AuTO: Scaling Deep Reinforcement Learning for Datacenter-Scale Automatic Traffic Optimization

Li Chen, Justinas Lingys, Kai Chen, Feng Liu†
SING Lab, Hong Kong University of Science and Technology, †SAIC Motors
{lchenad,jlingys,kaichen}@cse.ust.hk,liufeng@saicmotor.com

ABSTRACT

Traffic optimizations (TO, e.g. flow scheduling, load balancing) in datacenters are difficult online decision-making problems. Previously, they are done with heuristics relying on operators' understanding of the workload and environment. Designing and implementing proper TO algorithms thus take at least weeks. Encouraged by recent successes in applying deep reinforcement learning (DRL) techniques to solve complex online control problems, we study if DRL can be used for automatic TO without human-intervention. However, our experiments show that the latency of current DRL systems cannot handle flow-level TO at the scale of current datacenters, because short flows (which constitute the majority of traffic) are usually gone before decisions can be made.

Leveraging the long-tail distribution of datacenter traffic, we develop a two-level DRL system, AuTO, mimicking the Peripheral & Central Nervous Systems in animals, to solve the scalability problem. Peripheral Systems (PS) reside on end-hosts, collect flow information, and make TO decisions locally with minimal delay for short flows. PS's decisions are informed by a Central System (CS), where global traffic information is aggregated and processed. CS further makes individual TO decisions for long flows. With CS&PS, AuTO is an end-to-end automatic TO system that can collect network information, learn from past decisions, and perform actions to achieve operator-defined goals. We implement AuTO with popular machine learning frameworks and commodity servers, and deploy it on a 32-server testbed. Compared to existing approaches, AuTO reduces the TO turn-around time from weeks to ~100 milliseconds while achieving superior performance. For example, it demonstrates up to 48.14% reduction in average flow completion time (FCT) over existing solutions.

SIGCOMM '18, August 20–25, 2018, Budapest, Hungary

ACM ISBN 978-1-4503-5567-4/18/08…$15.00
https://doi.org/10.1145/3230543.3230551

CCS CONCEPTS

• **Networks** → **Network resources allocation**; **Traffic engineering algorithms**; **Data center networks**; • **Computing methodologies** → **Reinforcement learning**;

KEYWORDS

Datacenter Networks, Reinforcement Learning, Traffic Optimization

1 INTRODUCTION

Datacenter traffic optimizations (TO, e.g. flow/coflow scheduling [1, 4, 8, 14, 18, 19, 29, 61], congestion control [3, 10], load balancing &routing [2]) have significant impact on application performance. Currently, TO is dependent on handcrafted heuristics for varying traffic load, flow size distribution, traffic concentration, etc. When parameter setting mismatches traffic, TO heuristics may suffer performance penalty. For example, in PIAS [8], thresholds are calculated based on a long term flow size distribution, and is prone to mismatch the current/true size distribution in run-time. Under mismatch scenarios, performance degradation can be as much as 38.46% [8]. pFabric [4] shares the same problem when implemented with limited switch queues: for certain cases the average FCT can be reduced by over 30% even if the thresholds are carefully optimized. Furthermore, in coflow scheduling, fixed thresholds in Aalo [18] depend on the operator's ability to choose good values upfront, since there is no run-time adaptation.

Apart from parameter-environment mismatches, the turn-around time of designing TO heuristics is long—at least weeks. Because they require operator insight, application knowledge, and traffic statistics collected over a long period of time. A typical process includes: first, deploying a monitoring system to collect end-host and/or switch statistics; second, after collecting enough data, operators analyze the data, design heuristics, and test it using simulation tools and optimization tools to find suitable parameter settings; finally, tested heuristics are enforced[1] (with application modifications [19, 61], OS kernel module [8, 14], switch configurations [10], or any combinations of the above).

Automating the TO process is thus appealing, and we desire an automated TO agent that can adapt to voluminous, uncertain, and volatile datacenter traffic, while achieving operator-defined goals. In this paper, we investigate reinforcement learning (RL) techniques [55], as RL is the subfield of machine learning concerned with decision making and action control. It studies how an agent can learn to achieve goals in a complex, uncertain environment. An RL agent observes previous environment states and rewards, then decides an action in order to maximize the reward. RL has achieved good results in many difficult environments in recent years with advances in deep neural networks (DNN): DeepMind's Atari results [40] and AlphaGo [52] used deep RL (DRL) algorithms which make few assumptions about their environments, and thus can be generalized in other settings. Inspired by these results, we are motivated to enable DRL for automatic datacenter TO.

We started by verifying DRL's effectiveness in TO. We implemented a flow-level centralized TO system with a basic DRL algorithm, policy gradient [55]. However, in our experiments (§2.2), even this simple algorithm running on current machine learning software frameworks[2] and advanced hardware (GPU) cannot handle traffic optimization tasks at the scale of production datacenters ($>10^5$ servers). The crux is the computation time ($\sim$100ms): short flows (which constitute the majority of the flows) are gone before the DRL decisions come back, rendering most decisions useless.

Therefore, in this paper we try to answer the key question: *How to enable DRL-based automatic TO at datacenter-scale?* To make DRL scalable, we first need to understand the long-tail distribution of datacenter traffic [3, 11, 33]: most of the flows are short flows[3], but most of the bytes are from long flows. Thus, TO decisions for short flows must be generated quickly; whereas decisions for long flows are more influential as they take longer time to finish.

We present AuTO, an end-to-end DRL system for datacenter-scale TO that works with commodity hardware. AuTO is a two-level DRL system, mimicking the Peripheral & Central Nervous Systems in animals. Peripheral Systems (PS) run on all end-hosts, collect flow information, and make instant TO decisions locally for short flows. PS's decisions are informed by the Central System (CS), where global traffic information are aggregated and processed. CS further makes individual TO decisions for long flows which can tolerate longer processing delays.

The key to AuTO's scalability is to detach time-consuming DRL processing from quick action-taking for short flows. To achieve this, we adopt Multi-Level Feedback Queueing (MLFQ) [8] for PS to schedule flows guided by a set of thresholds. Every new flow starts at the first queue with highest priority, and is gradually demoted to lower queues after its sent bytes pass certain thresholds. Using MLFQ, AuTO's PS makes per-flow decisions instantly upon local information (bytes-sent and thresholds)[4], while the thresholds are still optimized by a DRL algorithm in the CS over a relatively longer period of time. In this way, global TO decisions are delivered to PS in the form of MLFQ thresholds (which is more delay-tolerant), enabling AuTO to make globally informed TO decisions for the majority of flows with only local information. Furthermore, MLFQ naturally separates short and long flows: short flows complete in the first few queues, and long flows descend down to the last queue. For long flows, CS centrally processes them individually using a different DRL algorithm to determine routing, rate limiting, and priority.

We have implemented an AuTO prototype using Python. AuTO is thus compatible with popular learning frameworks, such as Keras/TensorFlow. This allows both networking and machine learning community to easily develop and test new algorithms, because software components in AuTO are reusable in other RL projects in datacenter.

We further build a testbed with 32 servers connected by 2 switches to evaluate AuTO. Our experiments show that, for traffic with stable load and flow size distribution, AuTO's performance improvement is up to 48.14% compared to standard heuristics (shortest-job-first and least-attained-service-first) after 8 hours of training. AuTO is also shown to learn steadily and adapt across temporally and spatially heterogeneous traffic: after only 8 hours of training, AuTO achieves 8.71% (9.18%) reduction in average (tail) FCT compared to heuristics.

In the following, we first overview DRL and reveal why current DRL systems fail to work at large scale in §2. We describe system design in §3, as well as the DRL formulations and solutions in §4. We implement AuTO in §5, and evaluate it with extensive experiments in §6 using a realistic testbed. Finally, we review related works in §7, and conclude in §8.

2 BACKGROUND AND MOTIVATION

In this section, we first overview the RL background. Then, we describe and apply a basic RL algorithm, policy gradient,

[1]After the heuristic is designed, its parameters can usually be computed in a short time for average scenarios: minutes [8, 14, 19] or hours [61]. AuTO seeks to automate the entire TO design process, rather than just parameter selection.

[2]e.g. TensorFlow [57], PyTorch [48], Ray [42]

[3]The threshold between short and long flows is dynamically determined in AuTO based on current traffic distribution (§4).

[4]For short flows, AuTO relies on ECMP[30] (which is also not centrally controlled) for routing/load-balancing and makes no rate-limiting decisions.

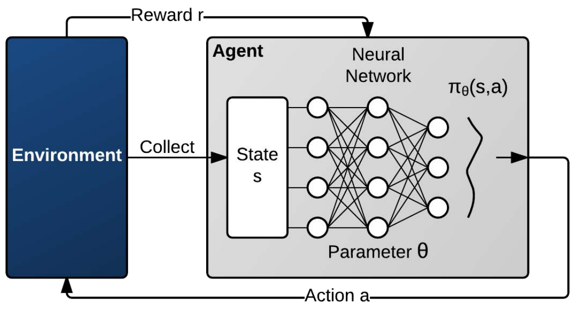

Figure 1: A general reinforcement learning setting using neural network as policy representation.

to enable flow scheduling in TO. Finally, we show the problem of an RL system running PG using testbed experiments, motivating AuTO.

2.1 Deep Reinforcement Learning (DRL)

As shown in Figure 1, *environment* is the surroundings of the *agent* with which the *agent* can interact through observations, actions, and feedback (rewards) on actions [55]. Specifically, in each time step t, the agent observes state s_t, and chooses action a_t. The state of the environment then transits to s_{t+1}, and the agent receives reward r_t. The state transitions and rewards are stochastic and Markovian [36]. The objective of learning is to maximize the expected cumulative discounted reward $E[\sum_{t=0}^{\infty}\gamma^t r_t]$ where $\gamma_t \in (0,1]$ is the discounting factor.

The RL agent takes actions based on a *policy*, which is a probability distribution of taking action a in the state s: $\pi(s,a)$. For most practical problems, it is infeasible to learn all possible combinations of state-action pairs, thus function approximation [31] technique is commonly used to learn the policy. A function approximator $\pi_\theta(s,a)$ is parameterized by θ, whose size is much smaller (thus mathematically tractable) than the number of all possible state-action pairs. Function approximator can have many forms, and recently, deep neural networks (DNNs) have been shown to solve practical, large-scale dynamic control problems similar to flow scheduling. Therefore, we also use DNN as the representation of function approximator in AuTO.

With function approximation, the agent learns by updating the function parameters θ with the state s_t, action a_t, and the corresponding reward r_t in each time period/step t. We focus on one class of updating algorithms that learn by performing gradient-descent on the policy parameters. The learning involves updating the parameters (link weights) of a DNN so that the aforementioned objective could be maximized.

$$\theta \leftarrow \theta + \alpha \sum_t \nabla_\theta \log \pi_\theta(s_t,a_t)\, v_t \qquad (1)$$

Training of the agent's DNN adopts a variant of the well-known REINFORCE algorithm [56]. This variant uses a modified version of Equation (1), which alleviates the drawbacks of the algorithm: convergence speed and variance. To mitigate the drawbacks, Monte Carlo Method [28] is used to compute an empirical reward, v_t, and a baseline value (the cumulative average of experienced rewards per server) is used for reducing the variance [51]. The resultant update rule (Equation (2)) is applied to the policy DNN, due to its variance management and guaranteed convergence to at least a local minimum [56]:

$$\theta \leftarrow \theta + \alpha \sum_t \nabla_\theta \log \pi_\theta(s_t,a_t)\,(v_t - baseline) \qquad (2)$$

2.2 Example: DRL for Flow Scheduling

As an example, we formulate the problem of flow scheduling in datacenters as a DRL problem, and describe a solution using the PG algorithm based on Equation (2).

Flow scheduling problem We consider a datacenter network connecting multiple servers. For simplicity, we adopt the *big-switch* assumption by previous works in flow scheduling [4, 14], where the network is non-blocking with full-bisection bandwidth and proper load-balancing. Following this assumption, the flow scheduling problem is simplified to the problem of deciding the sending order of flows. We consider an implementation that enables preemptive scheduling of flows using strict priority queueing. We create K priority queues for flows in each server [23], and enforce strict priority queuing among them. K priority queues are also configured in the switches, similar to [8]. The priority of each flow can be changed dynamically to enable pre-emption. The packet of each flow is tagged with its current priority number, and will be placed in the same queue throughout the entire datacenter fabric.

DRL formulation
Action Space: The action provided by the agent is a mapping from active flows to priorities: for each active flow f, at time step t, its priority is $p_t(f) \in [1,K]$.

State space: The big-switch assumption allows for a much simplified state space. As routing and load balancing are out of our concern, the state space only includes the flow states. In our model, states are represented as the set of all active flows, F_a^t, and the set of all finished flows, F_d^t, in the entire network at current time step t. Each flow is identified by its 5-tuple [8, 38]: source/destination IP, source/destination port numbers, and transport protocol. Active flows have an additional attribute, which is its priority; while finished flows have two additional attributes: FCT and flow size[5].

Rewards: Rewards are feedback to the *agent* on how good its actions are. The reward can be obtained after the completion of a flow, thus is computed only on the set of finished flows F_d^t for time step t. The average throughput of each finished flow f is $Tput_f = \frac{Size_f}{FCT_f}$. We model the reward as the ratio between the average throughputs of two consecutive time steps.

$$r_t = \frac{\sum_{f^t \in F_d^t} Tput_f^t}{\sum_{f^{t-1} \in F_d^{t-1}} Tput_f^{t-1}} \tag{3}$$

It signals if the previous actions have resulted in a higher per-flow throughput experienced by the agent, or it has degraded the overall performance. The objective is to maximize the average throughput of the network as a whole.

DRL algorithm We use the update rule specified by Equation (2). The DNN residing on the agent computes probability vectors for each new state and updates its parameters by evaluating the action that resulted in the current state. The evaluation step compares the previous average throughput with the corresponding value of the current step. Based on the comparison, an appropriate reward (either negative or positive) is produced which is added to the baseline value. Thus, we can ensure that the function approximator improves with time and can converge to a local minimum by updating DNN weights in the direction of the gradient. The update which follows (2) ensures that poor flow scheduling decisions are discouraged for similar states in the future, and the good ones become more probable for similar states in the future. When the system converges, the policy achieves a sufficient flow scheduling mechanism for a cluster of servers.

2.3 Problem Identified

Using the DRL problem of flow scheduling as an example, we implement PG using popular machine learning frameworks: Keras/TensorFlow, PyTorch, and Ray. We simplify the DRL agent to have only 1 hidden layer. We use two servers: DRL agent resides in one, and the other sends mock traffic information (states) to the agent using an RPC interface. We set the sending rate of the mock server to 1000 flows per second (fps). We measure the processing latency of different implementations at the mock server: the time between finish sending the flow information and receiving the action. The servers are Huawei Tecal RH1288 V2 servers running 64-bit Debian 8.7, with 4-core Intel E5-1410 2.8GHz CPU, NVIDIA K40 GPU, and Broadcom 1Gbps NICs.

[5]Flow size and FCT can be measured when the flow ends using either OS utility[44] or application layer mechanisms[49, 61].

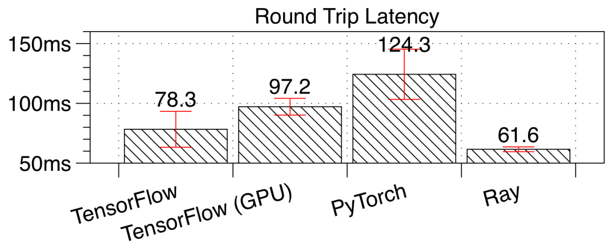

Figure 2: Current DRL systems are insufficient.

As shown in Figure2, even for small flow arrival rate of 1000fps and only 1 hidden layer, the processing delays of all implementations are more than 60ms, during which time any flow within 7.5MB would have finished on a 1Gbps link. For reference, using the well-known traffic traces of a web search application and a data mining application collected in Microsoft datacenters[3, 8, 26], a 7.5MB flow is larger than 99.99% and 95.13% of all flows, respectively. This means, most of the DRL actions are useless, as the corresponding flows are already gone when the actions arrive.

Summary Current DRL systems' performance is not enough to make online decisions for datacenter-scale traffic. They suffer from long processing delays even for simple algorithms and low traffic load.

3 AUTO DESIGN

3.1 Overview

The key problem of current DRL systems is the long latency between collection of flow information and generation of actions. In modern datacenters with $\geq 10Gbps$ link speed, to achieve flow-level TO operations, the round-trip latency of actions should be at least sub-millisecond. Without introducing specialized hardware, this is unachievable (§2.2). Using commodity hardware, the processing latency of DRL algorithm is a hard limit. Given this constraint, how to scale DRL for datacenter TO?

Recent studies [3, 11, 33] have shown that most datacenter flows are short flows, yet most traffic bytes are from long flows. Informed by such long-tail distribution, our insight is to delegate most short flow operations to the end-host, and formulate DRL algorithms to generate long-term (sub-second) TO decisions for long flows.

We design AuTO as a two-level system, mimicking the Peripheral and Central Nervous Systems in animals. As shown in Figure 3, Peripheral Systems (PS) run on all end-hosts, collect flow information, and make TO decisions locally with minimal delay for short flows. Central System (CS) makes individual TO decisions for long flows that can tolerate longer processing delays. Furthermore, PS's decisions are informed by the CS where global traffic information are aggregated and processed.

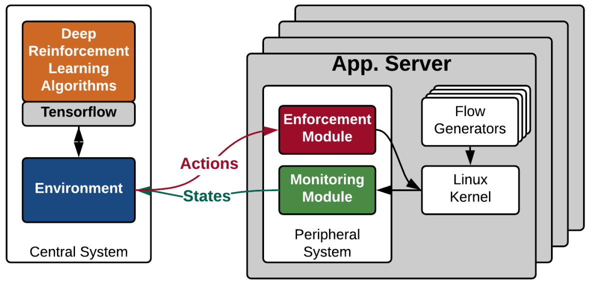

Figure 3: AuTO overview.

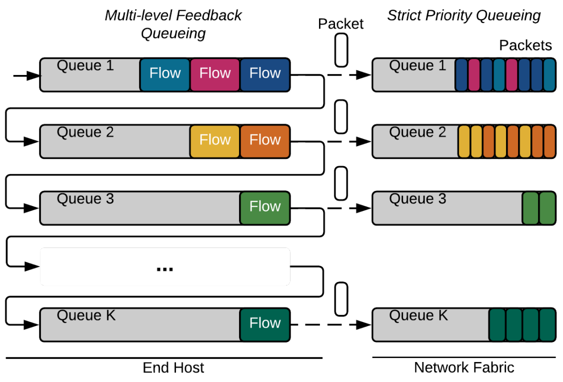

Figure 4: Multi-Level Feedback Queuing.

3.2 Peripheral System

The key to AuTO's scalability is to enable PS to make globally informed TO decisions on short flows with only local information. PS has two modules: an enforce module and a monitoring module.

Enforcement module To achieve the above goal, we adopt Multi-Level Feedback Queueing (MLFQ, introduced in PIAS [8]) to schedule flows without centralized per-flow control. Specifically, PS performs packet tagging in the DSCP field of IP packets at each end-host as shown in Figure 4. There are K priorities, $P_i, 1 \leq i \leq K$, and $(K-1)$ demotion thresholds, $\alpha_j, 1 \leq j \leq K-1$. We configure all the switches to perform strict priority queueing based on the DSCP field. At the end host, when a new flow is initialized, its packets are tagged with P_1, giving them the highest priority in the network. As more bytes are sent, the packets of this flow will be tagged with decreasing priorities P_j $(2 \leq j \leq K)$, thus they are scheduled with decreasing priorities in the network. The threshold to demote priority from P_{j-1} to P_j is α_{j-1}.

With MLFQ, PS has the following properties:

- It can make instant per-flow decisions based only on local information: bytes-sent and thresholds.

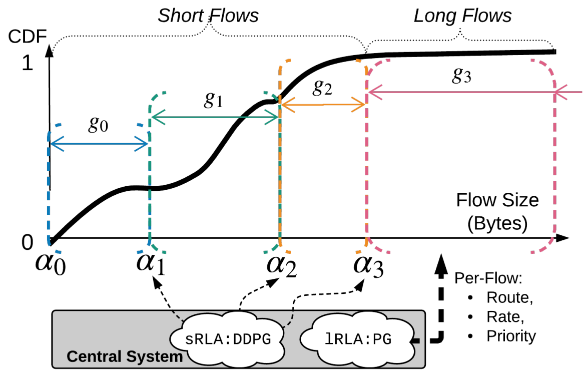

Figure 5: AuTO: A 4-queue example.

- It can adapt to global traffic variations. To be scalable, CS must not directly control small flows. Instead, CS optimizes and sets MLFQ thresholds with global information over a longer period of time. Thus, thresholds in PS can be updated to adapt to traffic variations. In contrast, PIAS [8] requires weeks of traffic traces to update the thresholds.
- It naturally separates short and long flows. As shown in Figure 5, short flows finished in the first few queues, and long flows drop to the last queue. Thus, CS can centrally process long flows individually to make decisions on routing, rate limit, and priority.

Monitoring module For CS to generate thresholds, the monitoring module collects flow sizes and completion times of all finished flows, so that CS can update flow size distribution. The monitoring module also reports on-going long flows that have descended into the lowest priority on its end-host, so that CS can make individual decisions.

3.3 Central System

The CS is composed of two DRL agents (RLA): short flow RLA (sRLA) is for optimizing thresholds for MLFQ, and long flow RLA (lRLA) is for determining rates, routes, and priorities for long flows. sRLA attempts to solve a FCT minimization problem, and we develop a Deep Deterministic Policy Gradient algorithm for this purpose. For lRLA, we use a PG algorithm (§2.2) to generate actions for the long flows. In the next section, we describe the two DRL problems and solutions.

4 DRL FORMULATIONS AND SOLUTIONS

In this section, we describe the two DRL algorithms in CS.

4.1 Optimizing MLFQ thresholds

We consider a datacenter network connecting multiple servers. Scheduling of flows is imposed by using K strict priority queues at hosts and network switches (Figure 4) by setting the DCSP field in each of the IP headers. The longer the flow is, the lower priority is assigned to the flow as it is demoted through host priority queues in order to approximate Shortest-Job-First (SJF). The packet's priority is preserved throughout the entire datacenter fabric till it reaches the destination.

One of the challenges of MLFQ is the calculation of the optimal demotion thresholds for the K priority queues at the host. Prior works [8, 9, 14] provide mathematical analysis and models for optimizing the demotion thresholds: $\{\alpha_1,\alpha_2,...,\alpha_{K-1}\}$. Bai et al. [9] also suggest weekly/monthly re-computation of the thresholds with collected flow-level traces. AuTO takes a step further and proposes a DRL approach to optimizing the values of the α's. Unlike prior works that used machine learning in datacenter problems [5, 36, 60], AuTO is unique due to its target - optimization of real values in continuous action space. We formulate the threshold optimization problem as an DRL problem and try to explore the capabilities of DNN for modeling the complex datacenter network for computing the MLFQ thresholds.

As shown in §2.2, PG is a basic DRL algorithm. The agent follows a policy $\pi_\theta(a|s)$ parameterized by a vector θ and improves it with experience. However, REINFORCE and other regular PG algorithms only consider stochastic policies, $\pi_\theta(a|s)=P[a|s;\theta]$, that select action a in state s according to the probability distribution over the action set $\mathcal{A}$ parameterized by θ. PG cannot be used for value optimization problem, as a value optimization problem computes real values. Therefore, we apply a variant of *Deterministic Policy Gradient* (DPG) [53] for approximating optimal values $\{a_0,a_1,...,a_n\}$ for the given state s such that $a_i=\mu_\theta(s)\ for\ i=0,...,n$. Figure 6 summarizes the major differences between stochastic and deterministic policies. DPG is an actor-critic [12] algorithm for deterministic policies, which maintains a parameterized actor function μ_θ for representing current policy and a critic neural network $Q(s,a)$ that is updated using the Bellman equation (as in Q-learning [41]). We describe the algorithm with Equation (4,5,6) as follows: The actor samples the environment and has its parameters θ updated according to Equation (4). The result of Equation (4) follows from the fact that the objective of the policy is to maximize the expected cumulative discounted reward Equation(5) and its gradient can be expressed in the following form Equation(5). For more details, please refer to [53].

$$\theta^{k+1} \leftarrow \theta^k + \alpha E_{s\sim\rho^{\mu^k}}\left[\nabla_\theta\mu_\theta(s)\nabla_a Q^{\mu^k}(s,a)\Big|_{a=\mu_\theta(s)}\right] \tag{4}$$

whereρ^{μ^k} is the state distribution at time k.

$$\begin{aligned}J(\mu_\theta)=&\int_S \rho^\mu(s)r(s,\mu_\theta(s))ds\\ =&E_{s\sim\rho^\mu}[r(s,\mu_\theta(s))]\end{aligned} \tag{5}$$

Algorithm 1: DDPG Actor-Critic Update Step

```
1 Sample a random mini-batch of $N$ transitions
  $(s_i,a_i,r_i,s_{i+1})$ from buffer
2 Set $y_i=r_i + \gamma Q'_{\theta^{Q'}}(s_{i+1},\mu'_{\theta^{\mu'}}(s_{i+1}))$
3 Update critic by minimizing the loss:
  $L=\frac{1}{N}\sum_{i=1}^N(y_i-Q_{\theta^Q}(s_i,a_i))^2$
4 Update the actor policy using the sampled policy
  gradient:
  $$\nabla_{\theta^\mu}J\approx\frac{1}{N}\sum_{i=1}^N\nabla_{\theta^\mu}(s_i)\mu_{\theta^Q}(s_i)\nabla_{a_i}Q_{\theta^Q}(s_i,a_i)\Big|_{a_i=\mu_{\theta^Q}(s_i)}$$
5 Update the target networks:
  $$\theta^{Q'} \leftarrow \tau\theta^Q + (1-\tau)\theta^{Q'}$$
  $$\theta^{\mu'} \leftarrow \tau\theta^\mu + (1-\tau)\theta^{\mu'}$$
  where $\gamma$ and $\tau$ are small values for stable learning
```

$$\begin{aligned}\nabla_\theta J(\mu_\theta)=&\int_S \rho^\mu(s)\nabla_\theta\mu_\theta(s)\nabla_a Q^{\mu^k}(s,a)\Big|_{a=\mu_\theta(s)}ds\\ =&E_{s\sim\rho^\mu}[\nabla_\theta\mu_\theta(s)\nabla_a Q^{\mu^k}(s,a)\Big|_{a=\mu_\theta(s)}]\end{aligned} \tag{6}$$

Deep Deterministic Policy Gradient (DDPG) [35] is an extension of DPG algorithm, which exploits deep learning techniques [41]. We use DDPG as our model for the optimization problem and explain how it works in the following. Same as DPG, DDPG is also an actor-critic [12] algorithm, and it maintains four DNNs. Two DNNs, critic $Q_{\theta^Q}(s,a)$ and actor $\mu_{\theta^\mu}(s)$ with weights θ^Q and θ^μ, are trained on sampled mini-batches of size N, where an item represents an experienced transition tuple (s_i,a_i,r_i,s_{i+1}) while the agent interacts with the environment. The DNNs are trained on random samples, which are stored in a buffer, in order to avoid correlated states which cause the DNNs to diverge [41]. The other two DNNs, target actor μ'_θ and target critic $Q'_{\theta^{Q'}}(s,a)$, are used for smooth updates of the actor and critic networks, respectively (Algorithm (1) [35]). The update steps stabilize training the actor-critic networks and achieve state-of-the-art results on continuous space actions [35]. AuTO applies DDPG for optimizing threshold values to achieve better flow scheduling decisions.

DRL formulation Next, we show that the optimization of thresholds can be formulated as an actor-critic DRL problem solvable by DDPG. We first develop an optimization problem of choosing an optimal set of thresholds $\{\alpha_i\}$ to minimize the average FCT of flows. Then we translate this problem into DRL problem to be solved using DDPG algorithm.

Denote the cumulative density function of flow size distribution as $F(x)$, thus $F(x)$ is the probability that a flow size is no larger than x. Let L_i denote the number of packets a given flow brings in queue Q_i for i=1,...,K. Thus, $E[L_i]\leq$

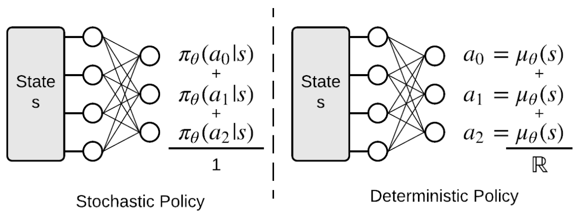
State s
$\pi_\theta(a_0|s)$ + $\pi_\theta(a_1|s)$ + $\pi_\theta(a_2|s)$
1
Stochastic Policy
State s
$a_0 = \mu_\theta(s)$ + $a_1 = \mu_\theta(s)$ + $a_2 = \mu_\theta(s)$
$\mathbb{R}$
Deterministic Policy

Figure 6: Comparison of deep stochastic and deep deterministic policies.

$(\alpha_i - \alpha_{i-1})(1-F(\alpha_{i-1}))$. Denote flow arrival rate as λ, then the packet arrival rate to queue Q_i is $\lambda_i = \lambda E[L_i]$. The service rate for a queue depends on whether the queues with higher priorities are all empty. Thus, P_1 (highest priority) has capacity $\mu_1 = \mu$ where μ is the service rate of the link. The idle rate of Q_1 is $(1-\rho_1)$ where $\rho_i = \lambda_i/\mu_i$ is the utilization rate of Q_i. Thus, the service rate of Q_2 is $\mu_2 = (1-\rho_1)\mu$ since its service rate is μ (the full link capacity) given that P_1 is empty. We have $\mu_i = \Pi_{j=0}^{i-1}(1-\rho_j)\mu$, with $\rho_0 = 0$. Thus, $T_i = 1/(\mu_i - \lambda_i)$ which is the average delay of queue i assuming M/M/1 queues. For a flow with size in $[\alpha_{i-1}, \alpha_i)$, it experiences the delays in different priority queues up to the i-th queue. Denote T_i as the average time spent in the i-th queue. Let $i_{max}(x)$ be the index of the smallest demotion threshold larger than x. So the average FCT for a flow with size x, $T(x)$, is upper-bounded by: $\sum_{i=1}^{i_{max}(x)} T_i$.

Let $g_i = F(\alpha_i) - F(\alpha_{i-1})$ denote the percentage of flows with sizes in $[\alpha_{i-1}, \alpha_i)$. Thus, g_i is the gap between two consecutive thresholds. Using g_i to equivalently express α_i, we can formulate the FCT minimization problem as[6]:

$$\min_{\{g\}} \quad \mathcal{T}(\{g\}) = \sum_{l=1}^{K}(g_l \sum_{m=1}^{l} T_m) = \sum_{l=1}^{K}(T_l \sum_{m=l}^{K} g_m) \tag{7}$$
$$\text{subject to} \quad g_i \geq 0, \quad i = 1, \ldots, K-1$$

We proceed to translate Problem (7) into a DRL problem.

State space: In our model, states are represented as the set of the set of all finished flows, F_d, in the entire network in the current time step. Each flow is identified by its 5-tuple[8, 38]: source/destination IP, source/destination port numbers, & transport protocol. As we report only finished flows, we also record the FCT and flow size as flow attributes. In total, each flow has 7 features.

Action Space: The action space is computed by a centralized agent, sRLA. At time step t, the action provided by the agent is a set of MLFQ threshold values $\{\alpha_i^t\}$.

[6]For a solution to this problem, e.g. $\{g_i'\}$, we can retrieve the thresholds $\{\alpha_i'\}$ with $\alpha_i' = F^{-1}(\sum_{j=1}^{i} g_j)$, where $F^{-1}(\cdot)$ is the inverse of $F(\cdot)$.

Rewards: Rewards are delayed feedback to the *agent* on how good its actions are for the previous time step. We model the reward as the ratio between objective functions of two consecutive time steps: $r_t = \frac{\mathcal{T}^{t-1}}{\mathcal{T}^t}$. It signals if the previous actions have resulted in a lower average FCT, or it has degraded the overall performance.

DRL algorithm We use the update rule specified by Equation (4) (Algorithm 1). The DNN computes g_i's for each newly received state from a host, and stores a tuple: (s_t, a_t, r_t, s_{t+1}) in its buffer for later learning. Reward r_t and the next state s_{t+1} are only known when the next update comes from the same host, so the agent buffers s_t and a_t until all needed information is received. Updates of parameters are performed in random batches to stabilize learning and to reduce probability of divergence [35, 41]. The reward r_t is computed at a host at step t and is compared to the previous average FCT. Based on the comparison, an appropriate reward (either negative or positive) is produced which is sent to the agent as a signal for evaluating action a_t. By following Algorithm 1, the system can improve the underlying actor-critic DNNs and converge to an solution for Problem (7).

4.2 Optimizing Long Flows

The last threshold, α_{K-1}, separates long flows from short flows by sRLA, thus α_{K-1} is updated dynamically according to current traffic characteristics, in contrast to prior works with fixed threshold for short and long flows [1, 22]. For long flows and lRLA, we use a PG algorithm similar to the flow scheduling problem in §2.2, and the only difference is in the action space.

Action Space: For each active flow f, at time step t, its corresponding action is $\{Prio_t(f), Rate_t(f), Path_t(f)\}$, where $Prio_t(f)$ is the flow priority, $Rate_t(f)$ is the rate limit, and $Path_t(f)$ is the path to take for flow f. We assume the paths are enumerated in the same way as in XPath [32].

State space: Same as §2.2, states are represented as the set of all active flows, F_a^t, and the set of all finished flows, F_d^t, in the entire network at current time step t. Apart from its 5-tuple [8, 38], each active flow has an additional attribute: its priority; each finished flow has two additional attributes: FCT and flow size.

Rewards: The reward is obtained for the set of finished flows F_d^t. Choices for the reward function can be: difference or ratios of sending rate, link utilization, and throughput in consecutive time steps. For modern datacenters with at least 10Gbps link speed, it is not easy to obtain timely flow-level information for active flows. Therefore, we choose to compute reward with finished flows only, and use the ratio between the average throughputs of two consecutive time steps as reward, as in Equation (3). The reward is capped to achieve quick convergence [35].

5 IMPLEMENTATION

In this section, we describe the implementation. We develop AuTO in Python 2.7. The language choice facilitates integration with modern deep learning frameworks [17, 45, 57], which provide excellent Python interfaces [45]. The current prototype uses the *Keras* [17] deep learning library (with TensorFlow as backend).

5.1 Peripheral System

PS is a daemon process running on each server. It has a Monitoring Module (MM) and an Enforcement Module (EM). The MM thread collects information about flows including recently finished flows and the presently active long flows (in the last queue of MLFQ). At the end of each period, the MM aggregates collected information, and sends to CS. The PS's EM thread performs tagging based on the MLFQ thresholds on currently active flows, as well as routing, rate limiting, and priority tagging for long flows. We implement a Remote Procedure Call (RPC) interface for communications between PS and CS. CS uses RPC to set MLFQ thresholds and to perform actions on active long flows.

5.1.1 Monitoring Module (MM):. For maximum efficiency, the MM can be implemented as a Linux kernel module, as in PIAS[8]. However, for the current prototype, since we are using a flow generator (as seen in [8, 10, 20]) to produce workloads, we choose to implement the MM directly inside the flow generator. This choice allows us to obtain the ground truth and get rid of other network flows that may interfere with the results.

For long flows (flows in the last queue of MLFQ), every T seconds, MM merges n_l active long flows (each with 6 attributes), and m_l finished long flows (each with 7 attributes) into an list. For short flows (in the first few queues of MLFQ) in the same period, MM collects m_s finished flows (each with 7 attributes) into an list. Finally, MM concatenates the two lists and sends them to CS as an observation of of the environment.

AuTO's parameters, $\{n_l,m_l,m_s\}$, are determined by traffic load and T: for each server, n_l (m_l) should be the upper-bound of number of active (finished) long flows within T, and m_s should also be the upper-bound of finished short flows. In the case that the actual number of active (finished) flow is less than $\{n_l,m_l,m_s\}$, the observation vector is zero-padded to the same size of the corresponding agent's DNN(s). We make this design choice because the number of input neurons of the DNN in CS is fixed, therefore can take only fixed-sized inputs. We leave dynamic DNN and recurrent neural network structure as future work. For the current prototype and experiments on the prototype, since we control the flow generator, it is easy to comply with this constraint. We choose $\{n_l{=}11,m_l{=}10,m_s{=}100\}$ in the experiments.

5.1.2 Enforcement Module (EM):. EM receives actions from CS periodically. The actions include new MLFQ thresholds, and TO decisions on local long flows. For MLFQ thresholds, EM builds upon the PIAS [8] kernel module, and adds dynamic configuration of demotion thresholds.

For short flows, we leverage ECMP [30] for routing and load-balancing, which does not require centralized per-flow control, and DCTCP [3] for congestion control.

For long flows, the TO actions include priority, rate limiting, and routing. EM leverages the same kernel module for priority tagging. Rate limiting is done using hierarchical token bucket (HTB) queueing discipline in Linux traffic control (tc). EM is configured with a parent class in HTB with outbound rate limit to represent the total outbound bandwidth managed by CS on this node. When a flow descends into the last queue in MLFQ, EM creates a HTB filter matching the exact 5-tuple for that flow. When EM receives rate allocation decisions from the CS, EM updates the child class of the particular flow by sending Netlink messages to Linux kernel: the rate of the TC class is set as the rate that centralized scheduler decides, and its ceiling is set to the smaller of the original ceiling and twice of the rates from CS.

5.2 Central System

CS runs RL agents (sRLA & lRLA) to make optimized TO decisions. Our implemented CS follows a SEDA-like architecture [58] when handling incoming updates and sending actions to the flow generating servers. The architecture is subdivided into different stages: http request handling, deep network learning/processing, and response sending. Each stage has its own process(es) and communicate through queues to pass required information to the next stage. Such an approach ensures that multiple cores of the CS server are involved in handling the requests from the hosts and load is distributed. The multi-processing architecture has been adopted due to the Global lock problem [24] in the CPython implementation of the Python programming language. The states and actions are encapsulated at the CS as an "environment" (similar to [47]), with which the RL agents can interact directly and programmatically.

5.2.1 sRLA. As discussed in §4.1, we use Keras to implement the sRLA running the DDPG algorithm with the aforementioned DNNs (actor, critic, target actor, and target critic).

Actors: The actors have two fully-connected hidden layers with 600 and 600 neurons, respectively, and the output layer with $K{-}1$ output units (one for each threshold). The input layer takes states (700 features per-server (m_s=100)) and outputs MLFQ thresholds for a host server for time step t.

Critics: The critics are implemented with three hidden layers, thus the networks are a bit more complicated as compared to the actor network. Since the critic is supposed to 'criticize' the actor for bad decisions and 'compliment' for good ones, the critic neural network also takes as its input the outputs of the actor. However, as [53] suggests, the actor outputs are not direct inputs, but are only fed into the critic's network at a hidden layer. Therefore, the critic has two hidden layers same as the actor and one extra hidden layer which concatenates the actor's outputs with the outputs of its own second hidden layer, resulting in one additional hidden layer. This hidden layer eventually is fed into the output layer consisting of one output unit - approximated value for the observed/received state.

The neural networks are trained on a batch of observations periodically by sampling from a buffer of experience: $\{s_t, a_t, r_t, s_{t+1}\}$. The training process is described in Algorithm (1).

5.2.2 lRLA. For lRLA, we also use Keras to implement the PG algorithm with a fully connected NN with 10 hidden layer of 300 neurons. The RL agent takes a state (136 features per-server (n_l=11, m_l=10)) and outputs probabilities for the actions for all the active flows.

Summary The hyper-parameters (structure, number of layer, height, and width of DNN) are chosen based on a few empirical training sessions. Our observation is that more complicated DNNs with more hidden layers and more parameters took longer to train and did not perform much better than the chosen topologies. Overall, we find that such RLA configurations leads to good system performance and is rather reasonable considering the importance of computation delay, as we reveal next in the evaluations.

6 EVALUATION

In this section, we evaluate the performance of AuTO using real testbed experiments. We seek to understand: 1) With stable traffic (flow size distribution and traffic load are fixed), how does AuTO compare to standard heuristics? 2) For varying traffic characteristics, can AuTO adapt? 3) how fast can AuTO respond to traffic dynamics? 4) what are the performance overheads and overall scalability?

Summary of results (grouped by scenarios):

- **Homogeneous:** For traffic with fixed flow size distribution and load, AuTO-generated thresholds converge, and demonstrate similar or better performance compared to standard heuristics, with up to 48.14% average FCT reduction.

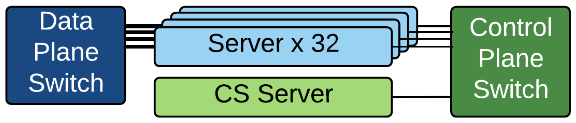

Figure 7: Testbed topology.

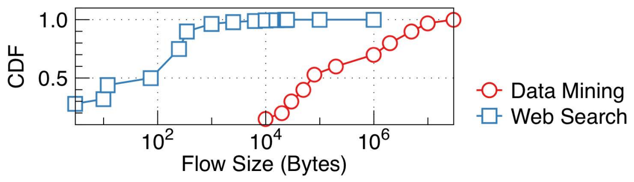

Figure 8: Traffic distributions in evaluation.

- **Spatially Heterogeneous:** We divide the servers into 4 clusters; each is configured to generate traffic with different flow size distribution and load. In these experiments, AuTO-generated thresholds also converge, with up to 37.20% average FCT reduction.
- **Spatially & Temporally Heterogeneous:** Building upon the above scenario, we then change the flow size distribution and load periodically for each cluster. For time-varying flow size distributions and traffic load, AuTO exhibits learning and adaptation behavior. Compared to fixed heuristics that excel only for certain combinations of traffic settings, AuTO demonstrates steady performance improvement across all combinations.
- **System Overhead:** The current AuTO implementation can respond to state updates within 10ms on average. AuTO also exhibits minimal end-host overhead in terms of CPU utilization and throughput degradation.

Setting We deploy AuTO on a small-scale testbed (Figure 7) that consists of 32 servers. Our switch supports ECN and strict priority queuing with at most 8 class of service queues[7] Each server is a Dell PowerEdge R320 with a 4-core Intel E5-1410 2.8GHz CPU, 8G memory, and a Broadcom BCM5719 NetXtreme Gigabit Ethernet NIC with 4x1Gbps ports. Each server runs 64-bit Debian 8.7 (3.16.39-1 Kernel). By default, advanced NIC offload mechanisms are enabled to reduce the CPU overhead. The base round-trip time (RTT) of our testbed is 100us.

We adopt the traffic generator [20] used in prior works [2, 7, 9, 15] that produces network traffic flows based on given flow size distribution and traffic load. We use two realistic workloads (Figure 8): web search workload [8] and data mining workload [10]. 15 servers hosting flow generators are called application servers, and the remaining one hosts

[7]As in most production datacenters [8, 10, 14], some queues are reserved for other services, such as latency-sensitive traffic and management traffic[15].

the CS. Each application server is connected to a data plane switch using 3 of its ports, as well as to a control plane switch to communicate with the CS server using the remaining port. The 3 ports are configured to different subnets, forming 3 paths between any pair of application servers. Both switches are Pronto-3297 48-port Gigabit Ethernet switch. States and actions are sent on the control plane switch (Figure 7).

Comparison Targets We compare with two popular heuristics in flow scheduling: Shortest-Job-First (SJF), and Least-Attained-Service-First (LAS). The main difference between the two is that SJF schemes [1, 4, 29] require flow size at the start of a flow, while LAS schemes [8, 14, 43] do not. For these algorithms to work, sufficiently enough data should be collected before calculating their parameters (thresholds). The shortest period to collect enough flow information to form an accurate and reliable flow size distribution is an open research problem [9, 14, 21, 34], and we note that previously reported distributions are all collected over periods of at least weeks (Figure 8), which indicates the turn-around time are also at least weeks for these algorithms.

In the experiments, we mainly compare with quantized version of SJF and LAS with 4 priority levels. The priority levels are enforced both in the server using Linux `qdisc` [23] and in the data plane switch using strict priority queueing [8]:

- **Quantized SJF (QSJF):** QSJF has three thresholds: α_0,α_1,α_2. We can obtain flow size from the flow generator at its start. For flow size s, if $x \leq \alpha_0$, it is given highest priority; if $x \in (\alpha_0, \alpha_1]$, it is given the second priority; and so on. In this way, shorter flows are given higher priority, similar to SJF.
- **Quantized LAS (QLAS):** QLAS also has thresholds: β_0,β_1,β_2. All the flows are given high priority at the start. If a flow sends more than β_i bytes, it is then demoted to the $(i+1)$-th priority. In this way, longer flows gradually drop to lower priorities.

The thresholds for both schemes can be calculated using methods described in [14] for "type-2/3 flows", and they are dependent on the flow size distribution and traffic load. In each experiment, unless specified, we use the thresholds calculated for DCTCP distribution at 80% load (i.e. the total sending rate is at 80% of the network capacity).

6.1 Experiments

6.1.1 Homogeneous traffic. In these scenarios, the flow size distribution and the load generated from all 32 servers are fixed. We choose Web Search (WS) and Data Mining (DM) distributions at 80% load. These two distributions represent different group of flows: a mixture of short and long flows (WS) and a set of short flows (DM). The average and 99th percentile (p99) FCT are shown in Figure 9. We train AuTO for 8 hours and use the trained DNNs to schedule flows for another hour (shown in Figure 9 as AuTO).

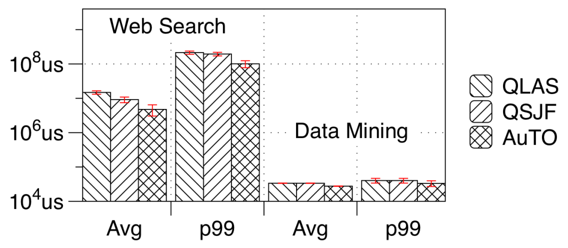

Figure 9: Homogeneous traffic: Average and p99 FCT.

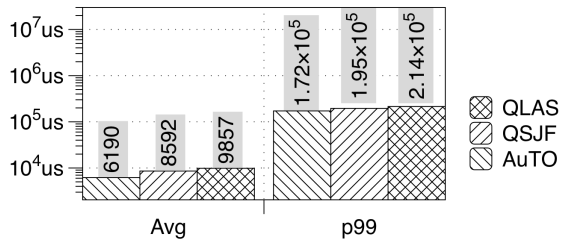

Figure 10: Spatially heterogeneous traffic: Average and p99 FCT.

We make the following observations:

- For a mixture of short and long flows (WS), AuTO outperforms the standard heuristics, achieving up to 48.14% average FCT reduction. This is because it can dynamically change priority of long flows, avoiding the starvation problem in the heuristics.
- For distribution with mostly short flows (DM), AuTO performs similar to the heuristics. Since AuTO also gives any flow highest priority when it starts, AuTO performs almost the same as QLAS.
- Training the RL network results in average FCT reduction of 18.31% and 4.12% for WS&DM distribution respectively, which demonstrates AuTO is capable to learn and adapt to traffic characteristics overtime.
- We further isolate the incast traffic [16] from the collected traces, and we find that they are almost the same with both QLAS and QSJF. This is because incast behavior is best handled by the congestion control and parameter setting. DCTCP [3], which is the transport we used in the experiments, already handles incast very well with appropriate parameter settings [3, 9].

6.1.2 Spatially heterogeneous traffic. We proceed to divide the servers into 4 clusters to create spatially heterogeneous traffic. We configure the flow generators in each cluster with different distribution and load pairs: <WS, 60%>, <WS, 80%>, <DM, 60%>, <DM, 60%>. We use AuTO to control all 4 clusters, and plot the average and p99 FCTs in Figure 10. For the heuristics, we compute the thresholds for

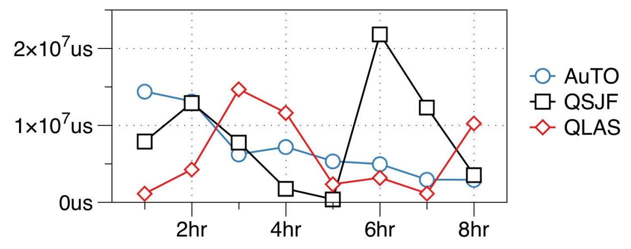

Figure 11: Dynamic scenarios: average FCT.

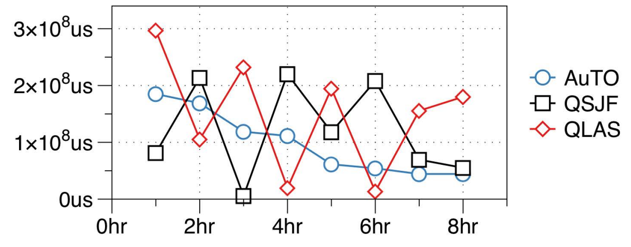

Figure 12: Dynamic scenarios: p99 FCT.

each cluster individually according to its distribution and load. We observe similar results compared to the homogeneous scenarios. Compared to QLAS(QSJF), AuTO is shown to reduce the average FCT by 37.20%(27.95%) and p99 FCT by 19.78%(11.98%). This demonstrates that AuTO can adapt to spatial traffic variations.

6.1.3 Temporally & spatially heterogeneous traffic. In these scenarios, we change the flow size distribution and network load every hour: The load value is chosen from {60%,70%,80%}, and the distribution is randomly chosen from the ones in Figure 8. We ensure that the same distribution/load does not appear in consecutive hours. The experiment runs for 8 hours.

The average and p99 FCTs are plotted against time in Figure 11&12. We can see:

- For heuristics with fixed parameters, when the traffic characteristics match the parameter setting, both average and p99 FCTs outperform the other schemes. But when mismatch occurs, the FCTs sharply drop. This shows that heuristics with fixed parameter setting cannot adapt to dynamic traffic well. Their parameters are usually chosen to perform well in the average case, but in practice, the traffic characteristics always change [8].
- AuTO is shown to steadily learn and adapt across time-varying traffic characteristics, in the last hour, AuTO achieves 8.71% (9.18%) reduction in average (p99) FCT compared to QSJF. This is because that AuTO, using 2 DRL agents, can dynamically change the priorities of flows in different environments to achieve better performance. Without any human involvement, this process can be done quickly and scalably.

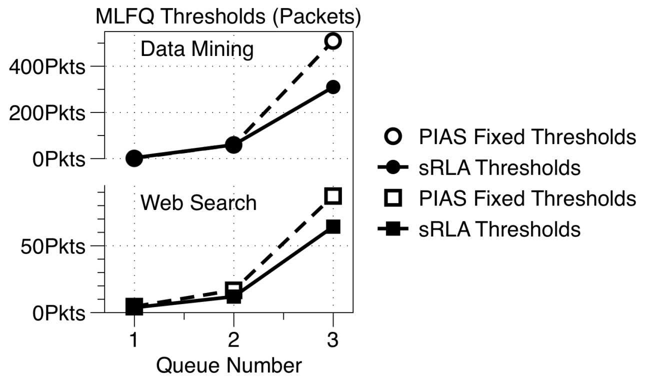

Figure 13: MLFQ thresholds from sRLA vs. optimal thresholds.

Considering AuTO, a constant decline in FCTs indicates learning behavior, which eventually, as we have discussed in §4, lead to convergence to a local optimum for the dynamic traffic generation process. Figure 11&12 confirms our assumption that datacenter traffic scheduling can be converted into a RL problem and DRL techniques (§4) can be applied to solve it.

6.2 Deep Dive

In the following, we inspect the design components of AuTO.

6.2.1 Optimizing MLFQ thresholds using DRL. We first examine the MLFQ thresholds generated by sRLA. In Figure 13, we compare the MLFQ thresholds generated by sRLA and those by an optimizer [9, 14]. We obtain a set of 3 thresholds (for 4 queues) from sRLA in CS after 8 hours of training for each flow size distribution at 60% load. We observe that both sets of thresholds are similar in the thresholds of the first 3 queues, and the main difference is in the last queue. For example, the last sRLA threshold (α_3) for Web Search distribution is 64 packets, while α_3 from optimizer is 87 packets. The same is true for Data Mining distribution. However, the discrepancy does not reflect in significant difference in terms of performance. We plot the average and p99 FCT results for both sets of thresholds in Figure 14&15. The results are grouped by flow size. For sRLA generated thresholds and optimizer-generated thresholds, we observe that the difference in FCT is small in all groups of flow sizes. We conclude that, after 8 hours of training, sRLA generated thresholds are similar to optimizer-generated ones in terms of performance.

6.2.2 Optimizing Long Flows using DRL. Next we look at how lRLA optimizes long flows. During the experiments in §6.1.3, we log the number of long flows on each link for 5 minutes in lRLA. Denote L as the set of all links, $N_l(t)$ as the number of long flows on link $l \in L$ at time t, and $N(t)=\{N_l(t), \forall l\}$. We plot $max(N(t))-min(N(t)), \forall t$ in Figure16, which is the

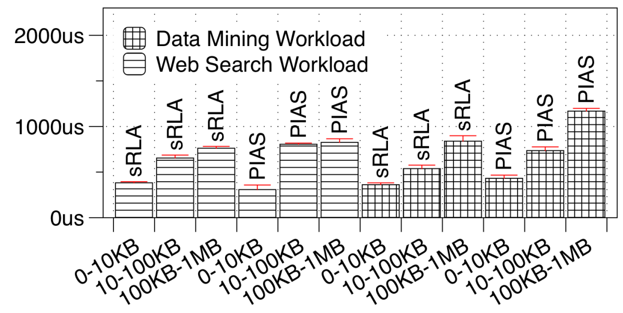

Figure 14: Average FCT using MLFQ thresholds from sRLA vs. optimal thresholds.

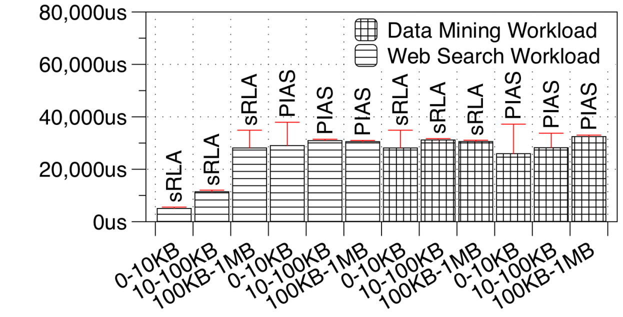

Figure 15: p99 FCT using MLFQ Thresholds from sRLA vs. optimal thresholds.

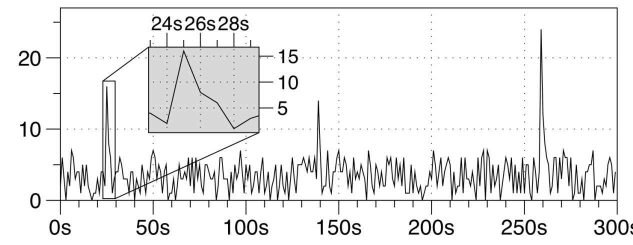

Figure 16: Load balancing using lRLA (PG algorithm): difference in number of long flows on links.

difference in number of long flows on the link that have the most long flows and the link that have the least. This metric is an indicator of load imbalance. We observe that this metric is less than 10 most of the time. When temporary imbalance occurs, as shown in the magnified portion of Figure 16 (from 24s to 28s), lRLA reacts to the imbalance by routing the excess flows onto the less congested links. This is because, as we discussed in §2.2, the reward of the PG algorithm is directly linked to the throughput: when long flows share a link, the total throughput is less than when they are using different links. lRLA is rewarded when it places long flows on different links, thus it learns to load balance long flows.

6.2.3 System Overhead. We proceed to investigate the performance and overheads of AuTO modules. First, we look

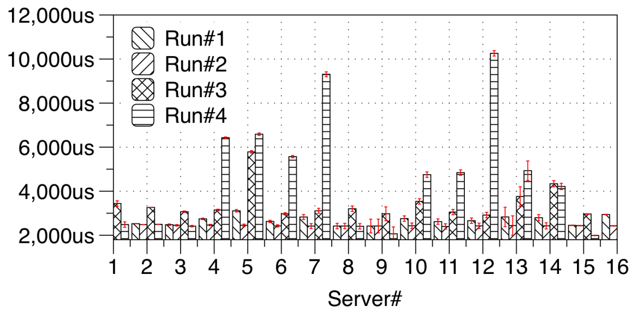

Figure 17: CS response latency: Traces from 4 runs.

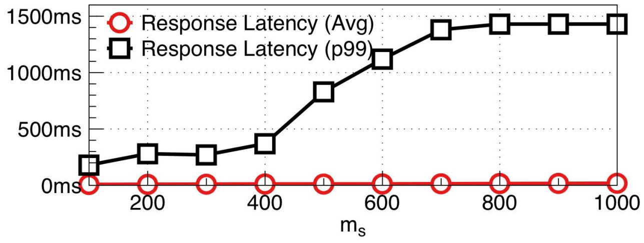

Figure 18: CS response latency: Scaling short flows (m_s)

at the response latency of CS, as well as its scalability. Then we examine the overheads of the end-host modules in PS.

CS Response Latency During experiments, response delay of the CS server (Figure 17) is measured as follows: t_u is the time instant of CS receiving an update from one server, and t_s is the time instant of CS sending the action to that server, so the response time is $t_s - t_u$. This metric directly shows how fast can the scheduler adapt to traffic dynamics reported by PS. We observe that CS can respond to an update within 10ms on average for our 32-server testbed. This latency is mainly due to the computation overhead of DNN, as well as the queueing delay of servers' updates at CS. AuTO currently only uses CPU. To reduce this latency, one promising direction is CPU-GPU hybrid training and serving [46], where CPUs handle the interaction with the environment, while GPUs train the models in the background.

Response latency also increases with computation complexity of DNN. In AuTO, the network size is defined by $\{n_l, m_l, m_s\}$. Since long flows are few, the increment of n_l, m_l are expected to be moderate even for datacenters with high load. We increase $\{n_l, m_l\}$ from {11,10} to {1000,1000}, and find the average response time for lRLA becomes 81.82ms (median 25.14ms). However, m_s can increase significantly in high load scenarios, and we conduct an experiment to understand the impact on response latency of sRLA. In Figure 18, we vary m_s from 100 (used in the above experiments) to 1000, and measure the response latency. We find that the average response time only slightly increase for larger m_s. This is because, m_s determines the input layer size, which only affects the matrix size of link weights between the input layer and the first hidden layer. Moreover, if in the future AuTO employs more complex DNNs, we can reduce the response latency with parallelization techniques proposed for DRL [6, 25, 27, 39].

CS Scalability Since our testbed is small, the NIC capacity of CS server is not fully saturated. Using the same parameter settings as in the experiments (§6.1.3), the bandwidth of monitoring flows is 12.40Kbps per server. Assuming 1Gbps network interface, the CS server should support 80.64K servers, which should be able to handle the servers in typical production datacenters [50, 54]. We also intend to achieve higher scalability in the following ways: 1) 1Gbps link capacity is chosen to mimic the experiment environment, and in current production datacenters, the typical bandwidth of of server is usually 10Gbps or above [50, 54]; 2) we expect CS to have GPUs or other hardware accelerators [46], so that the computation can complete faster; 3) we can reduce the bandwidth requirement of monitoring flows by implementing compression and/or sampling in PS.

PS Overhead End-host overhead refers to the additional work done for each flow to collect information and enforce actions. The overhead can be measured by CPU utilization and reduction in throughput when PS is running. We measured both metric during the experiments, and rerun the flows without enabling MM and EM. We find that the throughput degradation is negligible, and the CPU utilization is less than 1%. Since EM is similar to the tagging module in PIAS [8], our results confirm that both the throughput and CPU overhead are also minimal as PIAS.

7 RELATED WORKS

There have been continuous efforts on TO in datacenters. In general, three categories of mechanisms are explored: load balancing, congestion control, and flow scheduling. We focus on the proposals using machine learning techniques.

Routing and load balancing on the Internet have employed RL-based techniques [13] since 1990s. However, they are switch-based mechanisms, which are difficult to implement at line rate in modern datacenters with >10 GbE links. RL techniques are also used for adaptive video streaming in Pensieve [37].

Machine learning techniques [59] have been used to optimize parameter setting for congestion control. The parameters are fixed given a set of traffic distributions, and there is no adaptation of parameters at run-time.

For flow scheduling, CODA [61] uses unsupervised clustering algorithm to identify flow information without application modifications. However, its scheduling decisions are still made by a heuristic algorithm with fixed parameters.

8 CONCLUDING REMARKS

Inspired by recent successes of DRL techniques in solving complex online control problems, in this paper, we attempted to enable DRL for automatic TO. However, our experiments show that the latency of current DRL systems is the major obstacle to TO at the scale of current datacenters. We solved this problem by exploiting long-tail distribution of datacenter traffic. We developed a two-level DRL system, AuTO, mimicking the Peripheral & Central Nervous Systems in animals, to solve the scalability problem. We deployed and evaluated AuTO on a real testbed, and demonstrated its performance and adaptiveness to dynamic traffic in datacenters. AuTO is a first step towards automating datacenter TO, and we hope many software components in AuTO can be reused in other DRL projects in datacenters.

For future work, while this paper focuses on employing RL to perform flow scheduling and load balancing, RL algorithms for congestion control and task scheduling can be developed. In addition to the potential improvements we mentioned in §5&6, we also plan to investigate applications of RL beyond datacenters, such as WAN bandwidth management.

Acknowledgements: This work is supported in part by Hong Kong RGC ECS-26200014, GRF-16203715, CRF-C703615G, & China 973 Program No.2014CB340300. We thank the anonymous SIGCOMM reviewers and our shepherd David Andersen for their constructive feedback and suggestions.

REFERENCES

[1] Mohammad Al-Fares, Sivasankar Radhakrishnan, Barath Raghavan, Nelson Huang, and Amin Vahdat. [n. d.]. Hedera: Dynamic Flow Scheduling for Data Center Networks. In *USENIX NSDI '10*.
[2] Mohammad Alizadeh, Tom Edsall, Sarang Dharmapurikar, Ramanan Vaidyanathan, Kevin Chu, Andy Fingerhut, Francis Matus, Rong Pan, Navindra Yadav, George Varghese, et al. [n. d.]. CONGA: Distributed congestion-aware load balancing for datacenters. In *ACM SIGCOMM '14*.
[3] Mohammad Alizadeh, Albert Greenberg, David A Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. [n. d.]. Data Center TCP (dctcp). In *ACM SIGCOMM '10*.
[4] Mohammad Alizadeh, Shuang Yang, Milad Sharif, Sachin Katti, Nick McKeown, Balaji Prabhakar, and Scott Shenker. [n. d.]. pFabric: Minimal Near-Optimal Datacenter Transport. In *ACM SIGCOMM '13*.
[5] Behnaz Arzani, Selim Ciraci, Boon Thau Loo, Assaf Schuster, and Geoff Outhred. 2016. Taking the Blame Game out of Data Centers Operations with NetPoirot. In *ACM SIGCOMM'16*.
[6] Mohammad Babaeizadeh, Iuri Frosio, Stephen Tyree, Jason Clemons, and Jan Kautz. [n. d.]. Reinforcement learning through asynchronous advantage actor-critic on a gpu. In *ICLR'16*.
[7] Wei Bai, Kai Chen, Li Chen, Changhoon Kim, and Haitao Wu. [n. d.]. Enabling ECN over Generic Packet Scheduling. In *ACM CoNEXT'16*.
[8] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Weicheng Sun. [n. d.]. Information-Agnostic Flow Scheduling for Data Center Networks. In *USENIX NSDI '15*.

[9] Wei Bai, Li Chen, Kai Chen, Dongsu Han, Chen Tian, and Hao Wang. 2017. PIAS: Practical Information-Agnostic Flow Scheduling for Commodity Data Centers. *IEEE/ACM Transactions on Networking (TON)* 25, 4 (2017), 1954–1967.

[10] Wei Bai, Li Chen, Kai Chen, and Haitao Wu. [n. d.]. Enabling ECN in Multi-Service Multi-Queue Data Centers. In *USENIX NSDI '16*.

[11] Theophilus Benson, Aditya Akella, and David Maltz. [n. d.]. Network Traffic Characteristics of Data Centers in the Wild. In *ACM IMC'10*.

[12] Shalabh Bhatnagar, Mohammad Ghavamzadeh, Mark Lee, and Richard S Sutton. 2008. Incremental Natural Actor-Critic Algorithms. (2008).

[13] Justin A Boyan, Michael L Littman, et al. 1994. Packet routing in dynamically changing networks: A reinforcement learning approach. *Advances in neural information processing systems* (1994).

[14] Li Chen, Kai Chen, Wei Bai, and Mohammad Alizadeh. [n. d.]. Scheduling Mix-flows in Commodity Datacenters with Karuna. In *ACM SIGCOMM '16*.

[15] Li Chen, Jiacheng Xia, Bairen Yi, and Kai Chen. [n. d.]. PowerMan: An Out-of-Band Management Network for Datacenters Using Power Line Communication. In *USENIX NSDI'18*.

[16] Yanpei Chen, Rean Griffith, Junda Liu, Randy H Katz, and Anthony D Joseph. 2009. Understanding TCP incast throughput collapse in datacenter networks. *The 1st ACM workshop on Research on enterprise networking* (2009).

[17] Francois Chollet. [n. d.]. Keras Documentation. https://keras.io/. ([n. d.]). (Accessed on 04/18/2017).

[18] Mosharaf Chowdhury and Ion Stoica. [n. d.]. Efficient Coflow Scheduling Without Prior Knowledge. In *ACM SIGCOMM '15*.

[19] Mosharaf Chowdhury, Yuan Zhong, and Ion Stoica. [n. d.]. Efficient coflow scheduling with Varys. In *ACM SIGCOMM '14*.

[20] Cisco. [n. d.]. Simple client-server application for generating user-defined traffic patterns. https://github.com/datacenter/empirical-traffic-gen. ([n. d.]). (Accessed on 04/24/2017).

[21] Ralph B Dell, Steve Holleran, and Rajasekhar Ramakrishnan. 2002. Sample size determination. *ILAR journal* (2002).

[22] Nathan Farrington, George Porter, Sivasankar Radhakrishnan, Hamid Hajabdolali Bazzaz, Vikram Subramanya, Yeshaiahu Fainman, George Papen, and Amin Vahdat. [n. d.]. Helios: A Hybrid Electrical/Optical Switch Architecture for Modular Data Centers. In *ACM SIGCOMM'10*.

[23] Linux Foundation. [n. d.]. Priority qdisc - Linux man page. https://linux.die.net/man/8/tc-prio. ([n. d.]). (Accessed on 04/17/2017).

[24] Python Software Foundation. [n. d.]. Global Interpreter Lock. https://wiki.python.org/moin/GlobalInterpreterLock. ([n. d.]). (Accessed on 04/18/2017).

[25] Kevin Frans and Danijar Hafner. 2016. Parallel Trust Region Policy Optimization with Multiple Actors. (2016).

[26] Albert Greenberg, James R. Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A. Maltz, Parveen Patel, and Sudipta Sengupta. [n. d.]. VL2: A Scalable and Flexible Data Center Network. In *ACM SIGCOMM'09*.

[27] Shixiang Gu, Ethan Holly, Timothy Lillicrap, and Sergey Levine. [n. d.]. Deep reinforcement learning for robotic manipulation with asynchronous off-policy updates. In *Proceedings of Robotics and Automation (ICRA), 2017 IEEE International Conference on*.

[28] W. K. Hastings. 1970. *Biometrika* (1970). http://www.jstor.org/stable/2334940

[29] Chi-Yao Hong, Matthew Caesar, and P Godfrey. [n. d.]. Finishing flows quickly with preemptive scheduling. In *ACM SIGCOMM '12*.

[30] C. Hopps. 2000. Analysis of an Equal-Cost Multi-Path Algorithm. *RFC 2992* (2000).

[31] Kurt Hornik. [n. d.]. Approximation Capabilities of Multilayer Feedforward Networks. *Neural Netw., 1991* ([n. d.]).

[32] Shuihai Hu, Kai Chen, Haitao Wu, Wei Bai, Chang Lan, Hao Wang, Hongze Zhao, and Chuanxiong Guo. 2016. Explicit path control in commodity data centers: Design and applications. *IEEE/ACM Transactions on Networking* (2016).

[33] Srikanth Kandula, Sudipta Sengupta, Albert Greenberg, Parveen Patel, and Ronnie Chaiken. [n. d.]. The Nature of Datacenter Traffic: Measurements and Analysis. In *ACM IMC'09*.

[34] Russell V Lenth. 2001. Some practical guidelines for effective sample size determination. *The American Statistician* (2001).

[35] Timothy P. Lillicrap, Jonathan J. Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. 2015. Continuous control with deep reinforcement learning. *CoRR* abs/1509.02971 (2015). arXiv:1509.02971 http://arxiv.org/abs/1509.02971

[36] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. [n. d.]. Resource Management with Deep Reinforcement Learning. In *ACM HotNets '16*.

[37] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. [n. d.]. Neural Adaptive Video Streaming with Pensieve. In *ACM SIGCOMM '17*.

[38] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner. 2008. OpenFlow: Enabling innovation in campus networks. *ACM CCR* (2008).

[39] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. [n. d.]. Asynchronous methods for deep reinforcement learning. In *ICML'16*.

[40] 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).

[41] 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).

[42] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, William Paul, Michael I Jordan, and Ion Stoica. 2017. Ray: A Distributed Framework for Emerging AI Applications. *arXiv preprint arXiv:1712.05889* (2017).

[43] Ali Munir, Ihsan A Qazi, Zartash A Uzmi, Aisha Mushtaq, Saad N Ismail, M Safdar Iqbal, and Basma Khan. [n. d.]. Minimizing flow completion times in data centers. In *IEEE INFOCOM '13*.

[44] Netfilter.Org. [n. d.]. The netfilter.org project. https://www.netfilter.org/. ([n. d.]). (Accessed on 04/17/2017).

[45] NVIDIA. [n. d.]. Deep Learning Frameworks. https://developer.nvidia.com/deep-learning-frameworks. ([n. d.]). (Accessed on 04/18/2017).

[46] NVlabs. [n. d.]. Hybrid CPU/GPU implementation of the A3C algorithm for deep reinforcement learning. https://github.com/NVlabs/GA3C. ([n. d.]). (Accessed on 06/13/2018).

[47] OpenAI. [n. d.]. OpenAI Gym. https://gym.openai.com/. ([n. d.]). (Accessed on 04/24/2017).

[48] Adam Paszke, Sam Gross, Soumith Chintala, and Gregory Chanan. 2017. Pytorch. (2017).

[49] Yang Peng, Kai Chen, Guohui Wang, Wei Bai, Ma Zhiqiang, and Lin Gu. [n. d.]. HadoopWatch: A First Step Towards Comprehensive Traffic Forcecasting in Cloud Computing. In *IEEE INCOCOM '14*.

[50] Arjun Roy, Hongyi Zeng, Jasmeet Bagga, George Porter, and Alex C Snoeren. [n. d.]. Inside the Social Network's (Datacenter) Network. In *ACM SIGCOMM'15*.

[51] John Schulman, Sergey Levine, Philipp Moritz, Michael I. Jordan, and Pieter Abbeel. 2015. Trust Region Policy Optimization. *CoRR* (2015).

[52] David Silver, Aja Huang, Chris J Maddison, Arthur Guez, Laurent Sifre, George Van Den Driessche, Julian Schrittwieser, Ioannis Antonoglou, Veda Panneershelvam, Marc Lanctot, et al. 2016. Mastering the game of Go with deep neural networks and tree search. *Nature* (2016).

[53] David Silver, Guy Lever, Nicolas Heess, Thomas Degris, Daan Wierstra, and Martin Riedmiller. 2014. Deterministic Policy Gradient Algorithms. In *Proceedings of the 31st International Conference on International Conference on Machine Learning - Volume 32 (ICML'14)*. JMLR.org, I–387–I–395. http://dl.acm.org/citation.cfm?id=3044805.3044850

[54] Arjun Singh, Joon Ong, Amit Agarwal, Glen Anderson, Ashby Armistead, Roy Bannon, Seb Boving, Gaurav Desai, Bob Felderman, Paulie Germano, et al. [n. d.]. Jupiter Rising: A Decade of Clos Topologies and Centralized Control in Google's Datacenter Network. In *ACM SIGCOMM'15*.

[55] Richard S. Sutton and Andrew G. Barto. 1998. *Introduction to Reinforcement Learning*.

[56] Richard S Sutton, David A. McAllester, Satinder P. Singh, and Yishay Mansour. 2012. Policy Gradient Methods for Reinforcement Learning with Function Approximation. (2012).

[57] TensorFlow. [n. d.]. API Documentation: TensorFlow. https://www.tensorflow.org/api_docs/. ([n. d.]). (Accessed on 04/18/2017).

[58] Matt Welsh, David Culler, and Eric Brewer. [n. d.]. SEDA: An Architecture for Well-conditioned, Scalable Internet Services. In *SOSP '01*.

[59] Keith Winstein and Hari Balakrishnan. [n. d.]. Tcp ex machina: Computer-generated congestion control. In *ACM SIGCOMM '13*.

[60] Neeraja J. Yadwadkar, Ganesh Ananthanarayanan, and Randy Katz. [n. d.]. Wrangler: Predictable and Faster Jobs Using Fewer Resources. In *SOCC '14*.

[61] Hong Zhang, Li Chen, Bairen Yi, Kai Chen, Mosharaf Chowdhury, and Yanhui Geng. [n. d.]. CODA: Toward automatically identifying and scheduling coflows in the dark. In *ACM SIGCOMM '16*.