Reinforcement-Learning-Empowered MLaaS Scheduling for Serving Intelligent Internet of Things

Heyang Qin, *Student Member, IEEE*, Syed Zawad, *Student Member, IEEE*, Yanqi Zhou, *Member, IEEE*, Sanjay Padhi, *Member, IEEE*, Lei Yang, *Senior Member, IEEE*, and Feng Yan, *Member, IEEE*

***Abstract*—Machine learning (ML) has been embedded in many Internet of Things (IoT) applications (e.g., smart home and autonomous driving). Yet it is often infeasible to deploy ML models on IoT devices due to resource limitation. Thus, deploying trained ML models in the cloud and providing inference services to IoT devices becomes a plausible solution. To provide low-latency ML serving to massive IoT devices, a natural and promising approach is to use parallelism in computation. However, existing ML systems (e.g., Tensorflow) and cloud ML-serving platforms (e.g., SageMaker) are service-level-objective (SLO) agnostic and rely on users to manually configure the parallelism at both request and operation levels. To address this challenge, we propose a region-based reinforcement learning (RRL)-based scheduling framework for ML serving in IoT applications that can efficiently identify optimal configurations under dynamic workloads. A key observation is that the system performance under similar configurations in a region can be accurately estimated by using the system performance under one of these configurations due to their correlation. We theoretically show that the RRL approach can achieve fast convergence speed at the cost of performance loss. To improve the performance, we propose an adaptive RRL algorithm based on Bayesian optimization to balance the convergence speed and the optimality. The proposed framework is prototyped and evaluated on the Tensorflow Serving system. Extensive experimental results show that the proposed approach can outperform state-of-the-art approaches by finding near-optimal solutions over eight times faster while reducing inference latency up to 88.9% and reducing SLO violation up to 91.6%.**

***Index Terms*—Internet of Things (IoT), machine-learning-as-a-service (MLaaS), model inference, parallelism parameter tuning, reinforcement learning, service-level-objective (SLO), workload scheduling.**

Manuscript received October 30, 2019; revised December 15, 2019; accepted December 23, 2019. Date of publication January 9, 2020; date of current version July 10, 2020. This work was supported in part by the National Science Foundation under Grant CCF-1756013, Grant IIS-1838024 (using resources provided by Amazon Web Services as part of the NSF BIGDATA Program), and Grant EEC-1801727; and in part by Amazon Web Services Cloud Credits for Research Award. This article was presented in part at the International Conference for High Performance Computing, Networking, Storage and Analysis. *(Corresponding author: Lei Yang.)*

Heyang Qin, Syed Zawad, Lei Yang, and Feng Yan are with the Department of Computer Science and Engineering, University of Nevada, Reno, NV 89557 USA (e-mail: heyang_qin@nevada.unr.edu; szawad@nevada.unr.edu; leiy@unr.edu; fyan@unr.edu).

Yanqi Zhou is with Google, Google Brain, Mountain View, CA 94043 USA (e-mail: yanqiz@google.com).

Sanjay Padhi is with U.S. Education, Amazon Web Services, Seattle, WA 98109 USA (e-mail: sanpadhi@amazon.com).

Digital Object Identifier 10.1109/JIOT.2020.2965103

I. Introduction

Recent years have witnessed the proliferation of Internet of Things (IoT) in every aspect of people's life, work, and entertainment. Meanwhile, artificial intelligence (AI) has recently shown a remarkable success in a wide range of fields, spanning from computer vision [2], speech recognition [3], natural language processing [4] to chess playing (e.g., AlphaGo [5]) and robotics. With the emergence of diverse IoT applications (e.g., smart home, smart city, industrial automation, and connected car), it is envisaged that AI could deal with these heterogeneous IoT environments. However, limited by the IoT device capability, it is challenging to deploy machine learning (ML) models on IoT devices, which prompts the development of machine-learning-as-a-service (MLaaS) [6] that provides the *machine model inference* service in the cloud. Many major cloud service providers, including Google, Microsoft, and Amazon have offered MLaaS in their cloud environment. Different from local inference, IoT device will use MLaaS and require a timely and highly available ML service to function properly. MLaaS frees the IoT device from the burden of providing storage and computation for ML models and allows the manufacturer to update the models without having to push the update to all the IoT devices. At the same time, it becomes challenging to provide low-latency machine model inference service to massive IoT devices in heterogeneous environments.

In this article, we focus on providing low-latency *model inference* service (also known as ML serving) for IoT devices in heterogeneous environments. Unlike offline ML model training, which may take hours or even days, one main requirement of ML serving is to achieve consistently low latency to meet the need of interactive and real-time IoT applications like smart home and autonomous driving. However, the challenge lies in the fact that productional ML models for many complicated tasks often contain billions of neural connections, and it may take seconds or even minutes to fulfill users' requests [7][1] if executed in a *sequential* manner, leading to unacceptably long latency for IoT applications.

A natural and promising approach to meet the strict latency service-level-objective (SLO) is to use parallelism in computation [9], [10]. ML is an ideal application of

[1]The time for fulfilling users' requests includes the processing time and the queuing time.

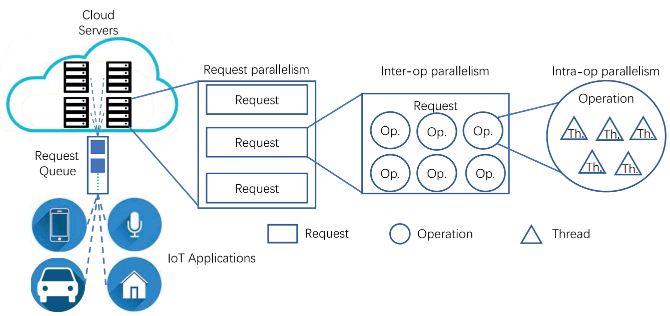


Fig. 1. Flow graph of how requests from IoT applications are handled in MLaaS.

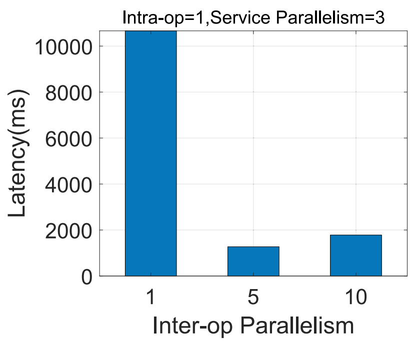


Fig. 2. Tensorflow Serving performance under different parallelism configurations for the Inception V3 model running on CPU. Appropriate parallelism improves system performance yet excessive parallelism decreases it because of interference. This observation is consistent with the previous study [8]. Experimental setup is detailed in Section VI-A.

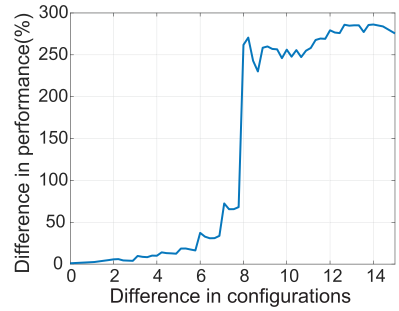


Fig. 3. Difference in performance versus difference in configuration. The difference in configurations is calculated by their Euclidean distance.

parallelization because most underlying operations in these models are vector–matrix multiplications or matrix–matrix multiplications [11]. Parallelization usually have two levels [7] for modern ML systems on CPU-based infrastructure. Upon arriving at the system, multiple requests can be served in parallel, which is noted as *request parallelism*. Each request can usually be decomposed into many operations. Further parallelization happens at the operation level, including *inter-op parallelism* where multiple operations executing simultaneously and *intra-op parallelism* where multiple threads working on each operation. Fig. 1 illustrates these three parallel implementations. Distinct parallelization mechanism can be found on hardware-accelerator-based infrastructure. For example, GPU has the built-in parallelism, such as thread blocks and scheduling partitions that are controlled by its own hardware schedulers. These low-level parallelisms are difficult to control directly through software-based approaches [12]. Fortunately, even for GPU infrastructure, we can still indirectly impact the parallelization by a few user-defined parameters. All of the parallel implementations and related configurations become control knobs in the ML-serving system. System performance is significantly determined by parallelism configurations. As indicated in Fig. 2, system performance can be boosted up to ten times by a well-tuned parallelism configuration compared to sequential execution (e.g., running Inception V3 on CPU infrastructure).

To provide low-latency ML serving, we propose a swift ML-serving scheduling framework for IoT applications. The proposed framework is driven by a lightweight region-based reinforcement learning (RRL) [1] approach that can efficiently identify optimal parallelisms configurations under heterogeneous IoT environments. The key insight is that the system performances under different similar configurations in a region can be accurately estimated by using the system performance under one of these configurations, due to their similarity (see Fig. 3). This key finding motivates us to develop RRL that can speed up the learning process by orders of magnitude faster than the state-of-the-art deep reinforcement learning methods with very limited training data. Theoretical analysis shows that the speedup increases with the size of the region; however, our initial results [1] show a performance gap between the RRL and the optimal solution due to the estimation error. To reduce such a performance gap, we propose an adaptive algorithm, namely, RRL Plus to adaptively adjust the region size to achieve fast learning speed as well as near-optimal performance.

We prototype the proposed framework on top of the popular Tensorflow Serving [13] ML-serving system and support both CPU and GPU-based hardware infrastructure. We release the source code for public access.[2] Extensive experimental evaluations on both CPU and GPU clusters show that by continuously learning the new traffic patterns and updating the scheduling policies, RRL Plus can quickly adapt to the ever-changing dynamics of IoT workloads and system environments. Compared to the state-of-the-art approaches (e.g., DeepRM [14] and CAPES [15]), RRL Plus can reduce the average latency up to 88.9% on CPU-based infrastructure and up to 71.5% on GPU-based infrastructure. In the SLO-aware scenario, RRL Plus can offer SLO guarantee under strict targets and provide up to 89.3% SLO violation reduction compared to CAPES and up to 91.6% compared to DeepRM. In addition, the proposed framework does not make assumptions on workload or ML applications and thus is applicable to most modern IoT applications.

II. Challenges and Observations

ML in IoT applications is often interactive and latency sensitive [16] in contrast to model training or other cloud applications which are usually throughput oriented (i.e., SLO-agnostic). Compared with traditional services (e.g., Web

[2]https://github.com/SC-RRL/RRL

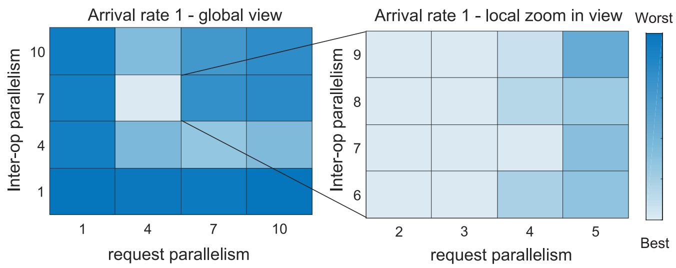


Fig. 4. Latency under the arrival rate of 14 requests per second on CPU with different parallel configurations (intra-op parallelism is set to ten) using Inception V3 deployed in Tensorflow Serving. The lighter the color, the lower the latency. The left plot shows a global performance view of configurations and the right plot is the zoomed-in view of the performance in a small region of configurations. The coarse-grained plot shows the latency is quite versatile globally while the zoomed-in fine-grained plot shows the latency is smooth locally (i.e., the neighboring points in the heatmap).

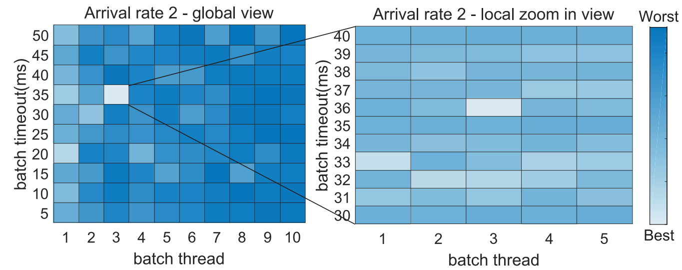


Fig. 5. Latency under the arrival rate of 61 requests per second on GPU with different parallel configurations (batch size is set to 50) using the Inception V3 model deployed in Tensorflow Serving. The lighter the color, the lower the latency. Left plot shows a global performance view of configurations and the right plot is the zoomed-in view of the performance in a small region of configurations.

service), ML service usually involves hundreds to thousands of operations together with complex correlation among them [17], which makes it challenging to model or to breakdown and fine-tune at operation level. How to optimally control these knobs is an important yet challenging problem as the overall performance depends on the performance requirement, workload characteristic, and available computing resources.

Many recent works focus on parallelism configuration tuning [18], [19]. However, existing methods rely on domain specific information and techniques to tune the parallelism configuration (see the detailed discussion in Section VII), which may not be applicable to many ML applications. Notably, Yan *et al.* proposed SERF in [8] and [20] using an analytical queuing model to achieve optimal parallelism configuration for ML serving, which works on exponential arrival process and homogeneous request size in certain image classification applications. Unfortunately, the arrival process may not be exponential for many other applications (such as video, speech, and natural language processing) and their request sizes can be heterogeneous. In addition, SERF supports only request level parallelism and CPU-based hardware. Therefore, there is a pressing need for a novel approach that can support two levels of parallelisms and hardware accelerators like GPU to effectively and efficiently tune parallelism configuration for ML applications with diverse arrival processes and heterogeneous request sizes.

There are many challenges for tuning parallelism configuration in modern ML-serving systems. For CPU-based infrastructure, the multilevel parallelism results in a relatively large configuration space. Fig. 4 illustrates request latency under only two parallelism configurations with fixed intra-op parallelism on a machine with only ten cores. The number of configurations will be magnitude larger with more parallelism parameters or complicated hardware environments, making it challenging for algorithms to locate the ideal one. Fig. 5 shows the similar observation on GPU-based infrastructure. As the configuration parameters have a wider range on GPU, the search space is even larger (e.g., batch timeout alone can have hundreds to thousands of possible choices). In addition, the indirect impact of configuration parameters in the GPU case makes it even harder to model or predict the behaviors. Even for the optimal parallelism configuration, it is also very sensitive to the load. When load experiences a slight change, the latency distribution which composes of both service time and queuing waiting time under different parallelism configurations becomes quite different. Such sensitivity significantly increases the search space and prohibits the exhaustive search. Among parallel computations, there are also complex interference behavior [8], [20] as a result of the high computation and memory needs of ML models, which leads to nonlinear performance behavior of different configurations. All these together brings significant challenges for profiling and analytical modeling approaches [21].

Another challenge that could result in the state-of-the-art modeling techniques [8], [20] ineffective is the tens of thousands of operations with complex dependencies among them in modern ML models. Moreover, the workload and system environment in many IoT applications are often highly dynamic [22]–[24], which requires the scheduling policy with an agile adaptive ability, in order to meet the sensitive latency SLO [25] of IoT applications. In this case, traditional learning-based methods [21], requiring a large training set and a long convergence time, can hardly be applicable. Therefore, it is of paramount importance to provide ML serving with swift deployment that can learn the dynamics of the IoT workload and system environment and optimize the performance in an online manner.

III. RRL-Based Scheduling Framework

In this section, we present the RRL-based scheduling framework for ML service in IoT. The RRL-based scheduling framework is designed to dynamically adjust the parallelism configuration of ML-serving systems according to dynamic system load, in order to optimize ML performance in IoT (e.g., response latency and resource consumption). It is challenging to model the relationship among the system performance, parallelism configurations, and system load in a closed form. As illustrated in Fig. 4, system performance varies under different parallelism configurations even for the same load. To tackle this challenge, a learning approach is used in the proposed framework to find the optimal parallelism configuration. Specifically, the proposed framework consists of three main components: 1) profiler; 2) scheduler; and 3) RRL, as

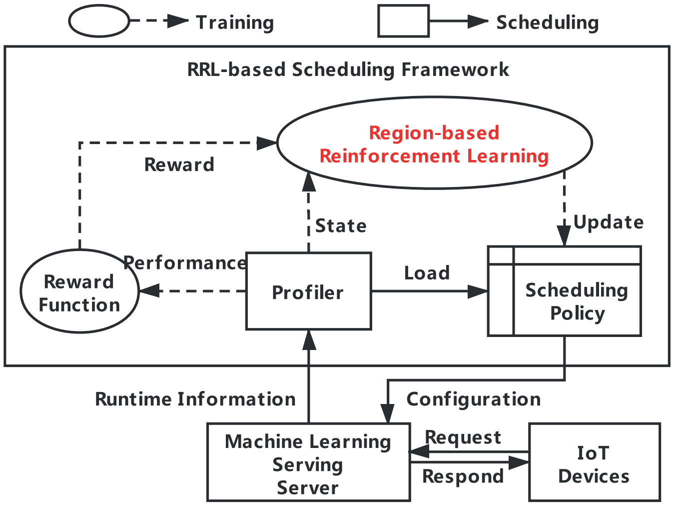


Fig. 6. Overview of the RRL-based scheduling framework for IoT applications.

illustrated in Fig. 6. Various system characteristics are collected by the profiler, including the current user traffic load and the corresponding system performance under this load and the present parallelism configuration. The scheduler then adjusts the parallelism configuration for the measured load level based on the current scheduling policy. Meanwhile, the RRL asynchronously updates the scheduling policy to adapt to the system dynamics based on the measured system load and corresponding performance.

Profiler: The profiler measures the system performance, including the system load (i.e., request arrival rate) and the latency (also known as response time) of each request. The profiler also collects hardware-related information (e.g., CPU core number, CPU utilization, available GPUs, GPU utilization, and network statistics). All the information can be used in reward functions to optimize the system performance for various scheduling objectives.

Scheduler: The scheduler takes into consideration the current system load, scheduling policies, and hardware information such as the availability of resource and adjusts the parallelism configuration accordingly.

RRL: As the core of the proposed framework, the RRL component aims to find the optimal scheduling policy and quickly adjusts the scheduling policy to adapt to the system dynamics. Specifically, the system performance measured by the profiler will be passed to the reward function in Fig. 6 to calculate the value of the system objective function, and then the learning component learns the scheduling policy based on this observed reward. Traditionally, the scheduling policy is incrementally improved in a point-by-point learning manner that makes the learning process significantly long. To address this challenge, the proposed RRL can speedup this learning process by leveraging the key feature of the system as illustrated in Fig. 3 that the system performances under different similar configurations are similar. Based on this feature, the system performance under one configuration can be used to estimate the system performances under other similar configurations, which would significantly reduce the number of samples needed to learn the optimal scheduling policy. For example, if we choose the radius of the configuration region is equal to 2, then we can use a single observation to update all configurations in this region and obtain a roughly ten times faster convergence with limited performance loss due to the estimation error. The detailed design is presented in Section IV.

IV. Region-Based Reinforcement Learning

In this section, we propose an RRL approach, in order to speed up the learning process of the scheduling policy to meet the requirements of IoT applications. Specifically, we first formulate the ML-serving scheduling as a Markov decision process (MDP), and then theoretically show that the RRL approach can achieve a near-optimal solution with fast convergence speed.

A. ML-Serving Scheduling: MDP View

The objectives of the ML-serving scheduling in IoT are 1) to minimize response latency using a given amount of resources [8] or 2) to minimize resource consumption while meeting latency SLO [26]. Our scheduling framework supports both objectives. We focus on the first objective of minimizing response latency due to the space limitation.

Define system state as $s \in \mathcal{S}$, where s denotes the overall load level and $\mathcal{S}$ denotes the set of possible load levels. System action is defined as the parallelism configuration $c \in \mathcal{C}$ which is a tuple of request parallelism c^{service}, inter-op parallelism c^{inter}, and intra-op parallelism c^{intra}, i.e., $c = (c^{\text{service}}, c^{\text{inter}}, c^{\text{intra}})$, where $\mathcal{C}$ denotes the set of possible parallelism configurations. For ML serving in IoT, it is challenging to characterize latency in a closed form as it can vary under different loads (system states) for the same parallelism configuration [8]. Instead we use the average request latency $r(s, c)$ under the system state s and the parallelism configuration c as reward. In this article, we assume that the scheduler does not have *a priori* knowledge of system state transitions, except the Markov property (i.e., the state transition depends on only the previous state).[3] Under this model, the ML-serving scheduling is cast as an MDP, aiming to minimize the expected cumulative discounted latency $\mathbb{E}[\sum_{t=0}^{\infty} \gamma^t r_t(s_t, c_t)]$, where $\gamma \in (0, 1]$ is a discount factor and $r_t(s_t, c_t)$ denotes the latency observed at time t under system state s_t and parallelism configuration c_t.

At each time t, the scheduler chooses a parallelism configuration based on a policy, defined as $\pi : \pi(s, c) \to [0, 1]$, where $\pi(s, c)$ is the probability that configuration c is used in state s. The Q-learning method can be applied to find the optimal policy yet its convergence is slow, especially, when the space of state–configuration pairs is large. One key reason for this slow convergence is that it searches the space point by point and incrementally improves the policy. To improve the convergence speed, many approaches [28], [29] have been proposed but they are still *point-based* learning essentially and would

[3]Markov models are often used to model the workload dynamics, e.g., [27] verifies the Markov property for different applications. In our application, the Markov property is also satisfied. The experiments in Section VI also corroborate the correctness of the Markov model in our application.

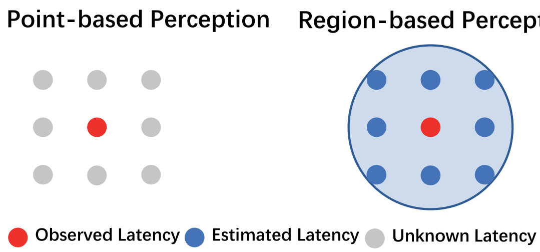


Fig. 7. Point-based versus region-based learning. The RRL approach can more efficiently learn the latency under different configurations.

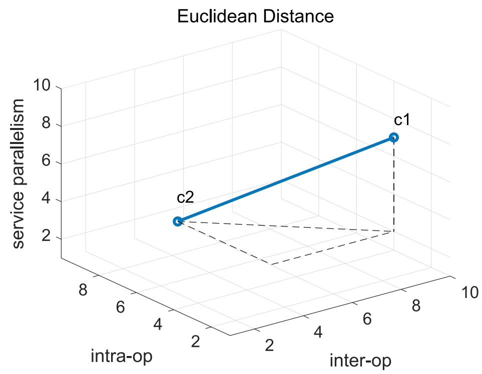


Fig. 8. Example of the Euclidean distance between two configurations c_1 and c_2, i.e., $\sqrt{(c_1^{\text{service}} - c_2^{\text{service}})^2 + (c_1^{\text{inter}} - c_2^{\text{inter}})^2 + (c_1^{\text{intra}} - c_2^{\text{intra}})^2}$.

not be applicable to our problem with large state-configuration space as shown in our experiments in Section VI.

B. RRL: From Point-Based to Region-Based Learning

To speed up the learning process, we propose the RRL approach. The key idea is that when observing the latency $r(s, c)$, we will estimate the latency in a region with configurations close to c under this state s, and then use the estimated latency in this region to learn the policy, as illustrated in Fig. 7. Intuitively, the learning speed would be significantly improved if this region-based learning approach uses a large region. However, the potential estimation errors of the latency associated with the region may make the converged policy deviate from the optimal one. In other words, the larger the region is, the larger the potential errors might be, which indicates a tradeoff between the learning speed and the optimality of the policy depending on the size of the region and the latency estimation scheme. When the region degenerates to a single point, the RRL approach would degenerate to the traditional reinforcement learning approaches. In this article, the Euclidean distance is used to measure the distance between two configurations since both CPU and GPU configurations are numeric (see Fig. 8). Note that other similarity measures can also be applied in RRL.

Specifically, the RRL approach consists of two main components: 1) latency-estimation-based perception and 2) policy update.

1) Latency-Estimation-Based Perception: Let $Q_t(s_t, c_t)$ denote the perception of the expected cumulative discounted latency under state s_t and configuration c_t. Define the region

Algorithm 1 RRL

Initialization: Choose β, δ, and γ. Set $t = 0$ and $Q_0(s, c) = 1/|\mathcal{C}|\ \forall c \in \mathcal{C}, s \in \mathcal{S}$.

For each time slot t

1) Choose a configuration based on the current policy π_t.
2) Update the perception based on Eq. (2).
3) Update the policy for the current state s_t using Eq. (3).

around c_t as $\mathcal{C}(c_t) = \{c|||c - c_t|| \leq \delta\ \forall c \in \mathcal{C}\}$, where $\delta \geq 0$ denotes the size of the region. Using the observed latency $r_t(s_t, c_t)$, the latency under other configurations in $\mathcal{C}(c_t)$ can be estimated as

$$\hat{r}_t(s_t, c) = f(c, r_t(s_t, c_t)) \quad \forall c \in \mathcal{C}(c_t) \tag{1}$$

where $f : \mathcal{C} \times \mathbb{R}^+ \to \mathbb{R}^+$ is the latency estimation function and $f(c_t, r_t(s_t, c_t)) = r_t(s_t, c_t)$. Based on (1), we update the perception of the expected cumulative discounted latency in the region by

$$\forall c \in \mathcal{C}(c_t),\ Q_{t+1}(s_t, c) = (1 - \alpha_t)Q_t(s_t, c) + \alpha_t\left(\hat{r}_t(s_t, c) + \gamma \min_{\tilde{c}\in\mathcal{C}} Q_t(s_{t+1}, \tilde{c})\right) \tag{2}$$

where $\alpha_t \in [0, 1]$ is the learning rate. As is standard, the learning rate is assumed to satisfy $\sum_{t=1}^{\infty} \alpha_t = \infty$ and $\sum_{t=1}^{\infty} \alpha_t^2 < \infty$. The perceptions of other configurations ($c \notin \mathcal{C}(c_t)$) will remain the same, i.e., $Q_{t+1}(s_t, c) = Q_t(s_t, c)\ \forall c \notin \mathcal{C}(c_t)$.

2) Policy Update: Based on the perceptions, we can use the Boltzmann distribution [30] to update the policy for state s_t

$$\pi_t(s_t, c) = \frac{\exp(-\beta Q_t(s_t, c))}{\sum_{\hat{c}\in\mathcal{C}} \exp\left(-\beta Q_t(s_t, \hat{c})\right)} \quad \forall c \in \mathcal{C} \tag{3}$$

where $\beta \geq 0$ controls the exploration–exploitation tradeoff. When β is very small, the scheduler would explore the space randomly; when β is large, the scheduler would tend to exploit the configuration with the lowest perceived latency.

It is worth noting that the accuracy of the latency estimation (1) directly impacts the performance of the RRL approach. Due to the stochastic nature of the state and the latency, it is challenging to characterize f in a closed form in practice. To tackle this challenge, we use the neural network to implement this estimation function as discussed in Section V. The detailed description of the RRL approach is given in Algorithm 1.

C. Performance Analysis of RRL

In this section, we will analyze the convergence rate and optimality performance of the RRL approach. To facilitate the analysis, we assume that the estimation error of the latency estimation (1) is upper bounded by $\Delta \geq 0$ for all state-configuration pairs in the space, i.e.,

$$\left|\hat{r}_t(s_t, c) - r_t(s_t, c)\right| \leq \Delta \quad \forall c \in \mathcal{C}(c_t),\ s_t \in \mathcal{S} \tag{4}$$

where $r_t(s_t, c)$ denotes the real latency that can be observed if the configuration c is chosen. In (4), Δ is intimately related to the size of the region δ. In general, Δ increases with δ, and Δ becomes zero when δ is zero.[4] The main results are summarized in the following theorem.

Theorem 1: The RRL approach can asymptotically converge to a near-optimal solution with probability one as t goes to infinity. The performance gap is upper bounded by $(\Delta/1-\gamma)$. The asymptotic convergence rate is $O(1/(n_\delta t)^{R(1-\gamma)})$ if $R(1-\gamma) < 1/2$ and $O(\sqrt{\log\log(n_\delta t)/(n_\delta t)})$ otherwise, where n_δ denotes the number of state-configuration pairs in the region with size δ and R denotes the ratio of the minimum and maximum state-configuration selection probabilities.

Proof: Proof can be found in [1]. ■

Remarks: Theorem 1 confirms our intuition that the RRL approach can accelerate the convergence speed of the reinforcement learning such that the larger n_δ (i.e., the larger δ), the faster the RRL converges. However, the fast convergence speed is at the cost of performance loss, i.e., there would be a gap $[\Delta/(1-\gamma)]$ between the RRL and the optimal solution. When $\delta = 0$, we have $n_\delta = 1$ and $\Delta = 0$, and the results of Theorem 1 degenerate to the results for the traditional point-based reinforcement learning [31]. Thanks to the unique structure of our problem (see Fig. 3), we use the Bayesian optimization to choose a suitable size of the region with fast learning speed as well as near-optimal performance (see Section IV-D).

D. Adaptive RRL

From the analysis of RRL, it is shown that excessive region size can lead to a large performance gap, whereas small region size leads to a low convergence rate. In order to find a suitable region size, we propose a Bayesian-based optimization approach to automatically adjust region size to achieve fast learning speed as well as near-optimal performance.

Specifically, we introduce an acquisition function to characterize the expected latency improvement under a give region size as our optimization target

$$\alpha(\delta) = \mathbb{E}\Big[\big(\bar{r}(\delta^*) - r(\delta)\big)^+\Big] \tag{5}$$

where $\bar{r}(\delta^*)$ is the best observed average latency and δ^* is the corresponding region size. $r(\delta)$ denotes the latency random variable following the Gaussian distribution $\mathcal{G} \sim \mathcal{N}(m(\delta), \sigma(\delta, \delta'))$ with mean $m(\delta)$ and covariance $\sigma(\delta, \delta')$. In each iteration, we choose the region size δ that maximizes the acquisition function α and uses this δ for RRL perception. Then, the observed average latency $\bar{r}(\delta)$ will be added into the sample set, and the mean $m(\delta)$ and covariance $\sigma(\delta, \delta')$ of $\mathcal{G}$ will be updated based on the Bayesian optimization [32]. The idea is to model the unknown function between the region size and the latency as a multivariate Gaussian distribution, and then use a computational cheap acquisition function to guide the search for the optimal point. Thus, we can reduce the latency by adaptively adjusting region size. The details are given in Algorithm 2.

[4]Note that Δ also highly depends on the accuracy of the estimation function. In this article, a neural-network-based estimation function is implemented, and the error bound is shown to be small in our experiments.

Algorithm 2 RRL Plus

Initialization: Initialize sample set D.

For each time slot t

1) Calculate the region size δ by maximizing α, i.e., $\delta = \underset{\delta}{argmax}\ \alpha(\delta)$.

2) Update the perception with region size δ using Eq. (2).

3) Update the policy for the current state s_t using Eq. (3).

4) Get the current average latency $\bar{r}_t(\delta)$, and update the sample set $D = \{D, (\delta, \bar{r}_t(\delta))\}$ and the parameters $m(\delta)$ and $\sigma(\delta, \delta')$ using D.

V. Implementation

In this section, we discuss the implementation of the proposed approach. Specifically, we focus on the neural-network-based estimation function design as the Tensorflow Serving integration of the proposed framework has been described in our previous work [1].

A. Neural-Network-Based Estimation Function

It is challenging to characterize the estimation function in a closed form as discussed in Section IV. Since neural-network-based approaches have shown great potentials in many applications [5], we propose a neural-network-based estimation function in this article. To support swift ML-serving scheduling, one key challenge is how to find a suitable neural network structure for the estimation function (1). Simple network structure may not effectively capture the structure of the underlying state-configuration space, which may lead to high estimation error [33]; complicated network structure may take a long training time, which is not suitable for online serving systems.

As indicated in the previous study [33], we need to strike a balance between complexity and efficiency. Our network design contains two hidden layers (one with 256 neurons and the other with 64 neurons) using ReLu [34] as activation function and one output layer with linear activation, after experimenting different network structures. Follow-the-regularized-leader (FTRL) [35] optimizer is used to optimize network parameters instead of the Adam method or other popular optimizers. This is because the number of training samples in our problem is far less than the number of state-configuration pairs in the space during online tuning, and thus FTRL performs well here. Moreover, FTRL is insensitive to model parameters. Our experiments in Tensorflow Serving show that FTRL performs well even where there is limited training data (see Section VI).

B. Tensorflow Serving Integration

The proposed scheduling framework is integrated into Tensorflow Serving [13], a popular production-ready ML-serving system. While we do a case study with Tensorflow Serving, we do not rely on any Tensorflow-specific features and nothing prevents the proposed work being integrated into other ML-serving systems. All the implementation details can be found in [1].

VI. Experimental Evaluation

In this section, we conduct extensive experimental evaluation simulating heterogeneous IoT environments and workloads to corroborate the effectiveness and robustness of the proposed RRL-based scheduling framework using a rich selection of the state-of-the-art ML applications on both CPU and GPU-based infrastructure. We first evaluate the sensitivity of RRL in convergence speed by adjusting the region size, and compare RRL Plus and RRL in terms of the convergence process. Then, we compare RRL Plus with the latest reinforcement learning approaches for the following four key features in IoT applications: 1) minimizing latency for image classification; 2) minimizing latency for speech recognition; 3) satisfying strict SLO guarantee; and 4) effectiveness of meeting SLO while minimizing resource usage.

A. Experimental Setup

ML-Serving System: We prototype the RRL-based scheduling framework and integrate it in Tensorflow Serving, refer to [1] for more details.

Service Workloads: We use three ML models commonly used in IoT applications for evaluation: 1) image classification models Inception V3 [36]; 2) Inception ResNet V2 [37]; and 3) speech recognition model Deep Speech V2 [3]. They cover popular ML tasks in IoT applications, such as smart home, smart city, and autonomous driving.

Arrival Process: We use two nonexponential arrival processes simulating IoT workloads for evaluation.

1) *WiKi:* An arrival process based on traces of user traffic visiting Wikipedia website [38] with unpredictable load spikes to simulate the request patterns in IoT applications.
2) *Dynamic:* A synthetic dynamic arrival process composed of periods of the Poisson process with a randomly changing average, which has pronounced changes from one period to the next.

Hardware: We use a cluster of ten identical servers. Each of them is equipped with dual-sockets Intel Xeon CPU E5-2630 v4 @ 2.20 GHz with hyperthreading disabled and four NVIDIA GeForce GTX 1080 Ti GPUs, 64 GB of memory, and connected through Infiniband.

Baseline Approaches: Since there is no alternative intelligent scheduling framework for a direct comparison, we opt to implement the state-of-the-art reinforcement learning approaches for tuning parallelism configuration in our scheduling framework: CAPES [15] and DeepRM [14], as they are the closest approaches for the online ML-serving scheduling. DeepRM is a job scheduling algorithm designed to work under limited resources and CAPES is a general-purpose parameter tuning algorithm.

SLO Setting: As our testbed is not production level, we set relatively loose SLOs in our evaluation, i.e., a range between 400 and 2500 ms to emulate different latency requirements for ML serving in production, which is consistent with previous studies [8], [20], [25].

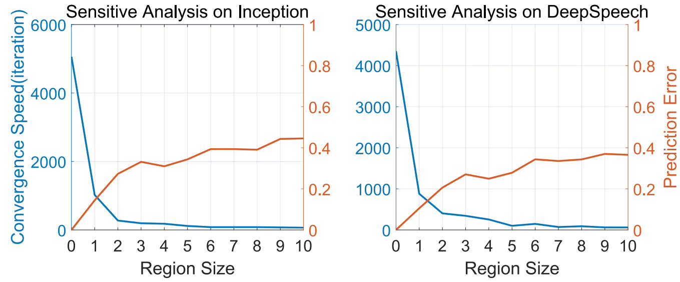


Fig. 9. Sensitivity analysis of RRL in terms of the convergence time in iteration (left y-axis) and the prediction error (right y-axis) as a function of region size using Inception and DeepSpeech.

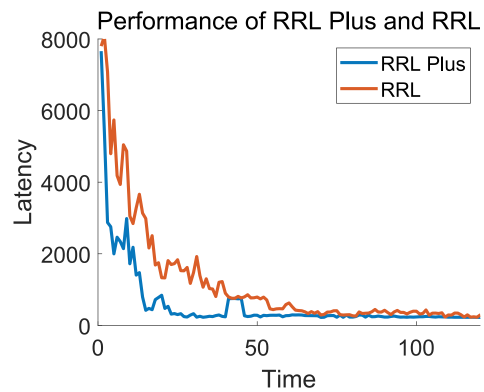


Fig. 10. Performance comparisons of RRL Plus with adaptive region size and RRL with fixed region size on Inception. RRL Plus has shorter learning process and lower latency.

B. Convergence Speed Analysis of RRL and RRL Plus

The key tuning parameter in RRL is the region size as it controls the tradeoff between convergence speed and optimality. We validate the theoretical results in Theorem 1 by sensitivity analysis of RRL using Inception, as illustrated in Fig. 9. The results show the convergence time measured in iteration (left y-axis) and distance from optimal Q-learning function (right y-axis) as a function of the region size. It is clear that convergence time drops very quickly when the region size increases while the prediction error increases in a much slower speed. For example, when the region size is one, RRL converges five times faster than Q-learning, which verifies the potential of the region-based methodology. When region size is zero, RRL degenerates to point-based learning, which has the same accuracy and the longest convergence time as Q-learning. We use RRL Plus to control the balance of performance and convergence time.

We evaluate the effectiveness of our adaptive algorithm RRL Plus by comparing it to the RRL with fixed region size. Fig. 10 shows the convergence process between RRL Plus and RRL. It can be inferred that during the converging process, RRL Plus performs better and has a shorter learning process. On average, RRL Plus has 17.54% less converge time and 15.31% less latency.

C. Minimizing Serving Latency for Image Classification

In this section, we evaluate the famous image classification model Inception on RRL Plus and the two baseline

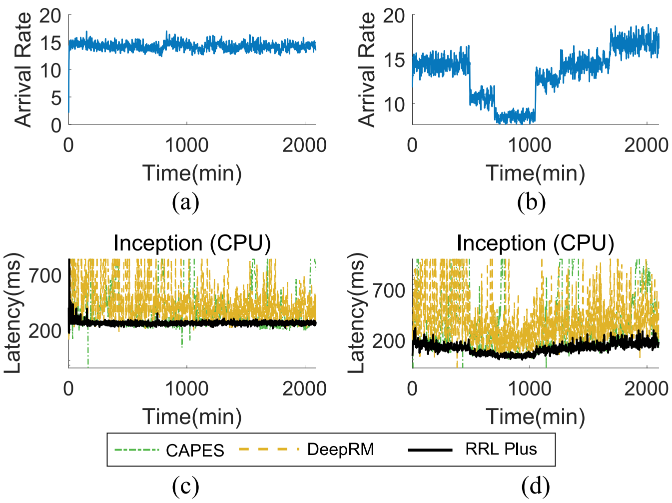


Fig. 11. Comparisons of RRL Plus with CAPES and DeepRM under different arrival processes and service workloads. (a) Arrival process (WiKi). (b) Arrival process (dynamic). (c) Latency (WiKi). (d) Latency (dynamic). (a) and (c) First scheduling objective of minimizing latency using WiKi as arrival process for Inception. (b) and (d) Scheduling objective of minimizing latency but under dynamic arrival process for Inception.

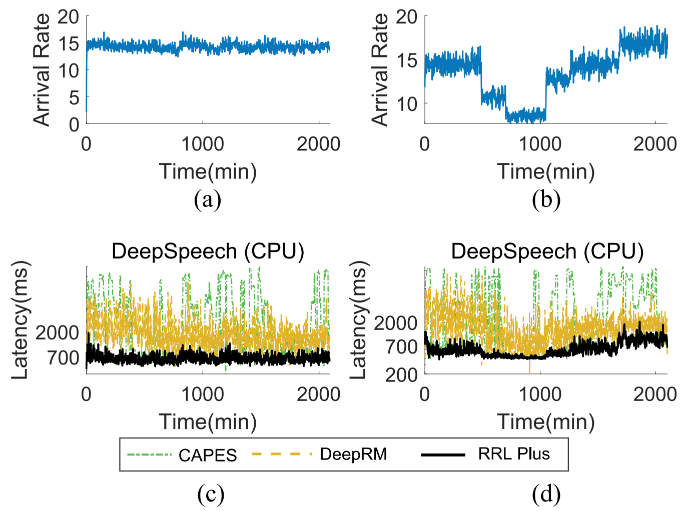


Fig. 12. Comparisons of RRL Plus with CAPES and DeepRM under different arrival processes and service workloads. (a) Arrival process (WiKi). (b) Arrival process (dynamic). (c) Latency (WiKi). (d) Latency (dynamic). (a) and (c) First scheduling objective of minimizing latency using WiKi as arrival process for DeepSpeech on CPU. (b) and (d) Scheduling objective of minimizing latency but under dynamic arrival process for DeepSpeech on CPU.

deep reinforcement learning approaches: 1) DeepRM [14] and 2) CAPES [15] to compare their effectiveness of minimizing serving latency on both CPU- and GPU-based infrastructure under WiKi and Dynamic arrival processes. This evaluation aims to test the algorithms' ability to keep low response latency under perturbation which is common in IoT applications.

WiKi Arrival Process: We show latency results of Inception running on CPU cluster in Fig. 11(c) using WiKi trace to drive the arrival process, which is demonstrated in Fig. 11(a). The results verify that RRL Plus converges much faster than the baseline approaches, i.e., RRL Plus converges to a near-optimal performance in about 150 min, while DeepRM roughly converges around 1400 min with variance and CAPES could not converge even after 2000 min. The results also show that RRL Plus is able to achieve better latency performance compared to deep-reinforcement-learning-based approaches, thanks to the swift learning capabilities. More specifically, the average latency of RRL Plus improves from CAPES and DeepRM by 70.1% and 75.2%, respectively, for Inception.

Dynamic Arrival Process: Workload can change dynamically over time in practice. In this section, we evaluate the robustness of the proposed scheduling framework in terms of the ability to quickly adapt to the workload change. We use a synthetic dynamic arrival process for evaluation, as shown in Fig. 11(b), the arrival change is more pronounced than the WiKi arrival process, which emulates the change of user traffic patterns over time.

The latency results are shown in Fig. 11(d). The results suggest that RRL Plus can adapt to the user traffic change quickly with a limited number of samples thanks to the region-based learning approach, which leads to a much shorter adapting time and more stable latency performance compared to CAPES and DeepRM. In contrast, DeepRM takes a much longer time to update scheduling polices and CAPES shows significant variation due to its slow learning process. On average, RRL Plus reduces the latency of Inception by 71.0% and 59.9% compared to CAPES and DeepRM, respectively.

D. *Minimizing Serving Latency for Speech Recognition*

One key feature of speech recognition application is that its requests are heterogeneous since the user can say a long sentence as well as just a few words, which is challenging to scheduling system as the system has no *a priori* knowledge of the computation cost of requests. Thus, it requires the scheduling system the ability to handle requests with various lengths. The application we use is DeepSpeech V2, a reputable speech recognition model.

Wiki Arrival Process: The latency results on CPU cluster for DeepSpeech are shown in Fig. 12(c). Similar to the previous evaluation, RRL Plus reaches a near-optimal configuration within shorter adapting time compared to CAPES and DeepRM. RRL Plus has better performance than CAPES by 88.9% and DeepRM by 80.7% on average.

Dynamic Arrival Process: As is shown in Fig. 12(d), even under ever-changing arrival process and heterogeneous request, RRL Plus is still able to keep a stable and low response latency whereas DeepRM has slower converge rate and CAPES shows significant variation. On average, RRL Plus reduces the latency of DeepSpeech by 86.0% and 63.3% compared to CAPES and DeepRM, respectively.

E. *Minimizing Serving Latency on GPU Infrastructure*

As explained in earlier sections, the parallelism on GPU is controlled by the hardware scheduler and difficult to be adjusted through software approaches. Here, we control the parallelism using an indirect approach by tuning the batching parameters (parallel batch threads, batch size, and batch

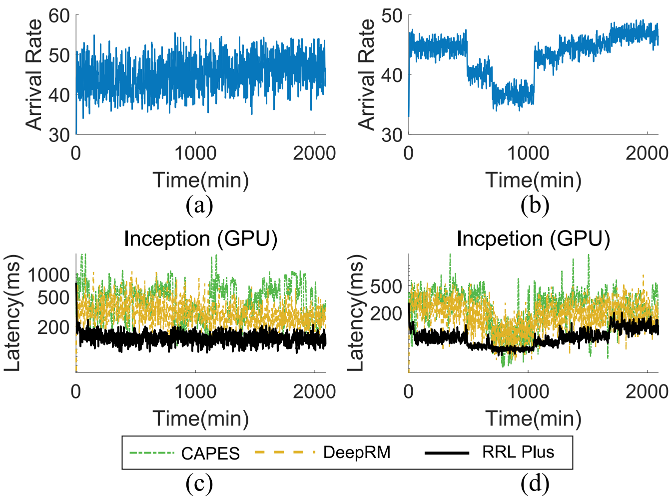


Fig. 13. Comparisons of RRL Plus with CAPES and DeepRM under different arrival processes and service workloads. (a) Arrival Process (WiKi). (b) Arrival process (dynamic). (c) Latency (WiKi). (d) Latency (dynamic). (a) and (c) First scheduling objective of minimizing latency using WiKi as arrival process for Inception on GPU. (b) and (d) Scheduling objective of minimizing latency but under dynamic arrival process for Inception on GPU.

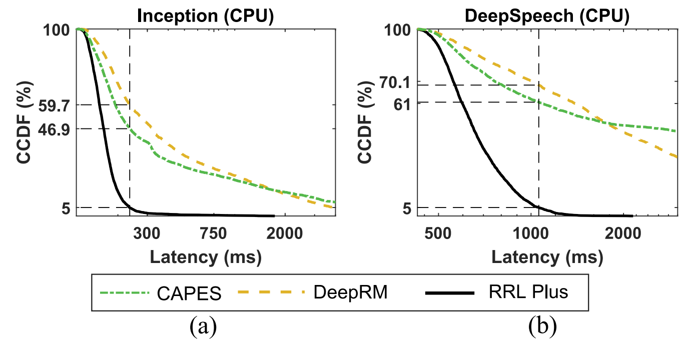


Fig. 14. Comparisons of RRL Plus with CAPES and DeepRM under strict SLO (95th percentile latency of 235 ms for Inception and 1060 ms for Deep Speech). (a) Latency (dynamic) for Inception (CPU). (b) Latency (dynamic) for DeepSpeech (CPU).

timeout). Similar as CPU case, we use scaled WiKi workload and CAPES and DeepRM as baselines and report the results in Fig. 13. Compared with CPU results, the variance in latency is higher on GPU which is caused by the indirect control mechanism. In spite of the challenge of high variance, RRL Plus still converges quickly and outperforms CAPES and DeepRM in latency. Specifically, RRL Plus performs 56.5% better than DeepRM and 68.1% than CAPES.

Fig. 13(d) shows the evaluation on GPU-based infrastructure under dynamic workload. The indirectly controlled parallelism on GPU leads to a slower adapt speed than CPU case. However, even in this challenging scenario, RRL Plus still consistently outperforms CAPES and DeepRM by 71.5% and 55.7% on average, respectively.

F. Meeting SLO With Minimum Resources

We evaluate our approach under the scenario of meeting strict SLO target, i.e., 95th percentile latency SLO of 235 ms for Inception and 1060 ms for DeepSpeech.[5] Fig. 14 (note that the x-axis is logscale) demonstrates the CCDF of RRL Plus latency on CPU cluster and GPU cluster compared with the baselines. The tail comparison indicates that RRL Plus has shorter tail latency and can provide a strict SLO guarantee. Compared with CAPES and DeepRM, RRL Plus achieves up to 89.3% and 91.6% SLO violation reduction, respectively, thanks to its SLO-aware design.

Our scheduling framework also supports another common scheduling objective in IoT applications which is meeting relatively loose SLO while minimizing the resource usage (e.g., cloud environment or shared cluster). The evaluations on CPU and GPU infrastructure of this scheduling objective using DeepSpeech, ResNet, and Inception under dynamic workload are shown in Fig. 15.

CPU Cluster: Fig. 15(b) shows the latency of DeepSpeech running on CPU cluster over the time using different scheduling methods, where both CAPES and DeepRM perform poorly on achieving the SLO target. DeepRM spent around 500 min before finding a scheduling policy that can achieve the SLO but at the expense of high CPU utilization whereas CAPES violates the SLO when the workload increases. RRL Plus in contrast always guarantees the SLO, even during abrupt workload changes. Another comparison is on resource utilization, which is critical for consolidating resources and achieve cost-efficient serving. We report the CPU utilization at Fig. 15(c), where RRL Plus consistently consumes much less CPU resource than both CAPES and DeepRM, which is especially important for commercial IoT applications with a rather large number of requests from all end devices. Similar observations hold for the ResNet results in Fig. 15(e) and (f), where all three methods achieve SLO in a short time, but RRL Plus uses only one-third CPU resources compared to the deep reinforcement-learning-based methods. On average, compare with CAPES and DeepRM, RRL Plus uses 32.0% and 35.0% less CPU resources, respectively, for DeepSpeech. For ResNet, the resource saving is even more significant: RRL Plus on average saved 61.8% compared to CAPES and 68.9% compared to DeepRM.

GPU Cluster: We show the GPU results in Fig. 15(h) and (i), where RRL Plus keeps a stable latency right under SLO and only uses half GPU resources compared with DeepRM. On average, RRL Plus saved 43.7% GPU resources compared with DeepRM. Compared with CAPES, RRL Plus uses the same level of GPU resources and achieves 38.8% latency reduction and 98.6% SLO violation reduction.

G. Discussion

Evaluation results show that RRL Plus outperforms RRL and other standard deep reinforcement learning methods in both speed and accuracy. RRL Plus uses the unique characteristics of ML serving to accelerate the learning process: when parallelism changes, the latency is quite versatile globally while smooth locally. Other methods do not have such insights. Compared with RRL, PPL Plus automatically sets

[5]It is worth to emphasize again that the relatively high latency is because our testbed is not enterprise scale nor equipped with the latest hardware, so both the processing time and the queuing waiting time is relatively high.

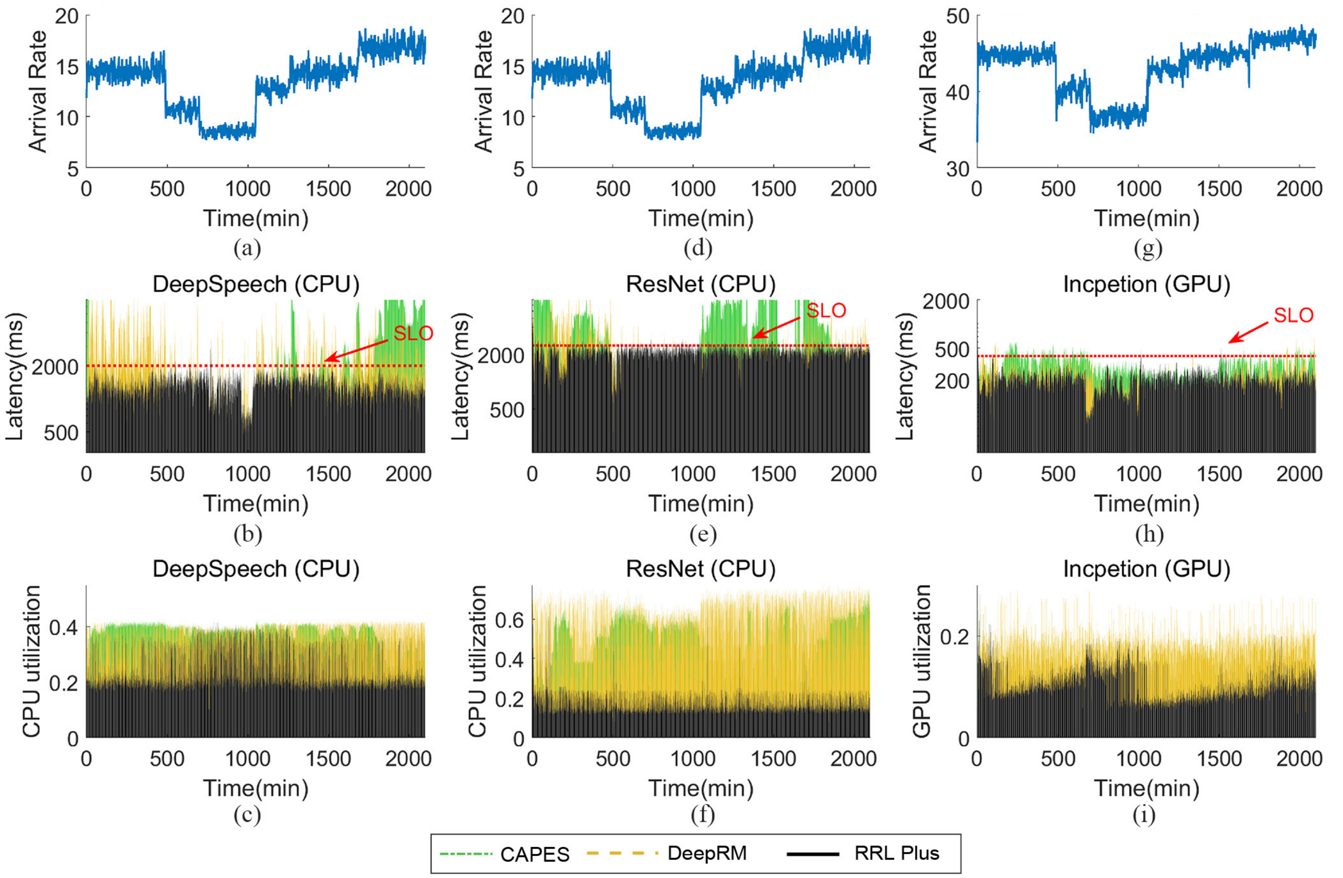


Fig. 15. Comparisons of RRL plus with CAPES and DeepRM when achieving SLO while optimizing resource usage (i.e., CPU and GPU utilization) under dynamic arrival processes and service workloads. (a), (d), and (g) Arrival process (dynamic). (b), (e), and (h) Latency (dynamic). (c) and (f) show CPU Util. (dynamic). (i) GPU Util. (dynamic). (a)–(f) Scheduling objective of minimizing CPU utilization with respect to given SLOs for model DeepSpeech and ResNet under dynamic workloads. (g)–(i) Second scheduling objective of achieving SLO while minimizing GPU usage with Inception under dynamic arrival process. The SLOs for DeepSpeech, ResNet, and Inception are 2000, 2500, and 400 ms, respectively.

the region size during optimization, which leads to less convergence time. When environment/workload changes, RRL Plus may have already converged to a near-optimal solution, whereas other methods may be still far away. Therefore, in online systems, RRL Plus outperforms the standard deep reinforcement learning methods in both speed and accuracy.

VII. Related Work

A. Machine Learning in IoT Applications

IoT applications have brought the number of end devices and the information they collected to a magnitude higher [39], [40]. To manage, process, and utilize such large amount of data, ML has been applied to many IoT scenarios.

Mohammadi *et al.* [41] applied a semisupervised reinforcement learning algorithm to smart city scenario that improves the accuracy of indoor localization. Cao *et al.* [42] combined SVM with belief network to optimize wireless network capacity. Chen *et al.* [43] used extreme learning machine to recognize human activity from data collected by smart health sensors. Liang *et al.* [44] detected soil moisture by applying neural network to sensor data.

However, a key challenge of combining ML with IoT applications is that IoT devices are usually low energy and embedded whereas ML models need considerable memory and computational power to run. Reagen *et al.* [45] proposed a design of hardware accelerator to accommodate DNN in IoT devices. Dhurandhar *et al.* [46] developed a method to compress RNN models to reduce resource usage. Even with the aid of these approaches, it is often infeasible to deploy ML to end devices, thus using ML as cloud service is a popular choice for many IoT applications.

B. Machine Learning Serving

How to efficiently deploy trained ML models in serving (or sometimes called inference) mode to provide low-latency services has drawn great attention in both academia and industry [8], [20], [47]. Several ML-serving systems have been open-sourced recently [13], [47], [48]. Hardware acceleration [49] has been used to accelerate the computation in ML serving. Software techniques using model compression and simplification [50], compiler techniques [51] and acceleration library [52] have also successfully reduced model computation time.

Another promising technique for reducing the latency of ML serving is parallelism [11]. *Request parallelism*, *inter-op parallelism*, and *intra-op parallelism* are the typical ways to parallel computation on CPU in today's ML-serving systems. On GPU, computation is parallelized through SMs and scheduling partitions which can be indirectly adjusted through batching parameters, such as *batch size*, *batch threads*, and *batch timeout*. As discussed in the introduction, existing methods [8], [18]–[20] either require domain-specific information to tune the parallelism configuration or are applicable for a special arrival process with homogeneous request size in certain applications. To achieve a more general solution, we design a scheduling framework that can work with general user traffic patterns and system environments on both CPUs and GPUs-based infrastructure.

C. Parameter Tuning Using Reinforcement Learning

During 1940s, reinforcement learning [53] was first proposed and has been widely used in different applications. Here, we focus on the works that apply reinforcement learning to system parameter tuning. Mao *et al.* [14] proposed a reinforcement-learning-based resource management method for multiresource cluster scheduling problem. Li *et al.* [15] developed a reinforcement-learning-based parameter tuning system for storage systems. Both works use traditional point-based reinforcement learning and suffer from slow convergence and adaptivity. Mirhoseini *et al.* [54] proposed to optimize Tensorflow operation placement between CPU and GPU using long short-term memory (LSTM), which is applicable for only CPU–GPU co-design architecture. In our previous work [1], we present the initial results of performance tuning using the RRL approach. However, the region size is hand-tuned and fixed throughout the optimization process, which leads to a performance gap between the RRL solution and the optimal one. In this article, we develop an enhanced RRL-based framework using the Bayesian optimization to dynamically update the region size, in order to improve the convergence speed and the agility in a dynamic environment.

VIII. Conclusion

In this article, we proposed an RRL-based scheduling framework for ML serving in IoT applications that can efficiently identify optimal configurations under dynamic workloads. A key observation is that the system performance under similar configurations in a region can be accurately estimated by using the system performance under one of these configurations due to their correlation. We theoretically showed that the RRL approach can achieve fast convergence speed at the cost of performance loss. To reduce the performance loss, we proposed an adaptive RRL algorithm, namely, RRL Plus, to balance the convergence speed and the optimality. The proposed framework was prototyped and evaluated on the Tensorflow Serving system. Convergence analysis indicates that RRL Plus can shorten the average convergence time by 17.54% and reduce the average latency by 15.31%, compared to RRL. Extensive experimental evaluation on both CPU cluster and GPU cluster show that the RRL Plus can quickly adapt to the dynamics of workloads and system environments. The proposed scheduling framework can reduce the average latency by up to 88.9% on the CPU cluster and 71.5% on the GPU cluster, compared to the state-of-the-art deep reinforcement-learning-based methods (DeepRM and CAPES). In the SLO-aware scenario, the RRL Plus can reduce up to 91.6% SLO violation under strict SLO requirements, while reducing the resource usage by up to 68.9% on CPU and 43.7% on GPU under loose SLO requirements. In addition, the proposed solution does not have assumptions on workload or underlying systems and thus can be used for most modern ML systems and applications.

Acknowledgment

The authors would like to thank Research and Innovation and the Office of Information Technology at the University of Nevada, Reno, NV, USA, for computing time on the Pronghorn High-Performance Computing Cluster.

References

[1] H. Qin, S. Zawad, Y. Zhou, L. Yang, D. Zhao, and F. Yan, "Swift machine learning model serving scheduling: A region based reinforcement learning approach," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal. (SC)*, 2019, pp. 1–23.

[2] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "ImageNet classification with deep convolutional neural networks," *Commun. ACM*, vol. 60, no. 6, pp. 84–90, 2017.

[3] D. Amodei *et al.*, "Deep speech 2: End-to-end speech recognition in English and Mandarin," in *Proc. 33rd Int. Conf. Mach. Learn. (ICML)*, vol. 48, 2016, pp. 173–182.

[4] J. Andreas, M. Rohrbach, T. Darrell, and D. Klein, "Learning to compose neural networks for question answering," in *Proc. Conf. North Amer. Assoc. Comput. Linguist. Human Lang. Technol. (NAACL HLT)*, 2016, pp. 1545–1554.

[5] D. Silver *et al.*, "Mastering the game of Go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[6] M. Ribeiro, K. Grolinger, and M. A. Capretz, "MLaaS: Machine learning as a service," in *Proc. IEEE 14th Int. Conf. Mach. Learn. Appl. (ICMLA)*, 2015, pp. 896–902.

[7] M. Zhang *et al.*, "Accelerating large scale deep learning inference through DeepCPU at Microsoft," in *Proc. USENIX Conf. Oper. Mach. Learn. (OpML)*, May 2019, pp. 5–7.

[8] F. Yan, Y. He, O. Ruwase, and E. Smirni, "SERF: Efficient scheduling for fast deep neural network serving via judicious parallelism," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal. (SC)*, Nov. 2016, pp. 300–311.

[9] J. Dean *et al.*, "Large scale distributed deep networks," in *Proc. NIPS*, 2012, pp. 1232–1240.

[10] T. M. Chilimbi, Y. Suzue, J. Apacible, and K. Kalyanaraman, "Project Adam: Building an efficient and scalable deep learning training system," in *Proc. OSDI*, 2014, pp. 571–582.

[11] L. Bottou, "Large-scale machine learning with stochastic gradient descent," in *Proc. COMPSTAT*, 2010, pp. 177–186.

[12] I. Tanasic, I. Gelado, J. Cabezas, A. Ramírez, N. Navarro, and M. Valero, "Enabling preemptive multiprogramming on GPUs," in *Proc. ACM/IEEE 41st Int. Symp. Comput. Archit. (ISCA)*, Jun. 2014, pp. 193–204.

[13] C. Olston *et al.*, "TensorFlow-serving: Flexible, high-performance ML serving," 2017. [Online]. Available: https://arxiv.org/abs/1712.06139.

[14] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, "Resource management with deep reinforcement learning," in *Proc. 15th ACM Workshop Hot Topics Netw.*, 2016, pp. 50–56.

[15] Y. Li, K. Chang, O. Bel, E. L. Miller, and D. D. E. Long, "CAPES: Unsupervised storage performance tuning using neural network-based deep reinforcement learning," in *Proc. Int. Conf. High Perform. Comput. Netw. Storage Anal.*, 2017, p. 42.

[16] C. Chen, A. Seff, A. L. Kornhauser, and J. Xiao, "DeepDriving: Learning affordance for direct perception in autonomous driving," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Santiago, Chile, Dec. 2015, pp. 2722–2730.

[17] M. Abadi *et al.*, "TensorFlow: Large-scale machine learning on heterogeneous distributed systems," 2016. [Online]. Available: https://arxiv.org/abs/1603.04467

[18] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang, "CherryPick: Adaptively unearthing the best cloud configurations for big data analytics," in *Proc. NSDI*, vol. 2, 2017, pp. 469–482.

[19] M. Jeon, Y. He, S. Elnikety, A. L. Cox, and S. Rixner, "Adaptive parallelism for Web search," in *Proc. EuroSys*, 2013, pp. 155–168.

[20] F. Yan, Y. He, O. Ruwase, and E. Smirni, "Efficient deep neural network serving: Fast and furious," *IEEE Trans. Netw. Service Manag.*, vol. 15, no. 1, pp. 112–126, Mar. 2018.

[21] M. Li *et al.*, "MRONLINE: MapReduce online performance tuning," in *Proc. 23rd Int. Symp. High Perform. Parallel Distrib. Comput.*, 2014, pp. 165–176.

[22] T. Wang, H. Luo, X. Zheng, and M. Xie, "Crowdsourcing mechanism for trust evaluation in CPCS based on intelligent mobile edge computing," *ACM Trans. Intell. Syst. Technol.*, vol. 10, no. 6, pp. 1–19, Oct. 2019.

[23] T. Wang, P. Wang, S. Cai, Y. Ma, A. Liu, and M. Xie, "A unified trustworthy environment establishment based on edge computing in industrial IoT," *IEEE Trans. Ind. Informat.*, to be published.

[24] C. Zhang, H. Tian, W. Wang, and F. Yan, "Stay fresh: Speculative synchronization for fast distributed machine learning," in *Proc. 38th IEEE Int. Conf. Distrib. Comput. Syst. (ICDCS)*, Vienna, Austria, Jul. 2018, pp. 99–109.

[25] C. Zhang, M. Yu, W. Wang, and F. Yan, "MARK: Exploiting cloud services for cost-effective, SLO-aware machine learning inference serving," in *Proc. USENIX Annu. Tech. Conf. (USENIX ATC)*, 2019, pp. 1049–1062.

[26] Z. Zhang, L. Cherkasova, and B. T. Loo, "Optimizing cost and performance trade-offs for MapReduce job processing in the cloud," in *Proc. NOMS*, 2014, pp. 1–8.

[27] J. Oly and D. A. Reed, "Markov model prediction of I/O requests for scientific applications," in *Proc. 16th Int. Conf. Supercomput. (ICS)*, 2002, pp. 147–155.

[28] M. G. Azar, R. Munos, M. Ghavamzadeh, and H. J. Kappen, "Speedy Q-learning," in *Proc. Adv. Neural Inf. Process. Syst.*, 2011 pp. 2411–2419.

[29] A. M. Devraj and S. P. Meyn, "Fastest convergence for Q-learning," 2017. [Online]. Available: https://arxiv.org/abs/1707.03770.

[30] E. H. L. Aarts and J. H. M. Korst, *Simulated Annealing and Boltzmann Machines—A Stochastic Approach to Combinatorial Optimization and Neural Computing* (Wiley-Interscience Series in Discrete Mathematics and Optimization). Hoboken, NJ, USA: Wiley, 1990. [Online]. Available: https://dblp.org/rec/bib/books/daglib/0067313

[31] C. Szepesvári, "The asymptotic convergence-rate of Q-learning," in *Proc. Adv. Neural Inf. Process. Syst.*, 1998, pp. 1064–1070.

[32] J. Snoek, H. Larochelle, and R. P. Adams, "Practical Bayesian optimization of machine learning algorithms," in *Proc. Adv. Neural Inf. Process. Syst. 26th Annu. Conf. Neural Inf. Process. Syst.*, Dec. 2012, pp. 2960–2968.

[33] J. Fu, A. Kumar, M. Soh, and S. Levine, "Diagnosing bottlenecks in deep Q-learning algorithms," in *Proc. 36th Int. Conf. Mach. Learn. (ICML)*, vol. 97, Jun. 2019, pp. 2021–2030.

[34] V. Nair and G. E. Hinton, "Rectified linear units improve restricted Boltzmann machines," in *Proc. 27th Int. Conf. Mach. Learn. (ICML)*, 2010, pp. 807–814.

[35] J. D. Abernethy, E. Hazan, and A. Rakhlin, "Competing in the dark: An efficient algorithm for bandit linear optimization," in *Proc. 21st Annu. Conf. Learn. Theory (COLT)*, Jul. 2008, pp. 263–274.

[36] C. Szegedy, V. Vanhoucke, S. Ioffe, J. Shlens, and Z. Wojna, "Rethinking the inception architecture for computer vision," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2016, pp. 2818–2826.

[37] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-V4, inception-RESNET and the impact of residual connections on learning," in *Proc. AAAI*, vol. 4, 2017, p. 12.

[38] G. Urdaneta, G. Pierre, and M. van Steen, "Wikipedia workload analysis for decentralized hosting," *Comput. Netw.*, vol. 53, no. 11, pp. 1830–1845, Jul. 2009.

[39] J. A. Stankovic, "Research directions for the Internet of Things," *IEEE Internet Things J.*, vol. 1, no. 1, pp. 3–9, Feb. 2014.

[40] T. Wang, Y. Mei, W. Jia, X. Zheng, G. Wang, and M. Xie, "Edge-based differential privacy computing for sensor-cloud systems," *J. Parallel Distrib. Comput.*, vol. 136, pp. 75–85, Feb. 2020. [Online]. Available: https://doi.org/10.1016/j.jpdc.2019.10.009

[41] M. Mohammadi, A. Al-Fuqaha, M. Guizani, and J.-S. Oh, "Semisupervised deep reinforcement learning in support of IoT and smart city services," *IEEE Internet Things J.*, vol. 5, no. 2, pp. 624–635, Apr. 2018.

[42] X. Cao, R. Ma, L. Liu, H. Shi, Y. Cheng, and C. Sun, "A machine learning-based algorithm for joint scheduling and power control in wireless networks," *IEEE Internet Things J.*, vol. 5, no. 6, pp. 4308–4318, Dec. 2018.

[43] M. Chen, Y. Li, X. Luo, W. Wang, L. Wang, and W. Zhao, "A novel human activity recognition scheme for smart health using multilayer extreme learning machine," *IEEE Internet Things J.*, vol. 6, no. 2, pp. 1410–1418, Apr. 2019.

[44] J. Liang, X. Liu, and K. Liao, "Soil moisture retrieval using UWB echoes via fuzzy logic and machine learning," *IEEE Internet Things J.*, vol. 5, no. 5, pp. 3344–3352, Oct. 2018.

[45] B. Reagen *et al.*, "MINERVA: Enabling low-power, highly-accurate deep neural network accelerators," in *Proc. 43rd ACM/IEEE Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 267–278.

[46] A. Dhurandhar, K. Shanmugam, R. Luss, and P. A. Olsen, "Improving simple models with confidence profiles," in *Proc. Adv. Neural Inf. Process. Syst. Annu. Conf. Neural Inf. Process. Syst. (NeurIPS)*, Dec. 2018, pp. 10317–10327.

[47] D. Crankshaw, X. Wang, G. Zhou, M. J. Franklin, J. E. Gonzalez, and I. Stoica, "Clipper: A low-latency online prediction serving system," in *Proc. 14th USENIX Symp. Netw. Syst. Design Implement. (NSDI)*, Mar. 2017, pp. 613–627.

[48] AWSLABS. (2019). *MXNET Model Server*. [Online]. Available: https://github.com/awslabs/mxnet-model-server

[49] T. Chen *et al.*, "DianNao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *Proc. ASPLOS*, 2014, pp. 269–284.

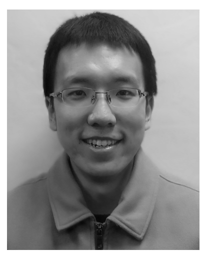
[50] F. N. Iandola *et al.*, "SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and < 0.5 mb model size," 2016. [Online]. Available: https://arxiv.org/abs/1602.07360.

[51] TensorFlow. *XLA: Optimizing Compiler for Machine Learning: TensorFlow*. Accessed: Jan. 24, 2020. [Online]. Available: https://www.tensorflow.org/xla

[52] *intel/mkl-dnn*, GitHub, San Francisco, CA, USA. Accessed: Jan. 24, 2020. [Online]. Available: https://github.com/intel/mkl-dnn

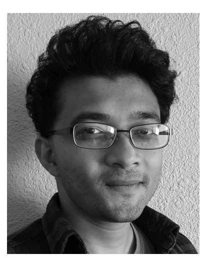
[53] S. P. Singh, T. S. Jaakkola, and M. I. Jordan, "Reinforcement learning with soft state aggregation," in *Proc. Adv. Neural Inf. Process. Syst.*, 1995, pp. 361–368.

[54] A. Mirhoseini *et al.*, "Device placement optimization with reinforcement learning," in *Proc. 34th Int. Conf. Mach. Learn. (ICML)*, Sydney, NSW, Australia, Aug. 2017, pp. 2430–2439. [Online]. Available: http://proceedings.mlr.press/v70/mirhoseini17a.html



Heyang Qin (Student Member, IEEE) received the bachelor's degree from the University of Electronic Science and Technology of China, Chengdu, China, in 2017. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering, University of Nevada, Reno, NV, USA.

He works as a Teaching Assistant and a Research Assistant with the University of Nevada, where he conducts research in areas of deep learning and reinforcement learning under the supervision of Dr. F. Yan and Dr. L. Yang.



Syed Zawad (Student Member, IEEE) received the B.Sc. degree in computer science and engineering from BRAC University, Dhaka, Bangladesh. He is currently pursuing the Ph.D. degree with the Department of Computer Science and Engineering, University of Nevada, Reno, NV, USA.

He has interned as a Researcher with Baidu, Sunnyvale, CA, USA. He also has three years of work experience as a Software Engineer for Web applications. His research area of interest is in high-performance computing, deep learning, federated learning, and neural architecture search.

Yanqi Zhou (Member, IEEE) received the bachelor's degree from the University of Michigan at Ann Arbor, Ann Arbor, MI, USA, and the Ph.D. degree from Princeton University, Princeton, NJ, USA.

She is a Research Scientist with Google Brain, Mountain View, CA, USA. Her research interest lies in computer systems and machine learning.

Sanjay Padhi (Member, IEEE) received the Ph.D. degree in high energy physics from McGill University, Montreal, QC, Canada.

He worked as a Physicist for about 15 years and had vast experience in predictive analytics, machine learning, and algorithm developments including analytics with streaming data. He leads a Research Initiatives with Amazon Web Services, Seattle, WA, USA. He is also an Adjunct Professor of physics with Brown University, Providence, RI, USA. He led various groups with hundreds of members with CERN, Meyrin, Switzerland, in physics, simulations, as well as distributed computing. Created and operated worldwide late-binding-based resource management systems, currently used by the CMS collaboration for all its computing activities across 140 institutions worldwide.

Lei Yang (Senior Member, IEEE) received the B.S. and M.S. degrees in electrical engineering from Southeast University, Nanjing, China, in 2005 and 2008, respectively, and the Ph.D. degree from the School of Electrical Computer and Energy Engineering, Arizona State University, Tempe, AZ, USA, in 2012.

He was a Postdoctoral Scholar with Princeton University, Princeton, NJ, USA, and an Assistant Research Professor with the School of Electrical Computer and Energy Engineering, Arizona State University. He is currently an Assistant Professor with the Department of Computer Science and Engineering, University of Nevada, Reno, NV, USA. His research interests include big data analytics, edge computing and its applications in IoT and 5G, stochastic optimization and modeling in smart cities and cyber-physical systems, data privacy and security in crowdsensing, and optimization and control in mobile social networks.

Dr. Yang was a recipient of the Best Paper Award Runner-Up at the IEEE INFOCOM 2014. He is currently an Associate Editor of IEEE ACCESS.

Feng Yan (Member, IEEE) received the M.S. and Ph.D. degrees in computer science from the College of William and Mary, Williamsburg, VA, USA, in 2011 and 2016, respectively.

He worked with Microsoft Research, Redmond, WA, USA, from 2014 to 2015 and HP Labs, Palo Alto, CA, USA, from 2013 to 2014. He is an Assistant Professor with the Department of Computer Science and Engineering, University of Nevada, Reno, NV, USA. He has a broad interest in big data and system areas. His current research focus includes machine learning, cloud/edge/fog computing, high-performance computing, storage, and cross-disciplinary topics among them and others.