Park: An Open Platform for Learning-Augmented Computer Systems

Hailiang ZHAO @ ZJU-CS

http://hliangzhao.me

January 5, 2021

This slide is a report on Hongzi Mao's paper *Park: An Open Platform for Learning Augmented Computer Systems*, published on **NeurIPS '19**. This paper won the ICML Workshop Best Paper Award.

Outline

Outline

Sequential Decison-Making Problems (SDMP)

Computer systems are full of **sequential decision-making tasks** that can naturally be expressed as Markov decision processes (MDP).

Examples

1. Caching (operating systems)
2. Congestion control (networking)
3. Query optimization (databases)
4. Scheduling (distributed systems)
5. ...

Since real-world systems are difficult to model accurately, state of the art systems often rely on **human-engineered heuristics**, which are

- *complex (e.g., a commercial database query optimizer involves hundreds of rules), and*
- *difficult to adapt across different environments.*

SDMP in Networking

- **Congestion Control**
  ▶ Hosts determine the rate to send traffic, accounting for both *the capacity of the underlying network infrastructure* and *the demands of other hosts*
  ▶ The sender side (agent) sets the sending rate based on how previous packets were acknowledged

- **Bitrate Adaptation in Video Streaming**
  ▶ At watch time, an agent decides the bitrate (affecting resolution) of each video chunk based on the network and video characteristics
  ▶ The goal is to learn a policy that maximizes the resolution while minimizing chance of stalls

SDMP in Databases

- **Query Optimization**
  ▶ Modern query optimizers are complex heuristics which use a combination of rules, handcrafted cost models, data statistics, and dynamic programming, with the goal to **re-order the query operators (e.g., joins, predicates) to ultimately lower the execution time**
  ▶ With RL, the goal is to learn a query optimization policy based on the feedback from optimizing and running a query plan

SDMP in Distributed Systems

- **Job Scheduling**
  ▶ Distributed systems handle computations that are too large to fit on one computer. A job scheduler decides how the system should **assign compute and memory resources to jobs** to achieve fast completion times
  ▶ Jobs can have complex structure
  (e.g., Spark jobs are structured as dataflow graphs, Tensorflow models are computation graphs)
  ▶ The agent in this case observes a set of jobs and the status of the compute resources (e.g., how each job is currently assigned)
  ▶ The action decides how to place jobs onto compute resources.
  The goal is to complete the jobs as soon as possible

SDMP in Operating Systems

- **Caching Optimization**
  ▶ Operating systems provide caching mechanisms which multiplex limited memory amongst applications
  ▶ The RL agent can observe the information of both the existing objects in the cache and the incoming object; it then decides whether to admit the incoming object and which stale objects to evict from the cache. The goal is to maximize the cache hit rate

- **CPU Power State Management**
  ▶ Operating systems control whether the CPU should run at an increased clock speed and boost application performance, or save energy with at a lower clock speed
  ▶ An RL agent can dynamically control the clock speed based on the observation of how each application is running. The goal is to maintain high application performance while reducing the power consumption

Why Not Use Reinforcement Learning?

Deep reinforcement learning (RL) has emerged as a general and powerful approach to sequential decision making problems in *games* and *simulated robotics tasks*. However,

Real-world SYSTEMS of deep RL have far been limited!

Challenges

- The problems in systems are vast, ranging from
  ▶ **centralized control**
  (e.g. scheduling for entire VM cluster)
  to
  ▶ **distributed multi-agent problem** where *multiple entities with partial information* collaborate to optimize system performance (e.g. network congestion control with multiple connections sharing bottleneck links)

Reinforcement Learning for Systems: Challenges

Challenges (cont'd)

- The control tasks manifest at **a variety of timescales**, from
  ▶ **fast, reactive control systems** with sub-second response-time requirements
  (e.g., admission and eviction algorithms for caching objects in memory)
  to
  ▶ **longer term planning problems** that consider a wide range of signals to make decisions
  (e.g., VM allocation and placement in cloud computing)

Reinforcement Learning for Systems: Challenges

Challenges (cont'd)

- Learning algorithms themselves also face new challenges:
  - ▶ **time-varying state or action spaces**
  (e.g. dynamically varying number of jobs and machines in a computer cluster)
  - ▶ **structured data sources**
  (e.g., graphs to represent data flow of jobs or a network's topology)
  - ▶ **highly stochastic environments**
  (e.g., random time-varying workloads)

Bridge the Gap between RL and Computer Systems

Why only little work on *RL for computer systems*?

1. The challenges mentioned above
2. Lack of good benchmarks for evaluating solutions
3. Absence of an easy-to-use platform for experimenting with RL algorithms in systems

Conducting research on learning-based systems requires

- *significant expertise to implement solutions in real systems,*
- *collect suitable real-world traces, and*
- *evaluate solutions rigorously.*

Thus, the authors present the open platform **Park** $\Longrightarrow$ which presents **a common RL interface to connect to a suite of 12 computer system environments**

https://github.com/park-project

Overview on the Open Platform Park

- The 12 representative environments span a wide variety of problems across networking, databases, and distributed systems, and range from centralized planning problems to distributed fast reactive control tasks
- In the backend, the environments are powered by both **real systems** (in 7 environments) and **high fidelity simulators** (in 5 environments)
- For each environment, Park defines the MDP formulation, e.g., **events that triggers an MDP step, the state and action spaces and the reward function**. This allows researchers to focus on the core algorithmic and learning challenges, without having to deal with low-level system implementation issues

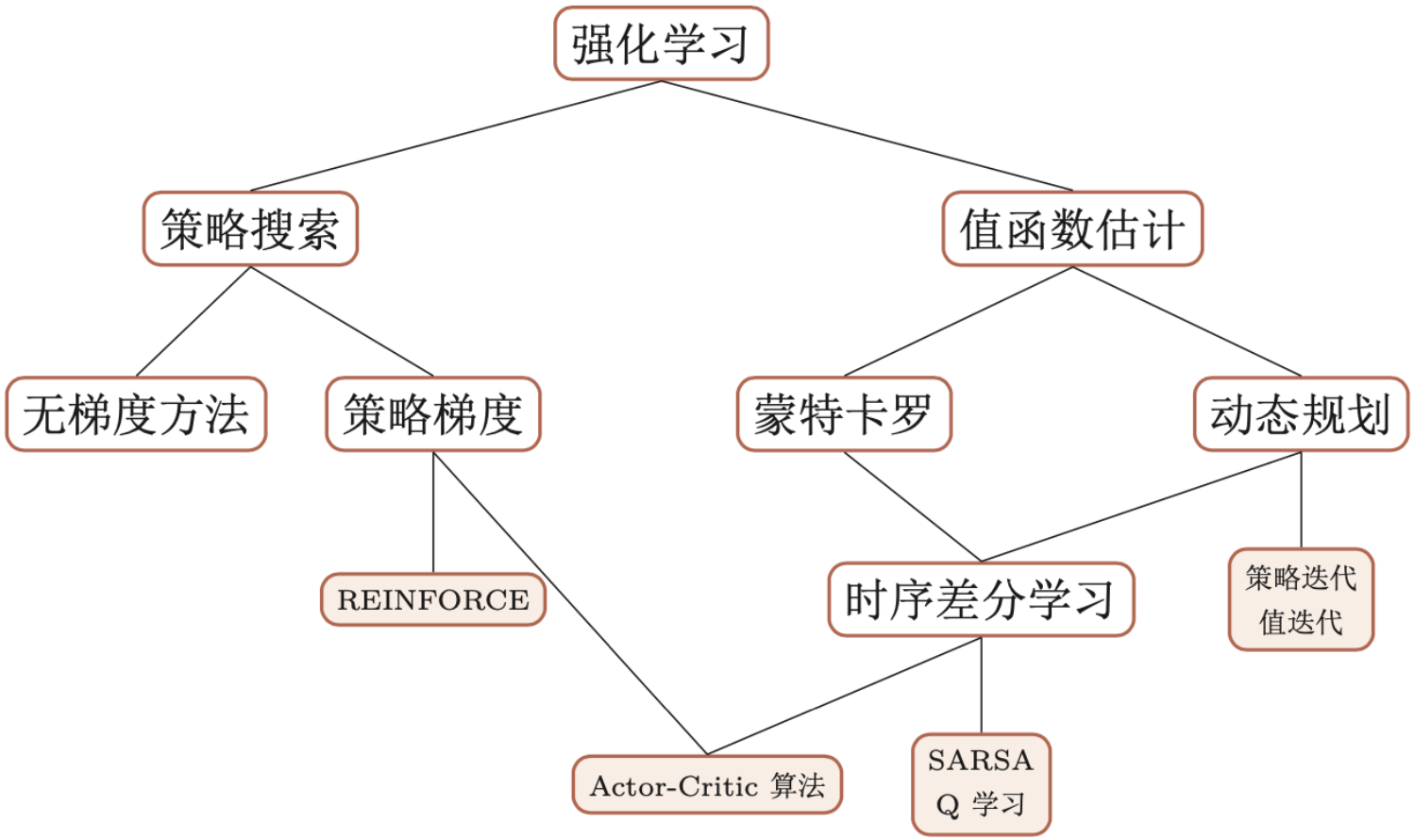
Overview on the Open Platform Park

- Park makes it easy to compare different proposed learning agents on a common benchmark, similar to how OpenAI Gym has standardized RL benchmarks for robotics control tasks
- Park defines **a RPC interface** between the RL agent and the backend system, making it easy to extend to more environments in the future
- The authors benchmark the 12 systems in Park with both RL methods and existing heuristic baselines. The empirical results are mixed: RL is able to outperform state-of-the-art baselines in several environments where researchers have developed problem-specific learning methods; for many other systems, RL has yet to consistently achieve robust performance.

Reinforcement Learning: An Overview

An agent learns to act by interacting with the environment (observing the state $\rightarrow$ generating an action $\rightarrow$ obtaining a reward).

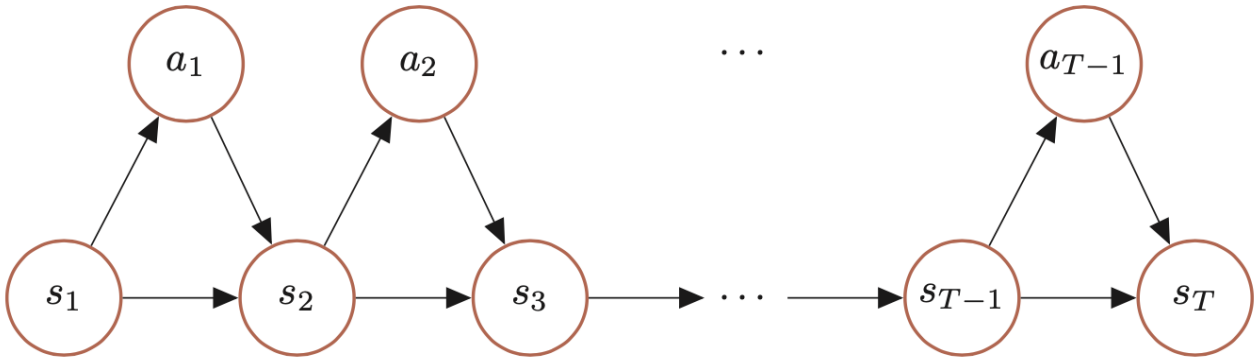
- **Value function estimation**
- **Policy search**



Baisc Symbols and Definitions

- MDP and trajectory:

  $\tau = s_0, a_0, s_1, r_1, a_1, \ldots, s_{t-1}, r_{t-1}, a_{t-1}, s_t, r_t, \ldots$



- Expected return $\mathcal{J}$: $\mathbb{E}_{\tau\sim p(\tau)}[G(\tau)] = \mathbb{E}\left[\sum_{t=0}^{T-1}\gamma^t r_{t+1}\right]$
- State value function V: $V^\pi(s) = \mathbb{E}_{\tau\sim p(\tau)}\left[\sum_{t=0}^{T-1}\gamma^t r_{t+1}|\tau_{s_0} = s\right]$
- Bellman equation:

$$V^\pi(s) = \mathbb{E}_{a\sim\pi(a|s)}\mathbb{E}_{s'\sim p(s'|s,a)}[r(s,a,s') + \gamma V^\pi(s')]$$

Baisc Symbols and Definitions

- State-action value function Q: $V^{\pi}(s) = \mathbb{E}_{a\sim\pi(a|s)}[Q^{\pi}(s,a)]$
- Bellman equation (for Q):

$$Q^{\pi}(s,a) = \mathbb{E}_{s'\sim p(s'|s,a)}[r(s,a,s') + \gamma \mathbb{E}_{a'\sim\pi(a'|s')}[Q^{\pi}(s',a')]$$

- The value functions (Q and V) are the evaluation on the action policy π. A larger V (or Q) leads to a better policy
- For dimensionally increasing state and action spaces, use deep neural networks to approximate the value functions, which leads to *Deep Q-Networks* (DQN)

Value function-based methods

- **Model based: Dynamic programming**
  (1) Policy Iteration: evaluate V, then update π
  (2) Value Iteration: update V (thus π) directly
- **Model free: Monte carlo search**
  (3) Vanilla version: sampling a trajectory with *ϵ-greedy method*
- **Temporal-Difference Learning**
  - **Off-policy** (evaluate π^{ϵ}, update π): e.g., (4) Q-learning:

$$\hat{Q}(s,a) \leftarrow \hat{Q}(s,a) + \alpha\Big(r(s,a,s') + \gamma \max_{a'} \hat{Q}(s',a') - \hat{Q}(s,a)\Big)$$

  - **On-policy** (evaluate and update π^{ϵ}): e.g., (5) SARSA:

$$\hat{Q}^{\pi}(s,a) \leftarrow \hat{Q}^{\pi}(s,a) + \alpha\Big(\underbrace{r(s,a,s') + \gamma\hat{Q}^{\pi}(s',a')}_{reality} - \underbrace{\hat{Q}^{\pi}(s,a)}_{estimate}\Big)$$

Value function-based methods

- (6) **Deep Q-Network**

  ▶ Based on Q-Learning, use a neural network (with param ϕ) to approximate the value function Q. The Loss function is the gap between *reality* and *estimate*:

$$\mathcal{L}(s, a, s'|\phi) = \Big(\underbrace{r(s, a, s') + \gamma \max_{a'} Q_\phi(\boldsymbol{s}', \boldsymbol{a}')}_{\textit{true reward as target}} - Q_\phi(\boldsymbol{s}, \boldsymbol{a}) \Big)^2,$$

  where

$$Q_\phi(\boldsymbol{s}) = \begin{bmatrix} Q_\phi(\boldsymbol{s}, a_1) \\ Q_\phi(\boldsymbol{s}, a_2) \\ \vdots \\ Q_\phi(\boldsymbol{s}, a_m) \end{bmatrix} \approx \begin{bmatrix} Q^\pi(\boldsymbol{s}, a_1) \\ Q^\pi(\boldsymbol{s}, a_2) \\ \vdots \\ Q^\pi(\boldsymbol{s}, a_m) \end{bmatrix}$$

  ▶ Other tricks:

  (i) freeze target network, and use (ii) experience reply

Policy Search-based methods

How Policy Search Works

Update the gradients of agent param θ to maximize the expected return $\mathcal{J}(\theta)$:

$$\frac{\partial \mathcal{J}(\theta)}{\partial \theta} = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_{t=0}^{T-1} \left(\frac{\partial}{\partial \theta} \log \pi_\theta(a_t|s_t) \gamma^t G(\tau_{t:T}) \right) \right],$$

where $G(\tau_{t:T}) = \sum_{t'=t}^{T-1} \gamma^{t'-t} r_{t'+1}$.

- (7) **REINFORCE algorithm**
  Use random walk to sample several trajectories $\tau^{(1)}, ..., \tau^{(n)}, ...$
  Then, with each trajectory, update θ by

$$\frac{\partial \mathcal{J}(\theta)}{\partial \theta} = \frac{1}{N} \sum_{n=1}^{N} \left[\sum_{t=0}^{T-1} \left(\frac{\partial}{\partial \theta} \log \pi_\theta(a_t^{(n)}|s_t^{(n)} \gamma^t G(\tau_{t:T}^{(n)}) \right) \right]$$

Policy Search-based methods

- (8) **REINFORCE with baseline**
  Take a baseline function $b(s_t)$ (usually use $V^{\pi_\theta}(s_t)$) to reduce the variance of the policy gradients.

$$\frac{\partial \mathcal{J}(\theta)}{\partial \theta} = \mathbb{E}_{\tau \sim p_\theta(\tau)} \left[\sum_{t=0}^{T-1} \left(\frac{\partial}{\partial \theta} \log \pi_\theta(a_t|s_t) \gamma^t \big(G(\tau_{t:T}) - V_\phi(s_t) \big) \right) \right]$$

If we can update the param θ in each step (s, a, r, s'), rather than a whole trajectory been sampled $\Longrightarrow$

- (9) **Actor-Critic algorithm**
  It combines REINFORCE with TD learning to udpate θ incrementally during each (s, a, r, s') step. ▶ Actor: the action policy $\pi_\theta(s, a)$ ▶ Critic: the value function $V_\phi(s)$

$$\theta \leftarrow \theta - \alpha \gamma^t \Big(\underbrace{\hat{G}(\tau_{t:T})}_{\text{true reward: } r_{t+1} + \gamma V_\phi(s_{t+1})} - \underbrace{V_\phi(s_t)}_{\text{approximation}} \Big) \frac{\partial}{\partial \theta} \log \pi_\theta(a_t|s_t)$$

Conclusion (from the NNDL book)

算法	步骤
SARSA	(1)执行策略，生成样本：s, a, r, s', a' (2)估计回报：$Q(s,a) \leftarrow Q(s,a) + \alpha\Big(r + \gamma Q(s',a') - Q(s,a)\Big)$ (3)更新策略：$\pi(s) = \arg\max_{a \in \lvert\mathcal{A}\rvert} Q(s,a)$
Q学习	(1)执行策略，生成样本：s, a, r, s' (2)估计回报：$Q(s,a) \leftarrow Q(s,a) + \alpha\Big(r + \gamma \max_{a'} Q(s',a') - Q(s,a)\Big)$ (3)更新策略：$\pi(s) = \arg\max_{a \in \lvert\mathcal{A}\rvert} Q(s,a)$
REINFORCE	(1)执行策略，生成样本：$\tau = s_0, a_0, s_1, a_1, \cdots$ (2)估计回报：$G(\tau) = \sum_{t=0}^{T-1} r_{t+1}$ (3)更新策略：$\theta \leftarrow \theta + \sum_{t=0}^{T-1} \Big(\frac{\partial}{\partial\theta} \log \pi_\theta(a_t \lvert s_t) \gamma^t G(\tau_{t:T})\Big)$
Actor-Critic	(1)执行策略，生成样本：s, a, s', r (2)估计回报：$G(s) = r + \gamma V_\phi(s')$ $\phi \leftarrow \phi + \beta\Big(G(s) - V_\phi(s)\Big)\frac{\partial}{\partial\phi} V_\phi(s)$ (3)更新策略：$\lambda \leftarrow \gamma\lambda$ $\theta \leftarrow \theta + \alpha\lambda\Big(G(s) - V_\phi(s)\Big)\frac{\partial}{\partial\theta} \log \pi_\theta(a \lvert s)$

Outline

RL for Systems: Characteristics and Challenges

The following demonstrates the **unique characteristics and challenges** that prevent off-the-shelf RL methods from achieving strong performance in different computer system problems.

- State-Action Space
- Decision Process
- Simulation-Reality Gap
- Understandability over Existing Heuristics

State-Action Space

Problem 1: The needle-in-the-haystack problem

The majority of the state-action space presents little difference in reward feedback for exploration, which provides **no meaningful gradient** during RL training.

The following are several examples:

- **Congestion Control**
  - 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 bad state, the agent must set a low sending rate **for multiple consecutive steps** to drain the queue before receiving any positive reward

State-Action Space

- **Circuit Design**
  - When any of the circuit components falls outside the operating region (the exact boundary is unknown before invoking the circuit simulator), the circuit cannot function properly and the environment returns a constant bad reward

$\Longrightarrow$ SOLUTIONS: Use **domain-knowledge** to confine the search space is necessary:

- environment-specific reward shaping
- bootstrap from existing policies
- ...

State-Action Space

Problem 2: Representation of state-action space

When designing RL methods for problems with complex structure, properly encoding the state-action space is the key challenge.

- space grows exponentially large as the problem increases
- size of the state/action space is constantly changing over time

$\Longrightarrow$ SOLUTIONS: **domain specific representations** that capture inherent structure

- Spark jobs, Tensorflow components, and circuit design: **dataflow graphs**, use Graph Convolutional Neural Networks (GCNs)
- However, finding the right representation for each problem is a central challenge

State-Action Space

$\Longrightarrow$ SOLUTIONS: **domain specific representations** that capture inherent structure

- Generalizability of GCN and LSTM state representation in *the Tensorflow device placement environment*. The numbers are average runtime in seconds. ± spans one standard deviation. Bold font indicate the runtime is within 5% of the best runtime. "Transfer" means testing on unseen models in the dataset

	GCN direct	GCN transfer
CIFAR-10 (Krizhevsky & Hinton, 2010)	**1.73**±0.41	**1.81**±0.39
Penn Tree Bank (Marcus et al., 1993)	**4.84**±0.64	**4.96**±0.63
NMT (Bahdanau et al., 2014)	**1.98**±0.55	**2.07**±0.51

LSTM direct	LSTM transfer	Random
1.78±0.38	1.97±0.37	2.15±0.39
5.09±0.63	5.28±0.6	5.42±0.57
2.16±0.56	2.88±0.66	2.47±0.48

Decision Process

Problem 3: Stochasticity in MDP causing huge variance

Queuing systems environments (e.g., job scheduling, load balancing, cache admission) have dynamics partially dictated by *an exogenous, stochastic input process*. Their dynamics are governed by both

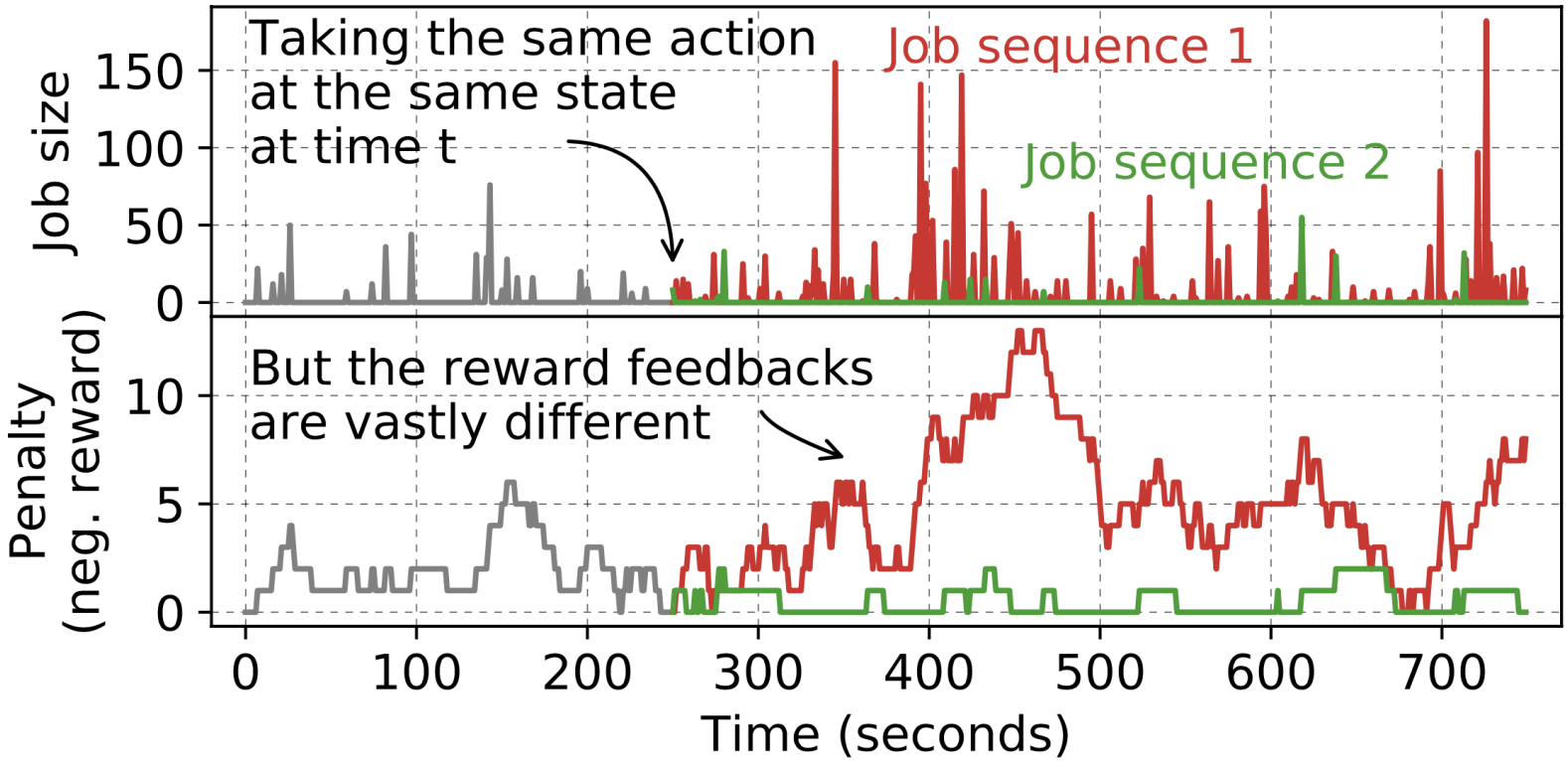
- **the decisions made within the system**, and
- **the arrival process that brings work** into the system

The stochasticity causes **huge variance** in the reward!

Decision Process

- **Huge Variance in Reward**

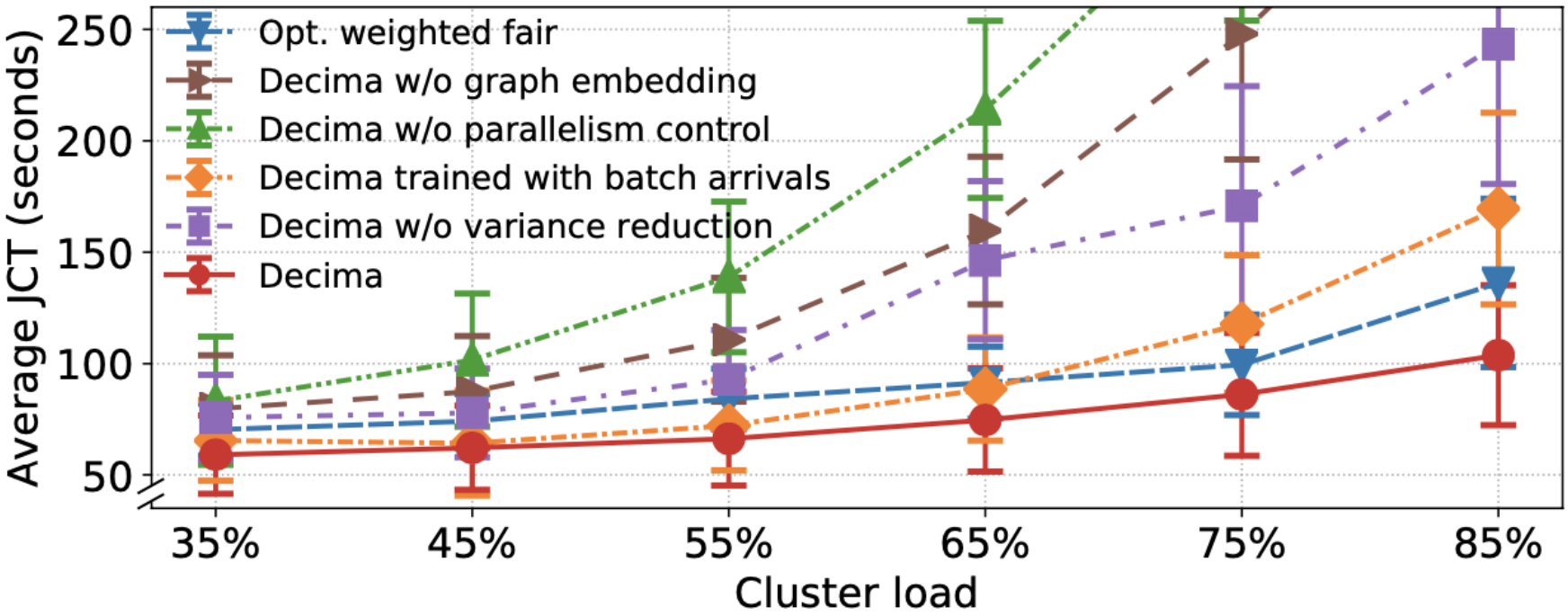
The agents cannot tell whether two reward feedbacks differ due to disparate input processes, or due to the quality of the actions.



Decision Process

$\Longrightarrow$ SOLUTIONS: use **input-dependent baseline** to effectively reduce the variance from the input process

- **With/Without Input-dependent Baselines**



Decision Process

Problem 4: Infinite horizon problems

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 that prevents the RL agents from performing **episodic training**.

Leads to:

- great difficulties for bootstrapping a value estimation since there is **no terminal state** to easily assign a known target value
- 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 window [e.g., scheduling a large job on a slow server blocks future small jobs]

Simulation-Reality Gap

Problem 5: Simulation-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.

- (5.1) discrepancies between simulation and reality prevent direct generalization
- (5.2) interactions with some real systems can be slow
- (5.3) live training or directly deploying an agent from simulation can degrade the system performance

Simulation-Reality Gap

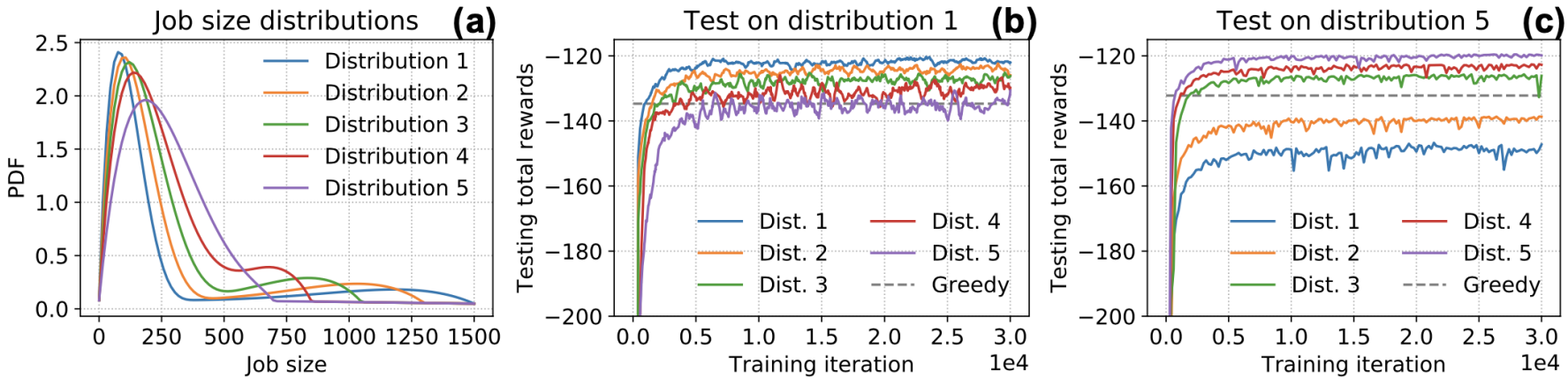
(5.1) **Discrepancies between simulation and reality**
For example, in [database query optimization], existing simulators or query planners use offline cost models to predict query execution time (as a proxy for the reward). However, the accuracy of the cost model quickly degrades as the query gets more complex due to both variance in the underlying data distribution and system-specific artifacts.

(5.2) **Slow interactions with real systems**
In adaptive [video streaming], for example, the agent controls the bitrate for each chunk of a video. Thus, the system returns a reward to the agent only after a video chunk is downloaded, which typically takes a few seconds. Naively using the same training method from simulation would take a single-threaded agent more than 10 years to complete training in reality.

Simulation-Reality Gap

(5.3) **System performance degrade**

For [load balancing], when training with a bimodal distribution job sizes, the RL agent learns to reserve a certain server for small jobs to process them quickly. However, when the distribution changes, blindly reserving a server wastes compute resource and reduces system throughput.



(a) Distribution of job sizes in the training workload. (b, c) Testing agents on a particular distribution. An agent trained with distribution 5 is more robust than one trained with distribution 1.

Understandability over Existing Heuristics

Problem 6: Understandability over existing heuristics

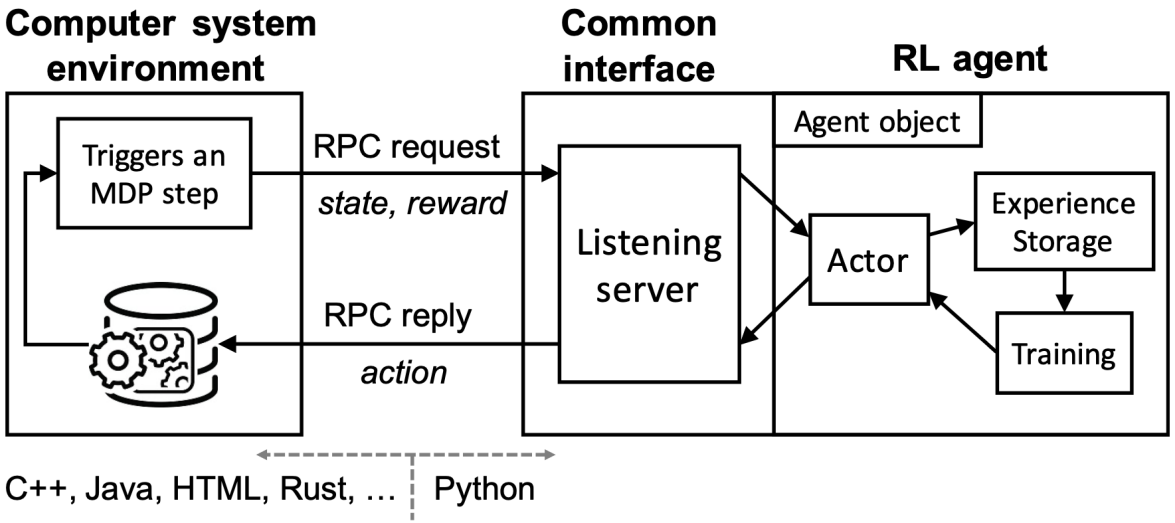
Heuristics are often easy to understand and to debug, whereas a learned approach is often not. Hence, making learning algorithms in systems as **debuggable and interpretable** as existing heuristics is a key challenge.

$\Longrightarrow$ SOLUTIONS: build **hybrid solutions**, which combine learning-based techniques with traditional heuristics
(a learned scheduling algorithm could fall back to a simple heuristic if it detects that the input distributions ignificantly drifted)

Outline

Architecture and Implementation Details

Park follows a standard *request-response* design pattern. The backend system runs continuously and **periodically send requests to the learning agent** to take control actions. To connect the systems to the RL agents, Park defines *a common interface* and hosts a server that *listens for requests from the backend system*. The backend system and the agent run on different processes (which can also run on different machines) and they **communicate using remote procedure calls (RPCs)**. This design essentially structures *RL as a service*.



Architecture and Implementation Details

The computer system connects to an RL agent through a canonical request/response interface, which hides the system complexity from the RL agent.

- Algorithm 1 describes a cycle of the system interaction with the RL agent
- By wrapping with an "agent-centric" environment in Algorithm 2, Park's interface also supports OpenAI Gym like interaction for simulated environments

Algorithm 1 Interface for real system interaction.

```
def env.run(agent):
    while not done:
        state, reward, done = server.listen()
        # reward for the previous action
        action = agent.act(state, reward, done)
        server.reply(action)
```

Algorithm 2 Interface for simulated interaction.

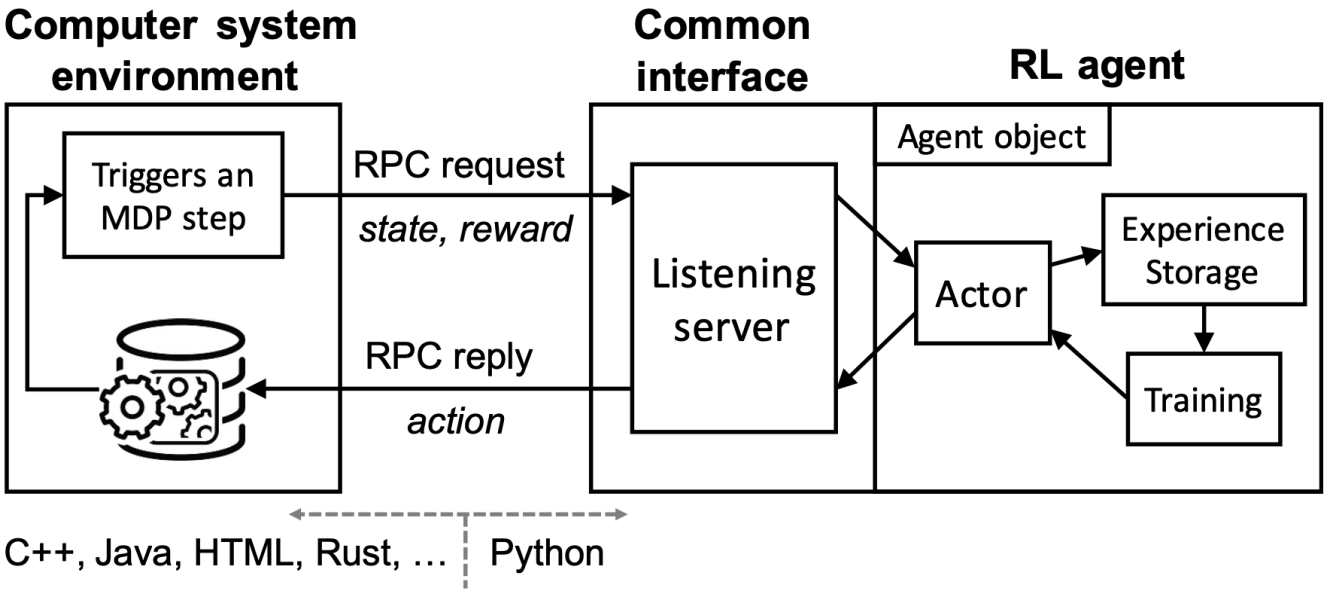
```
def env.step(action):
    # OpenAI Gym style of interaction
    server.reply(action)
    state, reward, done = server.listen()
    return state, reward, done
```

Real System Interaction Loop

- Each system defines its own events to trigger an MDP step. At each step, the system sends an RPC request that contains a current observation of the state and a reward corresponding to the last action
- Upon receiving the request, the Park server invokes the RL agent. Park provides the agent with all the information provided by the environment
- The implementation of the agent is up to the users (e.g., feature extraction, training process, inference methods). Once the agent returns an action, the server replies back to the system
- Invoking the agent incurs a physical delay for the RPC response from the server
  ▶ non-blocking RPC: the state observation received by the agent can be stale
  ▶ blocking RPC: taking a long time to compute an action

Wrapper for simulated interaction

- By wrapping the request-response interface with a shim layer, Park also supports an "agent-centric" style of interaction advocated by OpenAI Gym
- With this interface, we can directly reuse existing off-the-shelf RL training implementations benchmarked on Gym



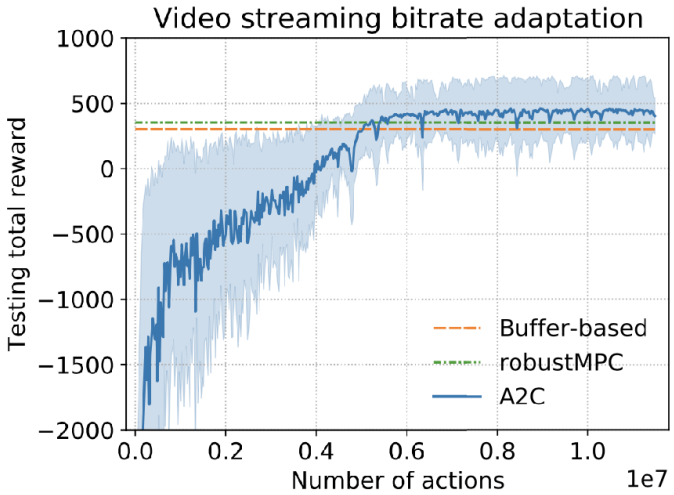
Other Properties

- **Scalability**
  ▶ The common interface allows multiple instances of a system environment to run concurrently (multiple environment instances, multiple actor instances, route with RPC request ID)
- **Environments**
  ▶ Park implements 12 environments. Seven of the environments use real systems in the backend. For the remaining five environments, which have well-understood dynamics, the authors provide a simulator to facilitate easier setup and faster RL training
  ▶ For these simulated environments, Park uses **real-world traces** to ensure that they mimic their respective real-world environments as faithfully as possible

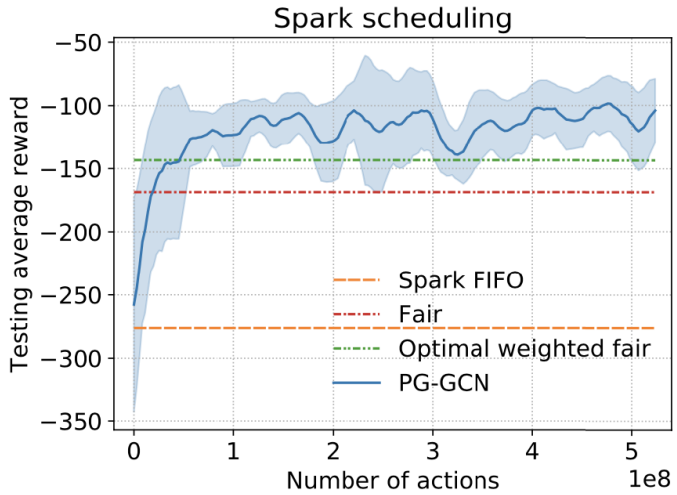
Other Properties

- **Extensibility**
  ▶ Adding a new system environment in Park 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
  (3) the function to calculate the reward feedback

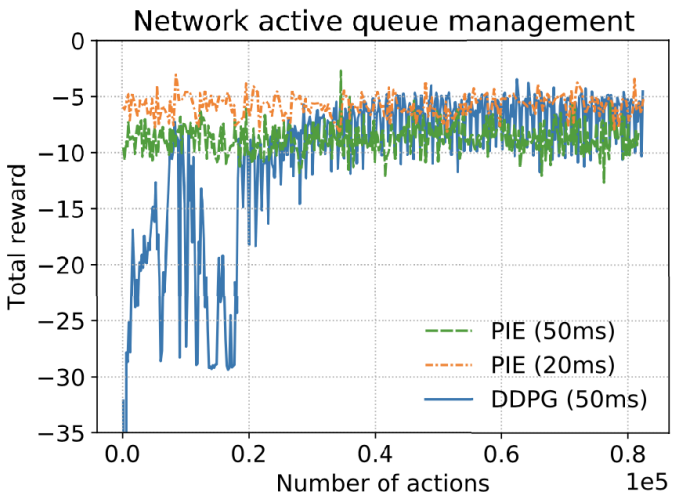
Benchmark Experiments



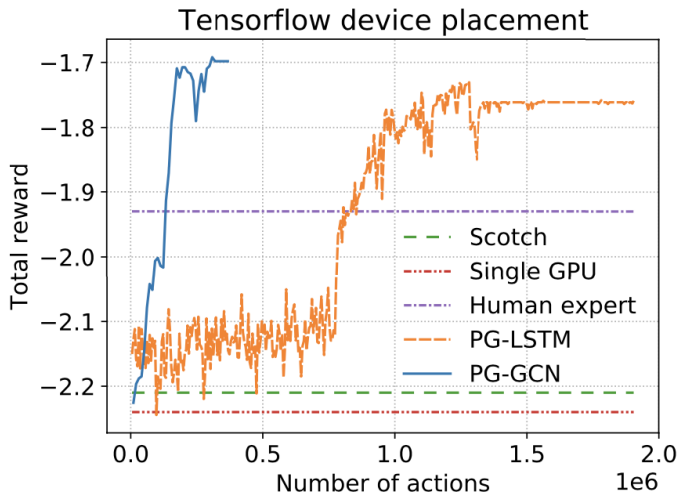
(a) Video streaming



(b) Spark scheduling

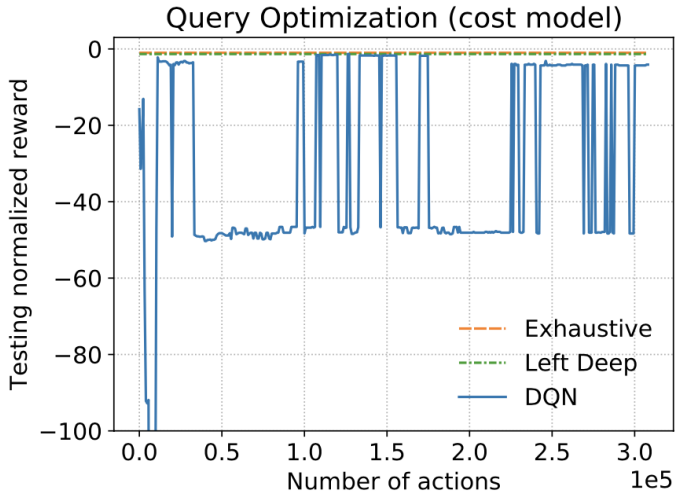


(e) Queue management

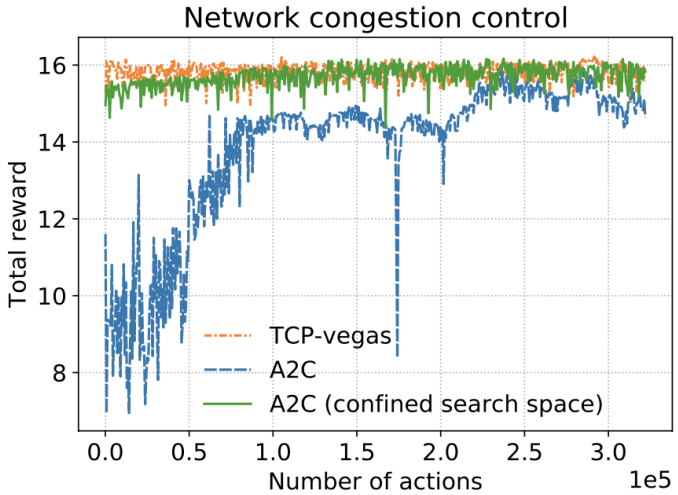


(f) TF device placement

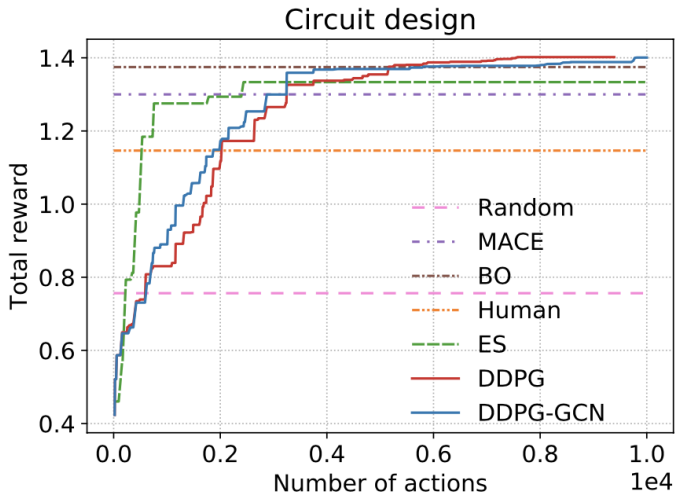
Benchmark Experiments



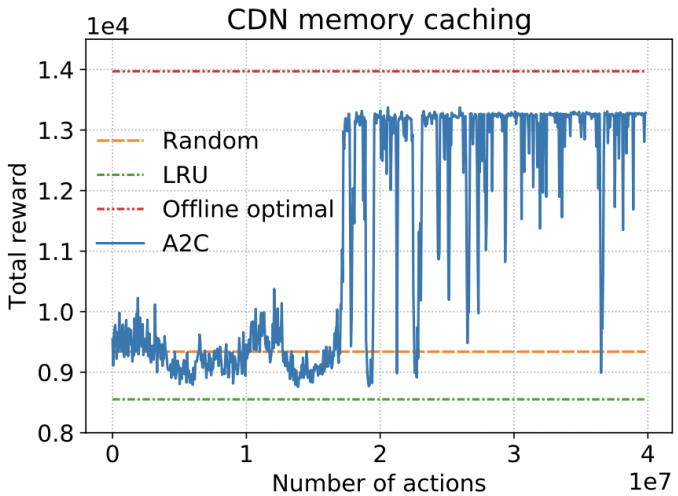
(c) Database optimization



(d) Congestion control

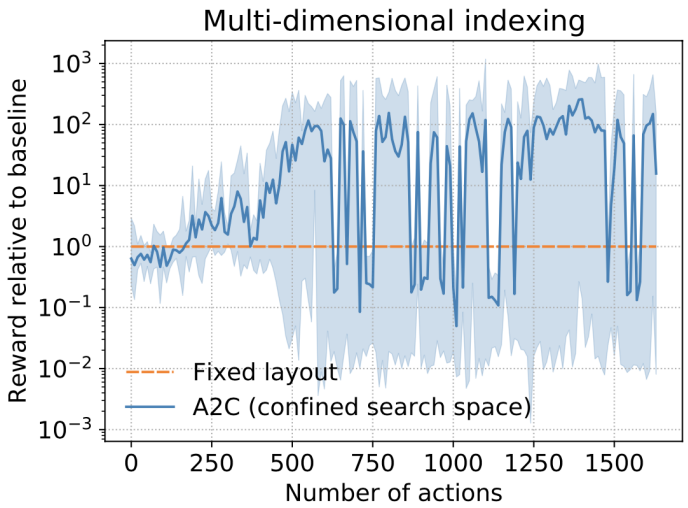


(g) Circuit design

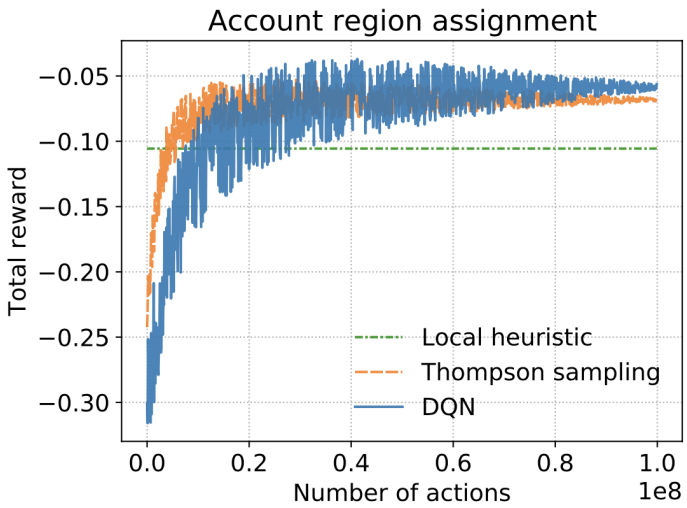


(h) CDN memory caching

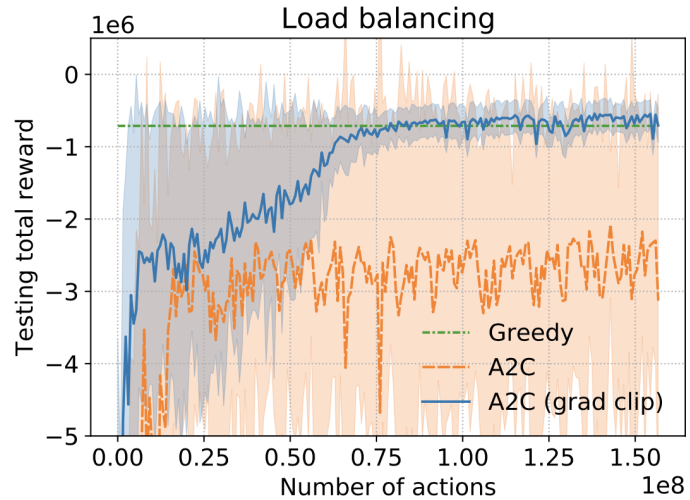
Benchmark Experiments



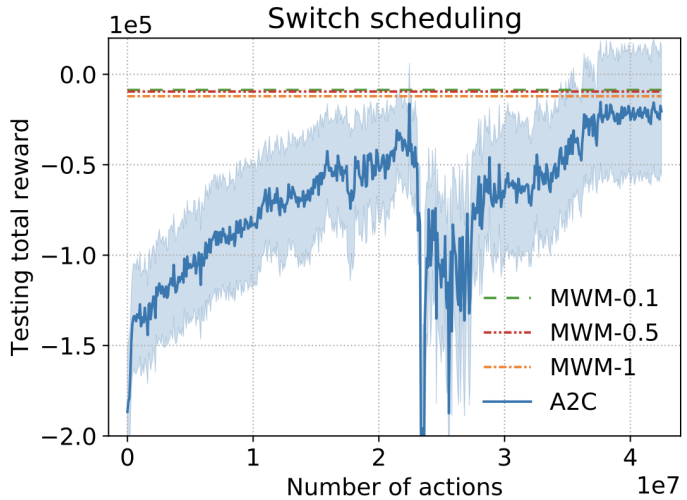
(i) Multi-dim indexing



(j) Account assignment



(k) Load balancing



(l) Switching scheduling