_t \in \mathcal{S}$ received by the agent; on that base, the agent selects an *action*, $A_t \in \mathcal{A}$, which changes the environment and receives a *reward* $R_t \in \mathcal{R}$ and a new state representation S_{t+1}. The function $p : \mathcal{S} \times \mathcal{R} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ defines the *dynamics* of these outcomes given certain preceding state and action:

$$p(s', r|s, a) \doteq \Pr\{S_t = s', R_t = r|S_{t-1} = s, A_{t-1} = a\} \tag{1}$$

The dynamic function in Eq.1 is suggested by Richard S. Sutton's textbook [36], simplifies the formulas, and can derive other conventional functions like three-argument *state-transition probabilities* $p : \mathcal{S} \times \mathcal{S} \times \mathcal{A} \to [0, 1]$ and expected rewards for state-action pairs as a two-argument function $r : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ as follows,

$$\begin{aligned} p(s'|s, a) &\doteq \Pr\{S_t = s'|S_{t-1} = s, A_{t-1} = a\} = \sum_{r \in \mathcal{R}} p(s', r|s, a) \\ r(s, a) &\doteq \mathbb{E}[R_t|S_{t-1} = s, A_{t-1} = a] = \sum_{r \in \mathcal{R}} r \sum_{s' \in \mathcal{S}} p(s', r|s, a) \end{aligned} \tag{2}$$

The *Markov property* assumed by the Eq.1 can be described as the state-transition probabilities and expected rewards depend on the past *only through* the current state of the system and the action selected by the agent in that state, which is also called *Markovian* or memoryless [27].

For an MDP, it is also assumed that all states, i.e., $S_t \in \mathcal{S}$ are *observable*. Otherwise, it turns into a generalization of MDP called *partially observable Markov decision process* (POMDP) that maintains an additional probability distribution over the set of possible states based on the a set of observations O, conditional observation probability, and the underlying MDP.

The purpose or goal of the agent is formalized as maximizing the cumulative numerical reward, denoted as the *return* G_t, fed by the environment after each step t, i.e., $G_t \doteq R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T$, if there is a final time step T. For some applications whose agent-environment interaction can break naturally into subsequences or *episodes*, they can always ends up in a special *terminal*

state where a reset is required to repeated the interaction, such as "Game Over" in video gaming. This kind of tasks are called *episodic tasks*.

More generally, in order to extend the formulation of return to another kind of tasks, named *continuing tasks*, that go on continually without limit, e.g., an on-going process-control task, a concept of *discounting* is introduced by a factor $\gamma \in [0, 1]$ named *discount rate*. It decreases the infinite possible reward in the future and make the return finite even if being a sum of infinite number of terms.

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

Most Markov reward and decision processes are discounted because it not only provides mathematically convenience, but also suits the nature of animal/human that favors immediate rewards than delayed ones; tuning the γ close to 0 or 1 can lead to a "myopic" or "far-sighted" evaluation respectively.

A (stochastic) *policy* is a distribution over actions given an observed state of the environment, which specifies the probability (or decision rule for deterministic policy) of selecting certain possible action to be taken at each step and can fully define the behavior of an agent.

$$\pi(a|s) \doteq \Pr[A_t = a | S_t = s] \tag{4}$$

In the terms of expected return and policy, *value functions* measure *how good* it is for the agent to be in a given state, or to perform a given action in a given state.

More specifically, *state-value function* $v_\pi(s)$ of an MDP is the expected return starting from state s, and then following policy π:

$$v_\pi(s) \doteq \mathbb{E}_\pi[G_t | S_t = s] = \mathbb{E}_\pi\Big[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \Big| S_t = s \Big] \tag{5}$$

The *action-value function* $q_\pi(s, a)$ of an MDP is the expected return starting from state s, taking action a, and then following policy π:

$$q_\pi(s, a) \doteq \mathbb{E}_\pi[G_t | S_t = t, A_t = a] = \mathbb{E}_\pi\Big[\sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \Big| S_t = s, A_t = a \Big] \tag{6}$$

Among all the state-value functions $v_\pi(s)$, the *optimal state-value function* $v_*(s)$ is defined as

$$v_*(s) \doteq \max_\pi v_\pi(s), \text{ for all } s \in \mathcal{S}. \tag{7}$$

Similarly, *optimal action-value function*, denoted $q_*(s, a)$, is defined as,

$$q_*(s, a) \doteq \max_\pi q_\pi(s, a) = \mathbb{E}_\pi[R_{t+1} + \gamma v_*(S_{t+1}) | S_t = t, A_t = a], \text{ for all } s \in \mathcal{S} \text{ and } a \in \mathcal{A}. \tag{8}$$

For any MDP, there always exists an *optimal policy* π_* whose state-value function is better than or equal to that of all other policies for all $s \in \mathcal{S}$. All optimal policies achieve both the optimal state-value function and optimal action-value function. Thus the optimal policy can be found by maximizing over $q_*(s, a)$,

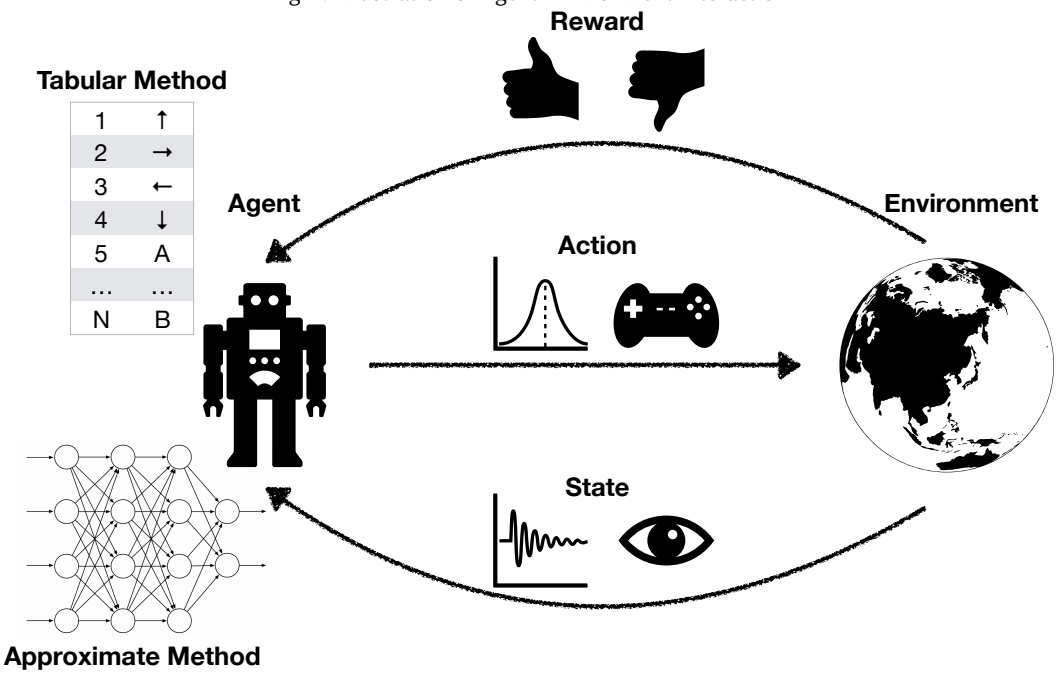
$$\pi_*(a|s) = \begin{cases} 1 & \arg\max_{a \in \mathcal{A}} q_*(s, a) \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

2.2 Reinforcement Learning Basics

Reinforcement learning is to learn the optimal policy, i.e., the mapping from situations to actions, so as to maximize the accumulative reward. Given the Markov Decision Process formulation above, this goal can be achieved by solving the MDPs (named *evaluation* or *prediction* problem) and retrieving the optimal policy π_* (named *control* problem).

As shown in Fig. 1, The decision maker or the learning *agent* continually interacts with the *environment*, which usually cannot be accurately modeled in RL, by observing current states, selecting actions, receiving corresponding rewards, and observing the next state. The agent makes its decision based on its stochastic *policy*, which is in fact a distribution over actions given an observed state of the environment.

Fig. 1. Illustration of Agent-Environment interaction



The simplest form of reinforcement learning, i.e., *tabular methods*, uses *tables* to represent value of states and actions as policies, which are assumed to be small enough to retrieve efficiently. The large-space situations that violates this assumption will be handled by *approximation methods*, for example, with deep neural networks (to be introduced in Sec. 2.3).

In this section, we will first elaborate tabular methods in reinforcement learning via three fundamental classes of methods for solving Markov decision problems: *dynamic programming* algorithms, *Monte Carlo* methods, and *temporal-difference* learning. After that we will introduce policy-based method and actor-critic method which typically work with a parameterized family of policies.

2.2.1 Dynamic Programming.

Dynamic Programming, in the term of Reinforcement Learning, refers to a collection of algorithms that computes the value functions to search for optimal policies in an organized approach, given a perfect model of the environment as a Markov decision process.

Following the discussion in Sec.2.1, we consider the state-value function being calculated with the current state s and its possible successor states s',

$$\begin{aligned} v_\pi(s) &\doteq \mathbb{E}_\pi[G_t|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)\Big[r + \gamma \mathbb{E}_\pi[G_{t+1}|S_{t+1} = s']\Big] \\ &= \sum_a \pi(a|s) \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \end{aligned} \tag{10}$$

Eq.10 is the *Bellman (expectation) equation* for v_π which sums over all values of a, s', and r starting from the current state, weighted by its probability, $\pi(a|s)p(s', r|s, a)$, to get an expected value of the state-value function $v_\pi(s)$.

Similarly, the Bellman (expectation) equation for action-value function $q_\pi(s, a)$ can be derived as,

$$\begin{aligned} q_\pi(s) &\doteq \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \\ &= \sum_{s',r} p(s', r|s, a)[r + \gamma v_\pi(s')] \end{aligned} \tag{11}$$

To enhance the fact that the value of a state under an optimal policy π_*, must equal the expected return for the best action from that state, we calculate the optimal state-value function $v_*(s)$ under the form of *Bellman optimality equation*,

$$\begin{aligned} v_*(s) &= \max_\pi \mathbb{E}_\pi[R_{t+1} + \gamma v_*(S_{t+1})|S_t = s] \\ &= \max_a \sum_{s',r} p(s', r|s, a)[r + \gamma v_*(s')] \end{aligned} \tag{12}$$

Similarly, the Bellman optimality equation for action-value function $q_*(s, a)$ is,

$$\begin{aligned} q_*(s, a) &= \mathbb{E}[R_{t+1} + \gamma \max_{a'} q_*(S_{t+1}, a')|S_t = s, A_t = a] \\ &= \sum_{s',r} p(s', r|s, a)[r + \gamma \max_{a'} q_*(s', a')] \end{aligned} \tag{13}$$

With the Bellman equations Eq.10,11,12,13, we can start from an arbitrary policy π, evaluate its state-value function v_π (*policy evaluation*), make a new policy π' based on the original one π by being greedy with respect to the original value function v_π and achieving greater value $v_{\pi'} \geq v_\pi$ (*policy improvement*), and keep doing it iteratively until we reach the optimal policy π_* (*policy iteration*):

$$\pi_0 \xrightarrow{E} v_{\pi_0} \xrightarrow{I} \pi_1 \xrightarrow{E} v_{\pi_1} \xrightarrow{I} \pi_2 \xrightarrow{E} \cdots \xrightarrow{I} \pi_* \xrightarrow{E} v_{\pi_*} \tag{14}$$

where $\xrightarrow{E}$ denotes a policy evaluation, $\xrightarrow{I}$ denotes a policy improvement, and the whole procedure is called *generalized policy iteration (GPI)*, under which dynamic programming often enjoys fast convergence where the model of environment is perfect and the spaces of state and action are small enough.

2.2.2 *Monte Carlo methods.*

Monte Carlo methods differ from dynamic programming mainly in only requiring *experience*, i.e., *sample* sequence of states, actions, and rewards, but not the model of environment that gives complete probability distribution.

To solve the MDP problem, Monte Carlo methods follows the overall scheme of generalized policy iteration (Eq.14) mentioned in dynamic programming; the difference lies in that Monte Carlo methods do not compute the value of states by a model, but

their values through averaging sample returns.

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t - V(S_t)] \tag{15}$$

Eq.15 is called the *incremental Monte-Carlo updates* where G_t (Eq.3) is the actual return given the sampled sequence, i.e., episodes of experience under policy π: $S_1, A_1, R_2, S_2, A_2, R_3, \cdots, S_k \sim \pi$, instead of the expected return calculated by the model (i.e., the dynamic function) in dynamic programming, for following time t. Apart from the discounted factor γ used in the return function, α is also a constant parameter that controls the step-size of updates.

The limitation of Monte Carlo methods is that it only support episodic tasks, i.e., all episodes should eventually terminate given whatever actions, and thus it can only be improved in an episode-by-episode sense but not in a step-by-step or online sense. Also, it does not bootstrap, i.e., update value estimates on the basis of other value estimates, so that it must wait until the end of the episode to determine the increment of the value functions.

On the other hand, Monte Carlo methods enjoy good convergence properties (even with function approximation to be discussed in Sec.2.3), robust to initial value, and are effective in non-Markovian environments since they do not exploit Markov property.

2.2.3 *Temporal-Difference Learning.*

Temporal-difference (TD) learning is sometimes considered as the most central and novel idea in reinforcement learning [36], which combines the idea of Monte Carlo methods and dynamic programming (DP) Like Monte Carlo methods, TD methods can learn from sampled experience rather than precise model of the environment. Like DP, TD methods support bootstrapping, i.e., updating value estimates on the basis of other value estimates.

Compared to the incremental Monte-Carlo policy evaluation updates in Eq.15, the simplest form of *TD policy evaluation*, called *TD(0)*, updates the $V(S_t)$ towards the estimated return, which is also called the *TD target*, $R_{t+1} + \gamma V(S_{t+1})$, instead of the actual return G_t.

$$V(S_t) \leftarrow V(S_t) + \alpha[R_{t+1} + \gamma V(S_{t+1}) - V(S_t)] \tag{16}$$

TD error, arising in various forms throughout reinforcement learning, is therefore defined as the difference between TD target and the state-value function,

$$\delta_t \doteq R_{t+1} + \gamma V(S_{t+1}) - V(S_t) \tag{17}$$

Considering the actual return of Monte Carlo and the one-step estimate return of TD(0) as two extremes in the spectrum, we can let TD target look n steps into the future by defining a *n-step return* $G_t^{(n)}$ as follows,

$$G_t^{(n)} \doteq R_{t+1} + \gamma R_{t+2} + \cdots + \gamma^{(n-1)} R_{t+n} + \gamma^n V(S_{t+n}) \tag{18}$$

when $n = 1$ it is the one-step TD target while $n = \infty$ it becomes the actual return used in Monte Carlo method. By selecting the value of n, we can shift from these two special cases smoothly as

needed to meet the demands of certain tasks. That is exactly the key idea of *n-step TD* methods, whose policy evaluation update rule can be expressed as follows,

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t^{(n)} - V(S_t)] \tag{19}$$

Sometimes the idea is also presented as the *TD(λ)* method by introducing a $\lambda \in [0, 1]$ instead of the discrete value n, and revises the update rules and the return function (named *λ-return*) as follows,

$$V(S_t) \leftarrow V(S_t) + \alpha[G_t^{\lambda} - V(S_t)] \tag{20}$$

$$G_t^{\lambda} \doteq (1-\lambda) \sum_{n=1}^{\infty} \lambda^{n-1} G_t^{(n)} \tag{21}$$

Based on the prediction methods discussed, we can seek the optimal policies and control the agent accordingly to get the maximum return in certain environment, which is called the *control problem*. However, one important challenge arising in the way, specific to reinforcement learning, is widely acknowledged as the trade-off between *exploration* and *exploitation*.

To obtain plenty of reward, a learning agent is preferred to take actions that give maximum expected return from its past experience, which appears *greedy* and *exploits* what it had learned. But on the other hand, some untried actions may lead to higher reward which are waited to be *explored* for better action selections in the future.

A common approach to surf this trade-off is applying *ϵ-greedy exploration* (Eq.22) where with probability of ϵ the agent will chooses an action uniformly at random from all m possible actions (i.e., explore), and with probability of $1 - \epsilon$ the agent will choose the best action learned so far (i.e., exploit).

$$\pi(a|s) = \begin{cases} \epsilon/m + 1 - \epsilon & \text{if } a = \arg\max_{a \in \mathcal{A}} Q_(s, a) \\ \epsilon/m & \text{otherwise} \end{cases} \tag{22}$$

On-policy vs. Off-policy. Another approach to explore possible optimal actions is to use two policies; one is learned from experience and becomes the optimal policy (called *target policy*) and another is tuned to be more exploratory and is used to generate behaviors (called *behavior policy*). In this case the learning is from data "off" the target policy and thus termed *off-policy learning*. The behavior policy that the agent is following could be the observation from humans, other agents, or previous policies to re-use past experience. Contrast to off-policy learning, *on-policy* methods, which learn about policy π directly from experience sampled from π ("learn on the job"), are usually more simpler but are less powerful and general than the off-policy methods. Regarding the sample efficiency, i.e., number of samples required to generated to get a good policy, on-policy methods are required to generate new samples for each the policy is changed while off-policy methods are free from such constraints and thus considered more sample-efficient.

SARSA: On-policy TD Control. Now we consider a basic approach towards control problem; the first step is to learn an action-value function rather than a state-value function, i.e., estimate $q_\pi(s, a)$ given policy π for all states s and actions a, which can guide us to select the optimal action a under current state s. Similar to the state-value function iteration under TD(0) settings (Eq.16), the convergence of action-value function can also be guaranteed if its value updated as follows,

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \tag{23}$$

This update is executed after every transition from a non-terminal state S_t and uses every element of each sequential quintuple of events, $(S_t, A_t, R_{t+1}, S_{t+1}, A_{t+1})$, which gives the name *SARSA* [30] for the algorithm.

Q-learning: Off-policy TD Control. Consider using an off-policy method, i.e., using two policies, to learn the action-values $Q(s,a)$ with sufficient exploration, we can choose the next action with ϵ-greedy of $Q(s,a)$ while update its value towards the optimal $Q(s,a)$ value; in the terms of off-policy learning, the target policy π of this method is greedy with respect to $Q(s,a)$ while the behavior policy, denoted as μ, is ϵ-greedy with respect to $Q(s,a)$. This method is called *Q-learning*, regarded as a critical breakthrough in reinforcement learning made by Watkins [39], and is defined as follows,

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{24}$$

2.2.4 Policy-based Methods.

The aforementioned value-based methods usually represent value function with *tables*, which assume the space is discrete and small enough to retrieve efficiently. For larger or continuous spaces, function approximation is usually employed to match real values or select actions.

Policy-based methods, also known as actor-only methods, or policy gradient methods, learn the policy directly with a parameterized function $\pi(a|s;\boldsymbol{\theta})$ respect to parameter $\boldsymbol{\theta}$ and do not use any form of a stored value function.

Its reward function is defined as:

$$J(\boldsymbol{\theta}) = \sum_{s\in\mathcal{S}} d^{\pi}(s) v^{\pi}(s) = \sum_{s\in\mathcal{S}} d^{\pi}(s) \sum_{a\in\mathcal{A}} Q^{\pi}(s,a)\pi(a|s;\boldsymbol{\theta}) \tag{25}$$

where $d^{\pi}(s)$ is the stationary distribution of Markov chain for $\pi(a|s;\boldsymbol{\theta})$ (on-policy state distribution under π). Assuming the parameterization is differentiable with respect to $\boldsymbol{\theta}$, the *policy gradient theorem* for episodic case establishes the gradient of the reward function as:

$$\nabla J(\boldsymbol{\theta}) \propto \sum_{s\in\mathcal{S}} d^{\pi}(s) \sum_{a\in\mathcal{A}} Q^{\pi}(s,a)\nabla\pi(a|s;\boldsymbol{\theta}) \tag{26}$$

By using standard optimization methods, a local optimal solution of the reward J can be retrieved by the following iterative updates:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha_t \nabla_{\boldsymbol{\theta}} J(\boldsymbol{\theta}_t) \tag{27}$$

REINFORCE: Monte Carlo Policy Gradient. As the most commonly used policy gradient method, REINFORCE [41] relies on an estimated return by Monte-Carlo methods (Eq.15) using episode samples to update the policy parameter $\boldsymbol{\theta}$, since the expectation of the sample gradient is equal to the actual gradient, following the derivation of Eq.26:

$$\begin{aligned}
\nabla J(\boldsymbol{\theta}) &\propto \sum_s d^{\pi}(s) \sum_a Q^{\pi}(s,a)\nabla\pi(a|s;\boldsymbol{\theta}) \\
&= \sum_s d^{\pi}(s) \sum_a Q^{\pi}(s,a)\pi(a|s;\boldsymbol{\theta}) \frac{\nabla_{\theta}\pi(a|s;\boldsymbol{\theta})}{\pi(a|s;\boldsymbol{\theta})} \\
&= \mathbb{E}_{\pi}\left[Q^{\pi}(s,a)\nabla_{\theta}\ln\pi(a|s;\boldsymbol{\theta})\right]
\end{aligned} \tag{28}$$

where $\mathbb{E}_{\pi}$ refers to $\mathbb{E}_{s\sim d_{\pi}, a\sim\pi_{\theta}}$ when both state and action distributions follow the policy $\pi(a|s;\boldsymbol{\theta})$ (i.e., on policy).

Thus the rule of updating policy parameters $\boldsymbol{\theta}$ for each time stamp, after estimating the return G_t is straightforward given $Q^{\pi}(S_t, A_t) = \mathbb{E}[G_t|S_t, A_t]$,

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha_t \gamma^t G_t \nabla_{\boldsymbol{\theta}} \ln \pi(A_t|S_t; \boldsymbol{\theta}) \tag{29}$$

2.2.5 Actor-Critic Methods.

Combining the advantages of value-based method (critic-only) and policy-based method (actor-only), *Actor-Critic* method can use value function to assist policy update by reducing gradient variance in vanilla policy gradients.

Actor-Critic learning: On-policy TD Control. Actor-critic methods consist of two models, actor, and critic, which may optionally share parameters.

The critic updates the value function parameters $\boldsymbol{w}$ (either action-value $Q(a|s; \boldsymbol{w})$ or state-value $V(s|\boldsymbol{w})$) to supply the actor with low-variance policy gradient estimates.

The actor updates the policy parameters $\boldsymbol{\theta}$ for $\pi(a|s; \boldsymbol{\theta})$ in policy improvement direction suggested by actor. Similar to policy gradient method, the policy is not inferred from the value function but directly generated by the actor.

A simple action-value actor-critic algorithm is shown in Alg.1 [40]

Result: Actor-Critic agents get parameters updated
Initialize $s, \theta, \boldsymbol{w}$ at random; sample $a \sim \pi(a|s; \boldsymbol{\theta})$;
for $t = 1 \ldots T$ **do**
  Sample reward $r_t \sim R(s, a)$ and next state $s' \sim P(s'|s, a)$;
  Sample the next action $a' \sim \pi(a'|s'; \boldsymbol{\theta})$;
  Update the policy parameters: $\boldsymbol{\theta} \leftarrow \boldsymbol{\theta} + \alpha_{\boldsymbol{\theta}} Q^{\boldsymbol{w}}(s, a) \nabla_{\boldsymbol{\theta}} \ln \pi(a|s; \boldsymbol{\theta})$;
  Compute the correction (TD error) for action-value at time t:
  $\delta_t = r_t + \gamma Q^{\boldsymbol{w}}(s', a') - Q^{\boldsymbol{w}}(s, a)$;
  Use TD error δ_t to update the parameters of action-value function:
  $\boldsymbol{w} \leftarrow \boldsymbol{w} + \alpha_{\boldsymbol{w}} \delta_t \nabla_{\boldsymbol{w}} Q^{\boldsymbol{w}}(s, a)$;
  Update $a \leftarrow a'$ and $s \leftarrow s'$
end

Algorithm 1: Action-value Actor-Critic Algorithm

2.3 Deep Reinforcement Learning

Deep reinforcement learning is the study of reinforcement learning using neural networks as function approximators. [31]

2.3.1 Basic of Deep Neural Networks.

Neural networks (NN), or multi-layer perceptrons (MLPs) are the quintessential machine learning models. Early in 1980s, G. Cybenko has demonstrated the power of continuous artificial neural networks, even with one single hidden layer, can be used to approximate arbitrary decision regions [8]. With the recent development of deep learning techniques, modern machine learning practitioners can deploy and train neural networks with tens or even hundreds of hidden layers (termed "deep neural networks") by the advent of fast graphics processing units (GPUs), stochastic gradient descent (SGD) optimization method, and regularization techniques, where arise the name "deep learning" [12, 17].

The basic components of a neural network layer consists of linear transformation, i.e., a weighted sum, of all the inputs, and a non-linear activation functions (e.g., sigmoid, tanh, ReLU) in concatenation to get the output. With backpropagation, we can compute the error derivative with respect to the output of each unit, and update the weight on each input link to adjust the layer's output.

2.3.2 Advanced DRL Algorithms and Applications.

As discussed in Sec.2.2.4, reinforcement learning with tabular methods assumes the space is discrete and small enough to retrieve efficiently. For larger or continuous spaces, function approximation is usually employed to match real values or select actions. In 1990s, D. Bertsekas and J. Tsitsiklis systematically presented the idea of using neural network as function approximators to overcome the "curse of dimensionality" in the practical application of dynamic programming and stochastic control to complex problems [2], named as *Neuro-Dynamic Programming*, which lay the foundation of modern deep reinforcement learning.

Deep Q-Networks. It is natural to extend the classic value-based approaches, e.g., Q-learning, with deep neural network as a function approximator. However, in practice, naive Q-learning training is usually unstable or even diverges with neural networks, which require much efforts paid to address instability issues. Mnih et al. propose Deep Q-network (DQN) as a modern variant of Q-learning, representing action-value (Q) function with deep neural networks [25]. By applying two techniques, namely *experience replay* and *periodically update*, the deep Q-network agent, receiving only the pixels and game scores as inputs, was able to outperform all previous algorithms in playing classic Atari 2600 games, approaching a level comparable to human game players.

Asynchronous Advantage Actor-Critic (A3C). proposed by Mnih et al [23] is a classic policy gradient method with special purpose on parallel training. The parallelism in multiple agent training enabled by A3C, not only accelerate the training speed, but also improve the stability a lot by allowing multiple actors learn in parallel and get synced with global parameters from time to time.

Trust Region Policy Optimization (TRPO). Schulman et al. took an analytic approach towards training stability issues in policy gradient method; the key idea is to avoid parameter updates that change the policy too much within one single step [32]. They formulate the updating process as an optimization problem that enforces a KL divergence constraint on the policy update step size.

Proximal Policy Optimization (PPO). To further simplify TRPO while maintaining similar performance, Schulman et al. propose PPO [33] that places a soft constraint of policy different between steps. Evaluation based on simulated robotic locomotion and Atari game playing show that PPO outperforms other online policy gradfient methods and strike good balance between simplicity and effectiveness.

Monte Carlo Tree Search (MCTS). AlphaGo [34] and AlphaGo Zero [35], developed by team at DeepMind, combining the deep convolution neural networks and Monte Carlo Tree Search, have achieved astonishing performance of professional human Go players. Given the intractable search space of game Go, directly approximate policy or value function is still infeasible; the introduction of MCTS method is mind-blowing and inspiring to researcher dealing with extremely large space.

3 CLOUD MANAGEMENT WITH REINFORCEMENT LEARNING

Cloud computing vendors provide virtualized resources, such as computation, memory, storage, networking, to the users, hide the implementation of how they are multiplexed and shared, and ensures their scalability and high availability. [11] Since the key to providing profitable cloud service lies in the efficient management of extremely large-scale, commodity-computer datacenters at low operating cost, through the years, many researchers have been dealing with these issues

from different aspects, including network optimization, virtual machine configuration, power management, cluster scheduling, etc. But the difficulty also arises with the growing scale of clusters and the heterogeneity of applications, which encourages the system community to seek automatic solutions that are self-adaptive to different workloads and scale to large clusters.

3.1 Network Optimization with Reinforcement Learning

3.1.1 Early application: Q-routing.

The idea of applying reinforcement learning to traffic optimization has a relatively long history that can be traced back to the 90s. In 1994, Boyan and Littman proposed the Q-routing [3] algorithm, a distributed adaptive traffic control scheme based on reinforcement learning, for packet routing. It enables the router to send packets not only based on the precomputed shortest path but also be adaptive to the dynamically chancing load level, traffic patterns, and topology. By experimenting with different routing policies and gathering statistics, the agents discover efficient routing policies in a dynamically changing network to minimize total delivery time, without having to know in advance the global information about the topology or the traffic, and without the need for a centralized routing control system as well. The algorithm was further improved by Choi and Yeung to achieve better learning speed and adaptability even under low network load condition by keeping the best experiences to predict future traffic trend. Even in these early ages, although the computation power and memory capacity is very limited, the overhead for implementing Q-learning-based algorithm is still considered to be acceptable [7].

3.1.2 Client-side Adaptive Birate Control.

After decades of development, the workload over the Internet increases dramatically; it also delivers contents to a large variety of heterogeneous devices and poses new challenges to the designers. For example, there has been a rapid increase in the volume of HTTP-based high-quality video streaming traffic, which consumes huge amounts of bandwidth. Under this situation, dynamic adaptive streaming over HTTP (DASH) becomes the predominant form of video delivery; it allows each client-side video players to employ adaptive bitrate (ABR) algorithms to optimize their own quality of experience (QoE). The adaptive controller sits at the client side, observes various information such as estimated network throughput and playback buffer occupancy, and makes bitrate decisions. The majority of existing ABR algorithms are developed either through fixed rules based on current information, or solving QoE optimization problem assuming perfect future network throughput prediction.

To further improve client-based DASH controller, Pensieve [20] was proposed to apply reinforcement learning techniques to automatically learn the optimal bitrate control policy purely from experience.

In order to train the learning agent, they firstly build a simple but faithful simulation environment that captures several main features of the network, e.g., slow-start-restart of underlying TCP connections. Then the agent interacts with the simulator, takes a set of observations, such as client playback buffer occupancy, past bitrate decisions, and network signals like throughput measurements, feeds these values into the policy network, and collects the action, i.e., the bitrate to use for the next chunk, from the output. As a result, the QoE will be sent back by the environment as a reward, guiding the improvement of the policy network, which forms a closed-loop control system. Apropos the policy network, the authors used neural networks as function approximator and applied the state-of-the-art actor-critic algorithm that training two neural networks simultaneously in a distributed asynchronous manner, i.e., A3C [23]. For the actor network, 6 categories of information, including $k = 8$ past chunk throughput, $k = 8$ past chunk download time, available next chunk sizes to select from, current buffer level, number of chunks remained, and last chunk's

bitrate, are feed to a 1D convolution layer (CNN) with 128 filters (size 4, stride 1); the intermediate results from the convolutional layers are then aggregated with each other by a fully connected layer with 128 neurons; the final output is derived from a softmax function, masking out unsupported bitrate, which gives a probability distribution over possible bitrate levels for the agent to select from. The critic network shares the same architecture with the actor network except for the final output layer replaced with a linear neuron, instead of the softmax activation function.

Compared with existing adaptive bitrate algorithms either considering current buffer or incoming rate, or solving optimization problems with future prediction of throughput, Pensieve achieves higher average QoE by 12%-25%; comparison between Pensieve and former proposed reinforcement learning based video streaming algorithm [6], which addressed online MDP solving problem with tabular Q-learning method without using neural networks as function approximator, the authors claim that Pensieve incorporates a large amount of throughput history records, which violates the Markov property assumed by former works, and thus enjoys at most 46.3% performance improvement.

3.1.3 Server Selection Framework for Network Optimization.

Apart from the adaptive control from the client side, many researchers propose to apply learning-based optimization mechanism on the cluster level and some of them follows the framework of reinforcement learning.

The authors of *Pytheas* [16] presented a general server selection framework that enables application-level quality of experience (QoE) optimization via a data-driven approach.

They argue that this data-driven approach should be treated as a *real-time exploration and exploitation (E2)* process rather than a prediction problem. Because the prediction-based methods suffer from the biases introduced through incomplete measurement and, given the logical and temporal separation between data collection and decision making, it cannot respond to sudden changes such as load-induced quality degradation. The key algorithm lying behind the framework is named discounted *Upper Confidence Bound (UCB)* [1] under multi-armed bandits problem formulation, which is considered as a simplified environment, termed *non-associative* setting [36], in reinforcement learning where the agent only act with one situation. To solve the scalability issue in large cluster, the authors adapt a *group-based E2* method utilizing the similar features, e.g., IP prefix, best delays of VoIP, web load time from edge proxy, shared by servers belonging to the same network group.

This data-driven QoE optimization framework is general enough to be applied to a variety of applications, including video on demand, live streaming, Internet telephony (i.e., VoIP), cloud storage and file sharing, and many other web services where server selection can benefit quality of experience. The authors evaluated the framework on a video streaming application and showed that it can improve video QoE up to 31% on average and 78% on 90th percentile over a state-of-the-art prediction-based system.

3.1.4 Datacenter-scale Automatic Traffic Optimization.

However, some researchers, based on their expertise of system implementation and deep reinforcement learning techniques, suggest underlying challenges in designing end-to-end DRL systems at the datacenter scale for real-world usages.

The authors of *AuTO* [5], an end-to-end Automatic Traffic Optimization system, criticize current DRL systems' failure to provide acceptable latency at datacenter-scale: even on modern computers equipped with accelerating hardware (e.g., GPU), the computation time of simple DRL algorithm inference still takes around 100ms during which short flows (constitute the majority of flows) have already gone. To overcome this key challenge while enjoying the automatic parameter setting optimization towards long flows learned by the agent, the authors develop a two-level DRL system,

viz. Peripheral Systems *(PS)* and Central Systems *(CS)*, which makes local decisions with minimal delay for short flows (in *PS*) and aggregate information for global traffic optimization decisions (in *CS*). By deploying two DRL agents with DDPG and PG algorithm for CS respectively, AuTO enables fast turn-around of traffic optimization settings compared to data mining and web searching approach and reduces up to 48.14% reduction in average flow completion time over existing solutions.

3.1.5 Summary.

This section reviews applications of reinforcement learning for network optimization. It can be traced back to early in the 90s when Boyan and Littman proposed the *Q-routing* [3] algorithm and then improved by Choi and Yeung with *predictive Q-routing* [7] for adaptive traffic control. Bouncing back to recent works, *Pensieve* [20], as a client-side adaptive bitrate controller, achieves astonishing performance improvement with the state-of-the-art deep reinforcement learning models, A3C. *Pytheas* [16] focus on applying RL to the system-level network optimization and suggest a data-driven framework general enough to be applied to a variety of applications. And finally, the authors of *AuTO* [5] tackle the unacceptable latency by DRL approaches with a two-level system that could work on datacenter scale.

In the next section, we introduce virtual machine (VM) configuration problem, which is more closely related to the management of cloud resources and performance.

3.2 Virtual Machine Configuration with Reinforcement Learning

Virtualization, as an enabling technology for cloud computing, allows one computer to "look like" multiple computers, doing multiple jobs, by sharing resources of a single machine across multiple environments. Highly elastic virtual machines (VMs) powers the elasticity of clouds: launching new VMs in a virtualized environment is cheap and fast than deploying real machines, and consolidating multiple VMs onto one physical machine help improve resource utilization. A cloud infrastructure is, to some extend, a VM management infrastructure. To this end, we firstly review reinforcement learning approaches towards virtual machine configuration.

3.2.1 Elastic Virtual Machine Provisioning. [28, 29]

Leveraging the virtualization techniques that partitions hardware resources into virtual machine (VM) instances, the authors intend to manage how much resource should be given to each virtual machine, in other words, VM resource provisioning. In order to satisfy Service Level Agreement (SLA) of individual applications while optimizing resource utilization of the whole system, such control should be applied in an on-the-fly manner given the factors of fluctuating user demand, complicated interference between co-located VMs and the arbitrary deployment of multi-tier applications. To facilitate self-adaptive virtual machines (VM) resource provisioning, the authors propose a distributed reinforcement learning mechanism, named *iBalloon*, that tackles the following challenges:

(1) It should guarantee virtual machines' application-level performance, i.e., SLA, given the complex relationship between their performance and resource allocated;
(2) It should adjust the resources assigned to the VMs correspondingly to time-varying user demands;
(3) It should make no assumptions about the virtual machines' membership and deployment topology thus being able to work in a real-world open cloud environment.

As a distributed method, the system treats each VM resource allocation as a distributed learning task; the host aggregates requests on each machine and provides feedback to individual VMs to make them to learn their capacity management policy accordingly. Decoupling each machine's

decision from the whole system's status help solve the scalability issue of managing a large cluster, in other words, the complexity of the learning algorithm does not grow exponentially with the number of VMs required to manage.

As a self-adaptive approach, the learning agent monitors each VM's running status and takes the utilization of CPU, I/O, memory, and disk swap $(u_{cpu}, u_{io}, u_{mem}, u_{swap})$ as *state*, borrows the design in the Cerebellar Model Articulation Controller (CMAC) to represent the Q value with multiple coarse-grained Q tables to efficiently handle limited data, and outputs an *action* in the form of $(nop, increase, decrease)$, i.e., an increase, decrease, or no-operation on certain type of resource. The *reward* is defined as follows; when there is a resource conflict, the feedback system penalizes the VMs requesting resource that exceeds aggregated demand by "-1" reward and responses neurally ("0" reward) to the existing VMs; when there is no resource conflict, the reward reflects application performance and resource efficiency as the ratio of *yield* to *cost*, where *yield* is a summarized gain averaged over all performance metrics x_i (ranges from 0 to 1 where 1 represents x_i satisfies its SLA) and *cost* is also a summarized utility based on all resource utilization status u_j.

Their evaluation, based on TPC-W and TPC-C workloads with a 22-node (8 CPU cores, 8 GB memory each) cluster and a 16-node (12 CPU cores, 32 GB memory each) cluster of DELL machines with Linux kernel 2.6, showed that iBalloon performed well in the aspects of performance, resource competition, and scalability.

On performance guarantee, iBalloon could keep almost 90% of the request below SLA compared to *over-provisioning* cases on single machine. In the scenario of multiple applications competing for resources, iBalloon suggested a smaller resource offer to alleviate severe contention, thus beating the static over-provisioning allocation as the number of VMs increases. Apropos scalability and overhead, iBalloon achieved <5% throughput compared to *optimal* assigned by human while incurred 20% degradation on request latency, in the scale of 128 VMs on a correlated 16-node cluster; it required no more than 3% CPU resources for Q computation and approximately 18MB of memory for Q table storage on each physical machine.

3.2.2 VMs and applications co-configuration. [42]

To further optimize the performance of not only virtual machines, but also applications, the authors of iBalloon proposed a unified reinforcement learning (*URL*) approach for autonomic cloud management for auto-configuration of both VMs and multi-tier web appliances. The system resides in the scenario of a three-tier Apache/Tomcat/MYSQL website with each component running on a virtual machine.

The VMs' state is represented as a vector of number of virtual CPUs, scheduling credit, and memory allocated, i.e., $(vcpu, time, mem)$ and the actions a *VM-Agent* could take is increasing or decreasing a parameter setting in a predefined step size or keeping it unchanged, i.e., (inc, dec, nop). The immediate reward of an action is calculated by the difference between the service level objective (SLO) and the measured response time (RT), i.e., $r = SLO - RT$, or, for multiple VMs running applications in the same pool of physical servers, the geometric mean of their normalized SLOs, i.e., $measuredSLO/targetSLO$.

Apropos the applications running on each VM, an *App-Agent* is proposed, as the highlight of the paper, with a model-enhanced reinforcement learning approach. It first samples the performance of a small portion of typical configurations as an initial *policy*, then runs the online RL process to drive appliances into good configuration. The state of the application is defined as a group of 8 performance-critic configuration parameters, selected from more than a hundred of candidates, inlucindg MaxClients, Keepalive timeout, MinSpareServers, MaxSpareServers in Apache server and MaxThreads, Sessions timeout, minSpareThreads, maxSpareThreads in Tomcast server (MySQL parameters are set as default). After deriving the Q-value of these configurations, they

use a polynomial regression to predict the their overall appliance performance due to the different settings.

The evaluation is based on benchmark handling three typical web services workload: TPC-W, TPC-C, and SPECweb on Xen-based VMs with the prototypes of VM-Agent and App-Agent implemented. It shows that VM-Agent can improve the throughput by around 10% and the App-Agent, compared to a heuristic trial-and-error configuration tuning approach, can converge to the optimal settings more steadily in response to workload changes.

3.2.3 Scalable VMs and applications co-configuration. [4]

The authors then further improve the system to become a coordinated tuning framework, *CoTuner*[4], which uses a model-free hybrid reinforcement learning technique to enable coordination among applications and virtual resources.

Instead of conducing RL search in the whole configurable state space, as in the former approaches, the authors first divide original state space into multiple exclusive subsets, and then use a Simplex method to locate the best configuration for each subsets, which forms a much smaller but "promising" candidate state set for RL search. Afterwards, during the RL search procedure, they employ a system knowledge guided exploration policy to tackle the long-standing trade-off between exploitation and exploration. They consider CPU and memory utilization as performance metrics and set certain upper bound and lower bound for each of them; once the current resource utilization goes beyond the corresponding upper bound, actions of reducing resource are forbidden, and vice versa. This approach help stabilize the performance from the impact of counter-intuitive explorations.

In respect of the problem formulation, similar to their former proposal URL[42], they model the problem as a finite Markov decision process (MDP) without requiring explicit model of either the managed system or the external environment. The *state* of VM-Agent is defined as the virtual resource configuration, i.e., $(vcpu_i, mem_i)$ for ith VM, of all the VMs within the VM cluster it manages; the *state* of App-Agent is defined as all the configurable parameter settings of its corresponding application. The *actions* of VM-Agent and App-Agent are limited to discrete ones as increasing/decreasing the resource allocation or certain parameter by a fixed unit. The *reward* is defined to reflect the overall system performance by applications' throughput and response time, with respect to service level agreement (SLA) and SLA violation penalty. The value function is updated by *temporal-difference (TD)* method.

As a result, CoTuner can scale up to 16 servers with 100 VMs, achieves more than 93% of the optimal performance, and outperforms at least 15% over six other existing policies, including searching based strategies like *Nelder Mead* and *Hill Climbing*, control-based strategies like *Adaptive proportional Integral(PI)* and *Auto-regressive-moving-average(ARMA)*, and two reinforcement learning strategies with no hybrid approach with Simplex and with less optimized value function updating (enforce $\lambda = 0$ in TD method).

3.2.4 Summary.

In this section, we review applications of reinforcement learning for virtual machine configuration. The reviewed projects, iBalloon [28, 29], URL [42], and CoTuner[4], take a value-based RL approach (Q-learning and TD learning) towards online adjusting of resources allocated to the virtual machines, in the purpose of higher overall performance. In response to dynamic incoming request rate, the authors started from pure reinforcement learning approaches, then extended the system to jointly VMs provisioning and multi-tier web appliances parameter tuning, and finally further improved the performance and scalability through a hybrid RL approach with Simplex methods as space reduction technique. In the next section, we review the power management problem in

cloud computing, which not only pursues optimal performance but also considers its trade-off with power consumption, which is also critical to cloud service providers in the real world.

3.3 Power Management with Reinforcement Learning

In [9, 18, 37, 38], the authors discuss about dynamic power management (DPM), i.e., selectively shuts-off or shows-down system components in idle. The power provisioning results from Google's warehouse-size computer [10] also verify the significant potential of power management schemes in energy savings.

3.3.1 Dynamic Power Management with Machine Learning. [9]

The authors summarize the state-of-the-art dynamic power management policies, both heuristic ones and stochastic ones, deploy these policies as *experts* to manage machines' power consumption, and propose a machine learning based *controller* atop these experts to select the best expert, i.e., DPM policy, to perform for the current idle period. This *policy selection mechanism* relies on the observation that no single policy fits perfectly all operating conditions.

The heuristic policies, including timeout policy (with fixed or adaptive timeout value) and idle predictive policy (using regression equation or exponential average of history), are simple to implement but no power/performance trade-off guarantee. On the other hand, the stochastic policies, assuming request arrival as discrete-time MDP, continuous-time MDP, semi-MDP, or Time Indexed Markov Chain SMDP (TISMDP) models, though are much more complicated to implement, do offer optimality and performance bounds, but only for stationary workloads. In this work, the authors select 4 policies as expert working set representing different classes of state-of-the-art DPM policies: fixed timeout, adaptive timeout, exponential predictive, and TISMDP policies.

The most critical part of this system, i.e., how to evaluate and select the experts, is handled by the controller using a machine learning. Although the authors did not mention reinforcement learning at all, the system forms a *feedback loop* and *update controller's policy* in an iterative manner; the controller selects the experts i $(i = 1, 2, \cdots, N)$ at time slot t with the probability of $r_1^t, r_2^t, \cdots, r_N^t$ respectively, where r_i^t is the weight w_i associated with each expert i after normalization, i.e., $r_i^t \doteq w_i^t / \sum_{i=1}^{N} w_i^t$. The weight w_i, which indicates the benefit gained if the correspondent expert was selected during the last idle period, is initialized randomly and updated in an iterative manner for each time slot, by the rate of a tunable parameter β powered by energy loss factor l_i^t, i.e., $w_i^{t+1} = w_i^t \beta^{l_i^t}$. The calculation of energy loss factor l_i^t is omitted here for brevity; it is related to relative importance of energy savings and performance delay, break-even time, sleep time, and idle period.

The experiments conduct on HP 2200A hard disk drive (HDD) I/O workload and Cisco Aironet 350 series Wireless Adapter (WLAN) network workload both show that this algorithm guarantees its performance at any point of time is closest to that of the best performing policy.

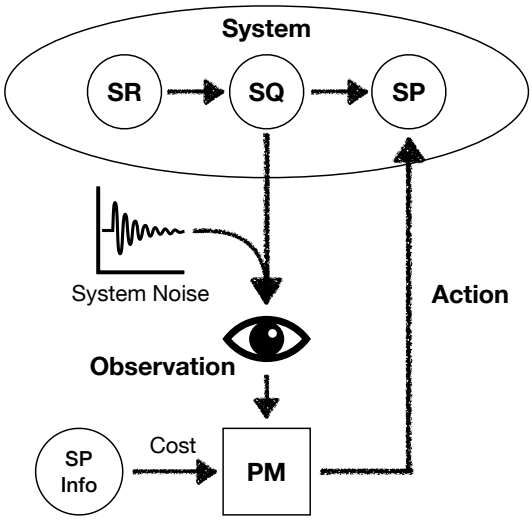
3.3.2 Adaptive DPM with RL. [37]

The authors were the first to apply model-free constrained reinforcement learning to learning the best power management (PM) policy providing minimum consumption under given performance constraints, without requiring any prior workload information. Compared with most previous power management work's approach of separating system modeling and policy optimization, the authors present a novel approach in modeling the system as a partially observable environment, and make the power manager learn a new power control policy, instead of learning how to selecting from a set of existing policies as proposed in [9].

Their problem formulation is based on the setting of a service requester *(SR)*, a service provider *(SP)*, and a service queue *(SQ)*. A power manager *(PM)* observes the system *state*, represented by the

3-tuple (SP, SR, SQ), through a noisy channel and then selects one of the available power modes as an *action* to impose on the system whenever there is a state transition (As shown in Fig.2). The authors choose Q-learning as the algorithm for its simplicity and robustness to noise, and further extend its value function to be the Lagrange cost of power consumption $c(s, a)$ constrained with delay caused by action $d(s, a)$, i.e., $C(s, a; \lambda) = c(s, a) + \lambda d(s, a)$, in order to strike trade-off between energy saving and performance. Q-value's learning rate $\epsilon(o, a)$ is inversely proportional to the number of times that certain observation-action pair (o, a) has been visited, for the purpose of faster convergence, i.e., $Q(o, a; \lambda) \leftarrow \left(1 - \varepsilon_{(o,a)}\right) Q(o, a; \lambda) + \varepsilon_{(o,a)} \left(c(o, a; \lambda) + \min_{a'} Q(o', a'; \lambda)\right)$.

Fig. 2. Illustration of Dynamic Power Management System [37]
SR: service requester, SQ: service queue, SP: service provider, PM: power manager.



The evaluation show that this model-free RL-based power management can achieve 24% and 3% reduction in power and latency without incurring any overhead in performance or energy, respectively. In addition, the improved Q-learning, modified by an adaptive factor in learning rate and updating multiple Q-values instead of one value in each step, outperforms traditional Q-learning significantly.

3.3.3 Near-optimal DPM. [38]

In [38], the authors formulate the DPM problem as a semi-Markov decision process (SMDP) which is an extension of Markov decision process to continuous-time discrete-state models. To solve such non-Markovian and non-stationary DPM problem, especially when the agent has no prior knowledge about state transition probabilities, i.e., model-free, the authors adopt the TD(λ) algorithm (introduced as Eq.20) that behaves more robustly and learns faster under this scenario. Besides, a workload predictor, powered by an online Naïve Bayes classifier, is also included in the framework to provide effective estimates of the workload states as the input of RL algorithm.

Compared to Dhiman et al.'s approach [9], this framework automate the policies selection process and can achieves better power-performance trade-off. Compared to Tan et al.'s Q-learning solution [37], this work solve it with a continuous-time model instead of a discrete-time one thus eliminating large overhead in real implementations. Besides, the state and action spaces of the RL

algorithm are reduced by designing two separate policies, i.e., timeout policy and N-policy, corresponding to different states, which help accelerate the learning process to less than 100 service requests as well.

The detailed settings are described as follows. The state of the system is a 3-tuple (SP, SQ, SP) where SR is the service request generating rate (high, low, etc.) or next inter-arrival time (short, long ,etc.), SQ is the number of requests in the service queue, and SP is the system power state (busy, idle, sleep). The *decision epochs* t_k are coincides the four states when the system power state is idle or sleep (SP=idle or sleep) with the service queue is either zero or non-empty ($SQ = 0$ or ≥ 1). The action, if system power state is idle, is to select a *timeout* value (including immediate shutdown) to power off the machine when the system power stats is idle; for the cases when the system power state is sleep and there have been some requests waiting in the service queue, the action is to select an N value as the threshold: the system will wake up to process requests only if number of requests in service queue reaches or exceeds N (i.e., $SQ \geq N$). The reward rate is replaced by the "cost rate", defined as a linearly-weighted combination of power consumption and the number of requests buffered in the service queue; given the Little's Law, the average number of requests in queue is proportional to the average latency for each request, which indicates the performance of certain service.

As a result, the Naïve Bayes classifier can reach around 80% accuracy for workload prediction; the whole model-free reinforcement learning based dynamic power management system can save up to 16.7% energy without any increase in the latency, and up to 28.6% latency reduction without any power addition power consumption, compared to the state-of-the-art expert-based approach.

3.3.4 Scalable DPM and Resource Allocation. [18]

Going beyond Wang et al.'s work [38] that achieves near optimal power management policy on single machine, Liu et al. [18] extend the scale of setting to power management of multiple machines and tackle the problem in the scenario of a cluster with tens or hundreds of servers, handling incoming jobs by allocating virtual machines to certain servers in an on-demand manner. Given the powerful adaptive power of learning agent, system administrator can also make trade-off between performance and operating costs, such as average job latency and energy consumption, strike a better balance, and tune the system to be more efficient in general.

To solve resource allocation and power management problem in such cloud computing systems, the authors propose a two-tier hierarchical framework, powered by reinforcement learning and deep learning techniques, to achieve best trade-off between latency and power/energy consumption in a server cluster. The global tier, in the proposed hierarchical framework, controls a job broker that dispatches incoming jobs with certain resource request to one of the servers at their arrival time. After that, each server maintains its own queue for assigned jobs and allocates resources to them in a first-come-first-serve manner. On the other hand, the local tier in the proposed hierarchical framework locates on each server and selectively powers off and turns on each single machine in a distributed manner, when it is idle or requested to execute jobs; it is regarded as dynamic power management that can effectively reduce power dissipation at system level.

The key challenge that the authors face is the scalability: the number of servers and characteristics of workloads can be over hundreds, and the naïve representation of cluster state could be the Cartesian product of the aforementioned two parts, which results in high dimensions in *state* space; similarly, the *action* space can be also large under naïve representation, like deciding the allocation of each job's virtual machine to certain servers and allocating certain resources to the servers for job execution. The high dimensions in state and action spaces prohibits the effectiveness of applying traditional tabular Reinforcement Learning techniques, such as Q-learning, to the

entire cluster in terms of both convergence speed and final performance. So as to tackle this difficulty, the authors use deep neural networks as value function approximators and follow the idea of divide-and-conquer: they divide the cluster into several server groups, calculate the value function for each group, considering other groups' information as well through the fixed-size outputs of an auto-encoder, compute the value for each action available to the whole cluster, and finally takes the action with highest score.

The evaluation compares the proposed hierarchical framework with the round-robin VM allocation policy with different fixed timeout value. It shows that RL-based resource allocation framework always outperforms the round-robin policy, especially in large-size server cluster with tens of thousands of jobs, and achieves better trade-off than any of fixed timeout value policy between job latency and power/energy consumption.

3.3.5 Summary.

This section reviews applications of machine learning and reinforcement learning for dynamic power management: starting from Dhiman et al.'s approach [9] selecting experts with a feedback loop, to Tan et al.'s Q-learning solution [37] and Wang et al.'s TD(λ) approach with Naïve Bayes workload predictor [38], and finally Liu et al. [18] extend the scale of setting from single machine to clusters on the cloud with the help of deep learning techniques. This reveals the long-standing trade-off between performance and power consumption and poses number of challenges like scalability and heterogeneity in the recent cloud computing scenario. In the next section, we review the workload scheduling problem in clusters, which provides more discussion on these issues.

3.4 Cluster Scheduling with Reinforcement Learning

Resource management is about how to efficiently allocate resources to achieve higher performance and/or lower operating cost.

3.4.1 Basic Job Scheduling with Deep Reinforcement Learning. [19]

As a pioneer work of applying reinforcement learning to resource management, Mao et al.[19] discuss the challenges that existing human-generated heuristics for cluster scheduling are facing:

(1) The underlying systems are often too complex to model accurately. For example, running time of a task may vary with data locality, server characteristics, interactions with other tasks, interference on shared resources, etc.
(2) Online decisions should be made with noisy inputs under diverse conditions. For example, adaptive bitrate controller should make decisions based on noisy forecasts of future workload and operate well on devices with different configurations.
(3) Some performance metrics of interest, such as tail performance, are hard to optimize in a principled manner.

And then they motivate taking an RL based approach, named *DeepRM*, to tackle these problems with the following reasons:

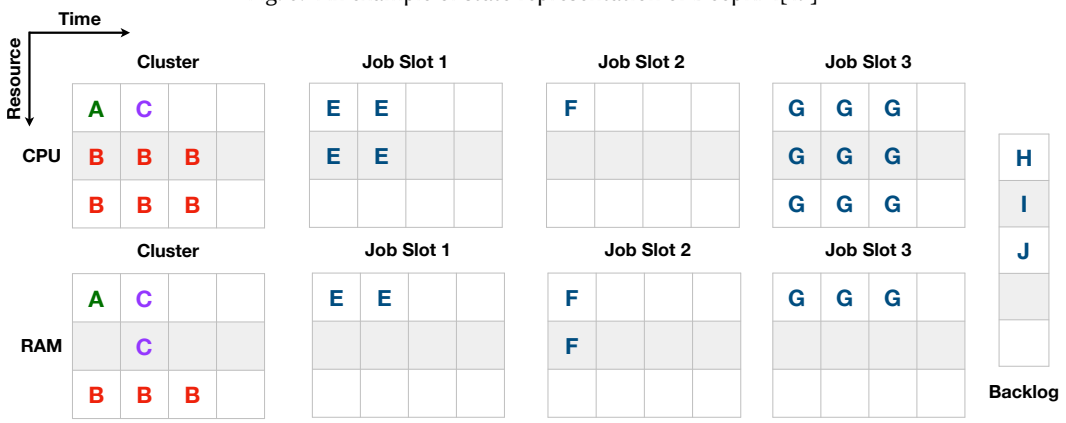
(1) Decisions made by these systems are often highly repetitive; abundant pairs of cluster scheduling decisions and the resulting performance can be treated as training data for RL algorithms.
(2) The complex system and decision making policies can be automatically modeled by deep neural networks without human-labor; noisy signals can also be incorporated as inputs to neural networks.
(3) As long as there exist reward signals that correlate with the performance metrics of interests, it is possible to train for these objectives directly.

(4) An RL agent can learn to optimize for a specific workload (e.g., small jobs, low load, periodicity) and to be adaptive to varying scenarios.

And also similar to other ground-breaking works, the authors propose DeepRM working in a simple, if not oversimplified, cluster setting with several assumptions. They consider the system with d resource types (e.g., CPU, memory, I/O bandwidth), and jobs arriving to the cluster in discrete time steps. They assume complete information about the jobs is known upon arrival, including the resource demand of each job j, represented as resource *profile* vector $\mathbf{r}_j = (r_{j,1}, \cdots, r_{j,d})$, as well as the *duration* of the job, T_j. The problem is further streamlined by assuming there are *no preemption* of jobs, *no malleability* of resources demand, and *no fragmentation* of machines. The primary system objective is *average job slowdown*; more specifically, for each job j, the slowdown is defined as $S_j \doteq C_j/T_j$, where C_j is the real duration of the job (the time period between its arrival and its completion of execution) and T_j is the ideal duration of the job.

The state of the system – including both the current allocation of cluster resources and the resource profiles of jobs waiting to be scheduled – are represented as distinct *images* (see Fig. 3). The cluster images (two leftmost images for CPU and memory resources respectively) represent the scheduling decision of jobs (A, B, and C) in terms of both units of resource (vertical axis) and units of time steps allocated (horizontal axis). The job slot images show the resource demand of awaiting jobs (E, F, and G). The backlog image serves as a counter of jobs (H, I, and J) waiting to be scheduled other than the first M jobs ($M = 3$ in Fig. 3), for the purpose of summarizing the information about any jobs beyond the first M thus limiting the number of images applied to $d \times (M + 1)$ as input to a neural network.

Fig. 3. An example of state representation of DeepRM[19]



Regarding the action space, the schedule can submit any subset of the M jobs, which leads to large action space of 2^M. To make the action space small enough to be learned, the authors design a trick to allowing the agent to execute more than one action in each "time step". The action space is given by $\{\emptyset, 1, \cdots, M\}$ where take action $a = i$ means "schedule the job at the i-th time slot"; time is frozen until the scheduler either selects an invalid action, or the action $a = \emptyset$, i.e., stopping scheduling more jobs in current time step, then the real-time proceeding is triggered. The decomposition of agent's decision sequence from real time, allowing the agent to schedule multiple jobs as the practical situation while keeping the action space linear in M in this situation,

serves as a basic but important idea in RL problem formulation and is widely adopted in many literature.

Since the objective is to minimize average slowdown, the reward is defined as $\sum_{j \in \mathcal{J}}(-1/T_j)$ where T_j is the (ideal) duration of the job, and is given only for each real time step (i.e., after $a = \emptyset$). Hence maximizing the cumulative reward mimics minimizing the average slowdown, which is given by $S_j = C_j/T_j$ where C_j is the real completion time of the job. The learning algorithm is selected as the text-book policy gradient method, i.e., REINFORCE algorithm (Eq.29).

The evaluation compares DeepRM with standard heuristics such as Shortest-Job-First (SJF) and a packing scheme inspired by Tetris [13]. It shows that DeepRM can performs comparably with there heuristics; DeepRM scheduler learns strategies like favoring short jobs over long jobs and keeping some resources free, via withholding long jobs, to serve possible future arriving short jobs, *directly from experience*.

3.4.2 DAG Job Scheduling with DRL and MCTS. [15]

Inspired by DeepRM [19], Hu et al. present a scheduling framework, *Spear* [15], designed to minimize the job completion time *(makespan)* of complex data processing jobs in big data systems. Not only considering the heterogeneous resource demands of different tasks as what DeepRM suggested, the authors also task tasks dependencies within each job into consideration, which serves as the main focus and contribution of Spear.

The dependencies between tasks are commonly represented by a direct acyclic graph (DAG), and how to schedule DAGs, especially when we need to consider the heterogeneous demands for multiple types of resources, is NP-hard [13]. Many existing approaches, though claim to be dependency-aware, is heavily relied upon a suite of manually-tuned parameters used to define a set of troublesome tasks that is critical to final scheduling outcomes, including Tetris [13] and Graphene [14].

Based on this observation, the authors believe that the ideal task scheduling algorithm should directly examine the search space and search for the best possible scheduling to minimize the makespan, with no assumptions or parameter settings based on empirical experience; the search should not be exhaustive but should be conduct in a disciplined manner. To this end, Spear deploys a deep reinforcement learning (DRL) agent working under the look-ahead search method called Monte Carlo Tree Search (MCTS), which was applied by AlphaGo Zero [35] for Go games.

More specifically, the resource cluster is modeled as a resource-time space for a fixed period of time; with the similar state representation as DeepRM in Fig.3, each resource dimension can be expressed as a separate rectangle with width for resource occupation and height as time span. The action space, $\{-1, 1, 2, \cdots, n\}$, is also modeled in a sequential decision that $a = i(i \neq -1)$ means scheduling the i-th task and action $a = -1$ means processing the tasks in the cluster and triggering real time proceeding. The agent will receive -1 reward for each processing action, which enforce it to pursue least processing actions, i.e., shortest DAG makespan.

In the MCTS algorithm, the state tree is built and grows coincide with each scheduling decisions. For example, starting from initial state node "$\emptyset$", if task A is scheduled first, the edge "schedule A" will lead to state node "A"; then task B is schedule, the edge "schedule B" will lead to state node "AB"; if task B is scheduled before task A in the first place, the final state node will be "BA".

Based on this method, the MCTS expends the search tree and gives different scheduling decisions by performing four steps, *Selection*, *Expansion*, *Simulation*, and *Back-propagation*. The authors improve the traditional MCTS in the expansion step by introducing the aforementioned resource-aware RL agent trained to select child path for further explore, instead of expand randomly as the common approach. Assisted with deep neural networks as function approximator, the RL agent will be able to choose more promising unexplored subtrees to minimize the makespan of a DAG.

Also in the simulation, the trained DRL agent is trained with *policy gradient* method and is applied to replace the random policy to choose actions until reaching a terminal state (*rollout*), which provides a more meaningful estimation of the makespan.

The evaluation is conduct on simulator with 10 DAGs (100 tasks each) and on traces collected from production clusters running hive workload with 99 MapReduce jobs (at least 5 tasks each); it compares Spear with four other heuristics, i.e., Tetris, shortest job first (SJF), greedy towards high critical path (CP), and Graphene, and shows that Spear can outperform those approaches by up to 20%.

3.4.3 Scalable DAG Scheduling with DRL. [21]

In the meanwhile, the authors of DeepRM, Mao et al., present their project, *Decima*, as a sophisticated cluster scheduler in order to achieve efficient data processing job scheduling as well. The key idea is to make the scheduler be aware of jobs' dependencies and make workload-specific decisions automatically.

(1) Dependency-aware task scheduling requires the scheduler to understand jobs' dependency structure and plan ahead. As mentioned before, it is commonly observed that many data-parallel jobs have complex data-flow graphs between different tasks. When a cluster runs multiple DAGs in parallel, the decision to figure out a *combined schedule* for all of them, termed "DAG packing problem", is NP-hard [14].
(2) Workload-specific decision requires setting the right level of parallelism for different jobs. For example, a job with large input or large intermediate data seeks additional parallelism for higher efficiency, while jobs with small size of data does not.

In addition to realizing dependency-aware task scheduling and workload-specific parallelism decision, there are three challenges need to tackle:

(1) Arbitrary DAGs to fixed-size state input. Similar to DeepRM, the authors apply deep neural networks to model the complex system and decision making policies, but the inputs to schedulers are DAGs with different attributes attached to nodes and edges. A new *graph embedding* technique is introduced to map job DAGs with arbitrary size and shape to fixed-size input vectors as states fed into the neural networks.
(2) Scalability. The cluster scheduler should scale to hundreds of jobs and thousands of machines in real-world cloud and decide among possibly hundreds of configurations per job. Thus careful design of the state space, i.e., amount of information considered, and action space, i.e., number of choices to select from, is critical. The authors propose decoupling task selecting decisions and parallelism determine decisions to reduce model complexity; parameter sharing among neural networks handling different number of jobs is also applied.
(3) Various arrival pattern. The variance introduced by continuous, streaming, stochastic job arrival could lead to variance in reward values, which confuses the outcomes of different arrival patterns with the outcomes of low-quality scheduling policy's decision. To handle this issue, the authors condition training feedback on the actual sequence of job arrivals experience, thus isolating the contributions of the scheduling policy in the overall feedback.

After addressing the aforementioned challenges, the authors train the RL-based scheduler with the well-known policy gradient method, REINFORCE algorithm. Some RL techniques, e.g., subtracting baseline when updating policy to reduce variance of the estimated gradient, and using differential reward instead of the standard one to work with infinite time horizon, are applied to enhance the convergence and training speed. The training happens offline with a simulator, which is built faithfully referring to an industrial trace and profiling information from a real Spark cluster consists of 25 worker VMs.

To evaluate the power of Decima in real-world cluster, the author implement it as a pluggable scheduler component enabling parallel data processing platforms like Spark, Dryad, or YARN to communicate with over an RPC interface. It shows that Decima outperforms all human-engineered heuristics schemes, both on batch and streaming job arrivals, achieving 21% to 3.1× lower average job completion time, because of its adaptive power to different high-level objectives, workloads, and environmental conditions.

3.4.4 Summary.

This section reviews applications of deep reinforcement learning for cluster scheduling. Mao et al. propose DeepRM [19] as a pioneer work and discuss the challenges that existing human-generated heuristics for cluster scheduling are facing. Following the similar formulation, Hu et al. apply DRL technique along with Monte Carlo Tree Search and design a dependency-aware DAG job scheduling agent. In the meanwhile, Mao et al. are also trying to solve DAG scheduling problem with a system called *Decima* [21], further considering stochastic arrival pattern and scalability issues.

3.5 Summary of Common Challenges in Cloud Management with RL

In this chapter, we review how researchers with a variety of background and interests, namely network optimization, VM configuration, power management, and cluster scheduling, formulate different system optimization problem with reinforcement learning concepts. Although their definition of states, actions, and rewards are usually distinct, some of them are facing similar characteristics of reinforcement learning agents and tackling these challenges with their own tricks. We summarize four of these problems and briefly recapitulate the authors' solutions as follows.

3.5.1 Adaptive to Dynamic Workload.

(1) Taking workload predictor's output as state input. In near-optimal DPM [38], Wang et al. adopt a Naïve Bayes classifier as a predictor offering effective estimation of the workload states as the input to the RL algorithm. Similarly, Liu et al. [18] deploy a workload predictor implemented by recurrent neural networks with LSTM cell to provide the next job interval estimation information to downstream RL agents.
(2) Conditioning training feedback with specific input pattern. Decima [21] isolates the contribution of scheduling policy under certain types of workload from others, builds different baselines for reward, and subtracts them when updating to reduce the variance of rewards. This input-dependent baseline technique has also adopted by the authors to improve training stability and quality across environments from queuing systems, computer networks, and MuJoCo robotic locomotion [22].

3.5.2 Cold Start of RL Agent.

(1) Initializing network by supervised learning prior to reinforcement learning. Spear [15] teach the network to imitate a greedy heuristic approach such as critical path algorithm, based on the observation that simulations with random policy network will result in extremely low convergence. AlphaGo [34] also use supervised learning on predicting the expert moves in the game of Go as the first state of training pipeline.
(2) Sampling typical configurations as initial policy. URL [42] initializes the App-Agent with default and typical configurations of applications, and then drive run the RL to drive them into better configurations.

3.5.3 Scalability.

(1) Divide and conquer. iBalloon [29] treats the resource allocation as distributed learning tasks by isolating each machine's observation and decision from the whole system's status. Similarly, Liu et al. [18] allocate a DPM RL agent to each single machine to make local decisions, only sharing neural network parameters among these agents.
(2) Compressing global information. Liu et al. [18] deploy an auto-encoder for each server to extract lower-dimensional high-level representation of other machines; after squeezing these information into a fixed length vector, the agent concatenates the compressed global information vector after the current machine's state and sends them into the RL algorithm.
(3) Using embeddings. Decima [21] adopts a graph neural networks to convert all jobs' DAGs, of arbitrary shape, into 3 fixed-size vectors, viz. per-node embeddings, per-job embeddings, and global information embeddings, as inputs for subsequent policy neural networks.

3.5.4 Space Reduction.

(1) Decoupling actions. Decima [21] decouples task scheduling decision and job parallelism degree selection. Similarly, Liu et al. [18] decompose the dynamic power management (DPM) from the job scheduling decision into two individual learning agents.
(2) Converting combinatorial decisions to sequential ones. DeepRM [19] converts the scheduling decision of any subset of M candidates (2^M possible actions) to a sequential of $\{\emptyset, 1, \cdots, M\}$ decisions compressed in one real time slot, which maintains the original decision structure while keeping the action space linear to M. Spear [15] adopts a similar representation when scheduling DAG jobs.
(3) Merging different action spaces with masking. Pensieve [20] generates a single model to handle videos of all type by taking the union set of all possible actions and masking out unavailable ones when dealing with certain type of videos.
(4) Ignoring lower-priority information. DeepRM [19] maintains job resource images for only the first M jobs since earlier-arriving ones may have been waiting longer thus enjoying higher priority.
(5) Preprocessing with other methods or heuristics. CoTuner[4] uses downhill Simplex method, a high-efficient heuristic-based method for optimization problems, before RL to locate the best configuration for each subset, which forms a much smaller but "promising" candidate state set for RL search. Spear[15] only considers the tasks that can be scheduled to start before the earliest finish time of tasks in the cluster when doing MCTS expansion.

4 CONCLUSIONS

This paper serves as a survey of selected cloud management topics addressed by reinforcement learning approaches. Firstly, we gave an overview of reinforcement learning with basic concepts, classic models, and state-of-the-art methods. Then, we presented several lines of applications conduct by researchers interested in different system topics, including *network optimization*, *virtual machine configuration*, *dynamic power management*, and *cluster scheduling*. After reviewing a variety of problem formulations and solutions, we conclude the survey with the discussion of common challenges encountered by the researchers and, hopefully, could motivate further research directions and industrial-oriented solutions.

ACKNOWLEDGMENTS

I would like to express my sincere appreciation to my advisor Prof. Wei WANG, for his insightful comments and continuous encouragement during my exploration of this topic. Also thanks to the rest of my PhD qualifying exam committee, Prof. Qian ZHANG, Prof. Kai CHEN, and Prof. Yangqiu SONG, for their kind attention and support.

REFERENCES

[1] Peter Auer, Nicolo Cesa-Bianchi, and Paul Fischer. 2002. Finite-time analysis of the multiarmed bandit problem. *Machine learning* 47, 2-3 (2002), 235–256.

[2] Dimitri P. Bertsekas and John N. Tsitsiklis. 1996. Neuro-dynamic programming. In *Optimization and neural computation series*.

[3] Justin A Boyan and Michael L Littman. 1994. Packet routing in dynamically changing networks: A reinforcement learning approach. In *Advances in neural information processing systems*. 671–678.

[4] Xiangping Bu, Jia Rao, and Cheng-Zhong Xu. 2013. Coordinated self-configuration of virtual machines and appliances using a model-free learning approach. *IEEE Transactions on Parallel and Distributed Systems* 24, 4 (2013), 681–690.

[5] Li Chen, Justinas Lingys, Kai Chen, and Feng Liu. 2018. Auto: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*. ACM, 191–205.

[6] Federico Chiariotti, Stefano D'Aronco, Laura Toni, and Pascal Frossard. 2016. Online learning adaptation strategy for DASH clients. In *Proceedings of the 7th International Conference on Multimedia Systems*. ACM, 8.

[7] Samuel PM Choi and Dit-Yan Yeung. 1996. Predictive Q-routing: A memory-based reinforcement learning approach to adaptive traffic control. In *Advances in Neural Information Processing Systems*. 945–951.

[8] George Cybenko. 1989. Approximation by superpositions of a sigmoidal function. *Mathematics of control, signals and systems* 2, 4 (1989), 303–314.

[9] Gaurav Dhiman and T. T. Rosing. 2006. Dynamic Power Management Using Machine Learning. *2006 IEEE/ACM International Conference on Computer Aided Design* (2006), 747–754.

[10] Xiaobo Fan, Wolf-Dietrich Weber, and Luiz André Barroso. 2007. Power provisioning for a warehouse-sized computer. In *ISCA*.

[11] Armando Fox, Rean Griffith, Anthony Joseph, Randy Katz, Andrew Konwinski, Gunho Lee, David Patterson, Ariel Rabkin, and Ion Stoica. 2009. Above the clouds: A berkeley view of cloud computing. *Dept. Electrical Eng. and Comput. Sciences, University of California, Berkeley, Rep. UCB/EECS* 28, 13 (2009), 2009.

[12] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep learning*. MIT press.

[13] Robert Grandl, Ganesh Ananthanarayanan, Srikanth Kandula, Sriram Rao, and Aditya Akella. 2014. Multi-resource packing for cluster schedulers. In *SIGCOMM*.

[14] Robert Grandl, Srikanth Kandula, Sriram Rao, Aditya Akella, and Janardhan Kulkarni. 2016. GRAPHENE: Packing and Dependency-Aware Scheduling for Data-Parallel Clusters. In *OSDI*.

[15] Zhiming Hu, James Tu, and Baochun Li. 2019. Spear: Optimized Dependency-Aware Task Scheduling with Deep Reinforcement Learning. *2019 39th IEEE International Conference on Distributed Computing Systems* (July 2019), 10.

[16] Junchen Jiang, Shijie Sun, Vyas Sekar, and Hui Zhang. 2017. Pytheas: Enabling data-driven quality of experience optimization using group-based exploration-exploitation. In *14th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 17)*. 393–406.

[17] Yann LeCun, Yoshua Bengio, and Geoffrey E. Hinton. 2015. Deep Learning. *Nature* 521 (2015), 436–444.

[18] Ning Liu, Zhe Li, Jielong Xu, Zhiyuan Xu, Sheng Lin, Qinru Qiu, Jian Tang, and Yanzhi Wang. 2017. A hierarchical framework of cloud resource allocation and power management using deep reinforcement learning. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 372–382.

[19] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. 2016. Resource Management with Deep Reinforcement Learning. In *HotNets*.

[20] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. 2017. Neural adaptive video streaming with pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*. ACM, 197–210.

[21] Hongzi Mao, Malte Schwarzkopf, Shaileshh Bojja Venkatakrishnan, Zili Meng, and Mohammad Alizadeh. 2018. Learning scheduling algorithms for data processing clusters. *arXiv preprint arXiv:1810.01963* (2018).

[22] Hongzi Mao, Shaileshh Bojja Venkatakrishnan, Malte Schwarzkopf, and Mohammad Alizadeh. 2018. Variance Reduction for Reinforcement Learning in Input-Driven Environments. *CoRR* abs/1807.02264 (2018).

[23] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*. 1928–1937.

[24] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602* (2013).

[25] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *Nature* 518, 7540 (2015), 529.

[26] Warren B Powell. 2007. *Approximate Dynamic Programming: Solving the curses of dimensionality*. Vol. 703. John Wiley & Sons.

[27] Martin L Puterman. 2014. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons.
[28] Jia Rao, Xiangping Bu, Kun Wang, and Cheng-Zhong Xu. 2011. Self-adaptive provisioning of virtualized resources in cloud computing. *ACM SIGMETRICS Performance Evaluation Review* 39, 1 (2011), 321–322.
[29] Jia Rao, Xiangping Bu, Cheng-Zhong Xu, and Kun Wang. 2011. A distributed self-learning approach for elastic provisioning of virtualized cloud resources. In *2011 IEEE 19th Annual International Symposium on Modelling, Analysis, and Simulation of Computer and Telecommunication Systems*. IEEE, 45–54.
[30] Gavin A Rummery and Mahesan Niranjan. 1994. *On-line Q-learning using connectionist systems*. Vol. 37. University of Cambridge, Department of Engineering Cambridge, England.
[31] John Schulman. 2016. *Optimizing expectations: From deep reinforcement learning to stochastic computation graphs*. Ph.D. Dissertation. UC Berkeley.
[32] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. 2015. Trust Region Policy Optimization. In *ICML*.
[33] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. *CoRR* abs/1707.06347 (2017).
[34] David Silver, Aja Huang, Chris J. Maddison, Arthur Guez, Laurent Sifre, George van den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Vedavyas Panneershelvam, Marc Lanctot, Sander Dieleman, Dominik Grewe, John Nham, Nal Kalchbrenner, Ilya Sutskever, Timothy P. Lillicrap, Madeleine Leach, Koray Kavukcuoglu, Thore Graepel, and Demis Hassabis. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* 529 (2016), 484–489.
[35] David Silver, Julian Schrittwieser, Karen Simonyan, Ioannis Antonoglou, Aja Huang, Arthur Guez, Thomas Hubert, Lucas Baker, Matthew Lai, Adrian Bolton, et al. 2017. Mastering the game of go without human knowledge. *Nature* 550, 7676 (2017), 354.
[36] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.
[37] Ying Tan, Wei Liu, and Qinru Qiu. 2009. Adaptive power management using reinforcement learning. *2009 IEEE/ACM International Conference on Computer-Aided Design - Digest of Technical Papers* (2009), 461–467.
[38] Yuankai Wang, Qing Xie, Ahmed C. Ammari, and Massoud Pedram. 2011. Deriving a near-optimal power management policy using model-free reinforcement learning and Bayesian classification. *2011 48th ACM/EDAC/IEEE Design Automation Conference (DAC)* (2011), 41–46.
[39] Christopher John Cornish Hellaby Watkins. 1989. *Learning from delayed rewards*. Ph.D. Dissertation. King's College, Cambridge.
[40] Lilian Weng. 2018. Policy Gradient Algorithms. https://lilianweng.github.io/lil-log/2018/04/08/policy-gradient-algorithms.html#policy-gradient-algorithms
[41] Ronald J Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning* 8, 3-4 (1992), 229–256.
[42] Cheng-Zhong Xu, Jia Rao, and Xiangping Bu. 2012. URL: A unified reinforcement learning approach for autonomic cloud management. *J. Parallel and Distrib. Comput.* 72, 2 (2012), 95–105.