DEGREE PROJECT IN ELECTRICAL ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2019

Towards Learning for System Behavior

LIANGCHENG YU

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ENGINEERING SCIENCES

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

KTH VETENSKAP OCH KONST

Master Thesis

Institute for Pervasive Computing, Department of Computer Science, ETH Zurich

School of Electrical Engineering and Computer Science, KTH, Sweden

Towards Learning for System Behavior

by Liangcheng Yu

Spring 2018

Student ID:	17-912-486(ETH) & 941005-7534(KTH)
E-mail address:	liayu@student.ethz.ch & liayu@kth.se
Supervisors:	Dr. Anwar Hithnawi
	Dr. Hossein Shafagh
	Prof. Dr. Friedemann Mattern
	Prof. Dr. Lars K. Rasmussen

Abstract

Traditional network management typically relies on clever heuristics to capture the characteristics of environments, workloads in order to derive an accurate model. While such methodology has served us well in early days, it is challenged by the growing intricacies of modern network design from various dimensions: the rocketing traffic volumn, proliferation of software applications and varied hardware, higher user-specific Quality of Experience (QoE) requirements with respect to bandwidth and latencies, overwhelming number of knobs and configurations and so forth. All these surging complexity and dynamics pose greater difficulty on us to understand and derive management rules to reach global optimum with heuristics that fits the dynamic context. Driven by the pulls of the challenges and encouraged by the success in machine learning techniques, this work elaborates on augmenting adaptive systems behaviors with learning approaches. This thesis specifically investigates the use case of the packet scheduling. The work explores the opportunity to augment systems to learn existing behaviors and explore custom behaviors with Deep Reinforcement Learning (DRL). We show the possibility to approximate the existing canonical behaviors with a generic representation, meanwhile, the agent is able to explore customized policy that are comparable to the state-of-art approaches. The results demonstrate the potentials of learning based approaches as an alternative to canonical scheduling approaches.

Sammanfattning

Traditionell nätverkshantering bygger vanligtvis på smart heuristik för att fånga egenskaper hos miljöer, arbetsbelastning för att få en exakt modell. Medan sådan metodik har fungerat bra i början av dagen, utmanas den av växande intricacies av modern nätverksdesign från olika dimensioner: rocketing Traumatiska volymer, spridning av programvaror och varierad hårdvara, högre Krav på användarspecifik kvalitet av erfarenhet (QoE) med avseende på bandbredd och latenser, överväldigande antal knoppar och konurationer och så vidare. allt Dessa växande komplexitet och dynamik gör oss mer engagerade i förståelsen och härleda ledningsregler för att nå global optimalt med heuristik som ts dynamiskt sammanhang. Drivs av utmaningarna och uppmuntras av framgången I maskininlärningsteknik utarbetar detta arbete på att öka adaptiva system beteenden med inlärningsmetoder. Denna avhandling anger användningsfallet eller paketplaneringen. Arbetet undersöker möjligheten att öka systemen att lära sig befintliga beteenden och utforska anpassade beteenden med Deep Reinforcement Lärande (DRL). Vi visar möjligheten att approximera den befintliga canonicalen Beteende med en generisk representation, under tiden kan agenten utforska anpassade policyer som är jämförbara med state-of-the-art-metoder. Resultaten visa potentialen att lära sig baserade metoder som ett alternativ till canonical planeringsmetoder.

Acknowledgements

I am specially thankful to Dr. Anwar Hithnawi and Dr. Hossein Shafagh for the comprehensive mentoring and tremendous support during the thesis. Their research style have influenced and shaped me into developing critical thinking and methodology of conducting research with ambition in future career. I am thankful to Prof. Friedeman Mattern for all the support and host throughout the thesis duration. I would like to express my great gratitude to Prof. Lars K. Rasmussen for examining the thesis, providing immediate support during the thesis and help. I am grateful for all the insightful advice and feedbacks received from Prof. Sylvia Ratnasamy. I would like to thank deeply all members from Distributed Systems Group at ETH Zurich for their support of facilities, encouragement and friendly atmosphere during the thesis.

I couldn't thank enough to Zeno Karl Schindler Foundation for their generous fellowship and I hope someday I would be able to give back helping others by contributing my own efforts. Last but not least, I am deeply indebted with my beloved parents, who always provide firm support to my life.

Contents

1 Introduction

In the past decade, we have witnessed the prosperity of widespread network services and increasingly complex and heterogeneous workloads inside modern networks. As more devices become Internet-enabled and as new applications are emerging, the usage of the network is becoming increasingly varied, ranging from content delivery to streaming media to IoT to social media to the tactile Internet and beyond. These applications have different network requirements regarding bandwidths, desired latencies and exhibit different flow characteristics. Moreover, such a high volume of flows with diverse patterns are contending in sharing the same network. Inevitably, they pose unique challenges on traditional network control strategies since not only the network itself becomes more complex, dynamic, and heterogeneous but also the expectation of user-specific Quality of Experience (QoE) grows as well. Traditional network management policies (e.g., congestion control) rely heavily on manually-crafted configurations and human heuristics which work only in general sense and often miss the actual context; thus these approaches have become less effective in response to actual operation patterns and large-scale network dynamics [30, 39].

The last decade has also witnessed breathtaking results brought by machine learning techniques, especially deep learning, in many areas, to name a few, computer vision, natural language processing, robotics, and medical healthcare [40]. Furthermore, the deep learning wave has ignited the renaissance of reinforcement learning and sparked exciting progress in deep reinforcement learning, with which amazing benchmarks in real-world decision-making tasks are achieved. In particular, the AI system AlphaGo Master developed by Google DeepMind which defeated world No.1 human Go-player demonstrated the real power of machine intelligence to exceed human potentials [1, 2, 85].

Behavioral machine learning leverages machines to self-learn the optimal sequential decision making via interactions with the surrounding environment. Such framework opens the door to tackling realistic decision-making tasks and shed light on the augmenting intelligence in systems behaviors. The mechanism incorporates deep learning models to process complex, large-volume sensory input that is faced by modern systems. Besides, while the current canonical approach relies on human heuristics that often work only in general case, this set of algorithms are catered to solving decision-making problems with respect to long-term objectives by taking the benefits of decision making history, which provides us not only the flexibility to express our intent to meet specific system performance goals but also the possibility of customizing network policies and behaviors to observed settings and workload of the system by the agent.

Driven by both pulls of the success of machine learning techniques and the complexity of decision making in systems, we have seen growing interests into applying machine learning approaches tackle many challenges in systems, to name a few, optimal virtual machine selection over the cloud [15,105], database management system configuration [98], resource management [54], video streaming tasks [55], optimal resource configurations for data analytics workloads [42,100] and so forth. In this thesis are particularly interested in augmenting intelligence for system behaviors.

This work looks explicitly at network packet scheduling which acts as one of the central decision-making components in modern networks. Packet scheduling is the process of deciding which packet is sent out next and when. It is crucial since such decisions have overall consequences on the fairness and the completion time of various contending flows. Besides, the end-to-end delay nowadays suffers largely from queueing delay that packets enduring in switches which is directly correlated with the opted scheduling policies. Echoing the promising directions of next-generation network systems [26, 30, 39, 57], the thesis aims to tackle packet scheduling as a medium to explore the implications underneath when augmenting deep behaviors into systems, paving the way for further implementation and design.

The main contributions of the thesis can be summarized as follows:

- We model packet scheduling as a decision-making problem and build a simulator prototype for evaluating various workloads and different scheduling agents for queue management.
- We propose a model-free DRL agent and exploit its capability to learn existing scheduling behaviors. Results are presented with theoretical analysis and empirical justifications.
- We explore the feasibility for the agent to adapt its behaviors to different settings and workloads, given the intended objective, and compare its performances with canonical approaches.
- With packet scheduling as a use case, we explore and identify the features, challenges, limitations, and implications when augmenting deep behaviors for systems, suggesting tips for future work towards this direction.

The remainder of the thesis is organized as follows. Chapter 2 overviews the background in reinforcement learning, packet scheduling, and related work of the thesis. Chapter 3 depicts in detail the design and the methodology for the exploration. Chapter 4 presents the results and evaluation for the design. Chapter 5 discusses the future work and Chapter 6 summarizes this work. Finally, the appendix attaches the instructions to reproduce the results in the thesis.

2 Background

2.1 Deep Reinforcement Learning

2.1.1 Framework

Intuition

Reinforcement learning is a branch of machine learning which offers a powerful set of tools for sequential decision making under uncertainty. Reinforcement learning leverages on evaluative feedbacks to self-learn the policy which maximizes cumulative reward. In reinforcement learning an agent learns to find an optimal policy directly from the locus of interacting experiences with the environment without manually specifying *how* the task is achieved [41]. Such property of reinforcement learning makes it appealing to many disciplines, including optimal control, economics, psychology, neuroscience, computer science.

The storyline of the agent during a task starts from an initial state. Upon each step, the agent exerts a valid action on the dynamic environment and typically receives immediate reward and observations of the next state, as shown in Figure 2.1. Intuitively, such a process is analogous to a dialogue between the environment and the agent [41]. The framework asks the agent a question and gives her a noisy score on her answer. During the process of these interaction loops, the agent goal is to find a policy mapping states to actions that maximize long-term reward.

Formalism

Behavioral machine learning typically involves the following abstractions and components: environment observations, policy derivation, exploration and exploitation, reward formulation, experiences, and policy improvement machinery.

Markov Decision Process (MDP) is the mathematical formulation of a typical reinforcement learning problem, for discrete time MDP, it is defined by $(\mathcal{S}, \mathcal{A}, r, \mathcal{T}, \gamma)$.

$$\begin{tikzcd}
\text{Agent} \arrow[r, bend right, "\text{action}"'] & \text{Environment} \arrow[l, bend right, "\text{observation, reward}"']
\end{tikzcd}$$

Figure 2.1: Basic Reinforcement Learning Framework

Starting from $s_0 \sim \mu(s)$, at each time step, the agent observes a state $s_t \in \mathcal{S}$ and selects an action $a_t \in \mathcal{A}$, following policy $\pi(a_t|s_t)$. It receives a scalar reward r_{t+1} and the environment transits to the next state s_{t+1}, according to reward function $r(s,a)$ and state transition operator $\mathcal{T}$ representing $p(s_{t+1}|s_t,a_t)$ respectively. Such dynamic process yields a sample trajetory $\tau = (s_0, a_0, r_1, s_1, a_1, r_2, ...)$. The agent seeks to maximize the expectation of such long term reward from each state $G_t = \sum_{k=0}^{\infty} \gamma^k R_{t+k}$, where $\gamma \in (0,1]$, i.e.,

$$\begin{aligned} \pi^* &= \underset{\pi}{\operatorname{argmax}}\, \mathbb{E}[\sum_{t\geq 0} \gamma^t R_{t+1}|\pi] \\ s.t. \quad & s_0 \sim p(s_0), a_t \sim \pi(.|s_t), s_{t+1} \sim p(.|s_t,a_t) \end{aligned} \tag{2.1}$$

Additonally, markov properties give rise to the distribution of state action sequences for a finite horizon task [43]:

$$\begin{aligned} \pi_\theta(\tau) = p(s_0, a_0, ...s_{T-1}, a_{T-1}|\theta, \pi) &= p(s_0) \prod_{t=0}^{T-1} \pi(a_t|s_t,\theta) p(s_{t+1}|s_t,a_t) \\ \theta^* &= \underset{\theta}{\operatorname{argmax}}\, \mathbb{E}_{\tau\sim p_\theta(\tau)}[\sum_t r(s_t,a_t)] \end{aligned} \tag{2.2}$$

In many real-world environments, it will not be possible for the agent to have perfect perception of the state of the environment. The resulting formulation is partially observable Markov decision process (POMDP) denoted as $(\mathcal{S}, \mathcal{A}, \mathcal{O}, \mathcal{T}, \varepsilon, r, \gamma)$, where ε stands for emission probability $p(o_t|s_t)$ and $o_t \in \mathcal{O}$. Fortunately, with function approximation, the partially observed setting is not much different conceptually from the fully-observed setting [77].

Value function $v_\pi(s) = \mathbb{E}_\pi[G_t|s_t = s]$ is typically used as a prediction of expected, accumulative future reward of each state. An optimal state value is therefore defined as $v_*(s) = \max_\pi v_\pi(s)$. They can be decomposes into Bellman Equation and Bellman Optimality Equation respectively,

$$\begin{aligned} v_\pi(s) &= \sum_a \pi(a|s) \sum_{s',r} p(s',r|s,a)[r + \gamma v_\pi(s')] \\ v_*(s) &= \max_a \sum_{s',r} p(s',r|s,a)[r + \gamma v_*(s')] \end{aligned} \tag{2.3}$$

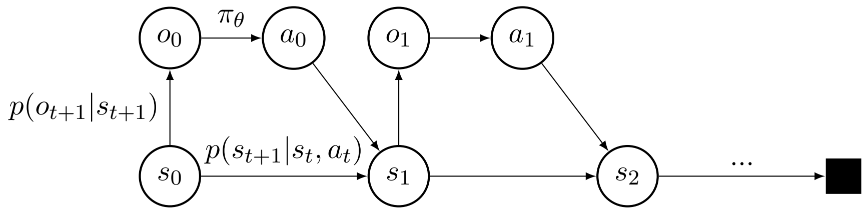

Figure 2.2: A Graphical Model for POMDP

Similarly, action value $q_\pi(s,a) = \mathbb{E}[G_t|s_t = s, a_t = a]$ and optimal action value function $q_*(s,a) = \max_\pi q_\pi(s,a)$ obey similar identities,

$$\begin{aligned} q_\pi(s,a) &= \sum_{s',r} p(s',r|s,a)[r + \gamma \sum_{a'} \pi(a'|s') q_\pi(s',a')] \\ q_*(s,a) &= \sum_{s',r} p(s',r|s,a)[r + \gamma \max_{a'} q_*(s',a')] \end{aligned} \tag{2.4}$$

Such identity is the cornerstone for *bootstrapping* an estimate of state or action value from subsequent estimates [63]. Given optimal value function, it is straightforward to act optimally via greedy method: $\pi_*(s) = \operatorname{argmax}_a q_*(s,a)$.

$$\begin{aligned} Q^*(s,a) &= \mathbb{E}_{s'}[r + \gamma \max_{a'} Q^*(s',a')|s,a] \\ \to Q_{i+1}(s,a) &= \mathbb{E}_{s'}[r + \gamma \max_{a'} Q_i(s',a')|s,a] \end{aligned} \tag{2.5}$$

2.1.2 Approximate Solution

Integration

One distinct feature of deep reinforcement learning, compared with "shallow" reinforcement learning, is that it integrates neural network approximator into the framework [20]. Such integration scales reinforcement learning algorithms to real-world problems [87, 97].

The approximation could be used to represent the policy function, Q function, state value, and even environment transition when it comes to model-based learning. With model-free deep reinforcement learning, the main task falls generally into three categories, as shown in Figure **??**: fitting action-value function with dynamic programming, optimizing policy directly, and combining policy optimization and value fitting, where corresponding policy and value function are parametrised.

Deep Models

Conventional machine-learning techniques were limited in their ability to process natural data in their raw form, besides, for decades, constructing a pattern-recognition or machine-learning system required careful engineering and considerable domain expertise to design a feature extractor that transformed the raw data into a suitable internal representation or feature vector from which the learning subsystem, often a classifier, could detect or classify patterns in the input [47]. Deep models are composed of multiple processing layers to learn representations of data with multiple levels of abstraction. These layers of features are not designed by human engineers: they are learned from data using a general-purpose learning procedure. Deep models have been tremendously successful in practical applications, among which, Convolutional Neural Networks (CNNs) have brought about breakthroughs in processing images, video, speech and audio, whereas Recurrent Neural Networks (RNNs) have sheded light on sequential data such as text and speech [47].

This thesis mainly leverages CNNs. CNNs are specialized neural networks for processing data that come in the form of grid-like topology, typically 3D volume (1D time series and 2D images could be viewed as special 3D volume in which certain dimension is with 1) [32, 47]. CNNs employ linear operations, namely *convolution* and *pooling* to improve the machine learning system. Essentially each CNN layer maps a 3D volume into another with such differentiable functions, transforming the representation towards a higher, slightly more abstract level, leveraging on the insights that real-world features typically come in a hierarchical pattern [32]. ConvNets are now the dominant approach for almost all recognition and detection tasks, where successful architectures include LeNet [48], AlexNet [45], GoogLeNet [95], ZFNet [106], VGGNet [86], ResNet [36] and so on.

2.1.3 Value Optimization

Value optimization methods seek to fit the state value function and derive the policy on top of the estimation. Deep Q-network (DQN) was proposed as a first novel deep learning agent which could process high-dimensional sensory inputs (pixel-level) and directly self-learn the policy with comparable performances to that of human gamers [62, 63]. Before DQN, most successful RL applications relied heavily on hand-crafted features. The success of DQN ignited the field of deep reinforcement learning.

DQN parametrizes state action value function with deep Q network. Reinforcement learning can be unstable or even divergent when off-policy Q learning is integrated with a nonlinear function like neural networks, this issue is known as deadly triad [93,

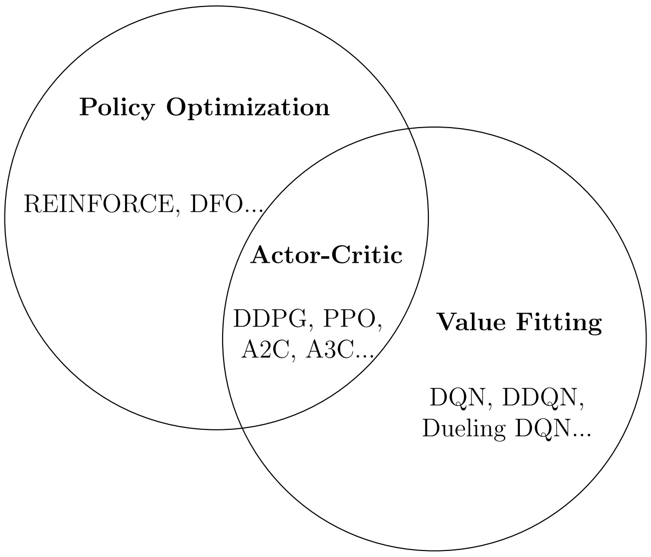

Figure 2.3: General Taxonomy

97]. To address the instabilities, researchers introduced experience replay for DQN, inspired by biological mechanism in order to remove temporal correlations in the observation sequence and smoothing over changes in the data distribution.

Experiences are stored in predefined replay memory data set $D = e_1, ..., e_t$ and drawn in batches with uniform probability. Such off-policy mechanism enables the agent to reuse the experiences instead of evicting immediately after a single update and to improve the target policy with generated samples from behavioural policy. The network is trained with respect to the loss defined in (2.6).

$$\ell_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(D)}[(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta i))^2] \tag{2.6}$$

In practice, stochastic gradient descent is applied rather than computing the full expectations in (2.7).

$$\nabla_{\theta_i}\ell_i(\theta_i) = \mathbb{E}_{(s,a,r,s')\sim U(D)}[(r + \gamma \max_{a'} Q(s', a'; \theta_i^-) - Q(s, a; \theta i))\nabla_{\theta_i} Q(s, a; \theta i)] \tag{2.7}$$

Another innovation of DQN is to update target network in a less frequent manner, as shown in Algorithm 1 (adapted from [63]). The policy is straightforward by directly applying a greedy policy to the Q function.

Algorithm 1: Deep Q-learning with experience replay

```
1  function Deep Q-learning ;
   Input  : replay memory size N
   Output : optimal state-action value approximation $Q^*$
2  initialize weights $\theta$ for primary Q network arbitrarity
3  initialize target $\hat{Q}$ network weights $\theta^- = \theta$
4  for each episode do
5      initialize state $s$
6      for each step $t$ do
7          $a_t \leftarrow$ action derived by Q and $\epsilon-$greedy at state $s_t$
8          execute action $a_t$ in simulator and observe $r_{t+1}, s_{t+1}$
9          store experience $e_t = (s_t, a_t, r_{t+1}, s_{t+1})$ in replay memory
10         sample random minibatch of experiences from replay memory
11         set $y_t = r_{t+1} + \gamma \max_{a'} \hat{Q}(s_{t+1}, a'; \theta^-)$
12         perform gradient descent of $\ell^{Huber}(y_t, Q_{s_t,a_t;\theta})$ with respect to $\theta$
13         clone $\theta^- = \theta$ every C steps
14     end
15 end
```

DDQN was later proposed to mainly tackle the over-estimate problem in DQN. (2.8) demonstrates the difference between DQN and DDQN. Double DQN removes the bias caused by $\text{argmax}_a Q(s, a, \theta)$. Current Q network is now used to select actions while older Q network is used to evaluate [99].

$$
\begin{aligned}
Y_t^{DQN} &\equiv R_{t+1} + \gamma Q(S_{t+1}, \arg\max_a Q(S_{t+1}, a; \boldsymbol{\theta}_t^-); \boldsymbol{\theta}_t^-) \\
Y_t^{DoubleQ} &\equiv R_{t+1} + \gamma Q(S_{t+1}, \arg\max_a Q(S_{t+1}, a; \boldsymbol{\theta}_t); \boldsymbol{\theta}_t^-)
\end{aligned}
\tag{2.8}
$$

Prioritized experience replay strategy was later applied to DQN to boost the efficiency of learning and improve further the bechmarks than DQN [75]. The main idea is to give more weights to those do not fit well to our current estimate of the Q function in the sampling distribution. Specifically, the collection of historical experiences are treated as a priority queue with key value calculated based on temporal-difference (TD) error, and, to scale the memory size N, the queue is represented with binary heap data structure with $O(logN)$ update complexity $O(1)$ for sampling [75].

There are many others DQN follow-ups, including bootstrapped DQN with better build-in exploration strategy [65], shallow RL structure that could reproduce DQN benchmarks [52], dueling network architechture which separates state-value and the advantages for each action [101] and many others. Despite its great success, deep Q learning methods are still risky of divergence and requires significant empirical engineering. It is still promising, though, since off-policy methods typically yield better policies if they work.

2.1.4 Policy Optimization

Unlike deep Q-learning family, policy gradient methods could select actions without consulting state-action value estimates. Policy gradient methods optimize the parametrised policy $\pi(a|s;\boldsymbol{\theta}) = P[a|s, \theta]$ directly by performing gradient ascent as:

$$\boldsymbol{\theta}_{t+1} = \boldsymbol{\theta}_t + \alpha \widehat{\nabla J(\boldsymbol{\theta}_t)} \tag{2.9}$$

$$\theta^* = \underset{\theta}{\text{argmax}}\, \mathbb{E}_{\tau \sim p_\theta(\tau)}[\sum_t r(s_t, a_t)] = \underset{\theta}{\text{argmax}}\, J(\theta) \tag{2.10}$$

where $J(\boldsymbol{\theta})$ denotes the performence measure [93]. In episodic environments we consider $J(\boldsymbol{\theta}) = V^{\pi_\theta}(s_0) = \mathbb{E}_{\pi_\theta}[G_0] = \mathbb{E}[\sum_{t\geq 0} \gamma^t R^{t+1} | \pi_\theta]$.

Introducing the notation $r(\tau) = \sum_t r(s_t, a_t)$, we arrive at the following identities.

$$J(\theta) = \mathbb{E}_{\tau \sim p_\theta(\tau)}[\sum_t r(s_t, a_t)] \approx \frac{1}{N} \sum_i \sum_t r(s_{i,t} a_{i,t}) \tag{2.11}$$

$$\nabla_\theta J(\theta) = \int \pi_\theta(\tau) \nabla_\theta \log \pi_\theta(\tau) r(\tau) d\tau = \mathbb{E}_{\tau \sim \pi_\theta(\tau)}[\nabla_\theta \log \pi_\theta(\tau) r(\tau)] \tag{2.12}$$

Taking logorithm of (2.2) both sides and substituting $\log \pi_\theta(\tau)$ in (2.12), we have

$$\nabla_\theta \log \pi_\theta(\tau) = \nabla_\theta[\log p(s_0) + \sum_t \log \pi_\theta(a_t|s_t) + \log p(s_{t+1}|s_t, a_t)] \tag{2.13}$$

$$\begin{aligned}\nabla_\theta J(\theta) &= \mathbb{E}_{\tau\sim\pi_\theta(\tau)}[(\sum_t \nabla_\theta \log \pi_\theta(a_t|s_t))(\sum_t r(s_t,a_t))] \\ &\approx \frac{1}{N}\sum_i(\sum_t \nabla_\theta \log \pi_\theta(a_{i,t}|s_{i,t}))(\sum_t r(s_{i,t},a_{i,t}))] \\ &= \frac{1}{N}\sum_i \nabla_\theta \log \pi_\theta(\tau_i) r(\tau_i)\end{aligned} \tag{2.14}$$

The REINFORCE method is derived directly from the policy gradient theorem [93, 94, 102]. The basic REINFORCE algorithm combined with Monte Carlo sampling is shown in Algorithm 2, where v_t is a shorthand for $q_{\pi_\theta}(s_t, a_t)$. REINFORCE formalizes the basic intuition of trial and error and requires no no knowledge regarding state transition.

There are many other form of policy gradient which can be unified with a generic form [79], shown in (2.15).

$$g = \mathbb{E}[\sum_{t=0}^{\infty} \Psi_t \nabla_\theta \log \pi_\theta(a_t|s_t)] \tag{2.15}$$

where Ψ_t could be of various forms, e.g., in REINFORCE, $\Psi_t = \sum_{t=0}^{\infty} r_t$, and if consider the causality of reward, i.e,, policy at time $t^{'}$ will not affect reward at $t \leq t^{'}$, $\Psi_t = \sum_{t'=t}^{\infty} r_{t'}$.

$$\hat{g} \approx \frac{1}{N}\sum_i \sum_{t'=0}^{T-1} \nabla_\theta \log \pi_\theta(a_{i,t'}|s_{i,t'}))(\sum_{t=t'}^{T-1} r(s_{i,t},a_{i,t}))] \tag{2.16}$$

One major downside is that it suffers from high variances and low sample-efficiency due to on-policy nature [93]. Adding a state-value function as a baseline could ease the issue without introducing bias, i.e., $\Psi_t = \sum_{t'=t}^{\infty} r_{t'} - b(s_t)$. A practical baseline is the average of historical rewards, as shown in (2.17). It is unbiased since $\mathbb{E}[\nabla_\theta \log \pi_\theta(\tau) b] = 0$. The intuition here is, instead of assigning the credit directly with the sampled rewards, we reinforce or penalize the agent based on how much better is the reward than average.

$$\begin{aligned} b &= \frac{1}{N}\sum_{i=1}^{N} r(\tau) \\ \nabla_\theta J(\theta) &\approx \frac{1}{N}\sum_i \nabla_\theta \log \pi_\theta(\tau_i)[r(\tau_i) - b]\end{aligned} \tag{2.17}$$

Other alternatives of Ψ_t are state-action value function $Q^\pi(s_t, a_t)$, advantage function $A^\pi(s_t, a_t) = Q^\pi(s_t, a_t) - V^\pi(s_t)$ and TD residual $r_t + V^\pi(s_{t+1}) - V^\pi(s_t)$ [79]. Besides, policy gradient could be turned into off-policy with importance sampling, in which $\Psi(t) = (\prod_{t'=0}^{\infty} \frac{\pi_\theta(a_{t'}|s_{t'})}{\pi_{\theta'}(a_{t'}|s_{t'})})(\sum_{t'=0}^{\infty} r(s_{t'}, a_{t'}))$ or $\Psi(t) = (\prod_{t'=0}^{t} \frac{\pi_\theta(a_{t'}|s_{t'})}{\pi_{\theta'}(a_{t'}|s_{t'})})(\sum_{t'=t}^{\infty} r(s_{t'}, a_{t'}))$ and the expectation is therefore taken over the behavioral policy $\pi_{\theta'}$.

Algorithm 2: REINFORCE (Monte-Carlo Policy Gradient)

```
1 function REINFORCE;
  Output: $\theta$
2 initialize $\theta$ arbitrarily
3 for each sampling trajectory $\tau$ following $\pi_\theta$ do
4     for t=0 to T-1 do
5         $\theta \leftarrow \theta + \alpha \nabla_\theta \log \pi_\theta(s_t, a_t) v_t$
6     end
7 end
```

Another promising alternative with quite dissimilar workflow than policy gradient is the evolutionary method, which is less sample efficient but exhibits favorable properties like ease of implementation, parallelism [71, 90]. Such methods typically follow the pipeline which starts from sampling, evaluation, and fitting the model with selected/survived instances from the original pool of sampling.

2.1.5 Actor-critic

Actor-critic methods improve policy gradients with extra policy evaluation via a biased critic: $Q_w(s, a) \approx Q_{\pi_\theta}(s, a)$ [93]. Hence, actor-critic algorithms follow an approximate policy gradient as

$$\nabla_\theta J(\theta) \approx \mathbb{E}_{\pi_\theta}[\nabla_\theta log \pi_\theta(s, a) Q_w(s, a)] \tag{2.18}$$

A collection of algorithms follow the acter-critic framework. Asynchronous Advantage Actor-Critic (A3C) [61] typically implements multiple workers in parallel on multiple cores and performs gradient update in a Hogwild pattern [69]. Similar to experience replay, asynchronous update provides an alternative to break the correlation of colelcted samples. Deterministic Policy Gradient (DPG) models the policy as a deterministic decision $a = \mu(s)$ and Deep Deterministic Policy Gradient (DDPG) combines it with DQN in a a off-policy actor-critic manner [53, 84]. They are suitable to address tasks involving continuous action. The exploration is realized via adding noise to the original action $\mu^{'}(s) = \mu_\theta(s) + \mathcal{N}$.

TRPO harnesses the intuition that to improve training stability, we should avoid parameter updates that change the policy too much at one step [78]. TRPO aims to maximize the objective function subject to trust region constraint which enforces the distance between old and new policies measured by KL-divergence to be small enough, within a parameter δ [78]. Proximal Policy Optimization (PPO) is an improvement over TRPO since it simplifies the complicated constraint by using a clipped surrogate objective while retaining similar performance [37, 80].

2.2 Packet Scheduling

2.2.1 Network Data Transmission

Computer networks are essentially a large set of edge nodes – personal computers, mobile phones, servers, and so on – connected with an interconnected group of forwarding devices like switches and routers[1]. Bit streams are generated by network applications and chunked into unit of "packets" by the implemented software stack over OS and hardware devices (e.g., NIC) of the source hosts. Packets form into set of flows according to the belonging communication sessions and traverse through the network. Routers and switches forward the packets leveraging on the meta data in packet headers and the network conditions, which ideally would be received by the destination endpoint and responded with corresponding acknoledgements [46, 59].

Networks are dynamic and input driven: high volume of contending data flows could overwhelm the network infrastructure and therefore impair the performance of the network in forms of packet losses, transmission latencies; besides, network infrastructure itself could suffer from malfunction, such as power down of the end servers, suspension of a link and so forth. Therefore, packet scheduling and congestion control are necessary to come in and act as two crucial decisions making mechanisms to ensure network performances. Congestion control focuses on preventing overwhelming traffic from the source host typically via dynamically observing network states and feedbacking to the source end with instructions to pause or continue sending packets. This thesis mainly looks at packet scheduling, which resides in local switches and determine when and which packet to send out next from the pool of queueing packets over the transmission link in order to achieve certain objective.

2.2.2 Local Packet Processing

Below is a summary of the typical processing pipeline of a unicast packet in a store-and-forward switch (output queued packet switch).

Prephase Processing

Upon arrival of a packet, it is first validated to insure correctness (checksum, time-to-live), compatibility (protocol, IP version) and security (DoS attacks). Additional processing typically includes decrement of packet time-to-live (TTL) field to prevent endless circulation of the packet. Eligible packets are then forward to corresponding egress port queue based on destination lookup (e.g., longest-prefix-match (LPM)) [96].

[1]We use router and switch interchangeably.

Queue Assignment

Typically, the packet is classified and appended to specific queue of the link depending on the meta data in the packet header (e.g., Type of Service (ToS) field, source/destination port). The representation of the queue is switch-specific, it could be find-grained, even per-flow queue, though it is prohibitively expensive due to the maintaining of per-flow statistics, or be coarse-grained so that each queue comprises packets from multiple flows. Off-the-shelf routers usually set up a fixed number of queues with predetermined rules to assign packets to queues. When the link is overwhelmed, the packet could be dropped or marked on the explicit congestion notification (ECN) field according to the implemented buffer management policies (drop-tail, random early detection (RED)) [96].

Scheduling

Triggered by buffer occupancy, each egress link scheduler makes independent decision on when and which packet to dequeue next, according to the hardcoded scheduling algorithms.

2.2.3 Canonical Approaches

Packet scheduling is essentially about the decision on when and which packet to dequeue next. Such decision is made based on the specific domain information. As an example, STFQ [34] bases the decision on the meta information of virtual start time for each packet which is maintained per-flow/per-queue and updated upon enqueue and dequeue events as shown [88]. The decision is directed to the packet with minimum virtual start value.

Packet scheduling algorithms are objective oriented: they are derived in order to achieve certain objectives (fairness [28,67,82,107], deadline awareness [50,73], prompt completion of flows [16]) in different network environments (Internet, datacenters, cellular networks). Hence, one would prioritize certain packet scheduling algorithms given specific deployment objectives. For instance, if fair share among flows is of top priority (e.g., in Internet), one would prefer WFQ algorithms like WDRR which strive for fairness. However, to advance flow completion (e.g., in Data Center Networks), one would probably apply SJF, SRPT which are customized to minimize flow completion time (FCT).

There is a large glossary of existing packet scheduling algorithms and its variants, combinations of hierarchical ones and so forth. Below is a gentle walkthrough of a subset of them from the perspective of decision-making criteria.

Time Awareness

First In First Out (FIFO), or First Come First Serve (FCFS) dequeues the packet with that arrives at the link earliest w.r.t. wall clock time, while Earliest Deadline First (EDF) prioritizes the flow with closest deadline.

Service Type

Strict Priority (SP) schedules packets based on the service priority of the flow. Such meta information is carried in the header of the packet, typically ToS field and is tagged at the end host. With SP, flows of higher priority will be favoured always, indicating that best effort queues might be starved for resources.

Fair Share

To achieve fair share, ideally, network resources should be served in a bit-by-bit fashion [28,66]. However, switches are store-and-forward devices and data is transmitted in the unit of separate packets. There are a number of algorithms to approximate fairness of network resource share. Round Robin (RR) and Weighted Round Robin (WRR) serve the queues in a circular fashion to achieve the concept of fairness. However, they suffer from limitations since they do not take into account the unfairness due to the packet size diversity. Deficit Round Robin (DRR) and Weighted Deficit Round Robin (WDRR) [83] instead define a quantum of resource and update the deficit of each queue upon enqueue and dequeue to achieve equal/weighted bandwidth share. In general, resources are allocated to the queue that contains the maximum deficit. Start Time Fair Queueing (STFQ) achieve fairness by maintaining a per-flow state of virtual start time. Therefore, the flow with minimum virtual start time is prioritized for transmission.

Flow Completion

Shortest Job First (SJF), Shortest Flow First (SFF) and Shortest Remaining Processing Time (SRPT) [16, 76] all advance flows that are expected to finish first. They dequeue the flow with least front packet size, minimum total flow size and shortest remaining flow size respectively. Flow size meta data is initialized at the end host and typically available for the switches. In practice, there are preemptive and non-preemptive variants.

2.3 Computational Framework

Apart from the increase of computing power, hardware capabilities and new algorithmic techniques, mature software packages and architectures are also what sit beneath the current AI success [92]. Modern frameworks like Tensorflow [12], MXNet [8], Torch/PyTorch [11], Caffe [3] and so on boost the productivity of deep learning pipeline with built-in support like auto differentiation. The thesis leverages the computation mainly over ETH Leonhard and Euler cluster [5, 6].

Tensorflow is an interface to *express* algorithms in the form of computation graph representation on a wide variety of heterogeneous systems, ranging from mobile devices to large-scale distributed systems with GPU cards [13, 14]. Unlike Torch which supports dynamic graphs [68], Tensorflow operations do not run at define time and computation graphs are predefined prior execution. The client uses the Session interface to communicate with the master and variable number of worker processes. Computational devices are instructed by the master before execution. A computing device are identified by its type, index within scope of the worker, and if with distributed setting, also job and task of the worker. Example device names are /job:localhost/device:cpu:0 or /job:worker/task:17/device:gpu:3 [13]. The placement algorithm is responsible to map operations into the set of available devices before the original graph paritioned into a set of subgraphs [13, 14].

2.4 Related Work

Network scheduling has been extensively studied during past decades, with a large glossary of algorithms and design which elaborates on different objectives and scenarios. However, these approaches share the same heuristic based mindset neglecting the potentials of augmenting behaviors through systems themselves. Meanwhile, we have also witnessed the emerging practices of applying machine learning into systems stimulated by the pushes from the success of deep learning, increasingly mature software instruments and hardware support. This section will first walk through the canonical approaches of network scheduling and then highlight the recent practices of machine learning for systems.

2.4.1 Canonical Scheduling

A broad spectrum of literature exists in order to deal with the diverse settings of scheduling problem, as mentioned in Section 2.2.3. Recent work continuously focuses on optimizing objectives under scenarios and assumptions that are of interest.

To sample a subset of these interesting work, pFabric [16] proposes a datacenter transport design that minimizes FCT in a near theoretically optimal sense, decoupling flow scheduling from rate control. D^3 presents a deadline-driven delivery control protocol that does not require routers to maintain per-flow state [103]. PDQ achieves both the intention of minimizing FCT and deadline awareness [38]. Universal packet scheduling (UPS) elaborates on finding the universal packet scheduling algorithm that could replay existing ones from both theoretical and practical viewpoints [60]. Programmable packet scheduling abstracts broad range of sophisticated scheduling algorithms with priority and calendar queues [88]. PIAS proposes multiple level feedback queue mechanism seeking to approximate SJF with no prior knowledge which supports existing commodity switch hardware [17]. QJUMP applies the concept of latency sensitivity level to datacenter applications that enables higher rank packets to jump the queue over those lower to mitigate network interference [35]. Remy [104] extracts congestion control as a function module and generate rules automatically for endpoints taking in the inputs like network assumptions and objectives, whose derived algorithms outperform the state-of-arts in NS-2 simulation.

These approaches for systems are typically derived from heuristics compound with meticulous tuning and are rigid with fixed commands that lacks cognitive capabilities, hence they are not suited to meeting the uncertainties and complexity of our objective as systems evolve [26].

2.4.2 Machine Learning for Systems

Current systems are filled with heuristics dicisions and user-tunable knobs which opens the opportunities for a recent surge of detected problems that could be exploited with machine learning techniques with potential of comparable or even exceeding performances than heuristics approaches [27].

Recent practices in research community have also shown the benefit of applying machine learning based approaches into systems decision-making. Machine learning techniques are used to reduce cooling cost bill in datacenters [4], virtual machine configuration [105], database management system (DBMS) configuration [98]. Besides, the progress in behavioral machine learning techniques like OpenAI Five [10], Atari Games [63], robotics [51] have ignited the enthusiasm of augmenting systems with such sequential decision making paradigm. For instance, DeepRM translates the resource management task into learning problem, and is comparably to state-of-the-art heuristics adaptive to different conditions and objective [54]. [25] presents a reinforcement learning (RL) based scheduler that can dynamically adapt to traffic variation in cellular networks. Pensieve applies deep reinforcement learning into video streaming tasks that generalizes across various network settings [55]. AuTo applies deep reinforcement learning to automate traffic optimizations instead of relying on heuristics of operators decoupling the decision making with respect to short flows and long ones [24]. Reinforcement learning has also been applied to replace heuristics in device placement [58], index structures like B-Trees [44] and graph-based cluster Scheduling [21].

3 Design

In this chapter, we start by motivating the need to design systems that can derive decisions based on their experiences to meet predefined goals. Then we dive into the packet scheduling problem and abstract it within a decision-making paradigm. We present the structure of the agent that is capable of adapting its behaviors. After that, we studied two concrete cases: cloning existing scheduling behaviors and exploring custom policies.

3.1 Motivation

Traditional network design solutions heavily rely on clever heuristics and manual configurations. As an example, today's TCP congestion control mechanism is filled with parameters that are tuned with heuristics, e.g., initial congestion window size (init_cwnd), additive increase/multiplicative decrease parameters in AIMD rate adaptation [39]. While in early days, these heuristic based systems have been hugely successful and effective, the overwhelming complexity, growing decision space, higher QoE expectations and increasing dynamics of modern networks are rendering such rigid network design paradigm to be sub-optimal and less satisfactory in response to variant conditions [29, 39, 104]. Such heuristic based methodology typically develops solutions based on a simplified model for ongoing problem that are intended to work well in general without adapting to the actual context. The growing degree of scale, heterogeneity and complexity of networks, it is increasingly hard to derive an accurate model and reach the global optimum with white-box design philosophy [74]. It is also increasingly challenging to capture workload-level characteristics with heuristics and even so, when certain aspects of the problem context change, the workflow of meticulous tuning has to be repeated, which compels us to explore the alternatives that can better address the ever-increasing complexity and dynamics of networks.

In this light, it is appealing to design systems that can learn to optimize their decision and behavior in the *local domain* by interacting with the environment and learning the system dynamics (i.e., exploration), and systematically exploit past experiences to make better decisions adaptively. Unlike heuristic based adaptive approaches which are fixed with hardcoded rules, here we are pursuing the one that exploit past experiences and wider sources of signal. We are enticed by the automatic&structured systems design setting where the human designers only states the intention (objectives, requirements, assumptions) as input triggering the automatic process synthesizing the information (intent, environment, workload) which will figure out the best corresponding behaviors [89]. Such paradigm enables us to equip the system with derived customized policy based on deep understanding of the environment and offers a promising direction to explore, meanwhile accompanying with challenges to be addressed, both from the statistical point of view regarding the used machine learning framework, as well as the systems point of view given we need to design systems to be more native and flexible with the continuous learning artifice.

3.2 Abstraction

3.2.1 Elements

To apply such paradigm, it is crucial to extract the key relevant components related to systems decision making behaviors.

- *Decision making interface*: the decision that exerts influence on the target environment of interests
- *Observation*: the information supporting the systems decision
- *Consequence*: the resulting behavior of the dynamic environment
- *Objective*: the ultimate goal of the system behaviors
- *Feedback loop*: the mechanism of improving the decision making mechanism
- *Metric*: the criteria to evaluate if the behavior excel in specific tasks

3.2.2 Formulation

We formulate packet scheduling as a decision making task on queue management. Although packet scheduling involves infinite steps and therefore exhibits a continuing task pattern, we treat it as an episodic task given the fact that packet flows come in a finite-step session nature. Upon each episode, a finite set of packet flows constitute a *workload* that would traverse the forwarding device(s) from the source to the destination.

Packet scheduling is essentially about making a decision on *when* and *which* packet to dequeue next. For simplicity, we assume that the scheduling is *non-preemptive*, i.e., the ongoing transmission of the packet can not be interrupted. Since the packet coming earliest within the same flow would take precedence always, we assume per-flow queue to maintain a generic representation across different scheduling algorithms. As shown in Figure 3.1, the classifier is to assign the packet to the flow based on its meta data [1]. By taking an action on a queue, the agent is dequeueing the front

[1]A flow can be identified by the tuple (source ip, destination ip, source port, destination port, protocol).

packet for the chosen queue. To be specific, we assume K queues to acquire a fixed state and action representation as neural net input. Upon each dequeue event, the agent decides on the queue to be scheduled in such linear action space, and if, the empty queue is chosen, the agent is taking an idle decision for a quantum time step, hence, it is not necessarily *work-conserving*.

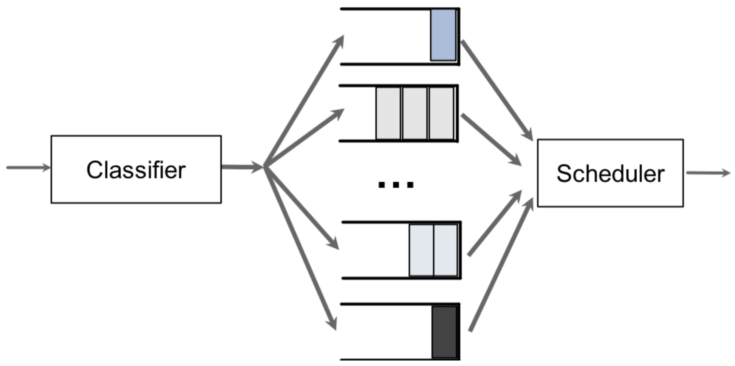

Figure 3.1: Packet Scheduling Abstraction

3.3 Agent Structure

The abstractions of the scheduling agent include the interface with the environment to exert actions and receive rewards, the internal representation of the policy, and the adaptive machinery.

3.3.1 Interface

The *observation* of the agent mainly consists of per flow statistics, including time of arrival, packet size of front packet in the queue, binary feature indicating presence of the queue, flow size, remaining flow size and so on. Besides, agent is also able to observe the historical scheduling decision log and information beyond local link statistics feedbacked by a global controller. The *action space* of the agent consists of the set of queues to dequeue, i.e., $\mathcal{A} = \{1, 2, 3, ..., K\}$. *Reward* is calculated depending on the context of interests, which guides the agent to explore the best policy achieving the intended objective.

3.3.2 Representation

Tabular Representation

Representation is about proposing hypothesis to model, e.g., the target policy function. In realistic tasks, it is necessary to leverage on parametrised representation since tabular approaches are with limited capabilities. Exact solutions with tabular form has the advantage of being straightforward and explainable, besides, it is *guaranteed* to reach global optima given enough visits of all possible states and actions.

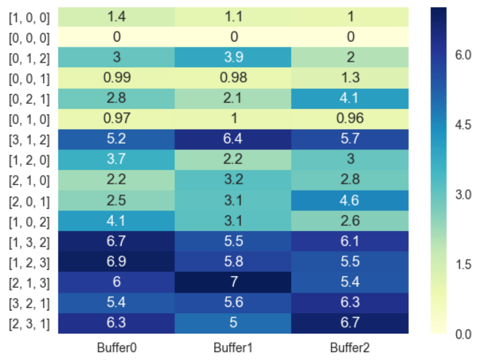

Figure 3.2: Tabular Representation Does Not Scale

However, it is impossible to store and visit all states given the explosion of state complexity. To illustrate, Figure 3.2 shows the learned $Q(s,a)$ table with tabular Q learning algorithm. Each tuple in the table stands for corresponding $Q(s,a)$ value. The agent determines the action a with $\epsilon-$greedy at state s and observes the environment transition $r, s^{'}$. The table gets updated by bootstrapping with $Q(s,a) \leftarrow Q(s,a) + \alpha[r + \gamma max_{a^{'}} Q(s^{'}, a^{'}) - Q(s,a)]$. In a simplified context where the action consists of 3 candidate buffers and the state is the priority of the front packet for the queue (0 indicates empty buffer and smaller tag indicates higher priority), after exploration, the policy encoded in the Q table is exactly SP when applying the greedy strategy, as demonstrated by the state action values. However, it does not scale since the state space possibilities corresponds to N^K, where N is number of possible priorities, not to mention the infinite case of continuous state like time stamps. Hence, it is inevitable to apply non-linear function approximators such as neural networks, that could interpret rich sensory inputs and enable generalization over limited experiences with a manageable number of learnable parameters, in order to scale to realistic complex tasks.

Parametrised Representation

As mentioned in section 2.1.2, deep models like CNNs are powerful for automatic feature extraction and detection. We consider to parametrise the policy and state value function with both CNNs and regular NNs, where $w, \theta \in \mathbb{R}^d$ correspond to weights, biases in the NNs.

$$\begin{aligned} V_w(s) &\approx V^\pi(s) \\ \pi_\theta(s,a) &\approx \pi(s,a) \end{aligned} \tag{3.1}$$

We use CNNs as the main form of representation in order to exploit the 3 dimensional structure of the input: flow, feature, and time frame. Considering that the correlation mainly exists along the dimension of flows instead of features, we apply 1D convolution [2] to each feature vector, i.e., the 1-D filter will only connect to a local region of each feature vector and share the weights along the dimension of flows.

It is worth mentioning that the value network is merely for assisting the experience learning during exploration, only policy network is triggered when the agent makes online decisions.

3.3.3 Internal Machinery

Preferences

The policy gradient algorithms are preferred over value fitting algorithms in our context. From the theoretical viewpoint, policy gradient methods are what actually performs gradient descent/ascent on desired objective $\mathbb{E}_{s_0 \sim p(s_0)}[V^\pi(s_0)]$. Policy gradient methods directly optimize the cumulative reward objective and can straightforwardly be used with nonlinear function approximators such as neural networks with guarantee to converge to (local) optima with gradient methods [94]. Value based methods, however, suffer from risks of divergence with non-linear function approximators. Besides, the process of minimising Bellman error is not the same as optimizing expected cumulative reward. Moreover, policy gradient methods could encode stochastic policies and could natually handle high-dimensional or continuous action space, compared with deterministic policies via value fitting methods combined with greedy methods.

From the practical perspective, although empirical techniques like fixed target, experience replay could alleviate the instability of approximate value fitting methods, it requires more engineering efforts and lacks the virtue of ease of use. What's more, such instability would undermine our trust into the systems behaviors. There are problems with policy gradient methods, though, they are inherently less sample efficient and suffer from high variances due to on-policy and Monto Carlo sampling. However, unlike robotics where acquiring sampling data is expensive, in network systems scenarios we consider the sample efficiency as trivial to some extent since there is a huge amount of input data with highly repetitive pattern. Thus, policy gradient methods are preferred for applications in our systems settings.

[2]Corresponding to tf.layers.conv1d in Tensorflow.

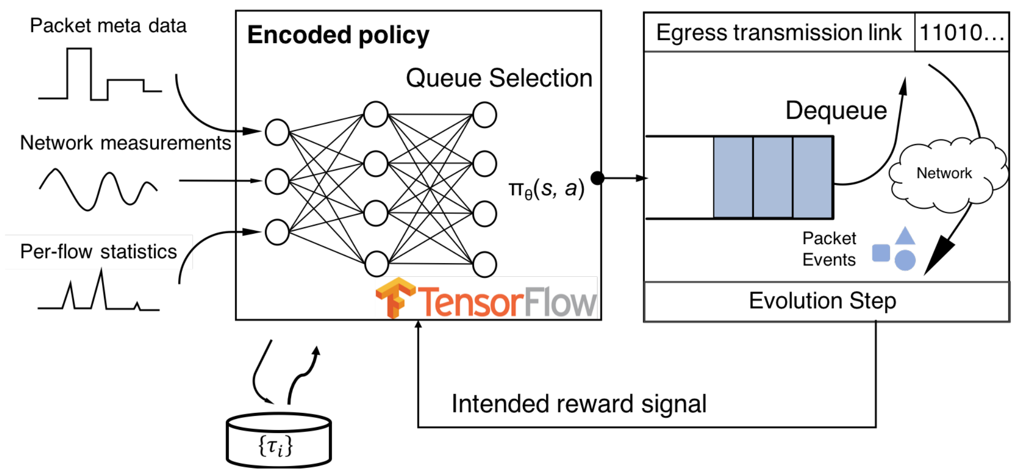

Figure 3.3: Framework Overview

Closed Loop Iteration

Figure 3.4 shows the general machinery of the agent. Starting with random policy indicating no prior knowledge of the task, the agent continuously interacts with the environment and receives the reward judging the quality of the agent's footprint, with which batches of experience tuples $E = \{e_0, e_1, ..., e_{T-1}\}$ are formed and used to improve the policy, typically by back propagating [49, 70] the fitted loss to the parametrised policy/value network with a specific learning rate[3]. The agent using policy gradient is *inherently* encoded with the mechanism to balance between exploration and exploitation: with more and more reinforces on positive actions, probability to sample poor actions is reduced, leading to further exploitation as the iteration proceeds, meanwhile, the opportunity of exploration is offered due to the sampling of actions during exploration.

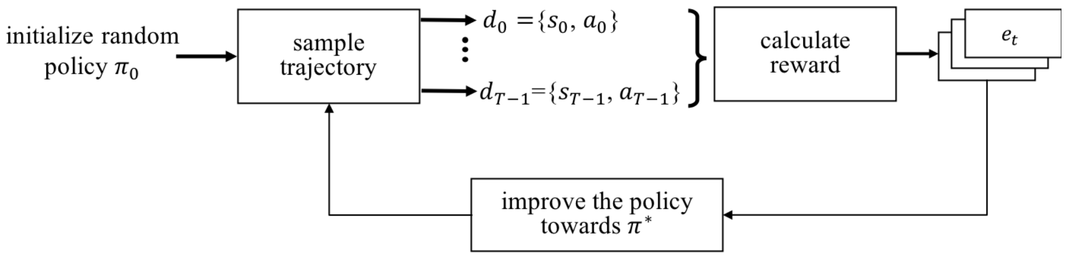
initialize random policy π_0
sample trajectory
$d_0 = \{s_0, a_0\}$
$d_{T-1} = \{s_{T-1}, a_{T-1}\}$
calculate reward
e_t
improve the policy towards π^*

Figure 3.4: General Pipeline

We consider a offline setting, i.e., separating the exploration and deployment phase. During the exploration, the agent explores the best behavior and actively adapts to

[3]We use adaptive Adam optimizer [22] to perform SGD.

the settings based on experiences, while during deployment, the agent machinery remains fixed. Though online setting seems appealing for full adaptiveness of systems behavior, it increases the risk when such exploration leads to dangerous outcomes, especially, when fallback mechanism is absent. What's more, exploration online involves significant computation burden overhead for real-time decision making. Therefore, in practice, the agent could be updated periodically depending on the frequency of scenario change in an offline fashion.

Baseline

As mentioned in Section 2.1.4, when trying to improve the policy, raw rewards could be too noisy for the agent during exploration and negate the learning process. Step-dependent simple average with "vanilla" policy gradient [77] requires fixed horizon. Although one could enforce the fixed step size across episodes via manual zero padding, such baseline gives little information judging the action at given state, and introduces significant noise when the steps of episodes are diverse even with exactly same driven traces, which is the case in our event triggered scenarios. We instead look at state-dependent baseline which reflects the average value of the state: during the improvement of the policy, the agent also fits the state value function with a parametrised network; during decision making, such value network is used to judge the expected reward of the state, and determine the quality of the action.

3.4 Learning Scheduling Policies

3.4.1 Formulation

The underlying question to explore could be framed as: is it *feasible* for an agent to self-learn different scheduling policies directly via experiences? Additionally, could we achieve such a goal while maintaining a consistent and generic representation of agent architecture, i.e., without elaborated feature engineering? Formally, the agent starts without any knowledge about the target scheduling policy π^*, i.e., an initial random policy π^0. The agent interacts with the environment and learns from historical experiences by updating current scheduling policy $\pi \rightarrow \pi'$ to approach π^*. Taking such road to learning existing approaches is appealing since we could not only reveal the potential of adaptive behaviors of cognitive agent, but also, in experience-hungry settings, cloning existing robust behaviors could be leveraged to bootstrap the system.

3.4.2 Taxonomy

First class of policies include FIFO, SP, SFF, SRPT, SJF and FQ. These policies filter out empty queues and dequeue the one with minimum relevant key (e.g., time of arrival, priority, packet size). More formally, the process of learning corresponds to $\pi_\theta(o) \approx \pi^*(o_{relevant}) = I_{g^{-1}\min o_{relevant}^{non-empty}}(a)$, where $g : A \to V$ is the mapping of action space to its feature space.

Second class of policies, RR and DRR (work-conserving), are more general in the sense it differs from just taking the flow with minimum key, i.e., $\pi_\theta(o) \approx \pi^*(o_{relevant}) = f(o_{relevant}^{non-empty})$. For instance, DRR takes into account both the deficit and front packet size of the queues and makes decision on the queue that will lead to largest remaining deficit.

For third class of policies, STFQ, WRR, WDRR, the agent does not observe the full space of features (e.g., predetermined weights), i.e., $\pi_\theta(o) \approx \pi^*(o_{relevant}, s_{internal})$. For example, for STFQ, it maintains a weighted virtual starting time for each flow, however, the agent neither observes the weights nor such per-flow statistics.

3.4.3 Methodology

To clone target scheduling behaviors, we need access to information regarding the target system. Cloning typical scheduling algorithms indicates the availability to full dynamics of the target scheduling algorithm (e.g., canonical ones like SJF), and we consider it straightforward that we run the target scheduling system under each step of the agent to signal the reward for the decision, e.g., if the agent successfully repeat the decision of the target scheduling algorithm, it will yield reward +1, otherwise 0. In the extreme case, such signal could indicate the exact truth to label the decision of the agent, which could be realized via supervised learning. We consider as well a more generic assumption where the only the input and output packet sequence of the target system are available and the machinery of the target system is a black box. To be more specific, we adopt the network model and definition in [60]. When we apply different scheduling policies $\pi_\alpha, \pi_\alpha^{'}$ for link α with same driven packets $\{(p, i(p), path(p))\}$, we consider π_α replays $\pi_\alpha^{'}$ with respect to the input if and only if for the set of output times $\{o(p)\}, \{o^{'}(p)\}$, $\forall p \in P, o^{'}(p) \geq o(p)$.

It is worth mentioning that it is infeasible to assign positive reward only if the output packet sequences produced by the agent and the target policies are exactly the same. The main reason is that the sparsity of the reward will grow exponentially. To illustrate, with 10 flows and 50 packets per flow on average in the workload, since the agent starts with random guess, it is expected to take 10^{500} episodes to just reach a first positive reinforce to reproduce the full sequence as target policy. Such extremely sparse reward will be discouraging the agent to learn effectively. It is also worth noticing that due to the avalanche effect of the decision making, a single deviation of the decision would lead to permanently different environment scenarios. Hence, for sequence level comparison, we assign positive rewards to those packet due to the target deadline and zero reward otherwise. As demonstrated by algorithm 3 using REINFORCE with state value baseline [93], we take advantage of the fact of the availability of the expert or oracle, to guide the agent with more dense reward signals. Each episode is of variant length, due to the nature of traffic settings, also, the agent updates its policy network and value estimates upon the end of the episode and receives no intermediate rewards during the scheduling process.

Algorithm 3: REINFORCE with State-value Baseline

```
1  function REINFORCEA;
   Output: policy network weights $\theta$
2  initialize $\theta$ and state value weights $w$ arbitrarily
3  for episode $i$ do
4      initialize the environment with traffic traces
5      for each scheduling step $t$ do
6          repeat
7              sample action $a_t \sim \pi_\theta$, $\hat{a}_t \sim \pi^*$ at state $s_t$
8              store record $d_t = (s_t, a_t, \hat{a}_t)$ in database $D$
9          until no new packets incoming;
10     end
11     for each record $d_t$ do
12         if $a_t == \hat{a}_t$ then $r_t \leftarrow 1$;
13         else $r_t \leftarrow 0$;
14         form experience tuple $e_t = (s_t, a_t, r_t)$
15     end
16     for $t = 0$ to $T_i - 1$ in sampled trajectory $\tau_i$ do
17         $G_t \leftarrow$ return from step $t$
18         $\delta \leftarrow G_t - \hat{v}(s_t, w)$
19         $w \leftarrow w + \beta\gamma^t\delta\nabla_w\hat{v}(s_t, w)$
20         $\theta \leftarrow \theta + \alpha\gamma^t\nabla_\theta \log \pi_\theta(s_t, a_t)\delta$
21     end
22 end
```

Algorithm 4: Workload exploration

```
1  Workload exploration;
2  sample $K$ workloads from the target workload distribution
3  initialize the parameterized policy $\pi_{\theta,w}$ randomly
4  for each iteration do
5      for each workload $\{z_t\}^{(k)}$ do
6          sample episodes $i = 1, 2, ..., N$
7          for $\{\tau_i\}$ do
8              $w \leftarrow w + \beta\gamma^t\delta\nabla_w\hat{v}(s_t, w)$
9          end
10         for $\{\tau_i\}$ do
11             $\delta \leftarrow G_t - \hat{v}(s_t, w)$
12             $\theta \leftarrow \theta + \alpha\gamma^t\nabla_\theta \log \pi_\theta(s_t, a_t)\delta$
13         end
14     end
15 end
```

3.5 Exploring Custom Policies

3.5.1 Formulation

Besides learning existing scheduling behaviors, we are compelled to explore the benefits underneath such intelligent agent, i.e., if the agent could explore herself policies that are comparable and even better than those leveraging on human heuristics with painstaking design workflow w.r.t. certain objective and certain workload.

3.5.2 Reward

Unlike learning scheduling policies, the design of reward is less explicit. One of the key challenges to explore custom policies in systems decision making is to design a proper reward scheme that both reflects *exactly* the intended objective and meanwhile maintains learnability for the agent towards the target.

Queueing Delay

End-to-end delay (half of RTT) of a packet consists of transmission delay, propagation delay, processing delay and queueing delay. While the former three sources of delay are relatively static and are mainly determined by the hardward configurations and network infrastructures, queueing delay is highly correlated with the decision making of the agent. Hence, minimizing queueing delay could help reducing end-to-end delay given fixed network infrastructures.

We are interested into minimizing average queueing delay for all the packets passing through the link. Hence, the objective could be formulated as $\sum_i^N T_i^Q$, where N is the total number of packets in the workload. The reward could be formulated as the penalty for the packets queueing in the buffer at each step, i.e., $R_t = -\sum_i^n (t_{next} - t_{current})$, where n refers to total number of packets in the buffers. To avoid bias on large size packets, a variant of objective normalized with packet size could be formulated as $\sum_i^N T_i^Q / L_i$.

3.5.3 Methodology

Analysis

Compared with Section 4.2.1, the agent is faced with a more challenging task: not only the reward signal is more noisy but also the observation of the environment is much more partial. The agent neither knows exactly the statistics of the packets apart from the front packet of the queue nor the time advance to the next state during the decision making, besides, the input driven traces are online with characteristics unknown to the agent a priori, which unfortunately are part of the calculation of the reward related to the true objective (queueing time, flow completion time and so on). The environment uncertainties also combines the dynamics of the online input traces in the form of input driven MDP [56]. All these makes the exploration task much more challenging.

Machinery Augmentation

When exploring custom policies with a noisy environment, the choosing of policy learning rate is crucial: a large learning rate will probably leads to a drastic hop to a poor policy and therefore generates bad experiences, leading to worse policy learned and worse experiences and so on; a small learning rate, however, will lead to slow improvement and significant decrease of exploration efficiency. Although the tuning of policy learning rate is intuitive, with noisy and challenging tasks, and it is typical to reduce the learning rate, yet, to find a best tradeoff between the exploration efficiency and stability can be time consuming, especially for the input driven environment where the full dynamics of the environment is determined also by the online arriving traces, leading to even more non-stationary experiences collected by the agent. Therefore, the agent maximizes surrogate objective subject to a constraint on the size of the policy update, as suggested by TRPO [78]. In practice, it corresponds to unconstrained optimization problem with clipped surrogate objective [80].

Incorporating parallel mechanism could speed up the training and overcome the limitation of correlated experiences collected with single agent.

Hence, each fixed workload is exposed to N workers separately and collect the experience tuples e_t to the global agent which performs the gradient update and shares the encoded policy with the workers. This could happen asynchronously without locking the workers when the global agent is performing update.

Reward Shaping

We also consider changing the reward as the exploration proceeds [64, 93], specifically, we find it empirically helpful to punish the agent in the early phase when it is non-work-conserving, and then proceeds with the exploration of the intended objective. The reward formulation for directing the agent to be work-conserving is straightforward, when the agent makes a decision on an empty queue, we punish with reward -1, otherwise 0.

4 Evaluation

4.1 Simulator

Unlike games, learning via trial and error in real world network systems can be extremely expensive and risky. Besides, network operators are often reluctant to carry out the deployment with concerns of security, cost, and SLA violations [18]. Hence, we adopt the typical practice of using simulator in reinforcement learning to study the prototype before we gain real confidence in them and move it into real-world system testbed. Such simulating approach also allows the agent to experience the environment with stronger flexibility without being restricted by the wall-clock time interactions. To be specific, we build our own packet-level simulator with extracted interesting elements and feed with realistic traffic traces and settings, such simulated environment allows the agent to gain experience without constraints of wall-clock time and heavy overheads. Existing simulators like NS-3 [9], or network emulator like mininet [7] involve heavy stack of burden for the exploration of machine learning approach.

Simplification

For packet scheduling, the key is to abstract the life time of the packets, which includes the synthesizing of the packet, forwarding, enqueue, dequeue at a switch link, and sink at the destination host. Hence, we simplify the routing behavior of routers via hardcoding the forwarding table prior simulation based on the synthesized flows. The intricacies related to finite state machine of end hosts, protocol stacks as well as congestion control mechanism at network layer are neglected as well. Besides, we set the buffer size to be large enough to prevent packets from dropping, since we are mainly concerned with packet scheduling, not buffer management. Sequences of packets are therefore generated at the source nodes and traverse through the forwarding devices until arrivals at the destination endpoints. These aspects of simplifications could be important for scheduling in commercial routers, however, the simplified model captures the essence of packet life time and provides a non-trivial and basic setup.

Traffic Synthesizer

The dynamics of each packet flow is modeled as poisson process with certain rate λ_i, correspondingly, the time intervals between successive arrivals follow exponential distribution. Hence, packet arrivals at the edge ingress link is a superposition of n

mutually independent poisson process, with parameter $\lambda = \sum_{i=0}^{n-1} \lambda_i$. However, each output link is not exactly the same as M/M/1 queue. Though it is assumed with constant packet size for each flow and constant processing speed at each transmission link, the transmission link is not necessarily work-conserving and scheduler does not necessarily follow FIFO policy. Hence, the service time does not follow exponential distribution.

Evolution

The construction of environment dynamics generally follows the methodology of event-driven simulation [23,31]. The scheduler maintains a time-variant event list $L = \{(k_i, e_i)\}, 0 \leq i \leq N_s - 1 \ldots$ which consists of the feasible event set $\Gamma(s)$ together with associated clock value k_i at each simulator state. Such collection of key-value pairs are represented as priority queue and stored in the form of array-based binary heap [33,81]. The event set is abstracted with $E = \{enquque, dequeue, evict, sink\}$ in which enqueue, dequeue, evict, sink events could be triggered at different locations of the network. The scheduler determines the triggering event $e^{'} = \text{argmin}_{i \in \Gamma(s)} k_i$. The environment evolves to the next state $s^{'} = f(s, e^{'}), s \in S$ with the advance of system clock $t^{'} = t + k^*$ where $k^* = \text{argmin}_{i \in \Gamma(s)} k_i$ and accompanying update of L. The simulation is terminated based on the predetermined criteria.

More generally, the agent is dealing with dequeue event pulses, unlike typical reinforcement learning practices which assume a quantum time in the environment dynamics. With packet scheduling context, assuming a minuscule time step, e.g., time to transmit 1 byte, will overwhelm the scheduler by a huge number of void triggering, while presuming a more coarse grained time step, e.g., time to send 100 bytes would lead to numerous fragmentations and idle behaviors.

4.2 Learning Scheduling Policies

4.2.1 A Generic Example

Configurations

We assume $K = 10$ queues, and feed the agent with per-flow states including front packet size (Bytes), time of arrival of front packet (seconds), flow priority, flow size (Bytes)[1], remaining flow size (Bytes), binary feature indicating presence of the buffer, scheduling decision log, virtual finishing round, and deficit of each queue, which counts up to 9 features, corresponding to 9 1D-CNNs in the policy network, as shown in the Figure 4.1 visualization with Tensorboard flashlight. To exploit the auto differentiation mechanism of Tensorflow, the policy gradient loss is defined as the cross-entropy, same as the loss defined in maximum likelihood

[1]Though accurate flow size information is hard to obtain in many network scenarios [17]

$J_{ML}(\theta) = \frac{1}{N}\sum_{i=1}^{T}\sum_{t}^{T}\log\pi_\theta(a_{i,t}|s_{i,t})$, but weighted by the advantage rollout. Huber loss is applied as an alternative of clipped squared loss for the value network.

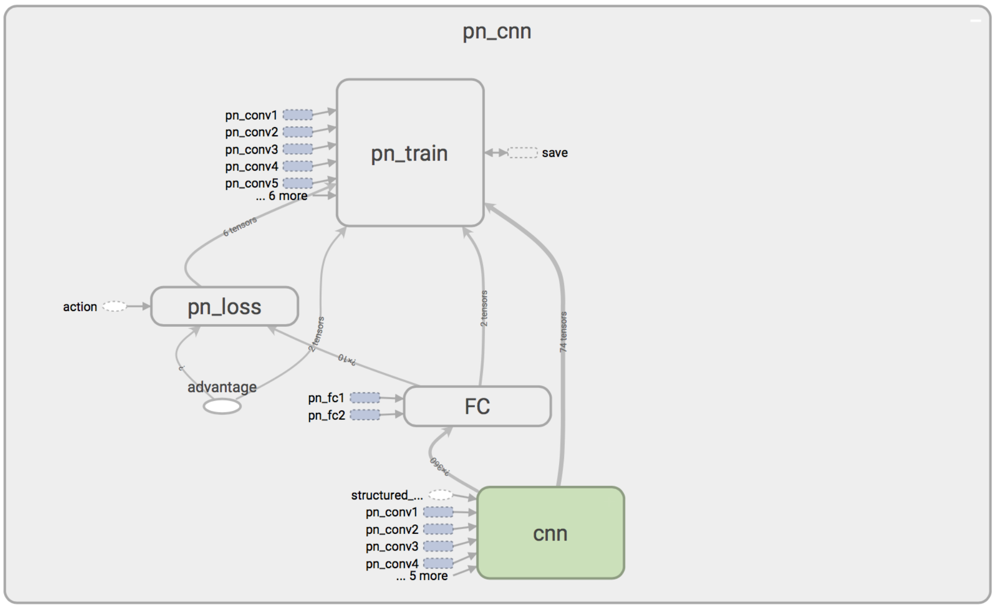

Figure 4.1: Tensorboard Visualization of Policy Network

We simulate the workload in which the characteristics of each flow are uniformly picked from corresponding pools in order to sweep uniformly. Specifically, number of packets for each flow ranges from 100 to 500, packet length ranges from 50 bytes to 500 bytes, flow priority is randomly assigned from $0 \sim 9$, starting time $10^{-10} \sim 10^{-5}$s, transmission speed 1Gbps, link utilization ranges from 80% to 150% and relative share among flows ranges $1 \sim 4$. Random seed is set to 2018 unless otherwise stated.

Across all the experiments, we adopt the *same* set of hyper parameters of the agent, in which reward decay $\gamma = 0.9$, kernel size $F = 3$, number of filters $K = 4$, stride $S = 1$, zero padding $P = 1$ (SAME padding), learning rates $\alpha = 0.001, \beta = 0.001$. Such configurations yield approximately 23788 learnable parameters for policy network and value network respectively.

Learning Curves

Since the agent will either collect reward 0 or 1 for the scheduling decision, the closer the similarity to 100%, the more the closeness of the encoded policy is to the target scheduling policy.

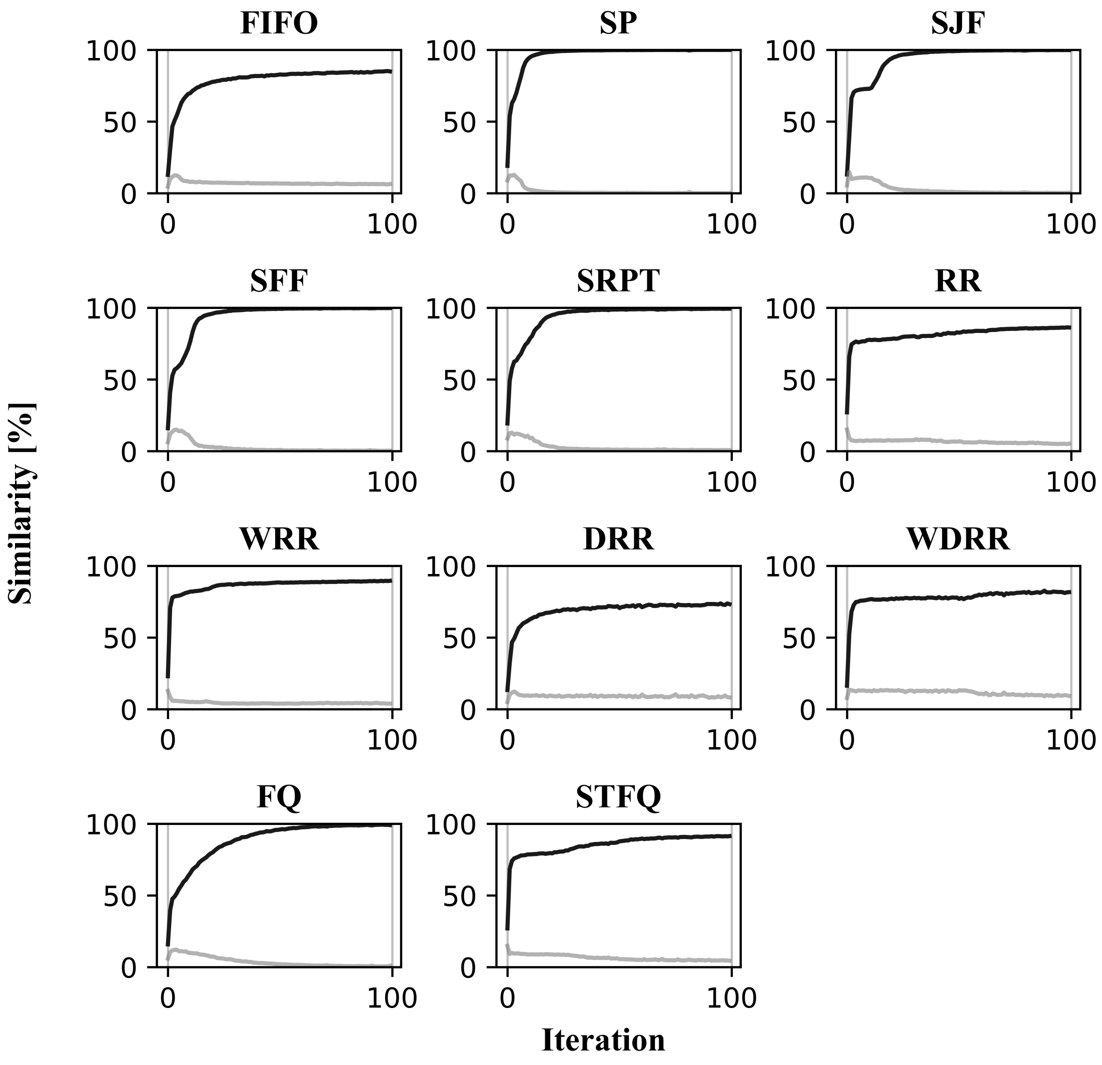

Figure 4.2: Learning curves for target policies during the primitive exploration. Highlighted curve is the mean similarity achieved for the workload set and the muted curve corresponds to the standard deviation.

We also test each learned policies under unseen examples, which is shown in Figure 4.3. During testing, the agent makes deterministic decision with maximum likelihood, with an extra mask [55] indicating valid queues.

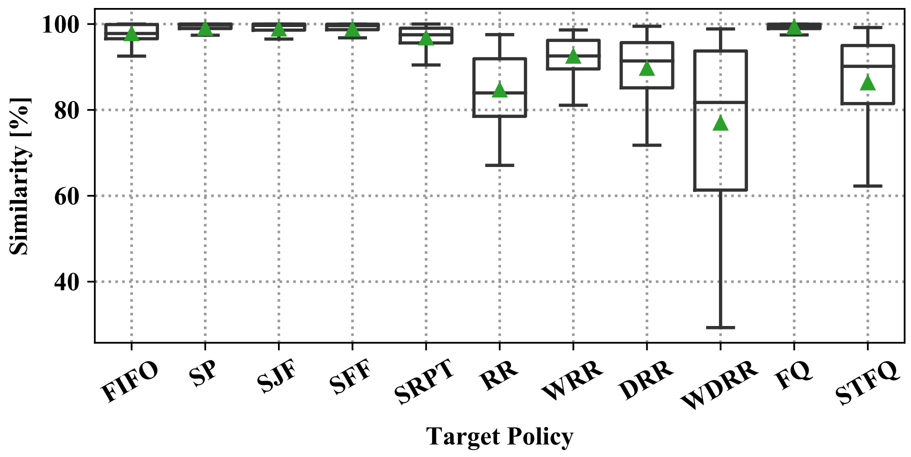

Figure 4.3: Generalization test with 500 workloads each with unseen samples, using seed 12345.

Implications

From the results above, we observe that effectiveness of learning degrade from type one policy to policy three, since the observation is getting more partial and policy logic is getting more sophisticated. Within specific type of target policy, state space complexity also acts as an factor influencing the learning. For instance, the state space of FIFO consists of continuous time stamp values, while for SP just labels of limited values of priorities. Note that, the agent is fully model-free with no knowledge a-priori, the hyper parameters are static across all experiments and are not fine-tuned for specific settings. Hence, the primitive training results show that the agent is able to adapt its behavior towards the intended policy.

Figure 4.3 shows that FQ is learned pretty well, however, given the assumption that we provide with per-flow statistics of virtual starting time, the states maintained by the switch when implementing FQ. It is therefore appealing to ask, considering the prohibitive cost of maintaining these statistics which afflicts the FQ implementation in reality, can we actually learn FQ without such input? Unfortunately, theoretically we can't exactly learn such FQ without the input of internal states. The policy encoded by the agent is simply a computing graph mapping observations to the probabilities of discrete actions. If there is deterministic mapping of agent state to scheduling decision for the target scheduling policy, the policy could be learned and encoded with 100% theoretically, otherwise, the best outcome is to reach a probabilistic policy that approximates the policy. If without the per flow statistics maintenance, which is the input for the FQ implementation, the same input of the agent will corresponds to multiple possible list of FQ input, such one-to-multiple mapping will lead to non-deterministic encoded policies of the agent. However, with FQ, such logic is deterministic, hence, theoretically, it is not possibly learn exactly the specific policy unless the agent input is one-to-one mapping of the target policy. Same arguments apply to DRR as well which maintains a per flow deficit list. However, it is interesting to ask instead if we could explore a fair share policy by the systems itself to approximate the goal without maintaining the state being expensive in implementation.

4.2.2 Discussion

Replayability

In section 4.2.1, we demonstrates that the agent is able to adapt its behavior with uniform internal structure to different target scheduling policies. We define the criteria of similarity as sanity check. However, in network scenarios, previous decisions will trigger cascading effects on the following states of the environment. Hence, we are interested into observing the closeness of learned policy to the target scheduling from network perspectives given such temporal impact. We take learned FIFO model as an example to evaluate the degree of replayability. As seen in Figure 4.4, the percentage of packets overdue increases as link utilization grows, since with sparse traffic, the cascading effect could be alleviated.

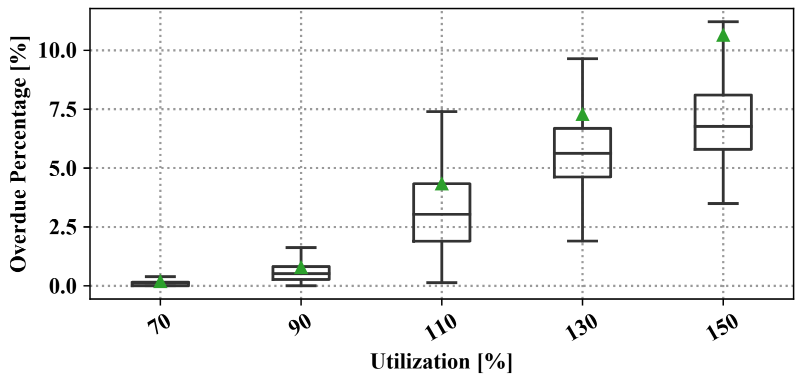

Figure 4.4: Fraction of packet overdue with various link utilization evaluated with 500 episodes, using random seed 12345.

Sequence Comparison

There are alternatives for learning existing approaches. To the extreme, if the target systems is fully a white box, we could rely on supervised learning that could even

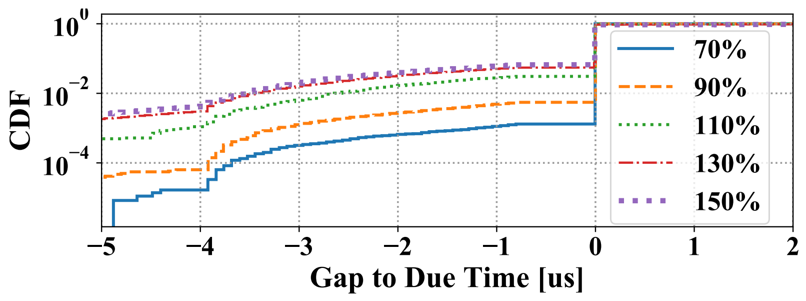

Figure 4.5: Cumulative distribution function for due gap of output packet sequences with various link utilization.

boost the training efficiency since the signal provides the exact label of correct action and is less noisy. Such effectiveness holds especially for learning policies where the agent has full access of the inputs of target policy, however, this comes at a cost of collecting exactly the decision trajectories of the target system. In reality, we are also interested if we have only access to the input and output for a real-world system with internal dynamics unknown. In this case, rewards are given by comparing the output packet sequence and positive rewards are directed to actions that lead to packets meeting the due. This could be useful for learning target policies where the agent has only access to partial observations. As an example, we let agent learns via both methodology without the input of per-flow virtual finishing round which is used to calculate FQ scheduling decisions. We evaluate two learned policies with the same trace sampled with a different random seed 12345. We observed that the percentage of packets meeting the due with sequence comparison approach is slightly better, corresponding to $82.72 \pm 4.78\%$ and $80.43 \pm 8.96\%$.

Performance of Baseline

In order to reduce the variance, we adopt the practice of adding a bias-free state-dependent baseline that predicts $v_\pi(s) = \mathbb{E}_\pi[G_t|s_t = s]$ and evaluates the relative quality of the action compared with the average. Figure 4.6 compare the performances between the case with and without a state value baseline, using the case of learning FIFO.

As seen in Figure 4.6, the addition of baseline not only reduces the variances but also increase the exploration efficiency. The main reason is that while improving the policy, the loss is computed as the cross entropy multiplied by the advantage instead of the raw cumulative return. The action that yields ”high” reward in an absolute sense can be below the baseline which will be treated with penalty while in the raw policy gradient, it is considered as a reinforce as well, which suffers the exploration with more variances.

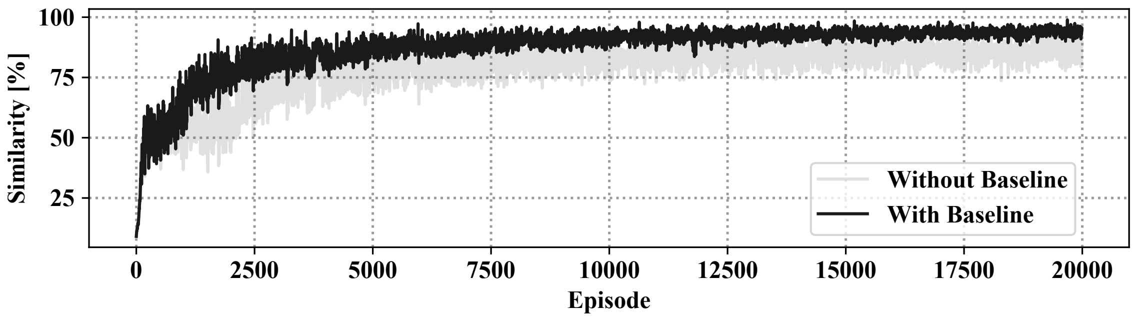

Figure 4.6: Performance of the state-dependent baseline in the FIFO example. Both scenarios share the same driven traces and hyper parameters except the baseline. Both curve is the moving average with window size 10 of raw reward curve.

N	1 (128)	2 (128-64)	3 (128-64-32)	4 (128-64-32-16)
$\eta^{explore}$ (%)	96.48 ± 2.92	98.79 ± 1.05	99.01 ± 0.85	99.36 ± 0.57
$\eta^{exploit}$ (%)	98.07 ± 1.84	99.15 ± 0.94	99.47 ± 0.65	99.52 ± 0.59

Table 4.1: Sensitivity of performance with respect to hyper parameters in network architecture.

Representation Architecture

In the example, we take the 1D CNN to locally filter the feature along the dimension of flows. Though we did not fine tune the parameters of the agent structure, it is still of interest that how would parameters tuning impact the result. We adopt the practice to sweep the hyper parameters for the architecture to observe its impact [55]. We vary the number of hidden layers N, in which the number of neurons are adjusted in tandem, for example, with 2 hidden layers we have 128-64. Table 4.1 shows that the addition of layer complexity could help increase the marginal performance. However, it is worth mentioning that there is a great deal of room for improving further (e.g., stacking more layers, testing other NN variants) and our prototype focuses mainly on the feasibility.

4.3 Exploring Custom Policies

We synthesize flows based the realistic trace distribution [19]. We found that exploring directly the optimal policy to minimize the queueing delay is hard for the agent given such partial observation. Besides, with Policy Gradient, it is risky when at some point the agent moves towards a bad policy, which will collect bad experiences and stuck at a worse policy.

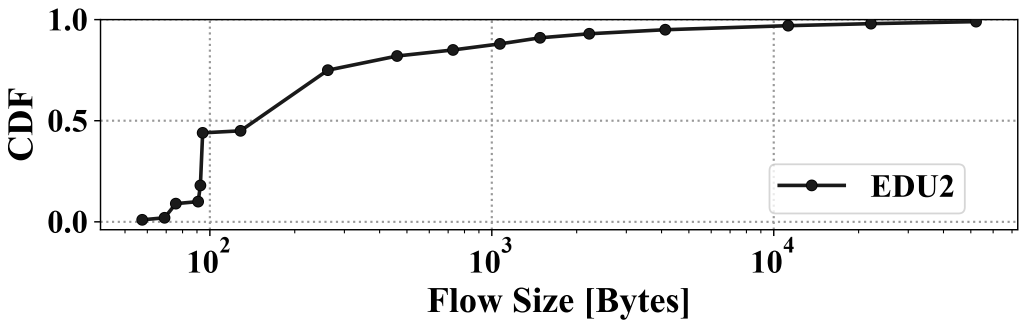

Figure 4.7: Flow Size Distribution of the Synthesized Trace

Hence, we leverage PPO to control the degree of policy change, with policy learning rate 0.0001, and critic network learning rate 0.001. We bootstrap the agent with 500 iterations trying to learn to be work-conserving and shortest job first. With both exploration pipelines, the agent explores the policy that is close to SJF, the optimal canonical policy in a single link.

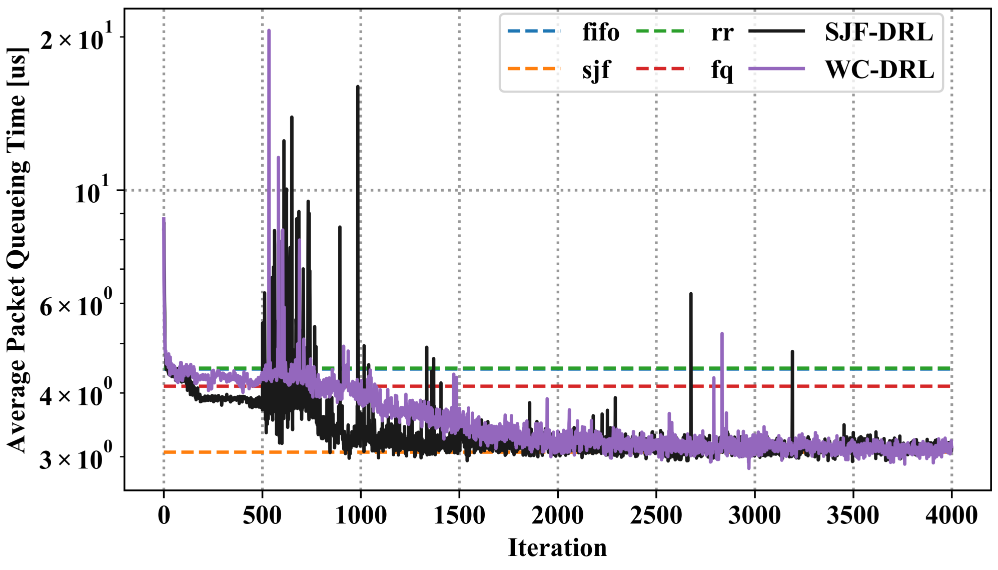

Figure 4.8: Exploration Curve with Proper Bootstrap

4.4 Practical Viewpoint

Although this work is actually looking at a prototype of queue management without actually diving into concrete real-world system implementation, we discuss several practical implications for future deployment.

4.4.1 Limitation

Scalability

We assume the agent is dealing with maximum 10 queues at the same time, others could be backup in the backlog. Yet in reality, there is need to augment such capacity, especially for core switches. As a proof of concept, we run a series of FIFO example with fixed wall clock time 120 hours (including the time burden of corresponding simulation) and single CPU core (moving to parallel implementation or GPU could accelerate the process), as shown in Figure 4.9.

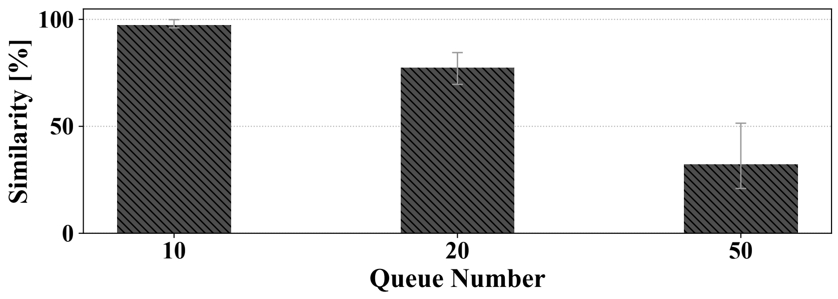

Figure 4.9: Training benchmarks obtained with limited 120h wall clock time and single CPU core for different queue numbers.

With the increment of queue capacities, the complications come in several dimensions. First, the policy network will need to be augmented with more output neurons and corresponding addition of neurons in the hidden layer to adjust the capacity of the model, which indicates we need more training epochs. Besides, the agent also has to deal with longer horizon if more scheduling steps could be involved in an episode. Though scaling to a larger magnitude could be feasible, but the accompanying cost might be prohibitively expensive. To illustrate, the recent work on OpenAI Five indicates the feasibility of exploring tasks with a long horizon, large discrete action space. They managed to deal with an action space of magnitude 1000 simultaneously and a horizon of 80000 ticks, yet at a heavy cost of 256 GPUs, 128000 CPU cores corresponding to around 180 years per day [10].

Real-time Constraint

In the simulation, we do not account for the processing time of deriving scheduling decisions, however, in real-world systems, the agent has to make timely decisions. Hence, we measure the processing time to make a single dequeue decision (i.e., inference) using the learned model with Tensorflow CPU module and 2.9 GHz Intel Core i7, as shown in 4.10.

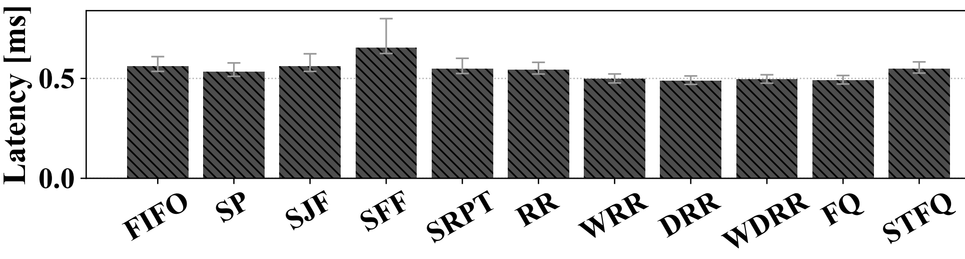

Figure 4.10: Processing delay of making single dequeue decision for each learned policies with 10^5 measurements respectively. The bar indicates the mean, upper quartile Q_3 and lower quartile Q_1.

For packet scheduling, 0.5 ms will corresponds to scheduling 62500 Bytes of data with 1Gbps egress link, during which around 80% of the short flows would vanish in traces like EDU2 shown in 4.7, let alone a single packet. Though the computation time is model specific as well as platform dependent and machine learning specialized hardware might alleviate the issue, the result reveals the potential concern of the real-time constraint in magnitudes when deploying machine learning computing model online, especially for line-rate frequent decision making.

4.4.2 Implications

The evidence shown above indicates the limitations and boundaries when we try to apply such paradigm into real-world systems. For real-time applications, we also have to consider the constraints on the frequency of making decisions. Besides, this work we mainly look at the line-rate interface of scheduler mainly because we want to be consistent with canonical approaches, however, with packet scheduling context, augmenting intelligence directly on current core switches is not feasible and ready for wide deployment due to the overwhelming cost of maintaining dynamic, large-volume pre flow statistics and incompatibilities to machine learning computation complexity.

Given the current router technology which supports only a simple priority queue based policy, it is more realistic to look at alternative interfaces with proper distributed design to reduce the overhead. Echoing the end-to-end argument [72], it is appealing to instead augment intelligence at the edges to cooperate with core devices. One typical example is Core Stateless Fair Queueing (CSFQ) [91], which decomposes a group of routers into hierarchies of edges and core routers and transfers the burden of maintaining per flow statistics to edge routers. Core routers is deployed with a simple policy (e.g., SP) for line-rate decision making while the edges makes decision on tagging the flows. By tagging the packets with fix range of priorities, we could fix the core routers with commercial priority queue scheduling.

5 Conclusion

5.1 Summary

In this work, we focus on the viability of augmenting systems ability to adaptively learn from its experiences in order to tailor for the specific settings. We have shown the promising potentials of learning approaches in systems with the case of packet queue management that could clone the existing policies and explore the policy end-to-end to meet certain objective. As apposed to conventional hardcoded, explicitly defined commands, such agent paradigm could explore and exploit from prior experiences on herself.

5.2 Future Work

This thesis is an early phase attempt to explore, identify, and understand the challenges and opportunities to augment systems adaptiveness with behavioral machine learning framework in response to the ever increasing heterogeneity and complex environment. We are enticed by the promising benefits of bringing the paradigm into implementation on real system platform. Towards this, there is a broad spectrum of promising potentials to exploit further towards this direction in the future.

From the systems point of view, we have to shift our paradigm to problem with global scale and formulate corresponding design. How can we design such systems that could meet the online requirements, e.g, making timely and robust decisions with robust responsiveness under environment variations? How do we cope with common security concerns, how do we ensure that the agent is not making disastrous decisions and how shall we introduce fallback mechanism to control the risks? Besides, with current hardware compatibilities, how can we position the decision making components properly to enable incremental augmentation in commercial platforms? How can we actually design such systems with minimal resource requirements in large scale deployment? How to determine the point when it is necessary to retrain the model catering to the new pattern of traffic or even enable a fully online setting? How would the agent perform when there are also many other objectives to explore, e.g., fairness, deadline. Besides, though this work exhibits a strong preference on model-free approaches, would it be better that we combine and benefit from the model of traditional design wisdom as well? Can we exploit the temporal and spatial locality in the scheme?

From the statistical perspective, applying machine learning mechanism into systems domain is faced with specific challenges. To illustrate, systems online decision making could be with long time horizon, sparse and delayed reward and so forth, which opens the opportunities to exploit hierarchical reinforcement learning, inverse reinforcement learning, transfer learning, neuro evolutionary methods and so forth. In order to really trust and exploit the paradigm as an alternative of canonical approaches, it is also of vital importance that we enable the learning process and policies to be explainable from human administrators in a systematic manner.

Bibliography

[1] Alphago at the future of go summit. deepmind.com/research/alphago/alphago-china/. Accessed: 2018-03-10.

[2] Alphago zero: Learning from scratch. deepmind.com/blog/alphago-zero-learning-scratch/. Accessed: 2018-03-10.

[3] Caffe. caffe.berkeleyvision.org. Accessed: 2018-08-01.

[4] Deepmind ai reduces google data centre cooling bill by 40%. deepmind.com/blog/deepmind-ai-reduces-google-data-centre-cooling-bill-40/. Accessed: 2018-03-10.

[5] Euler cluster. scicomp.ethz.ch/wiki/Euler. Accessed: 2018-05-20.

[6] Leonhard cluster. scicomp.ethz.ch/wiki/Leonhard. Accessed: 2018-05-20.

[7] Mininet. mininet.org. Accessed: 2018-08-10.

[8] Mxnet. mxnet.apache.org. Accessed: 2018-08-01.

[9] ns-3. www.nsnam.org. Accessed: 2018-08-10.

[10] Openai five. blog.openai.com/openai-five/. Accessed: 2018-08-10.

[11] Pytorch. pytorch.org. Accessed: 2018-08-01.

[12] Tensorflow. www.tensorflow.org. Accessed: 2018-08-01.

[13] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. *arXiv preprint arXiv:1603.04467*, 2016.

[14] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, et al. Tensorflow: A system for large-scale machine learning. In *OSDI*, volume 16, pages 265–283, 2016.

[15] O. Alipourfard, H. H. Liu, and J. Chen. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics.

[16] M. Alizadeh, S. Yang, M. Sharif, S. Katti, N. McKeown, B. Prabhakar, and S. Shenker. pfabric: Minimal near-optimal datacenter transport. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 435–446. ACM, 2013.

[17] W. Bai and K. Chen. Information-agnostic flow scheduling for commodity data centers.

[18] M. Bartulovic, J. Jiang, S. Balakrishnan, V. Sekar, and B. Sinopoli. Biases in data-driven networking, and what to do about them. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, pages 192–198. ACM, 2017.

[19] T. Benson, A. Akella, and D. A. Maltz. Network traffic characteristics of data centers in the wild. In *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*, pages 267–280. ACM, 2010.

[20] D. P. Bertsekas, D. P. Bertsekas, D. P. Bertsekas, and D. P. Bertsekas. *Dynamic programming and optimal control*, volume 1. Athena scientific Belmont, MA, 2005.

[21] H. M. M. S. S. Bojja and V. M. Alizadeh. Learning graph-based cluster scheduling algorithms.

[22] L. Bottou. Stochastic gradient descent tricks. In *Neural networks: Tricks of the trade*, pages 421–436. Springer, 2012.

[23] C. G. Cassandras and S. Lafortune. *Introduction to discrete event systems*. Springer Science & Business Media, 2009.

[24] L. Chen, J. Lingys, K. Chen, and F. Liu. Auto: scaling deep reinforcement learning for datacenter-scale automatic traffic optimization. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication*, pages 191–205. ACM, 2018.

[25] S. Chinchali, P. Hu, T. Chu, M. Sharma, M. Bansal, R. Misra, and M. Pavone. Cellular network traffic scheduling with deep reinforcement learning.

[26] D. D. Clark, C. Partridge, J. C. Ramming, and J. T. Wroclawski. A knowledge plane for the internet. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 3–10. ACM, 2003.

[27] J. Dean. Machine learning for systems and systems for machine learning.

[28] A. Demers, S. Keshav, and S. Shenker. Analysis and simulation of a fair queueing algorithm. In *ACM SIGCOMM Computer Communication Review*, volume 19, pages 1–12. ACM, 1989.

[29] N. Dukkipati, T. Refice, Y. Cheng, J. Chu, T. Herbert, A. Agarwal, A. Jain, and N. Sutin. An argument for increasing tcp's initial congestion window.

[30] N. Feamster and J. Rexford. Why (and how) networks should run themselves. *arXiv preprint arXiv:1710.11583*, 2017.

[31] G. S. Fishman. *Discrete-event simulation: modeling, programming, and analysis*. Springer Science & Business Media, 2013.

[32] I. Goodfellow, Y. Bengio, and A. Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[33] M. T. Goodrich, R. Tamassia, and M. H. Goldwasser. *Data structures and algorithms in Python*. John Wiley & Sons Ltd, 2013.

[34] P. Goyal, H. M. Vin, and H. Chen. Start-time fair queueing: a scheduling algorithm for integrated services packet switching networks. In *ACM SIGCOMM Computer Communication Review*, volume 26, pages 157–168. ACM, 1996.

[35] M. P. Grosvenor, M. Schwarzkopf, I. Gog, R. N. Watson, A. W. Moore, S. Hand, and J. Crowcroft. Queues don't matter when you can jump them! 2015.

[36] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.

[37] N. Heess, S. Sriram, J. Lemmon, J. Merel, G. Wayne, Y. Tassa, T. Erez, Z. Wang, A. Eslami, M. Riedmiller, et al. Emergence of locomotion behaviours in rich environments. *arXiv preprint arXiv:1707.02286*, 2017.

[38] C.-Y. Hong, M. Caesar, and P. Godfrey. Finishing flows quickly with preemptive scheduling. In *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*, pages 127–138. ACM, 2012.

[39] J. Jiang, V. Sekar, I. Stoica, and H. Zhang. Unleashing the potential of data-driven networking. In *International Conference on Communication Systems and Networks*, pages 110–126. Springer, 2017.

[40] M. I. Jordan and T. M. Mitchell. Machine learning: Trends, perspectives, and prospects. *Science*, 349(6245):255–260, 2015.

[41] L. P. Kaelbling, M. L. Littman, and A. W. Moore. Reinforcement learning: A survey. *Journal of artificial intelligence research*, 4:237–285, 1996.

[42] A. Klimovic, H. Litz, and C. Kozyrakis. Selecta: Heterogeneous cloud storage configuration for data analytics. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 759–773, Boston, MA, 2018. USENIX Association.

[43] D. Koller and N. Friedman. *Probabilistic graphical models: principles and techniques*. 2009.

[44] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis. The case for learned index structures. In *Proceedings of the 2018 International Conference on Management of Data*, pages 489–504. ACM, 2018.

[45] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc., 2012.

[46] J. Kurose and K. Ross. Computer networking: A top-down approach. *Cell*, 757(239):8573, 2013.

[47] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *nature*, 521(7553):436, 2015.

[48] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[49] Y. A. LeCun, L. Bottou, G. B. Orr, and K.-R. Müller. Efficient backprop. In *Neural networks: Tricks of the trade*, pages 9–48. Springer, 2012.

[50] J. Y.-T. Leung. A new algorithm for scheduling periodic, real-time tasks. *Algorithmica*, 4(1-4):209, 1989.

[51] S. Levine, C. Finn, T. Darrell, and P. Abbeel. End-to-end training of deep visuomotor policies. *The Journal of Machine Learning Research*, 17(1):1334–1373, 2016.

[52] Y. Liang, M. C. Machado, E. Talvitie, and M. Bowling. State of the art control of atari games using shallow reinforcement learning. In *Proceedings of the 2016 International Conference on Autonomous Agents & Multiagent Systems*, pages 485–493. International Foundation for Autonomous Agents and Multiagent Systems, 2016.

[53] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[54] H. Mao, M. Alizadeh, I. Menache, and S. Kandula. Resource management with deep reinforcement learning. In *Proceedings of the 15th ACM Workshop on Hot Topics in Networks*, pages 50–56. ACM, 2016.

[55] H. Mao, R. Netravali, and M. Alizadeh. Neural adaptive video streaming with pensieve. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication*, pages 197–210. ACM, 2017.

[56] H. Mao, S. B. Venkatakrishnan, M. Schwarzkopf, and M. Alizadeh. Variance reduction for reinforcement learning in input-driven environments. *arXiv preprint arXiv:1807.02264*, 2018.

[57] N. McKeown. Software-defined networking. *INFOCOM keynote talk*, 17(2):30–32, 2009.

[58] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, and J. Dean. Device placement optimization with reinforcement learning. *arXiv preprint arXiv:1706.04972*, 2017.

[59] R. Mittal. *Towards a More Stable Network Infrastructure*. PhD thesis, EECS Department, University of California, Berkeley, Aug 2018.

[60] R. Mittal, R. Agarwal, S. Ratnasamy, and S. Shenker. Universal packet scheduling. In *Proceedings of the 14th ACM workshop on hot topics in networks*, page 24. ACM, 2015.

[61] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937, 2016.

[62] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[63] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518(7540):529–533, Feb. 2015.

[64] A. Y. Ng. *Shaping and policy search in reinforcement learning*. PhD thesis, University of California, Berkeley, 2003.

[65] I. Osband, C. Blundell, A. Pritzel, and B. Van Roy. Deep exploration via bootstrapped dqn. In *Advances in neural information processing systems*, pages 4026–4034, 2016.

[66] A. K. Parekh and R. G. Gallager. A generalized processor sharing approach to flow control in integrated services networks: the single-node case. *IEEE/ACM Transactions on Networking (ToN)*, 1(3):344–357, 1993.

[67] A. K. Parekh and R. G. Gallagher. A generalized processor sharing approach to flow control in integrated services networks: the multiple node case. *IEEE/ACM Transactions on Networking (ToN)*, 2(2):137–150, 1994.

[68] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer. Automatic differentiation in pytorch. 2017.

[69] B. Recht, C. Re, S. Wright, and F. Niu. Hogwild: A lock-free approach to parallelizing stochastic gradient descent. In *Advances in neural information processing systems*, pages 693–701, 2011.

[70] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. Learning representations by back-propagating errors. *nature*, 323(6088):533, 1986.

[71] T. Salimans, J. Ho, X. Chen, S. Sidor, and I. Sutskever. Evolution strategies as a scalable alternative to reinforcement learning. *arXiv preprint arXiv:1703.03864*, 2017.

[72] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Transactions on Computer Systems (TOCS)*, 2(4):277–288, 1984.

[73] H. Sariowan, R. L. Cruz, and G. C. Polyzos. Sced: A generalized scheduling policy for guaranteeing quality-of-service. *IEEE/ACM Transactions on networking*, 7(5):669–684, 1999.

[74] M. Schapira and K. Winstein. Congestion-control throwdown. In *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*, pages 122–128. ACM, 2017.

[75] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. Prioritized experience replay. *arXiv preprint arXiv:1511.05952*, 2015.

[76] L. E. Schrage and L. W. Miller. The queue m/g/1 with the shortest remaining processing time discipline. *Operations Research*, 14(4):670–684, 1966.

[77] J. Schulman. *Optimizing expectations: From deep reinforcement learning to stochastic computation graphs.* PhD thesis, UC Berkeley, 2016.

[78] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz. Trust region policy optimization. In *International Conference on Machine Learning*, pages 1889–1897, 2015.

[79] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel. High-dimensional continuous control using generalized advantage estimation. *arXiv preprint arXiv:1506.02438*, 2015.

[80] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[81] R. Sedgewick and K. Wayne. *Algorithms.* Addison-Wesley Professional, 4th edition, 2011.

[82] M. Shreedhar and G. Varghese. Efficient fair queueing using deficit round robin. In *ACM SIGCOMM Computer Communication Review*, volume 25, pages 231–242. ACM, 1995.

[83] M. Shreedhar and G. Varghese. Efficient fair queuing using deficit round-robin. *IEEE/ACM Transactions on networking*, 4(3):375–385, 1996.

[84] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller. Deterministic policy gradient algorithms. In *ICML*, 2014.

[85] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, et al. Mastering the game of go without human knowledge. *Nature*, 550(7676):354, 2017.

[86] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[87] S. P. Singh, T. Jaakkola, and M. I. Jordan. Reinforcement learning with soft state aggregation. In *Advances in neural information processing systems*, pages 361–368, 1995.

[88] A. Sivaraman, S. Subramanian, M. Alizadeh, S. Chole, S.-T. Chuang, A. Agrawal, H. Balakrishnan, T. Edsall, S. Katti, and N. McKeown. Programmable packet scheduling at line rate. In *Proceedings of the 2016 ACM SIGCOMM Conference*, pages 44–57. ACM, 2016.

[89] A. Sivaraman, K. Winstein, P. Thaker, and H. Balakrishnan. An experimental study of the learnability of congestion control. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 479–490. ACM, 2014.

[90] K. O. Stanley and R. Miikkulainen. Evolving neural networks through augmenting topologies. *Evolutionary computation*, 10(2):99–127, 2002.

[91] I. Stoica, S. Shenker, and H. Zhang. *Core-stateless fair queueing: Achieving approximately fair bandwidth allocations in high speed networks*, volume 28. ACM, 1998.

[92] I. Stoica, D. Song, R. A. Popa, D. Patterson, M. W. Mahoney, R. Katz, A. D. Joseph, M. Jordan, J. M. Hellerstein, J. E. Gonzalez, et al. A berkeley view of systems challenges for ai. *arXiv preprint arXiv:1712.05855*, 2017.

[93] R. S. Sutton and A. G. Barto. *Reinforcement Learning : An Introduction*. MIT Press, 1998.

[94] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*, pages 1057–1063, 2000.

[95] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, A. Rabinovich, et al. Going deeper with convolutions.

[96] A. Tanenbaum. *Computer Networks*. Prentice Hall Professional Technical Reference, 4th edition, 2002.

[97] J. N. Tsitsiklis and B. Van Roy. Analysis of temporal-diffference learning with function approximation. In *Advances in neural information processing systems*, pages 1075–1081, 1997.

[98] D. Van Aken, A. Pavlo, G. J. Gordon, and B. Zhang. Automatic database management system tuning through large-scale machine learning. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1009–1024. ACM, 2017.

[99] H. Van Hasselt, A. Guez, and D. Silver. Deep reinforcement learning with double q-learning. In *AAAI*, volume 16, pages 2094–2100, 2016.

[100] S. Venkataraman, Z. Yang, M. J. Franklin, B. Recht, and I. Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics.

[101] Z. Wang, T. Schaul, M. Hessel, H. Van Hasselt, M. Lanctot, and N. De Freitas. Dueling network architectures for deep reinforcement learning. *arXiv preprint arXiv:1511.06581*, 2015.

[102] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. In *Reinforcement Learning*, pages 5–32. Springer, 1992.

[103] C. Wilson, H. Ballani, T. Karagiannis, and A. Rowtron. Better never than late: Meeting deadlines in datacenter networks. *ACM SIGCOMM Computer Communication Review*, 41(4):50–61, 2011.

[104] K. Winstein and H. Balakrishnan. Tcp ex machina: computer-generated congestion control. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 123–134. ACM, 2013.

[105] N. J. Yadwadkar, B. Hariharan, J. E. Gonzalez, B. Smith, and R. H. Katz. Selecting the best vm across multiple public clouds: A data-driven performance modeling approach. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 452–465. ACM, 2017.

[106] M. D. Zeiler and R. Fergus. Visualizing and understanding convolutional networks. In *European conference on computer vision*, pages 818–833. Springer, 2014.

[107] L. Zhang. Virtual clock: A new traffic control algorithm for packet switching networks. In *ACM SIGCOMM Computer Communication Review*, volume 20, pages 19–29. ACM, 1990.

A Appendix

A.1 Installation and Configuration

Install relevant python3 packages.

```
pip install --upgrade pip
pip install --upgrade setuptools
pip3 install -r dependencies.txt
```

Configure and test the project.

```
# under {user-prefix}/src folder.
root_dir=$(pwd)
if [ "$(uname)" == "Darwin" ]; then
    echo "export ROOT_DIR="$root_dir>>~/.bash_profile
    source ~/.bash_profile
elif [ "$(expr substr $(uname -s) 1 5)" == "Linux" ]; then
    echo "export ROOT_DIR="$root_dir>>~/.bash_rc
    source ~/.bash_rc
fi
# compile cython
rm -rf **/*.{c,so}; rm -rf *.{c,so};
python3 setup.py build_ext --inplace
# run modules test
. ${ROOT_DIR}/caller/test.sh
```

A.2 Reproduction of the Results

A.2.1 Learning Scheduling Policies

Choose or create one's own configuration file in `${ROOT_DIR}/config/` and include it as the argument of the target launcher. E.g., train the agent towards a target scheduling via evaluative feedback with lrps_train.py configuration.

```
python3 launcher/exec_lrps.py -w remote run lrps_train -lp ./tmp/semi/train/
```

Evaluate the approximate models with an unseen workload set.

```
python3 launcher/eval_lrps_mp.py -s 12345 -n 100 -t semi
```

Let the agent to learn a blackbox scheduling behavior with lrps_train_se.py configuration.

```
python3 launcher/exec_lrps_se.py lrps_train_se -lp ./tmp/se/train/ -eid 0
```

Visualize the tensorflow computation graph with tensorboard:

```
tensorboard --logdir=${ROOT_DIR}/tmp/model/ --host=0.0.0.0
```

`${ROOT_DIR}/caller/` stores the scripts to automate series of the experiments for reproduction.

A.2.2 Exploring Custom Policies

Expose the agent to explore a customized policy for a specific objective with exps_explor.py configuration.

```
python3 launcher/exec_exps.py exps_explor -lp ./tmp/exps/train/ -eid 0
```