Park: An Open Platform for Learning Augmented Computer Systems

Hongzi Mao 1 Akshay Narayan 1 Parimarjan Negi 1 Hanrui Wang 1 Jiacheng Yang 1 Haonian Wang 1
Mehrdad Khani 1 Songtao He 1 Ravichandra Addanki 1 Ryan Marcus 1 Frank Cangialosi 1 Wei-Hung Weng 1
Song Han 1 Tim Kraska 1 Mohammad Alizadeh 1

Abstract

This paper presents Park, an open extensible platform that uses a common interface to connect to a suite of real world computer systems for RL augmented optimizations. These systems cover a wide spectrum of problems, including both global vs. distributed control, and fast control loop vs. long term planning. This dataset unveils unique challenges that the existing off-the-shelf RL techniques cannot solve. The challenges occur in the representation and search of the state-action space, the special property of the decision process and the reality gap between simulations and actual systems. To understand the effect of these challenges, we benchmark several existing RL algorithms in Park with comparing heuristic baselines.

1. Introduction

Deep reinforcement learning (RL) has emerged as a general and powerful approach in complex games (Mnih et al., 2015; Silver et al., 2018; Tian et al., 2017; OpenAI, 2018; Vinyals et al., 2019) and robotics (Rajeswaran et al., 2017; OpenAI et al., 2018; Hwangbo et al., 2019). Despite these rapid algorithmic advancements, applications of modern RL remain largely confined to simulated and controlled environments. There has thus been a surge of interest seeking realistic impact for RL in practical domains (Li, 2019). To this end, optimizing computer systems presents a new arena bridging RL algorithms and real-world applications.

In many computer systems, RL is readily applicable since the optimization problems involve control loops that naturally fit into the Markov decision process (MDP) structure. Also, production software systems have faster interaction times compared to traditionally studied real-world robotic environments. It is therefore easier to generate abundant data to explore and train RL models, which mitigates one of the shortcomings of model-free RL approaches in practice — their high sample complexity (Amodei & Hernandez, 2018). Moreover, because the optimization problems in some computer systems are hard to model accurately, state-of-the-art schemes are forced to rely on suboptimal, human-engineered heuristics. This leaves large room for performance improvement (Mirhoseini et al., 2017).

This natural fit and potential benefit have attracted a surge of recent interest in computer system community to apply RL to multiple applications, including datacenter job scheduling (Mao et al., 2018), network congestion control (Jay et al., 2018) and database query optimization (Krishnan et al., 2018; Marcus et al., 2019). In contrast to RL in games or robotics, however, there is relatively little attention from the machine learning community focusing on the algorithmic development for these problems. The major drawback is the lack of an easy-to-use platform to develop RL algorithms for a wide range of systems problems. As a result, the common experience in designing RL for a system detours into a time consuming process that involves (1) understanding the specific system problem and its implementation, (2) collecting real-world traces, and (3) creating an interface between the system and the RL agents.

The primary goal of this paper is to make real-world systems problems easily accessible to the machine learning community. We present Park, an open extensible platform that uses a common RL interface to connect to a suite of representative computer system environments. For each environment, Park defines the MDP formulation (especially the event to trigger an MDP step) along with the state-action space and reward function. This standardization allows machine learning researchers to focus on the core algorithmic and learning challenges, without having to deal with low level system implementation issues. At the same time, this interface allows system operators to easily compare the performance among different proposed learned agents on a common standard.

In Park, the environments encompass three aspects of systems problems (§3). First, these systems range from global single-agent control (e.g., scheduling jobs in a Spark datacenter) to distributed multi-agent control (e.g., network congestion control). Second, some systems feature a fast control loop (e.g., memory cache admission) while others in-
volve long term planning (e.g., database query optimization). Third, Park uses faithful simulations when the the dynamics of a system are well understood (e.g., packet forwarding in switch scheduling) and, in other cases, it uses real system implementations to capture complex system artifacts (e.g., run time variation in Tensorflow device placement).

Importantly, Park exposes unique challenges that prior RL approaches have overlooked (§4). In some systems, the size of the state-action space changes over time, which cannot be expressed by off-the-shelf tensorized state-action encodings. Job scheduling is an example; the scheduling agent must decide where to assign a set of pending jobs while the size of the set is changing as new jobs arrive. These challenges present new opportunities for designing RL algorithms. For example, stochastic system environments have a unique structure where an input process introduces a substantial amount of noise in the RL training process. When the input process can be observed and controlled, this enables the RL agent to design new control variants to reduce the variance and train a more robust policy (Mao et al., 2019).

We benchmark the environments in Park with existing RL methods and provide existing heuristics tailored for each environment as comparing baselines. In §3.4, we present each agent’s training efficiency and final performance compared to the existing baselines. Park is open sourced at https://github.com/park-project/park.

2. Background

RL algorithms enable an agent to learn to make better control decisions for a task through interactions with an environment. At each step $t$, the agent observes some state $s_t$, and takes an action $a_t$. Following the action, the state of the environment transitions to $s_{t+1}$ and the agent receives a reward $r_t$ as feedback. The state transitions and rewards are stochastic and assumed to be Markov: the state transition to $s_{t+1}$ and the reward $r_t$ depend only on the state $s_t$ and the action $a_t$ at step $t$ (i.e., they are conditionally independent of the past).

In the model-free RL setting, the agent can only control its actions; and it has no a priori knowledge of the state transition probabilities or the reward function. For training, standard RL proceeds in episodes. Each episode consists of a sequence of (state, action, reward) observations. The standard goal of RL agent is to maximize the total reward: $E\left[\sum_{t=0}^{T} r_t\right]$, where $T$ is the episode length. In §3.3, we describe how each environment in Park maps to this standard RL framework.

3. Environments

In this section, we first categorize different properties for computer system problems (§3.1). Then we describe the common interface connecting RL to all the systems in Park and we discuss the interface’s extensibility for adding more system environments in the future (§3.2). Finally, we describe the details of each environment in Park, focusing on the MDP abstraction of the system interaction (§3.3).

3.1. Environment Categories

Park provides a wide spectrum of system problems. These problems can be broadly categorized in three axes.

Global vs. distributed control. Some environments in Park provide a global view of the system, in which the agent makes the decisions that affect the entire system as a whole. For example, in Spark job scheduling, agent observes all pending jobs to schedule and it knows the current allocation of the compute resources. In contrast, some other environments have multiple controlling agents running concurrently. Each agent observes only part of the system and all the agents optimize the system collaboratively in a distributed way. Network congestion control is a classic example. Multiple network connection can share a single link. Each connection has an agent controlling the dynamics of packet transfer (e.g., by setting the sending rate). The control policies for all the agents have to compete fairly for the shared link, while maximizing the available bandwidth.

Fast control loop vs. planning. When fitting the optimization problem into the RL framework, some systems have fast control loop with a clear MDP structure. For instance, the bitrate adaptation agent in video streaming controls the bitrate for every video chunk downloads and it receives reward feedback right after the chunk download event. In these problems, it is relatively straightforward to experiment with existing RL implementations. In some other environments, however, the optimization problem involves planning and searching at every action. For example, in database query optimization, making an action for early part of an query requires planning out the remaining query — similar to playing chess or Go (Silver et al., 2018), it is hard to directly make one-shot decisions for partial query without explicit searching (Marcus et al., 2019).

Real system vs. simulation. To maximally present the system artifacts during the optimization process, half of the environments in Park are interacting with real systems. For the systems with well understood dynamics, we provide simulated environment to facilitate easier setup. However, to make the simulation faithfully mimic the real world setting, we power the simulation with real world traces. For example, in memory caching environment, we use an open dataset containing 500 million requests, collected from a public CDN serving top-ten US websites (Berger, 2018).

3.2. Common Interface

In Park, the control agent and the underlying systems are independent. Park connects to an underlying system through
a remote procedure call (RPC) server. Algorithm 1 describes the pseudocode for this interaction. On the system side, when invoking the control agent, the system summarizes the state, reward, done = env.step(action). As a result, this interface hides the system complexity from the RL agent.

Algorithm 1 Interface for real system interaction.

Require: An RPC server that hosts action requests
1: def env.run(agent):
2: while not done:
3: state, reward, done = server.listen()
4: # reward corresponds to the previous action
5: action = agent.get_action(state, reward, done)
6: server.reply(action)

In simulated environments, we can implement a shim interface for the “agent-centric” style advocated by OpenAI Gym (Brockman et al., 2016). Algorithm 2 outlines this option. The agent explicitly steps the environment forward by sending the action to the underlying system through the RPC response (line 3). It then waits on the RPC server for the next action request (line 4). With this interface, we can directly reuse existing RL implementations.

Algorithm 2 Interface for simulated system interaction.

Require: An RPC server that hosts action requests
1: def env.step(action):
2: # OpenAI Gym style of interaction
3: server.reply(action)
4: state, reward, done = server.listen()
5: return state, reward, done

Conceptually, extending this interface to other system environments is straightforward. For a new system, it only needs to specify (1) the state-action space definition (e.g., tensor, graph, powerset, etc.), (2) the event to trigger an MDP step, at which it sends an RPC request and (3) the formula to calculate the reward feedback. From the agent’s perspective, it can use the same RL algorithm for the new environment.

3.3. Detailed Descriptions

In this section, we describe the details of the environments in Park. Table 1 provides an overview.

Adaptive video streaming. The volume of video streaming has reached almost 60% of all the Internet traffic (Sandvine, 2018). Streaming video over variable-bandwidth networks (e.g., cellular network) requires the client to adapt the video...
bitrate to optimize the user experience. In industrial DASH standard (Akamai, 2016), videos are divided into multiple chunks, each of which represents a few seconds of the overall video playback. Each chunk is encoded at several discrete bitrates, where a higher bitrate implies a higher resolution and thus a larger chunk size. For this problem, each MDP episode is a video playback with a particular network trace (i.e., a time series of network throughput). At each step, the agent observes the past network throughput measurement, the current video buffer size, and the remaining portion of the video. The action is the bitrate for the next video chunk. The objective is to maximize the video resolution and minimize the stall (which occurs when download time of a chunk is larger than the current buffer size) and the reward is structured to be a linear combination of selected bitrate and the stall when downloading the corresponding chunk. Prior adaptive bitrate approaches construct heuristic based on the buffer and network observations. For example, a control theoretic based approach (Yin et al., 2015) conservatively estimates the network bandwidth and use model predictive control to choose the optimal bitrate over the near-term horizon. In practice, the network condition is hard to model and estimate, making a fixed, hard-coded model-based approach insufficient to adapt to changing network conditions (Mao et al., 2017; Akhtar et al., 2018).

Spark cluster job scheduling. Efficient utilization of expensive compute clusters matters for enterprises: even small improvements in utilization can save millions of dollars at scale (Barroso et al., 2013). Cluster schedulers are key to realizing these savings. A good scheduling policy packs work tightly to reduce fragmentation (Verma et al., 2014), prioritizes jobs according to high-level metrics such as user-perceived latency (Verma et al., 2015), and avoids inefficient configurations (Ferguson et al., 2012). Since hard-tuning scheduling policies is uneconomic for many organizations, there has been a surge of interest in using RL to generate highly-efficient scheduling policies automatically (Mao et al., 2016; Chen et al., 2018; Mao et al., 2018).

We build our scheduling system on top of the Spark cluster manager (Zaharia et al., 2010). Each Spark job is represented as a DAG of computation stages, which contains identical tasks that can run in parallel. The scheduler maps executors (atomic computation units) to the stages of each job. We modify Spark’s scheduler to consult an external agent at each scheduling event (i.e., each MDP step). A scheduling event occurs when (1) a stage runs out of tasks (i.e., needs no more executors), (2) a stage completes, unlocking the tasks of one or more of its children, or (3) a new job arrives in the system. At each step, the cluster has some available executors and some runnable stages from pending jobs. Thus, the scheduling agent observes (1) the number of tasks remaining in the stage, (2) the average task duration, (3) the number of executors currently working on the stage, (4) the number of available executors, and (5) whether available executors are local to the job. This set of information is embedded as features on each node of the job DAGs. The scheduling action is two-dimensional—(1) which node to work on next and (2) how many executors to assign to the node. We structure the reward at step k as \( r_k = -(t_k - t_{k-1}) J_k \), where \( J_k \) is the number of jobs in the system during the physical time interval \( [t_{k-1}, t_k) \). Sum of such rewards penalize the agent in order to minimize the average job completion time. Park platform supports replaying an one-month industrial workload trace from Alibaba.

SQL Database query optimization. Queries in relational databases often involve retrieving data from multiple tables. The standard abstraction for combining data is through a sequential process that joins entries from two tables based on the provided filters (e.g., actor \( \text{JOIN} \) country \( \text{ON} \) ac-tor.country_id = country.id) at each step. The most important factor that affects the query execution time is the order of joining the tables (Krishnan et al., 2018). While any ordering leads to the same final result, an efficient ordering keeps the intermediate results small, which minimizes the number of entries to read and process. Finding the optimal ordering remains an active research area, because (1) the total number of orderings is exponential in the number of filters and (2) the size of intermediate results depends on hard-to-model relationship among the filters. There have been a few attempts to learn a query optimizer using RL (Krishnan et al., 2018; Ortiz et al., 2018; Marcus et al., 2019).

Building the sequence of joins naturally fits in the MDP formulation. At each step, the agent observes the remaining tables to join as a query graph, where each node represents a table and the edges represent the join filters. The agent then decides which edge to pick (corresponds to a particular join) as an action. Park supports rewards from a cost model (a join cost estimate provided by commercial engines) and the final physical duration. In our implementation, we use Calcite (Begoli et al., 2018) as the query optimization framework, which can serve as a connector to any database management system (e.g., Postgres (PostgreSQL, 2019)).

Network congestion control. Congestion control has been a perennial problem in networking for three decades (Jacobson, 1988), and governs when hosts should transmit packets. Transmitting packets too frequently leads to congestion collapse (affecting all users) (Nagle, 1984) while over-conservative transmission schemes under-utilize the available network bandwidth. Good congestion control algorithms achieve high throughput and low delay while competing fairly for network bandwidth with other flows in the network. Various congestion control algorithms, including learning-based approaches (Dong et al., 2018; Yan et al., 2018; Jay et al., 2018), optimize for different objectives in this design space. It remains an open research
question to design an end-to-end congestion control scheme that can automatically adapt to high-level objectives under different network condition (Shapira & Weinstein, 2017).

We implement this environment using CCP (Narayan et al., 2018), a platform for expressing congestion control algorithms in user-space. At each step, the agent observes the network state, including the throughput and delay. The action is a tuple of pacing rate and congestion window. The action interval is configurable (default interval 10 ms; can also go down to per packet level control). The reward function is adopted from the Copa (Arun & Balakrishnan, 2018) algorithm: \( \log(\text{throughput}) - \log(\text{delay})/2 - \log(\text{lost packets}) \). This environment supports different network traces, from cellular networks to fixed-bandwidth links (emulated by Mahimahi (Netravali et al., 2015)).

**Network active queue management.** In network routers and switches, active queue management (AQM) is a fundamental component that controls the queue size (Athuraliya et al., 2001). It monitors the queue dynamics and decides to drop packets when the queue gets close to full (Floyd & Jacobson, 1993). The goal for AQM is to achieve high throughput and low delay for the packets passing through the queue. Designing a strong AQM policy that achieves this high-level objective for a wide range of network condition can be complex. Standard methods — such as PIE (Hollot et al., 2001), based on PID control (Astrom et al., 2006) — construct a policy for a low-level goal that maintains the queue size at a certain level. In our setting, the agent observes the queue size and network throughput measurement; it then sets the packet drop probability. The action interval is configurable (default interval 10 ms; can also go down to per packet level control). The reward can be configured as a penalty for the difference between observed and target queue size, or a weighted combination of network throughput and delay. Similar to the congestion control environment, we emulate the network dynamics using Mahimahi with a wide range of real-world network traces.

**Circuits Design.** Analog integrated circuits often involve complex non-linear models relating the transistor sizes and the performance metrics. Common practice for optimizing analog circuits relies on expensive simulations and tedious manual tuning from human experts (Razavi, 2002). Prior work has applied Bayesian optimization (Lyu et al., 2018) and evolution strategy (Liu et al., 2010) as general black-box parameter tuning tools to optimize the analog circuit design pipeline. Wang et al. (2019a;b) recently proposed to use RL to end-to-end optimize the circuit performance.

Park supports transistor-level analog circuit design (Razavi, 2002), where the circuit schematic is fixed and the agent decides the component parameters. For each schematic, the agent observes a circuit graph where each node contains the component ID, type (e.g., NMOS or PMOS) and static parameters (e.g., Vth0). The corresponding action is also a graph in which each node must specify the transistor size, capacitance and resistance. Then, the underlying HSPICE circuit simulator (Synopsys, 2019) returns a configurable combination of bandwidth, power and gain as a reward. We refer the readers to Wang et al. (2019a) for more details.

**Tensorflow device placement.** Large scale machine learning applications use distributed training environments, where neural networks are split across multiple GPUs and CPUs (Mirhoseini et al., 2017). A key challenge for distributed training is how to split a large model across heterogeneous devices to speed up training. Determining an optimal device placement is challenging and involves intricate planning, particularly as neural networks grow in complexity and approach device memory limits (Mirhoseini et al., 2018). Motivated by these challenges, several learning based approaches have been proposed (Mirhoseini et al., 2017; 2018; Gao et al., 2018; Addanki et al., 2018).

We build our placement system on top of Tensorflow (Abadi et al., 2016). Each model is represented as a computational graph of neural network operations. A placement scheme maps nodes to the available devices. We formulate the MDP as an iterative process of placement improvement steps (Addanki et al., 2018). At each step, the agent observes an existing placement graph and tries to improve its runtime by updating the placement at a particular node. The state observation is the computation graph of a Tensorflow model, with features attached to each node which include (1) estimated node run time (2) output tensor size (3) current device placement (4) flag of the “current” node (5) flag if previously placed. The action places the current node on a device. Since the goal is to learn a policy that can iteratively improve placements, the reward \( r_i = -(t_i - t_{i-1}) \), where \( t_i \) is the runtime of the placement at step \( i \). Park supports optimizing placements for graphs with hundreds of nodes across a configurable number of devices. To speedup training, Park also provides a simulator for the runtime of a device placement (based on measurements from prior executions, see Appendix A4 in Addanki et al. (2018) for details).

**CDN memory caching.** In today’s Internet, the majority of content is served by Content Delivery Networks (CDNs) (Nygren et al., 2010). CDNs enable fast content delivery by caching content in servers near the users. To reduce the content retrieval cost from a data center, CDNs aim to maximize the fraction of bytes served locally from the cache, known as the byte hit ratio (BHR) (Hasan et al., 2014). The admission control problem of CDN caching fits naturally to the MDP setting. At each step when an uncached object arrives in the

---

1See Table 2 of Narayan et al. (2018) for full list.
CDN, the agent observes the object size, the time since the previous visit (if available) and the remaining CDN cache size. The agent then takes an action to admit or drop the uncached object. To maximize BHR, the reward at each step is the total byte hits since the last action (i.e., counting the size of cached objects served). Coupled with the admission policy is an eviction policy that decides which cached object to remove in order to make room for a newly admitted object. By default, our environment uses a fixed least-recently-used policy for object eviction. The environment also supports training an eviction agent together with the admission agent (e.g., via multi-agent RL). Our setup includes a real world trace with 500 million requests collected from a public CDN serving top-ten US websites (Berger, 2018).

**Account region assignment.** Social network websites reduce access latency by storing data on servers near their users. For each user-uploaded piece of content, the service providers must decide which region to serve the content from. These decisions have a multitude of tradeoffs: storing a piece of content in many regions incurs increased storage cost (e.g., from a cloud service provider), and storing a piece of content in the “wrong” region can substantially increase access latency, diminishing the end user’s experience (Alicherry & Lakshman, 2012).

To faithfully simulate this effect, our environment includes a real trace of one million posts created on a medium-sized social network over eight months from eight globally distributed regions. Park supports two variants of the assignment task. First, the agent chooses a region assignment when a new piece of content is initially created. At each content creation step, the observation includes the language, outgoing links, and posting user (anonymized) ID. The action is one of the eight regions to store the content. The reward is based on the fraction of accesses from within the assigned region. This variant can be viewed as a contextual multi-armed bandit problem (Lu et al., 2010). The second variant is similar to the first one, except that the agent has the opportunity to migrate any content to any region at the end of each 24 hour time period. The action space spans all possible bijection matchings between the users and the regions. In this case, the agent must balance the cost of a migration against the potential decrease in access latency.

**Server load balancing.** In this simulated environment, an RL agent balances jobs over multiple heterogeneous servers to minimize the average job completion time. Jobs have a varying size that we pick from a Pareto distribution (Grandl et al., 2016) with shape 1.5 and scale 100. The job arrival process is Poisson with an inter-arrival rate of 55. The number of servers and their service rates are configurable, resulting in different amounts of system load. For example, the default setting has 10 servers with processing rates ranging linearly from 0.15 to 1.05. In this setting, the load is 90%. The problem of minimizing average job completion time on servers with heterogeneous processing rates does not have a closed-form solution (Harchol-Balter & Vesilo, 2010); a widely-used heuristic is to join the shortest queue (Daley, 1987). However, understanding the workload pattern can give a better policy; for example, one strategy is to dedicate some servers for small jobs to allow them finish quickly even if many large jobs arrive (Feng et al., 2005). In this environment, upon each job arrival, the observed state is a vector \((j, s_1, s_2, ..., s_k)\), where \(j\) is the incoming job size and \(s_k\) is the size of queue \(k\). The action \(a \in \{1, 2, ..., k\}\) schedules the incoming job to a specific queue. The reward \(r_i = \sum_n \min(t_i, c_n) - t_{i-1}\), where \(t_i\) is the time at step \(i\) and \(c_n\) is the completion time of active job \(n\).

**Switch scheduling.** Switch scheduling poses a matching problem that transfers packets from the incoming ports to the outgoing ports (McKeown, 1999; Shah & Wischik, 2006; Maguluri & Srikant, 2016). This abstracted model is ubiquitous in many real world systems, such as datacenter routers (Giaccone et al., 2002) and traffic junctions (Hunter et al., 1997). At each step, the scheduling agent observes a matrix of queue lengths, with element \((i, j)\) indicating the packet queue from input port \(i\) to output port \(j\). The matching action is bijective — no two incoming packets shall pass through the same output ports. Notice that in a switch with \(n\) input/output ports, the action space is the \(n!\) possible bijection matchings. After each scheduling round, one packet is transferred per each input/output port pair. The goal is to maximize switch throughput while minimizing packet delay. The optimal scheduling policy for this problem is unknown and is conjectured to depend on the underlying traffic pattern (Shah & Wischik, 2006). For example, the max weight matching policy empirically performs well only under high load (Maguluri & Srikant, 2016). Adapting the scheduling policy under dynamics load to optimize an arbitrary combination of throughput and delay is challenging.

### 3.4. Preliminary experiments

With the environments in Park, we train several existing RL models, including A2C (Mnih et al., 2016), DDPG (Lillicrap et al., 2015), DQN (Mnih et al., 2015) and REINFORCE (Williams, 1992). For state space involving graph structure, we use LSTM (Gers et al., 1999) and GCN (Kipf & Welling, 2016) encodings. Figure 1 shows a set of preliminary results. As a sanity check, the performance of learned policy in these environments all improve over time. However, in many scenarios, the off-the-shelf RL algorithms only perform well when we largely simplify the settings or they struggle to outperform the existing deployed heuristics. We explain the challenges causing these results in §4.
4. Challenges

The system environments described in §3 present unique challenges that standard RL algorithms are not designed for. This section discusses the challenges in training a general strong agent in the environments we study.

4.1. State-action Space

Sparse meaningful space for exploration. In some system environments, most of the state-action space presents little difference in reward feedback for exploration. This provides no meaningful gradient during RL training, especially in the beginning, when policies are randomly initialized. Network congestion control is an example: even in the simple case of a fixed-rate link, setting the sending rate above the available network bandwidth saturates the link and the network queue. Then, changes in the sending rate above this threshold result in an equivalently bad throughput and delay, leading to constant, low rewards. To exit this saturation state, the agent must set a low sending rate for multiple consecutive steps to drain the queue before receiving any positive reward. Random exploration is not effective at learning this behavior because any large sending rate (by random sampling) would erase the effort from previous small sending rates and experience no difference in rewards.

In these environments, using domain-knowledge to confine the search space helps to train a strong policy. For example, Figure 1(d) shows a significant performance improvement when restricting the policy search to a pre-defined space for the congestion control problem. Also, environment-specific reward shaping (Ng et al., 1999) or bootstrapping from existing policies (Silver et al., 2016; Hester et al., 2017) can improve the policy search efficiency. However, finding a general algorithm that can efficiently prune a large space remains an open problem of interest to system operators.

Representation of state-action space. When designing RL algorithms for computer system problems, sometimes properly encoding the state-action space is the key challenge. In some systems, the action space grows exponentially large as the problem size increases. For example, in switch scheduling, the action is a bijection mapping between input and output ports—a standard 32-port would have $32!$ possible mappings. Encoding such a large action space is challenging. In Figure 1(j), we only train on a $3 \times 3$ mini-switch.

In other cases, the size of the action space is constantly changing over time. Account region migration is an example: as new accounts join the system, the space of possible migrations (i.e., number of accounts $\times$ number of regions) grows large. This requires the agent to handle a variable-sized action space in its representation.

Furthermore, using representations that capture inherent structure in the state space can improve training efficiency and generalization. Many system problems have a natural graph structure: Spark jobs are DAGs of computation stages; Tensorflow places components of a dataflow graph on different computational units; and circuits are undirected graphs with component features at each node. For these environments, leveraging Graph Convolutional Neural Networks (GCNs) (Kipf & Welling, 2016) captures the underlying problem structure. As shown in Figure 1(g), for example, GCNs make the agent converge to the final performance $5 \times$ faster than a LSTM based approach for the Tensorflow device placement problem.

4.2. Decision Process

Input-driven variance. Queuing systems environments (e.g., job scheduling, load balancing, cache admission) in Park have dynamics partially dictated by an exogenous, stochastic input process. Specifically, their dynamics are governed not only by the decisions made within the system,
Figure 2. Illustrative example of load balancing showing how different instances of a stochastic input process can have vastly different rewards. After time $t$, we sample two job arrival sequences from a Poisson process. Figure adopted from Mao et al. (2018).

but also the arrival process that brings work (e.g., jobs, packets) into the system. For the policy gradient family of RL training methods (Sutton et al., 1999; Mnih et al., 2016; Schulman et al., 2015), the stochastic processes in these environments introduce large variance in the gradient estimates. Standard baseline approaches (Weaver & Tao, 2016; Schulman et al., 2015) are tailored for policy gradient methods and require the environments to have a repeatable input sequence. Thus, coping with input-driven variance remains an open problem for value-based RL methods and for environments with uncontrollable input process.

Infinite horizon. In practice, production computer systems (e.g., Spark schedulers, load balancers, cache controllers, etc.) are long running and host services indefinitely. This creates an infinite horizon MDP (Baxter & Bartlett, 2001) that prevents the RL agents from performing the usual episodic training. In particular, this creates difficulties for bootstrapping a value function (whether training value-based RL, or using it for baseline estimation in policy gradients) since there is no terminal state to easily assign a known target value.

Moreover, the discounted total reward formulation in the episodic case might not be suitable — an action in a long running system can have impact beyond a fixed discounting range. For example, scheduling a large job on a slow server blocks future small jobs (affecting job runtime in the rewards), no matter whether the small jobs arrive immediately after the large job or over the course of the long lifetime of the large job. Average reward formulation can be a viable alternative in this setting (see §10.3 in Sutton & Barto (2017) for more details).

4.3. Reality Gap

Unlike training RL in simulation, robustly deploying a trained RL agent or directly training RL on an actual running computer systems has several difficulties. First, discrepancies between simulation and reality prevent direct generalization. For example, in database query optimization, a static offline cost model cannot represent an actual runtime due to both variance in the underlying data distribution and system-unique artifacts (Leis et al., 2015). Second, interactions with a real system can be slow, which requires a sample-efficient RL training method. In adaptive video streaming, for example, the agent obtains a reward only after a video chunk is downloaded, which typically takes multiple seconds. Third, an RL agent’s inference latency interferes with and degrades system performance, which may require an explicit embedding of the physical time into the agent’s decision making process. For example, a query optimization should integrate the the physical delay in search time when planning actions. Finally, exploration during training should not break down a production system. This requires safety and interpretability constraints for the RL agents to deploy training algorithms online (García & Fernández, 2015; Achiam et al., 2017; Kang et al., 2018).

5. Conclusion

Park provides an open platform that uses a common interface to connect to a wide spectrum of real world computer system problems. Park’s interface is extensible to adding new systems into the platform. Using Park, we also identify several unique challenges that require new algorithmic development in RL. We open source Park along with the benchmarking RL agents and the existing baselines in https://github.com/park-project.
References


