Spring 5-22-2021

Experience-driven Control For Networking And Computing

Zhiyuan Xu
Syracuse University, xuzhiyuan1588@gmail.com

ABSTRACT

Modern networking and computing systems have become very complicated and highly dynamic, which makes them hard to model, predict and control. In this thesis, we aim to study system control problems from a whole new perspective by leveraging emerging Deep Reinforcement Learning (DRL), to develop experience-driven model-free approaches, which enable a network or a device to learn the best way to control itself from its own experience (e.g., runtime statistics data) rather than from accurate mathematical models, just as a human learns a new skill (e.g., driving, swimming, etc). To demonstrate the feasibility and superiority of this *experience-driven control* design philosophy, we present the design, implementation, and evaluation of multiple DRL-based control frameworks on two fundamental networking problems, Traffic Engineering (TE) and Multi-Path TCP (MPTCP) congestion control, as well as one cutting-edge application, resource co-scheduling for Deep Neural Network (DNN) models on mobile and edge devices with heterogeneous hardware.

We first propose DRL-TE, a DRL-based framework that enables experience-driven networking for TE. DRL-TE maximizes a widely-used utility function by jointly learning network environment and its dynamics, and making decisions under the guidance of powerful DNNs. We propose two new techniques, TE-aware exploration and actor-critic-based prioritized experience replay, to optimize the general DRL framework particularly for TE. Furthermore, we propose an Actor-Critic-based Transfer learning framework for TE, ACT-TE, which solves a practical problem in experience-driven networking: when network configurations are changed, how to train a new DRL agent to effectively and quickly adapt to the new environment. In the new network environment, ACT-TE leverages policy distillation to rapidly learn a new control policy from both old knowledge (i.e., distilled from the existing agent) and new experience (i.e., newly collected samples).

In addition, we propose DRL-CC to enable experience-driven congestion control for

MPTCP. DRL-CC utilizes a single (instead of multiple independent) DRL agent to dynamically and jointly perform congestion control for all active MPTCP flows on an end host with the objective of maximizing the overall utility. The novelty of our design is to utilize a flexible recurrent neural network, LSTM, under a DRL framework for learning a representation for all active flows and dealing with their dynamics. Moreover, we integrate the above LSTM-based representation network into an actor-critic framework for continuous congestion control, which applies the deterministic policy gradient method to train actor, critic, and LSTM networks in an end-to-end manner.

With the emergence of more and more powerful chipsets and hardware and the rise of Artificial Intelligence of Things (AIoT), there is a growing trend for bringing DNN models to empower mobile and edge devices with intelligence such that they can support attractive AI applications on the edge in a real-time or near real-time manner. To leverage heterogeneous computational resources (such as CPU, GPU, DSP, etc) to effectively and efficiently support concurrent inference of multiple DNN models on a mobile or edge device, in the last part of this thesis, we propose a novel experience-driven control framework for resource co-scheduling, which we call COSREL. COSREL has the following desirable features: 1) it achieves significant speedup over commonly-used methods by efficiently utilizing all the computational resources on heterogeneous hardware; 2) it leverages DRL to make dynamic and wise online scheduling decisions based on system runtime state; 3) it is capable of making a good tradeoff among inference latency, throughput and energy efficiency; and 4) it makes no changes to given DNN models, thus preserves their accuracies.

To validate and evaluate the proposed frameworks, we conduct extensive experiments on packet-level simulation (for TE), testbed with modified Linux kernel (for MPTCP), and off-the-shelf Android devices (for resource co-scheduling). The results well justify the effectiveness of these frameworks, as well as their superiority over several baseline methods.

EXPERIENCE-DRIVEN CONTROL FOR NETWORKING AND COMPUTING

By

Zhiyuan Xu
B.S., University of Electronic Science and Technology of China, 2015

DISSERTATION

Submitted in partial fulfillment of the requirements for the degree of
Doctor of Philosophy in Computer and Information Science and Engineering

Syracuse University
May 2021

ACKNOWLEDGMENTS

The work presented in this thesis could not have been finished without the support of my committee members, friends, my family, and my wife. I would like to acknowledge those who have helped me throughout this effort.

First and foremost, I would like to express my deepest gratitude to my advisor, Dr. Jian Tang. He has not only taught me how to outstanding extensive research work but also how to effectively collaborate with colleagues. His passion for research and pursuit of excellence have set a great example to me.

There are many other faculty members from other universities who helped me a lot in my research work. Therefore, I would also like to thank the guidance of Dr. Guoliang Xue from Arizona State University, Dr. Dejun Yang from Colorado School of Mines, Dr. Weiyi Zhang from AT&T Labs Research, and Dr. Yanzhi Wang from Northeastern University. Without their inspiration and suggestion, I could not have completed this thesis.

I would like to thank all my labmates and many other friends. Many thanks for all their collaboration, encouragement, and company. They became a part of my life and made the long Ph.D. life in snowy Syracuse more enjoyable.

My final but everlasting gratitude goes to my family members. They were always trusting and supporting me in my choices. Most importantly, I would like to express my special and greatest appreciation to my wife, Ningxin Yue, for her love, care, support, and company in every critical moment of my life.

TABLE OF CONTENTS

CHAPTER 1
INTRODUCTION

1.1 Motivations

Many algorithms and protocols have been proposed to operate networking and computing systems and utilize their resources efficiently and effectively. Traditional methods for system control and resource allocation can be divided into two categories: *state-oblivious* and *optimization-based*. A state-oblivious method usually follows a pre-defined (fixed) policy for control and resource allocation. Typical examples include shortest-path routing that uses the hop-count as the routing metric, load-balancing routing that always splits traffic load evenly over all the candidate paths, and round-robin scheduling that assigns tasks in equal portions and in circular order. An optimization-based method usually consists of two steps: 1) formulating a resource allocation problem into a mathematical programming problem based on certain mathematical models; and 2) designing an algorithm to solve it according to its mathematical properties (such as convex programming). Typical examples include those well-known Network Utility Maximization (NUM) algorithms [76, 93]. We argue that neither of these two approaches will work well for modern or future systems (such as Software Defined Network (SDN) and edge systems), which have become or are expected to be very complicated and highly-dynamic. A state-oblivious method usually

leads to a simple algorithm or protocol, which, however, may suffer from very poor performance (such as throughput and delay) in a highly time-variant system due to its lack of careful consideration for runtime states and suboptimal solutions. An optimization-based method, however, needs to have an accurate prediction for future values of some key parameters (such as user demands, link usages, etc) as input; and accurate mathematical models to estimate/characterize network behavior (after applying a given resource allocation solution). Both of them are very challenging, especially in complex systems.

Take a fundamental networking problem, Traffic Engineering (TE), as an example. For a given a set of network flows with source and destination nodes, the goal of TE is to find a solution to effectively forward the data traffic with the objective of maximizing a utility function. Simple and widely-used state-oblivious solutions include always routing traffic via shortest paths (e.g., Open Shortest Path First (OSPF) [90]); or evenly distributing traffic via multiple available paths (e.g., Valiant Load Balancing (VLB) [111]). Obviously, neither of them are optimal. Better solutions could be developed if there exist accurate and mathematically solvable models for network environment, user demands and their dynamics. Queueing theory has been employed to model communication networks and assist resource allocation [69, 92, 93, 135]. However, it may not work well for those networking problems involving multi-hop routing and end-to-end performance (such as delay) due to the following reasons: 1) In the queueing theory, many problems in a queueing network (rather than a single queue) remain open problems, while a communication network with a mesh-like topology represents a fairly complicated multi-point to multi-point queueing network where data packets from a queue may be distributed to multiple downstream queues, and a queue may receive packets from multiple different upstream queues. 2) The queueing theory can only provide accurate estimations for queueing delay under a few strong assumptions (e.g, tuple arrivals follow a Poisson distribution, etc), which, however, may not hold in a complex communication network. Note that even if the packet arrival at every source node follows a Poisson distribution, packet arrivals at intermediate nodes may not.

In addition, NUM [76] has been widely-applied to provide a resource allocation solution for TE by formulating and solving an optimization problem. However, these methods may suffer from the following issues: 1) They usually assume that some key factors (such as user demands, link usages, etc) are given as input, which, however, are hard to estimate or predict. 2) It is hard to directly minimize end-to-end delay by explicitly including it in the utility function since given decision variables for resource allocation in TE, it is hard to express the corresponding end-to-end delay in a closed form with them since an accurate mathematical model is needed to achieve this (while queueing theory may not work here as described above). 3) Network dynamics have not been well addressed by these works. Most of them claimed to provide a "good" resource allocation solution, which is optimal or close-to-optimal but only for a snapshot of the network. However, most networks are highly time-varying. How resource allocation should be adjusted or re-computed to accommodate such dynamics has not been well addressed by these NUM methods.

Hence, we aim to study networking and computing problems from a whole new perspective by leveraging emerging Artificial Intelligence (AI) techniques, especially deep learning, to develop an experience-driven approach, which enables a network or a device to learn the best way to control itself from its own experience (e.g, runtime statistics data), just as a human learns a skill (such as swimming and driving). Unlike state-oblivious or optimization-based methods, the experience-driven approach does not rely on any mathematical model but is expected to make wise decisions on online control with full consideration for real-time runtime states.

Recent breakthrough of *Deep Reinforcement Learning (DRL)* [81] provides a promising technique for enabling effective experience-driven model-free control. *DRL* (originally developed by DeepMind) enables computers to learn to play games, including Atari 2600 video games [81] and one of the most complicated games, Go (AlphaGo [116]), and beat the best human players. Even though DRL has made tremendous successes on game-playing that usually has a limited action space (e.g., moving up/down/left/right), it has not

yet been investigated how DRL can be leveraged for system control and resource allocation problems in complex communication networks or computing systems, which usually have sophisticated states and huge or continuous action spaces. We believe DRL is especially promising for control in networking and computing systems because: 1) It has advantages over other dynamic system control techniques such as model-based predictive control in that the former is model-free and does not rely on accurate and mathematically solvable system models (such as queueing models), thereby enhancing its applicability in complex networks with random and unpredictable behaviors. 2) It is able to deal with highly dynamic time-variant environments such as time-varying system states and user demands. 3) It is capable of handling a sophisticated state space (such as AlphaGo [116]), which is more advantageous over traditional Reinforcement Learning (RL) [120].

To demonstrate the feasibility and superiority of this *experience-driven control* design philosophy, in this thesis, we focus on two fundamental networking problems, Traffic Engineering (TE) and Multi-Path TCP (MPTCP) congestion control, as well as one cutting-edge application, resource co-scheduling for Deep Neural Network (DNN) models on mobile and edge devices with heterogeneous hardware. Nevertheless, designing such a DRL-based control framework for these problems is not straightforward but quite challenging.

Direct application of the basic DRL technique, such as Deep Q-Network (DQN) based DRL [81], to networking problems, like TE or congestion control, does not work since they are continuous control problems; while DQN-based DRL is only capable of handling control problems with a limited action space. Although DRL methods have been proposed for continuous control very recently [33, 71], we show a state-of-the-art method, Deep Deterministic Policy Gradient (DDPG) [71], does not work well. Moreover, it is quite common that configurations of a communication network are changed over time. Thus, when applying DRL to realizing experience-driven control, we need to carefully address another important problem: when network configurations are changed, how to train a new DRL agent for a new network environment? Even though DRL works well for a given

network and path sets, it may lead to serious performance degradation if we keep using it whenever the topology and/or path sets are changed. While transfer learning seems a good solution to the re-configuration problem, there are still challenges that need to be addressed when applying it to experience-driven networking: 1) In DRL, instead of learning from a given dataset with targeting labels (values), like supervised learning, the DRL agent learns the best control policy through interacting with the environment in a trial-and-error manner. However, most of the previous transfer learning [7] techniques are proposed to solve the problem in the context of supervised learning, which cannot be directly applied to DRL; 2) Although some recent work [106, 17] proposed policy distillation that aims to solve the transfer learning problem in DRL, their methods are based on DQN and cannot be directly applied to the tasks with continuous action space, which, however, is required by the majority of the networking problems.

In addition to handling continuous action space, we have to address several other issues when designing an experience-driven congestion control framework for MPTCP. First, a straightforward congestion control solution is to use a DRL agent to perform congestion control for each MPTCP flow independently. However, this solution may not work well since it lacks necessary and effective cooperation among these agents, while concurrent flows may interfere with each other due to their competition for common resources, which, however, cannot be well addressed by independent agents. Moreover, it is quite challenging to use DRL to dynamically handle multiple flows that may come and go at any time since a DRL agent usually uses a deep feed-forward neural network or a deep convolutional neural network as the function approximator for action inference, which has a fixed input size.

To enable experience-driven control for mobile and edge computing, which aims to leverage heterogeneous computational resources (such as CPU, GPU, DSP, etc.) to effectively and efficiently support the concurrent inference of multiple DNN models on a device, we have to consider a practical problem during the DRL training. The DRL agent cannot be well trained only on a cloud server in an offline manner, since data (i.e., transition

samples) for training the agent need to be collected continuously via interactions with the mobile/edge device. On the other hand, it is also difficult to train a DRL agent only on a mobile/edge device. As we all know, complex mathematical operations (e.g., calculating gradients and backpropagation) are needed to be performed during the training process, which, however, are not supported by most mobile deep learning frameworks such as TensorFlow Lite [126] and Pytorch Mobile [99]. Moreover, training a DRL agent only on a mobile/edge device may take a long time to converge due to its very limited computational resources.

1.2 State of The Art and Literature Gap

1.2.1 Network Control and Resource Allocation

Traffic Engineering and Network Utility Maximization

TE and NUM have been well studied in the literature. In a seminal work [76], Low and Lapsley proposed asynchronous distributed algorithms to solve a flow control problem whose objective is to maximize the aggregate source utility over their transmission rates. In [93], Palomar and Chiang, introduced primal, dual, indirect, partial, and hierarchical decompositions, focusing on NUM problems and the meanings of primal and dual decompositions in terms of network architectures. In [92], the authors designed a congestion control system that scales gracefully with multiple objectives, which was built on decentralized control laws at end-systems. Xu *et al.* [135] proposed a new link-state routing protocol PEFT, which splits traffic over multiple paths with an exponential penalty on longer paths, with hop-by-hop forwarding, with the objective of achieving optimal TE. The authors of [69] proposed algorithms to solve a NUM problem in a network with delay sensitive/insensitive traffic, which is modelled by adding explicit delay terms to the utility function measuring QoS. Einhorn *et al.* [22] proposed a RL-based decentralized approach for QoS routing and

TE in MPLS networks. Recently, TE has been studied in the context of SDN. For example, Jain *et al.* [59] presented design and implementation of Google's SDN-based WAN, B4, and proposed a TE algorithm based on a bandwidth function for data transmissions among its data centers. The authors of [4] proposed approximation algorithms for TE problems with with partial deployment of SDN. NUM, TE and/or related problems have also been studied by quite a few works [8, 103, 127, 147] in the context of wireless networks, which were mainly focused on wireless-specific issues such as interference, time-varying link states, etc.

Congestion Control for Multi-Path TCP

Congestion control, as a fundamental problem in networking, has been widely studied in the context of TCP [13, 20, 52, 134, 141]. Unlike these related works targeting at the regular TCP, the congestion control in MPTCP is quite different. It has also been shown [96, 101] that MPTCP may suffer from serious performance degradation when directly applying a regular TCP congestion control algorithm separately on each sub-flow. In [53], Honda *et al.*proposed a congestion control scheme, which enables an end-to-end connection that uses flows along multiple paths to fairly compete with TCP flows at shared bottlenecks, and in the meanwhile, maximizes the utilization of different paths. Hassayoun *et al.* [47] proposed Dynamic Window Coupling (DWC), a multipath congestion control mechanism that seeks to be fair to other flows in the network while being able to maximize its own throughput. DWC detects shifting bottlenecks in the network and responds by dynamically regrouping subflows. In [101], Raiciu *et al.*designed Linked Increase Algorithm (LIA), which couples the congestion control policies running on different subflows by linking their increase functions. The authors of [64] presented Opportunistic Linked Increase Algorithm (OLIA), which resolves some performance issues of LIA while retaining non-flappiness and responsiveness. As an extension of the well-known TCP Vegas [13], the authors of [136] proposed weighted Vegas for MPTCP, which adopts the packet queu-

ing delay as a congestion signal, achieving fine-grained load balancing. In [96], Peng *et al.*proposed BAlanced LInked Adaptation (BALIA) to generalize existing congestion control algorithms through a fluid model and strike a good performance. In [24], the authors quantified the penalty of the coupled congestion control for links that do not share a bottleneck, then designed and implemented a practical shared bottleneck detection (SBD) algorithm for MPTCP, namely MPTCP-SBD, to overcome the penalty. A recent work [145] presented MPTCPD, an energy-efficient variant of MPTCP particularly for datacenters, which can provide energy efficiency by minimizing the flow completion time. Morevoer, Le *et al.* [68] developed ecMTCP, which is an energy-efficient congestion control algorithm. Dong *et al.* [21] designed mVeno particularly for wireless communications with multiple radio interfaces. Raiciu *et al.* [102] implemented MPTCP in Linux kernel and evaluated its performance. They mainly focused on the algorithms needed to efficiently use paths with different characteristics, notably send and receive buffer tuning and segment reordering. They also compared the performance of their implementation with regular TCP on web servers.

We summarize the differences between our work and these related works in the following: 1) Unlike [47, 53, 69, 92, 93, 101, 135] guided by queueing models or pre-defined policies, we develop experience-driven model-free approaches based on DRL, which learns the best control policy based on real-time runtime states. 2) Related works [4, 59, 76] have not explicitly addressed end-to-end delay, which, however, is one of the major concerns in our work. 3) We consider a TE problem in general networks, which is mathematically different from those problems in specific networks/scenarios [4, 8, 22, 59, 103, 127, 147]. 4) We are the first to leverage the emerging DRL for TE and MPTCP, which has been shown to be very effective.

1.2.2 Scheduling for Mobile and Edge Computing

Recently, there has been a growing interest on runtime optimization for DNNs on mobile and edge devices. Lane *et al.* [65] proposed DeepX, a software accelerator for DL model executions on mobile devices. DeepX first performs a runtime layer compression on a given DNN model to control the memory computation and energy consumption, then it decomposes the DNN model into unit-blocks of various types and schedules them on heterogeneous hardware (e.g., CPU and GPU) for efficient on-device inference. Yao *et al.* [139] presented a unified approach called DeepIoT to compress DNN models for sensing applications. DeepIoT compresses neural network structures into smaller dense matrices by finding the minimum number of non-redundant hidden elements, such as filters and dimensions required by each layer, while keeping the performance of sensing applications the same. Fang *et al.* [23] proposed NestDNN, which prunes a DNN model into a set of descendent models, each of which offers a unique resource-accuracy trade-off. At runtime, it dynamically selects a DNN model with the best resource-accuracy tradeoff to fit available resources in the system. Liu *et al.* [74] proposed a usage-driven selection framework, called AdaDeep, to automatically select a combination of compression techniques for a given DNN model, which leads to an optimal balance between user-specified performance goals (e.g., latency and energy) and resource constraints.

Another line of research has addressed the problem of scheduling DNN models among mobile/wearable devices, edge computing nodes and cloud servers. Han *et al.* [42] evaluated a variety of model optimization techniques to balance the resource usages in terms of memory, computation and accuracy. Then they proposed a framework called MCDNN to automatically optimize DNN models while conforming to the resource specification and assign DNN models to run either on a cloud or on a mobile device. Kang *et al.* [61] presented Neurosurgeon, a system that automatically partitions DNN models at the granularity of layers and assigns them to run on a mobile device or cloud for the best latency and energy consumption tradeoff. Zhao *et al.* [146] proposed ECRT, an edge computing system

for real-time object tracking on resource-constrained devices. By intelligently partitioning DNN models into two parts, which are executed locally on an IoT device or on an edge server, ECRT minimizes the power consumption of IoT devices while meeting the user requirement on end-to-end delay. Xu *et al.* [137] proposed DeepWear, which focuses on applying DNN models on wearable devices. DeepWear offloads DNN inference tasks from a wearable device to its paired hand-held device through local network connectivity (e.g., Bluetooth). Zeng *et al.* [143] proposed Boomerang, an on-demand cooperative DNN inference framework for edge systems in an Industrial Internet of Things (IIoT) environment. Boomerang first reshapes the amount of DNN computation via an early-exit mechanism to reduce the total runtime of DNN inference, and then it segments the DNN model between IIoT devices and an edge server to achieve the DNN inference immediacy.

In addition, Georgieve *et al.* [31] presented LEO, a scheduler designed to maximize the performance of multiple continuous mobile sensing applications by making use of the domain-specific signal processing knowledge to distribute sensor processing tasks to heterogeneous computational resources (e.g., CPU, GPU, DSP and cloud). Zhou *et al.* [148] proposed S^3DNN, a system that supports DNN-based real-time object detection workloads on GPUs in a multi-tasking environment, while simultaneously improving real-time performance, throughput, and GPU resource utilization. Yang *et al.* [138] proposed a framework to improve the number of simultaneous camera streams for object detection on embedded devices without significantly increasing per-frame latency or reducing per-stream accuracy by applying a combination of techniques, including parallelism, pipelining, and the merging of per-camera images.

Unlike those methods proposed in [23, 65, 74, 139], which applied model compression techniques to reduce the model size and speedup model inference, we represent a complementary and transparent solution, which does not change DNN models thus preserves their accuracies. Unlike [42, 61, 137, 143, 146], which addressed the problem of offloading DNN models (or part of models) to the cloud, we, however, focus on the on-device infer-

ence without the help of cloud servers, and consider a mathematically different problem of scheduling tasks on heterogeneous hardware. In addition, we study DNN inference on mobile and edge devices in general here, while some related works [31, 138, 146, 148] targeted at specific applications or models with application-specific goals and requirements.

1.2.3 Learning Methods

Deep Reinforcement Learning

DRL has won his world-wide fame due to its impressive successes on game-playing tasks such as Go and Atari games. It has recently attracted extensive research attention from both industry and academia. In a pioneering work [81], Mnih *et al.*proposed deep Q-learning and DQN, which can learn successful policies directly from high dimensional sensory inputs. Their work bridges the gap between high-dimensional sensory inputs and actions, resulting in the first artificial agent that is capable of learning to deal with a diverse array of challenging gaming tasks. As introduced above, they introduced two new techniques, experience replay and target network, to ensure learning stability. The authors of [48] proposed Double Q-learning as a specific adaptation of the DQN and an improvement to the earlier work [81]. Another improvement was introduced in [108] to use prioritized experience replay in DQN such that important transition samples can be replayed more frequently, which can lead to more efficient learning. In [129], Wang *et al.*presented a new dueling neural network architecture, which includes two separate estimators: one for the state value function and one for the state-dependent action advantage function. Fortunato *et al.* [28] presented NoisyNet, a DRL agent with parametric noise added to its weights. They showed that the induced stochasticity of the agent's policy can be used to aid efficient exploration and illustrated that replacing the conventional exploration heuristics for A3C [82], DQN [81] and Dueling [129] agents with NoisyNet yields substantially higher scores for a wide range of Atari games. Hessel *et al.* [46] empirically studied multiple extensions to the basic DQN algorithm and proposed Rainbow as a combination method, which represents the state-of-

the-art Q-learning method for Atari games. So far, we only discuss works related to discrete control with a limited action space. Continuous control has also be addressed in the context of DRL. Lillicrap *et al.* [71] adapted the ideas underlying the success of DQN to the continuous action domain, they proposed an actor-critic model-free algorithm based on the deterministic policy gradient named DDPG. Gu *et al.* [33] proposed normalized advantage functions for reducing sample complexity for continuous control. In [34], the authors proposed an interesting policy gradient method Q-Prop, which uses a Taylor expansion of the off-policy critic as a control variant. The authors of [82] proposed asynchronous gradient descent for optimizing learning with DNNs, and showed its successes on a wide variety of continuous motor control tasks. Haarnoja *et al.* [39] proposed soft actor-critic, an off-policy actor-critic DRL algorithm based on the maximum entropy RL framework. By combining off-policy updates with a stable stochastic actor-critic formulation, their method achieves state-of-the-art performance on a range of continuous control benchmark tasks.

Transfer Learning

Transfer learning can be defined as the ability of a system to apply knowledge and skills learned from previous tasks to new tasks [94], in other words, extracting the knowledge from one or more source tasks and applying the knowledge to a target task. Traditional transfer learning can be roughly categorized as three cases according to *what to transfer*. The first is instance transfer approach [14, 18, 60], which reuses the data from the source domain in the target domain. The second is feature-representation transfer approach [7, 11, 19], where the knowledge used to transfer across domains is encoded into the learned feature representation. The third case is parameter transfer approach [12, 67, 109], i.e., target tasks and source tasks share some parameters of models. Rusu *et al.* [106] first proposed policy distillation, which extended the idea of knowledge distillation into DRL. Czarnecki *et al.* [17] further analyzed the policy distillation under different circumstances.

Even though, DRL has made tremendous successes, the research on the feasibility and

effectiveness of using it in the context of quite different networking and computing problems is still in its infancy. To the best of our knowledge, we are the first to leverage DRL for experience-driven control in networking and computing systems. Moreover, none of these aforementioned transfer learning methods can be directly applied to the general network re-configuration problem in the context of experience-driven control, which is one of the target of this thesis.

1.2.4 Learning-based Control for Networking and Computing

Learning-based control methods, especially Machine Learning (ML) and Deep Learning (DL), have recently been widely applied for networking and computing problems. To improve the performance of congestion control, instead of manually formulating each endpoint's reaction to congestion signals, Winstein *et al.* [134] proposed to generate the congestion control algorithms named Remy Congestion Control (RemyCC), and achieved better performance compared with related congestion control algorithms and protocols. Different with Remy, Dong [20] designed an learning-based congestion control method called performance-oriented congestion control (PCC), which could continuously adjust the data sending rate while observing the network performance. Li *et al.* [70] proposed an ML-based two-stage intrusion detection framework in Software Defined Internet of Things (SD-IOT) networks. They improved the bat algorithm for feature selection and enhanced the random forest algorithm for flow classification. Wang *et al.* [130] proposed a hybrid DL model for spatiotemporal modeling and prediction in cellular networks, based on big system data. Shen *et al.* [112] presented a networking-slicing based architecture on RAN and elaborated how ML/DL can potentially empower this architecture in the aspect of RAN slicing, automated RAT selection/user association, and content placement and delivery. Bega *et al.* [10] presented a DNN architecture, DeepCog, as an DL-based cost-aware capacity forecast to optimize resource provisioning in network slicing. Zappone *et al.* [142] proposed to use DL-based approaches to assistant traditional mathematical-oriented models and ap-

proaches in future wireless communication networks. As a popular ML/DL technique, RL/DRL has attracted much attention due to its effectiveness and model-free property. Tesauro *et al.* [123] showed the feasibility of using online RL to learn resource valuation estimates (in lookup table form) that can be used to make high-quality server allocation decisions in multi-application prototype data center scenario. Vengerov *et al.* [128] combined RL with fuzzy rulebases to perform adaptive reconfiguration of a distributed system based on maximizing the long-term business value, solving the problem of dynamic resource allocation among multiple entities sharing a common set of resources. Shaio *et al.* [110] proposed a RL-based congestion control scheme in a high-speed network, which consists of a long-term policy evaluator and a short-term rate selector, to jointly determine the congestion threshold and sending rate. As a prior work to apply DRL for resource scheduling, Mao *et al.* [78] proposed DeepRM that enables a cluster system to gradually learn the optimal resource scheduling policy from previous experiences. To realize automatic video streaming adaption over HTTP, Mao *et al.* [79] proposed Pensieve that learns an Adaptive BitRate (ABR) policy by REINFORCE [132] DRL algorithm. Pensieve outperforms the best state-of-the-art ABR algorithms in terms of average quality of experience.

Unlike [10, 20, 70, 112, 134, 142], which only apply ML/DL to predict some key parameters, like flow classification and traffic amount, but still rely on mathematical-oriented models to make control decisions. Our method, which, however, controls the networking or computing systems in an model-free end-to-end manner (i.e., observe the system state and directly derive the action decision). Unlike [110, 123, 128], which applied the traditional RL to solve resource allocation and scheduling problems, we focus on leveraging the emerging DRL method with DNNs as function approximators to deal with high-dimensional complex system states. Moreover, different from the above mentioned work, in this thesis, we do not only attempt to apply learning methods (e.g., DRL) to solve a particular networking or computing problem but also to discuss the *experience-driven control* design philosophy.

1.3 Contributions

In this thesis, we aim to leverage emerging DRL to develop experience-driven control framework for networking and computing problems. More specifically, we make the following contributions.

- We propose a highly effective and practical DRL-based experience-driven control framework, DRL-TE, for TE in a communication network, to jointly learn network dynamics and making decisions under the guidance of powerful DNNs. We discuss and show that direct application of a state-of-the-art DRL solution for continuous control does not work well for the TE problem. Based on the analysis, we propose two new techniques, TE-aware exploration and actor-critic-based prioritized experience replay to optimize the general DRL framework particularly for TE.

- We propose a novel Actor-Critic-based Transfer learning framework, ACT-TE, for the re-configuration problem in TE. One of the critical insights of ACT-TE is that the critic network of an existing DRL agent could provide knowledge for a new DRL agent in a new environment since they belong to the same task domain. Instead of only randomly exploring solutions at the early stage of training like most of the existing works [10, 11], we used the critic network of an existing well-trained DRL agent to guide the training of the new agent. Furthermore, we applied the prioritized replay buffer [12] to filter out more knowledgeable transitions that help further improve the exploration efficiency.

- We propose the design, implementation, and evaluation of a DRL-based congestion control framework, DRL-CC, which realizes our experience-driven design philosophy on MPTCP congestion control. We, for the first time, integrate the a LSTM-based representation learning network into an actor-critic framework for continuous congestion control, and leverages the emerging deterministic policy gradient method [32] to train critic, actor and LSTM networks in an end-to-end manner. In addition,

a unique and desirable feature of our design is that we use a DRL agent to control all (instead of a single) active MPTCP flows.

- We conduct a preliminary empirical study for inference with a simple DNN model on an off-the-shelf smartphone, and make several interesting findings, which can serve as a guidance for the design of an efficient co-scheduling algorithm. We present the design and implementation of a novel experience-driven control framework, COSREL, for resource co-scheduling on mobile and edge devices. Moreover, We propose a novel device-server co-training algorithm, which makes a device and a server work collaboratively and efficiently to train the DRL agent of COSREL.

1.4 Outline

The rest of this thesis is organized as follows. We give a brief background introduction about DRL and two state-of-the-art algorithms in Chapter 2. We first present the design, implementation, and evaluation of the DRL-based experience-driven control framework for the TE problem in Chapter 3. Furthermore, we consider the re-configuration problem for experience-driven control in TE and present an actor-critic-based transfer learning framework in Chapter 4. In Chapter 5, we present our DRL-based congestion control framework for MPTCP. In Chapter 6, we conduct an empirical study for DNN scheduling over heterogeneous hardware and introduce our DRL-based resource co-scheduling framework. We conclude this thesis in Chapter 7.

CHAPTER 2
BACKGROUND ON DEEP REINFORCEMENT LEARNING

In this chapter, we give a brief introduction to Deep Reinforcement Learning (DRL). In a typical DRL framework, as shown in Fig. 2.1, an DRL agent interacts with an environment (e.g., a networking or computing system) in discrete decision epochs. At each epoch t, the agent makes an observation of the state $\mathbf{s}_t$ of the environment, takes an action $\mathbf{a}_t$ according to its policy (e.g., a deep neural network). After executing the action in the environment, the agent receives a reward t_t. The goal of the agent is to find a control policy π to map its state to a deterministic action or to a probability distribution over actions $\pi : \mathbf{s}_t \rightarrow \mathbf{a}_t$, such that the discounted cumulative reward $R_0 = \sum_{t=0}^{T} \gamma^t r(\mathbf{s}_t, \mathbf{a}_t)$ can be maximized, where $r(\cdot)$ is the reward function and $\gamma \in [0, 1]$ is a factor to determine the relative importance of the current reward compared with future rewards. If $\gamma = 0$, the DRL agent greedily considers to maximize its current reward at each epoch. In contrast, if $\gamma = 1$, the DRL agent focuses more on long-term higher cumulative reward.

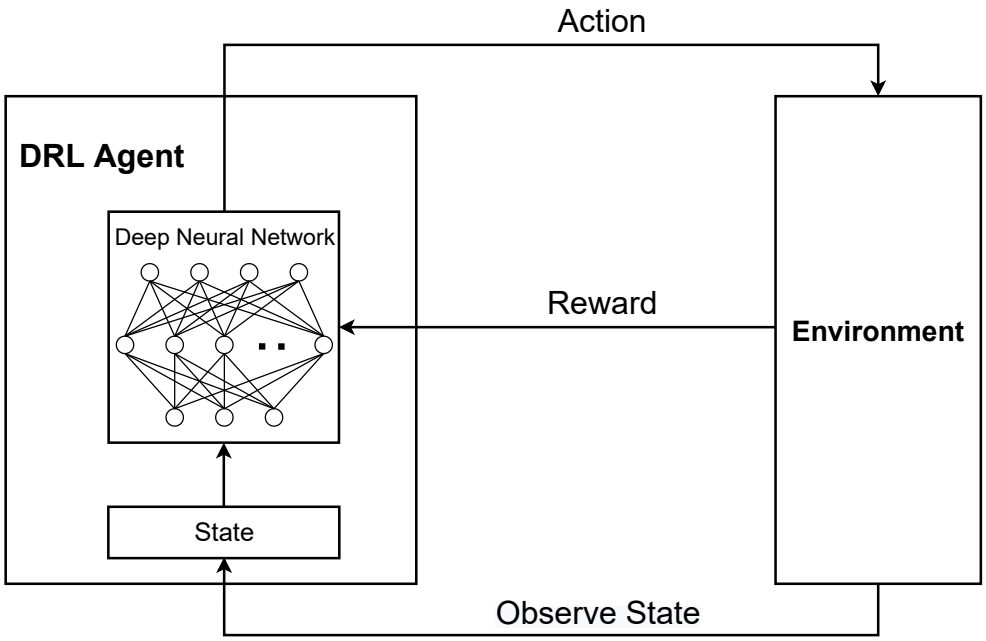


Fig. 2.1: The overview of deep reinforcement learning

2.1 Deep Q-Learning

The deep version of RL was introduced in a well-known work [81] by Mnih *et al.* from DeepMind, which extends the traditional Q-learning to bridge the gap between high-dimensional sensory inputs (e.g. raw images) and actions. A unique feature of the DRL agent in [81] is to use a Deep Neural Network (DNN) called DQN as the function approximator. A DQN takes a state-action pair $(\mathbf{s}_t, \mathbf{a}_t)$ as input and outputs the corresponding Q value $Q(\mathbf{s}_t, \mathbf{a}_t)$, which is the expected discounted cumulative reward:

$$Q(\mathbf{s}_t, \mathbf{a}_t) = \mathbb{E}\Big[R_t | \mathbf{s}_t, \mathbf{a}_t\Big], \tag{2.1}$$

where $R_t = \sum_{k=t}^{T} \gamma^k r(\mathbf{s}_t, \mathbf{a}_t)$. The action can be derived by applying a commonly-used greedy policy:

$$\pi(\mathbf{s}_t) = \underset{\mathbf{a}_t}{\operatorname{argmax}}\, Q(\mathbf{s}_t, \mathbf{a}_t). \tag{2.2}$$

According to Q-learning, the training target value for each state-action pair can be derived using the Bellman equation:

$$y_t = r(\mathbf{s}_t, \mathbf{a}_t) + \gamma Q(\mathbf{s}_{t+1}, \pi(\mathbf{s}_{t+1})|\boldsymbol{\theta}^Q), \tag{2.3}$$

$$y_t = r_t + \gamma \text{max}_{\mathbf{a}'} Q(\mathbf{s}_{t+1}, \mathbf{a}'|\boldsymbol{\theta}^Q) \tag{2.4}$$

where $\boldsymbol{\theta}^Q$ is the parameters of the DQN. Based on the target value, the DQN can be trained by minimizing the following loss:

$$L(\boldsymbol{\theta}^Q) = \mathbb{E}\Big[y_t - Q(\mathbf{s}_t, \mathbf{a}_t|\boldsymbol{\theta}^Q)\Big]. \tag{2.5}$$

$$L(\boldsymbol{\theta}^Q) = \mathbb{E}\Big[(y_t - Q(\mathbf{s}_t, \mathbf{a}_t|\boldsymbol{\theta}^Q))^2\Big]. \tag{2.6}$$

Even though neural network or DNN has been used as the function approximator for RL before, it is known that such a non-linear function approximator is not stable and may even lead to divergence. To improve the stability of learning, Mnih *et al.* [81] introduced two effective techniques: experience replay and target network. With experience relay, a DRL agent collects and stores state transition samples into a relay buffer, and then updates the DNN using a mini-batch sampled from the replay buffer instead of the immediately collected transition sample (used in traditional Q-learning). By doing so, the DRL agent could break correlations in the observation sequence, and learn from a more independently and identically distributed past experience, which is required by most of the training algorithms, such as Stochastic Gradient Descent (SGD). They proposed to use a separate target network to estimate target values $< y_t >$, which shares the same network structure as the original DQN. But its parameters are slowly updated every $C > 1$ epochs and are held fixed in between. These two techniques can smooth out the learning processing and avoid oscillations or divergence.

2.2 Deep Deterministic Policy Gradient

As mentioned above, DQN-based DRL is restricted to discrete control with a limited action space and there is no trivial extension to continuous control, which, however, is quite common in computer and communication networks (e.g., traffic engineering and congestion control). A commonly-used approach to continuous control is policy gradient [119]. In a recent work [71], Lillicrap *et al.*introduced an actor-critic approach called Deep Deterministic Policy Gradient (DDPG) for DRL, which leverages both DNNs and the emerging deterministic policy gradient [115] for continuous control. The key idea behind DDPG is to simultaneously maintain two functions: one is the parameterized actor function $\pi(\mathbf{s}_t|\boldsymbol{\theta}^\pi)$ used for deriving actions; and another is the parameterized critic function $Q(\mathbf{s}_t, \mathbf{a}_t|\boldsymbol{\theta}^Q)$ used for evaluating actions. The critic function is implemented using the DQN mentioned above, which takes a given state-action pair as input and outputs the corresponding Q-value. It can be trained as a regular DQN, which has been introduced above. The actor function can be implemented by another DNN, which takes a state as input and outputs the best action (that could be continuous). As shown in [71], to update the actor network, the chain rule can be applied to the the expected cumulative reward J with respect to the actor parameters $\boldsymbol{\theta}^\pi$:

$$\begin{aligned} \nabla_{\boldsymbol{\theta}^\pi} J &\approx \mathbb{E}\Big[\nabla_{\boldsymbol{\theta}^\pi} Q(\mathbf{s}, \mathbf{a}|\boldsymbol{\theta}^Q)|_{\mathbf{s}=\mathbf{s}_t, \mathbf{a}=\pi(\mathbf{s}_t|\boldsymbol{\theta}^\pi)}\Big] \\ &= \mathbb{E}\Big[\nabla_{\mathbf{a}} Q(\mathbf{s}, \mathbf{a}|\boldsymbol{\theta}^Q)|_{\mathbf{s}=\mathbf{s}_t, \mathbf{a}=\pi(\mathbf{s}_t)} \cdot \nabla_{\boldsymbol{\theta}^\pi} \pi(\mathbf{s}|\boldsymbol{\theta}^\pi)|_{\mathbf{s}=\mathbf{s}_t}\Big]. \end{aligned} \tag{2.7}$$

Note that both experience replay and target network can also be used together with DDPG to ensure the learning stability.

CHAPTER 3
EXPERIENCE-DRIVEN NETWORKING

3.1 Overview

To demonstrate the design philosophy of experience-driven networking, we first consider a classical fundamental networking problem, Traffic Engineering (TE), i.e., for given a set of network flows with source and destination nodes, to find a solution to effectively forward the data traffic with the objective of maximizing a utility function. In this chapter, we present a novel and highly effective Deep Reinforcement Learning (DRL)-based model-free control framework for TE in a communication network to jointly learn network dynamics and make decisions under the guidance of powerful Deep Neural Networks (DNNs). We summarize our contributions of this chapter in the following:

- We are the first to present a highly effective and practical DRL-based experience-driven control framework, DRL-TE, for TE.
- We discuss and show that direct application of a state-of-the-art DRL solution for continuous control, namely Deep Deterministic Policy Gradient (DDPG) [71], does not work well for the TE problem.
- We propose two new techniques, TE-aware exploration and actor-critic-based pri-

oritized experience replay to optimize the general DRL framework particularly for TE.

- We show via extensive packet-level simulation using ns-3 [87] with both representative and random network topologies that DRL-TE significantly outperforms several widely-used baseline methods.

To the best of our knowledge, we are the first to leverage the emerging DRL for enabling model-free control in communication networks. We aim to promote a simple and practical experience-driven approach based on DRL, which, we believe, can be easily extended to solve many other resource allocation problems in networking and computing systems.

3.2 Problem Statement

We describe the TE problem in this section. First, we summarize the major notations (used in this chapter) below for quick reference.

Table 3.1: Notation Definition

Variable	Definition
K	The number of communication sessions
$\mathbf{P}_k$	The set of candidate paths of session k
$\mathbf{E}$	The set of links of the network
B_k	Traffic demand of session k
C_e	Capacity of link e
$f_{k,j}$	The amount of traffic of the jth path of session k
$w_{k,j}$	Split ratio for the jth path of session k
x_k, z_k	Throughput and delay of session k
$\mathbf{s}$, $\mathbf{a}$, r	State, action and reward
$p_i, P(i)$	Priority and probability (being selected) of transition sample i
$\boldsymbol{\theta}^\pi, \boldsymbol{\theta}^Q$	Weights of actor and critic networks $\pi(\cdot)$ and $Q(\cdot)$

We consider a general communication network with K end-to-end communication sessions. We use a directed graph $\mathbf{G}(\mathbf{V}, \mathbf{E})$ to model the network, where each vertex corresponds to a node (router or switch) and each edge corresponds to a directed communication link connecting a pair of nodes. Each communication session k has a source node s_k, destination d_k and a set of candidate paths $\mathbf{P}_k$ (connecting s_k with d_k) that can carry its traffic load. As mentioned above, we aim to study a TE problem seeking a rate allocation solution, which specifies the amount of traffic load $f_{k,j}$ going through the jth path of $\mathbf{P}_k$. Note that once we have such a solution, then when a packet of session k arrives at s_k, path j is chosen to transmit the packet with a probability of $w_{k,j}$, where $w_{k,j} = f_{k,j}/(\sum_{j=1}^{|P_k|} f_{k,j})$, which is known as the split ratio.

The α-fairness [117, 134] model has been widely used for NUM. According to this model, the utility of a communication session with a steady-state throughput of x is $U_\alpha(x) = (\frac{x^{1-\alpha}}{1-\alpha})$. Particularly, as $\alpha \to 1$, in the limit $U_1(x)$ becomes $\log x$ [134]. For $\alpha > 0$, $U_\alpha(x)$ is monotonically increasing with x. The objective of the TE problem is usually set to maximizing the total utility of all the communication sessions, i.e., $\sum_{k=1}^{K} U_\alpha(x)$. α can be used to tradeoff fairness and efficiency. If $\alpha = 1$, the objective is to achieve the proportional fairness, which is widely used for resource allocation.

In order to address both throughput and delay, similar as in [134], we define a utility function $U(\cdot)$ for session k:

$$U(x_k, z_k) = U_{\alpha_1}(x_k) - \sigma \cdot U_{\alpha_2}(z_k), \tag{3.1}$$

where x_k and z_k are the end-to-end throughput and delay of session k respectively; and σ expresses the relative importance of delay vs. throughput. Similarly, the objective of the TE problem is to maximize the total utility of all the communication sessions in the network, i.e., $\sum_{k=1}^{K} U(x_k, z_k)$.

Note that we aim to consider a general communication network and show how DRL can

enable experience-driven networking rather than targeting at a specific physical network (such as SDN, multi-hop wireless network) or a specific scenario (such as WAN, MAN, LAN, etc). So we try to make system model and problem statement as general as possible. However, the proposed control framework (Section 3.3) is so flexible that it can be easily extended for a specific network or scenario with additional constraints.

3.3 DRL-based Control Framework

In this section, we present the proposed DRL-based control framework, DRL-TE, for the TE problem described above.

In order to utilize the DRL techniques (no matter which method/model to use), we first need to design the state space, action space and reward function.

STATE: The state consists of two components: throughput and delay of each communication session. Formally, the state vector $\mathbf{s} = [(x_1, z_1), \cdots, (x_k, z_k), \cdots, (x_K, z_K)]$.

ACTION: An action is defined as the solution to the TE problem, i.e., the set of split ratios for the communication sessions. Formally, the action vector $\mathbf{a} = [w_{1,1}, \cdots, w_{kj}, \cdots, w_{K,|P_k|}]$, where $\sum_{j=1}^{|P_k|} w_{k,j} = 1$.

REWARD: The reward is the objective of the TE problem, which is the total utility of all the communication sessions. Formally, $r = \sum_{k=1}^{K} U(x_k, z_k)$.

Note that the design of state space, action space and reward is critical to the success of a DRL method. Our design well captures network states and the key components of the TE problem without including useless/redudant information. The core of the proposed control framework is an agent, which runs a DRL algorithm (Algorithm 1) to find the best action at each decision epoch, takes the action to the network (e.g., through a network controller) observes the network state, and collects a transition sample.

The TE problem is obviously a continuous control problem. As explained above, the DQN-based DRL proposed in the well-known work [81] does not work here; so we choose

the state-of-the-art DRL-based solution for continuous control, DDPG [71], as the starting point for our design, whose basic idea has been introduced in Chapter 2.

Even though DDPG has been demonstrated to work well on quite a few continuous control tasks [71], our experimental results, however, show that direct application of DDPG to the TE problem does not lead to satisfying performance (Section 3.4). We suspect this is due to the following two reasons: 1) The DDPG framework in [71] does not clearly specify how to explore. A simple random noise based method or the exploration methods proposed for physical control problems (mentioned in [71]) do not work well for the TE problem here. 2) DDPG utilizes a simple uniform sampling method for experience replay, which ignores the significance of transition samples in the replay buffer. To address these two issues, we propose two new techniques to optimize DDPG particularly for TE, including TE-aware exploration which leverages a good TE solution as the baseline during exploration; and actor-critic-based prioritized experience replay which can employs a new method for specifying significance of samples with careful consideration for both the actor and critic networks. Exploration is an essential and important process for training a DRL agent because an inexperienced agent needs to see sufficient transition samples to gain experience and eventually learn a good (hopefully optimal) policy. For continuous control problems, exploration is quite challenging because there are infinite number of actions that can be chosen in each decision epoch and the commonly-used ϵ-greedy method [81] only works for tasks with a limited discrete action space, which obviously does not work here. DDPG generates an action for exploration by adding a random noise to the action returned by the current actor network.

3.3.1 TE-aware Exploration

For exploration, we propose a new randomized algorithm that guides the exploration process with a base TE solution. Specifically, with ϵ probability, the DRL agent derives action as $\mathbf{a}_{\text{base}} + \epsilon \cdot \mathcal{N}$; and with $(1 - \epsilon)$ probability, it derives action as $\mathbf{a} + \epsilon \cdot \mathcal{N}$; where $\mathbf{a}_{\text{base}}$

is a base TE solution, **a** is the output of actor network $\pi(\cdot)$ and ϵ is an adjustable parameter. ϵ can tradeoff exploration and exploitation by determining the probability of adding a random noise to the action rather than taking the derived action from the actor network. ϵ decays with decision epoch t, which means with more learning, more derived (rather than random) actions will be taken. The parameter $\mathcal{N}$ is a uniformly distributed random noise.

The proposed control framework is not restricted to any specific base TE solution for $\mathbf{a}_{\text{base}}$, which can be obtained in many different ways. For example, a simple solution is to use the shortest path to deliver all the packets for each communication session, which is not optimal in most cases but is good enough to sever as a baseline for exploration. Another solution is to evenly distribute traffic load of each communication session to all candidate paths. NUM-based methods can also be used to find base solutions. For example, we can obtain a TE solution by solving the following mathematical programming:
NUM-TE:

$$\max_{<x_k, f_{k,j}>} \sum_k U_\alpha(x_k) \tag{3.2a}$$

subject to:

$$\sum_{k=1}^{K} \sum_{\mathbf{p}_j \in \mathbf{P}_k : e \in \mathbf{p}} f_{k,j} \leq C_e, \forall\, e \in \mathbf{E}; \tag{3.2b}$$

$$x_k \leq B_k, k \in \{1, \cdots, K\}; \tag{3.2c}$$

$$\sum_{j=1}^{|\mathbf{P}_k|} f_{k,j} = x_k, k \in \{1, \cdots, K\}. \tag{3.2d}$$

In this formulation, the objective is to maximize the total utility in terms of throughput. Note that it is hard to include the end-to-end delay term in the utility function since there does not exists a mathematical model that can accurately establish a connection between end-to-end delay and the other decision variables $< x_k, f_{k,j} >$. This is why end-to-end delay has not been well addressed by most existing works on NUM. Constraints (3.2b)

ensure the aggregated traffic load on each link does not exceed its capacity C_e, where $\mathbf{p}_j$ is the jth path in $\mathbf{P}_k$. Constraints (3.2c) ensure the total throughput of each session k does not exceed its demand B_k, which can be estimated. Constraints (3.2d) establish the connections between two set of decision variables $< x_k >$ and $< f_{k,j} >$. If $\alpha = 1$, $U_\alpha(x_k) = \log x_k$, then this problem becomes a convex programming problem, which can be efficiently solved by the Gurobi Optimizer [37] that were used in our implementation.

3.3.2 Actor-critic-based Prioritized Experience Replay

DDPG simply uniformly samples transition data from the experience replay. It has been shown by [108] that an DRL agent can learn more effectively from some transitions than others. A method called prioritized experience replay has also been introduced in [108], which has been shown to lead to better performance on game-playing tasks when being combined with DQN. It assigns a priority for each transition sample. Based on this priority, transition data in the replay buffer are sampled in each epoch. However, this method was proposed only for DQN-based DRL and has never been used with the actor-critic method for continuous control. We extend this method to enable prioritized experience replay under the actor-critic framework. Specifically, since an actor-critic method uses two networks (actor and critic) to guide decision making, the priority should consist of two parts. The first part is the Temporal-Difference (TD) error, which corresponds to training of the critic network:

$$\delta = y - Q(\mathbf{s}, \mathbf{a}), \tag{3.3}$$

where y is the target value for training the critic network, which is defined in Equation (2.4). Note that to help understand the basic idea better, we omit the subscripts/superscripts here for clean presentation; the exact forms of these equations can be found at the formal algorithm presentation. The actor and critic network are jointly trained by transition samples in the replay buffer. The second part is related to training of the actor network, i.e., the

Q gradient $\nabla_{\mathbf{a}} Q = \nabla_{\mathbf{a}} Q(\mathbf{s}, \mathbf{a})|_{\mathbf{s}=\mathbf{s}_i, \mathbf{a}=\pi(\mathbf{s}_i)}$ (Equation (2.7)). Combining them together, the priority of a transition sample is given as:

$$p = \varphi \cdot (|\delta| + \xi) + (1 - \varphi) \cdot \overline{|\nabla_{\mathbf{a}} Q|}, \tag{3.4}$$

where φ is a parameter controlling the relative importance of TD error vs. Q gradient. $\overline{|\nabla_{\mathbf{a}} Q|}$ is the average of absolute values of the Q gradient (which is a vector). A small positive constant ξ is used to prevent the edge-cases of transitions not being revisited once their error is zero. The probability of sampling transition i is:

$$P(i) = \frac{p_i^{\beta_0}}{\sum_j^{|\mathbf{B}|} p_j^{\beta_0}}, \tag{3.5}$$

where the exponent β_0 determines how much prioritization is used; if $\beta_0 = 0$, then it becomes uniform sampling.

3.3.3 DRL-TE Framework

We formally present the proposed DRL-based control framework for TE, DRL-TE, as Algorithm 1. First the algorithm randomly initializes all the weights $\boldsymbol{\theta}^{\pi}$ of actor network $\pi(\cdot)$; and $\boldsymbol{\theta}^{Q}$ of the critic networks $Q(\cdot)$(line 1). As mentioned above, we employ target networks $\pi'(\cdot)$ and $Q'(\cdot)$ to improve learning stability. The target networks are clones of the original actor or critic networks, whose weights $\boldsymbol{\theta}^{\pi'}$ and $\boldsymbol{\theta}^{Q'}$ are initialized in the same way as their original networks (line 2) but are slowly following updated (line 23). The update rate is controlled by a parameter τ. In each decision epoch, the algorithm applies the TE-aware exploration method to obtain an action first (line 6), which is explained above.

We use a prioritized replay buffer for storing transition samples. We first store the sample into the replay buffer with maximal priority (line 8), and then sample a mini-batch of transition samples from **B** (lines 10-19) to train the actor and critic networks. The priority of transition is then updated using the method described right above (lines 16-18). Note that

Algorithm 1: DRL-TE

1: Randomly initialize critic network $Q(\cdot)$ and actor network $\pi(\cdot)$ with weights $\boldsymbol{\theta^Q}$ and $\boldsymbol{\theta^\pi}$ respectively;
2: Initialize target networks $Q'(\cdot)$ and $\pi'(\cdot)$ with weights $\boldsymbol{\theta^{Q'}} := \boldsymbol{\theta^Q}$, $\boldsymbol{\theta^{\pi'}} := \boldsymbol{\theta^\pi}$;
3: Initialize prioritized replay buffer $\mathbf{B}$ and $p_1 := 1$;
/**Online Learning**/
4: Receive the initial observed state $\mathbf{s}_1$;
/**Decision Epoch**/
5: **for** t = 1 **to** T **do**
6: Apply the TE-aware exploration method to obtain $\mathbf{a}_t$;
7: Execute action $\mathbf{a}_t$ and observe the reward r_t;
8: Store transition sample $(\mathbf{s}_t, \mathbf{a}_t, r_t, \mathbf{s}_{t+1})$ into $\mathbf{B}$ with maximal priority $p_t = \max_{j<t} p_j$;
9: /**Prioritized Transition Sampling**/
10: **for** i = 1 **to** N **do**
11: Sample a transition $(\mathbf{s}_i, \mathbf{a}_i, r_i, \mathbf{s}_{i+1})$ from $\mathbf{B}$ where $i \sim P(i) := p_i^{\beta_0} / \sum_j p_j^{\beta_0}$;
12: Compute important-sampling weight: $\omega_i := (|\mathbf{B}| \cdot P(i))^{-\beta_1} / \max_j \omega_j$;
13: Compute target value for critic network: $Q(\cdot)\ y_i := r_i + \gamma \cdot Q'(\mathbf{s}_{i+1}, \pi'(\mathbf{s}_{i+1}))$;
14: Compute TD-error: $\delta_i := y_i - Q(\mathbf{s}_i, \mathbf{a}_i)$;
15: Compute gradient: $\nabla_{\boldsymbol{\theta^\pi}} J_i := \nabla_{\mathbf{a}} Q(\mathbf{s}, \mathbf{a})|_{\mathbf{s}=\mathbf{s}_i, \mathbf{a}=\pi(\mathbf{s}_i)} \cdot \nabla_{\boldsymbol{\theta^\pi}} \pi(\mathbf{s})|_{\mathbf{s}=\mathbf{s}_i}$;
16: Update the transition priority: $p_i := \varphi \cdot (|\delta_i| + \xi) + (1 - \varphi) \cdot \overline{|\nabla_{\mathbf{a}} Q|}$;
17: Accumulate weight-change for critic network: $Q(\cdot)$ $\Delta_{\boldsymbol{\theta^Q}} := \Delta_{\boldsymbol{\theta^Q}} + \omega_i \cdot \delta_i \cdot \nabla_{\boldsymbol{\theta^Q}} Q(\mathbf{s}_i, \mathbf{a}_i)$;
18: Accumulate weight-change for actor network: $\pi(\cdot)\ \Delta_{\boldsymbol{\theta^\pi}} := \Delta_{\boldsymbol{\theta^\pi}} + \omega_i \cdot \nabla_{\boldsymbol{\theta^\pi}} J_i$;
19: **end for**
20: /**Network Updating**/
21: Update the weights of critic network: $Q(\cdot)\ \boldsymbol{\theta^Q} := \boldsymbol{\theta^Q} + \eta^Q \cdot \Delta_{\boldsymbol{\theta^Q}}$, reset $\Delta_{\boldsymbol{\theta^Q}} := 0$;
22: Update the weights of actor network: $\pi(\cdot)\ \boldsymbol{\theta^\pi} := \boldsymbol{\theta^\pi} + \eta^\pi \cdot \Delta_{\boldsymbol{\theta^\pi}}$, reset $\Delta_{\boldsymbol{\theta^\pi}} := 0$;
23: Update the weights of the corresponding target networks: $\boldsymbol{\theta^{Q'}} := \tau \boldsymbol{\theta^Q} + (1 - \tau) \boldsymbol{\theta^{Q'}}$; $\boldsymbol{\theta^{\pi'}} := \tau \boldsymbol{\theta^\pi} + (1 - \tau) \boldsymbol{\theta^{\pi'}}$;
24: **end for**

for every transition sample $(\mathbf{s}_i, \mathbf{a}_i, r_i, \mathbf{s}_{i+1})$ in the mini-batch, we first obtain its important-sampling weight ω (line 12), which is used to correct the bias introduced by prioritized replay [108]. The weight is integrated into the critic network updating in the form of $\omega \cdot \delta$ (rather than δ only) (line 17). Priorities ensure high-error transitions are seen more frequently. Those large steps (with large priority) can be very disruptive because of large updating values. As suggested by [108], we import annealing weight β_1 to correct this bias,

by linearly annealing it from its initial value to 1 (line 12). For learning stability, we always normalize ω by $1/\max_j \omega_j$, so they only scale the weight update downward. We obtain the action for the next state from target actor network $\pi'(\mathbf{s}_{i+1})$, and the target value y_i (line 13) for training the critic network; In addition, we compute the policy gradient by the chain rule, as described in Equation (2.7) (line 15). The weight-changes are accumulated (lines 17-18) and used to update the actor and critic networks (lines 21-22).

There are quite a few hyper-parameters in the proposed control framework. To maximize its performance, we conducted a comprehensive empirical study to find the best settings for them and the best structures of the actor and critical networks. In our design and implementation, we used a 2-layer fully-connected feedforward neural network to serve as the actor network, which includes 64 and 32 neurons in the first and second layer respectively and utilized the Leaky Rectifier [32] for activation. In the final output layer, we, however, employed the softmax [32] as activation function to ensure the sum of output values equals one. For the critic network, we also used a 2-layer fully-connected feedforward neural network, with 64 and 32 neurons in the first and second layer respectively and with the Leaky Rectifier for activation. In order to sample N transitions with probabilities given by Equation (3.5), the range [0, p_{total}] is divided into N sub-ranges, and a transition is uniformly sampled from each sub-range, where p_{total} is the sum of priorities of all transitions in replay buffer. As suggested by [108], we used a sum-tree to implement the priority probability, which is similar to a binary heap. The differences are 1) leaf nodes store the priorities of transitions; and 2) internal nodes store the sum of its children. In this way, the value of root is p_{total}, and the time complexity for updating and sampling is $O(\log N_{\text{tree}})$, where N_{tree} is the number of nodes in the sum-tree. During the empirical study, we also found good settings for the other important hyper-parameters: $\xi := 0.01$, $\beta_0 := 0.6$, $\beta_1 := 0.4$, $\gamma := 0.99$, $\varphi := 0.6$, $\eta^{\pi} := 0.001$, $\eta^{Q} := 0.01$, $\tau := 0.01$ and $N = 64$.

3.4 Performance Evaluation

We conducted extensive simulation to evaluate the performance of the proposed DRL-based framework. We present and analyze the simulation results in this section. We implemented the proposed framework and set up the environment in ns-3 [87] for packet-level simulation. The DNNs included in the framework (i.e., the actor and critic networks) were implemented using Tensorflow [1]. Due to the light wight of our design, we found that we could easily run and train the proposed framework (along with the corresponding DNNs) on a regular desktop with an Intel Quad-Core 2.6Ghz CPU with 8GB memory.

The simulation runs were performed on two well-known network topologies, NSF Network (NSFNET [88]) and Advanced Research Projects Agency Network (ARPANET [3]). Besides, we randomly generated a network topology with 20 nodes and 80 links, using the widely-used network topology generator, BRITE [80]. For each network topology, we assigned $K = 20$ communication sessions, each with randomly selected source and destination nodes. For each communication session, we selected 3-shortest paths (in terms of hop-count) as its candidate paths. The capacity of each link was set to 100Mbps. The packet arrival at the source node of each communication session (i.e., traffic demand) follows a Poisson process (note that the packet arrivals at intermediate nodes may not follow a Poisson process), with its mean value uniformly distributed within a window with a size of 20Mbps. In our experiments, we set the window to $[0, 20]$Mbps initially, and we increased the traffic demand by sliding the window with a step size of 5Mbps for each run. We set $\alpha := 1$ and $\sigma := 1$ for the utility function to balance throughput, delay and fairness, i.e. the objective/utility function became $\sum_{k=1}^{K}(\log x_k - \log z_k)$.

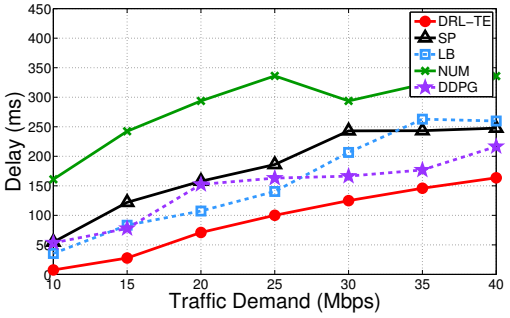
We compared our DRL-based control framework with three widely used baseline solutions as well as DDPG [71]:

- Shortest Path (SP): every communication session uses a shortest path to deliver all its packets.
- Load Balance (LB): every communication session evenly distributes its traffic load

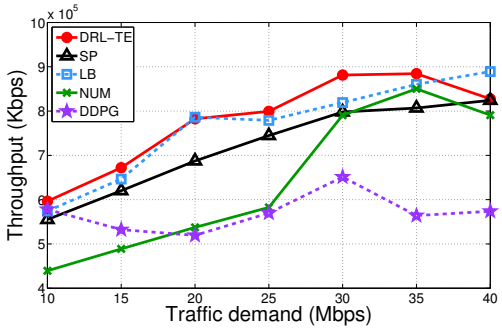
to all candidate paths.

- Network Utility Maximization (NUM): it obtains TE solutions by solving the convex programming problem, NUM-TE given in Section 3.3.1.
- DDPG: For fair comparison, we replaced the DRL-TE algorithm (Algorithm 1) with the DDPG algorithm [71], while keeping the other settings (such as state, action, reward and the DNNs) the same.

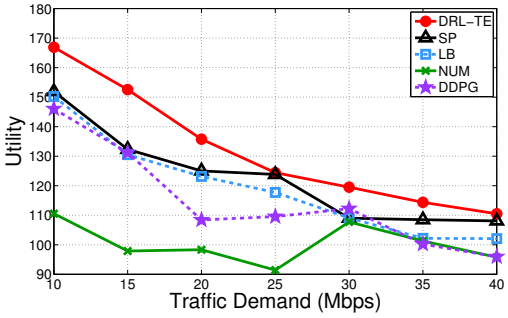
We used the total end-to-end throughput, the end-to-end average packet delay, and the network (i.e., total) utility value as the performance metrics for comparisons. We present the corresponding simulation results in Figs. 3.1-3.3, each of which corresponds to a network topology. Note that the numbers on the x-axis are the central values of the corresponding traffic demand windows (mentioned above). In addition, we show the performance of two DRL methods (DDPG and DRL-TE) over the three network topologies during the



(a) End-to-end delay

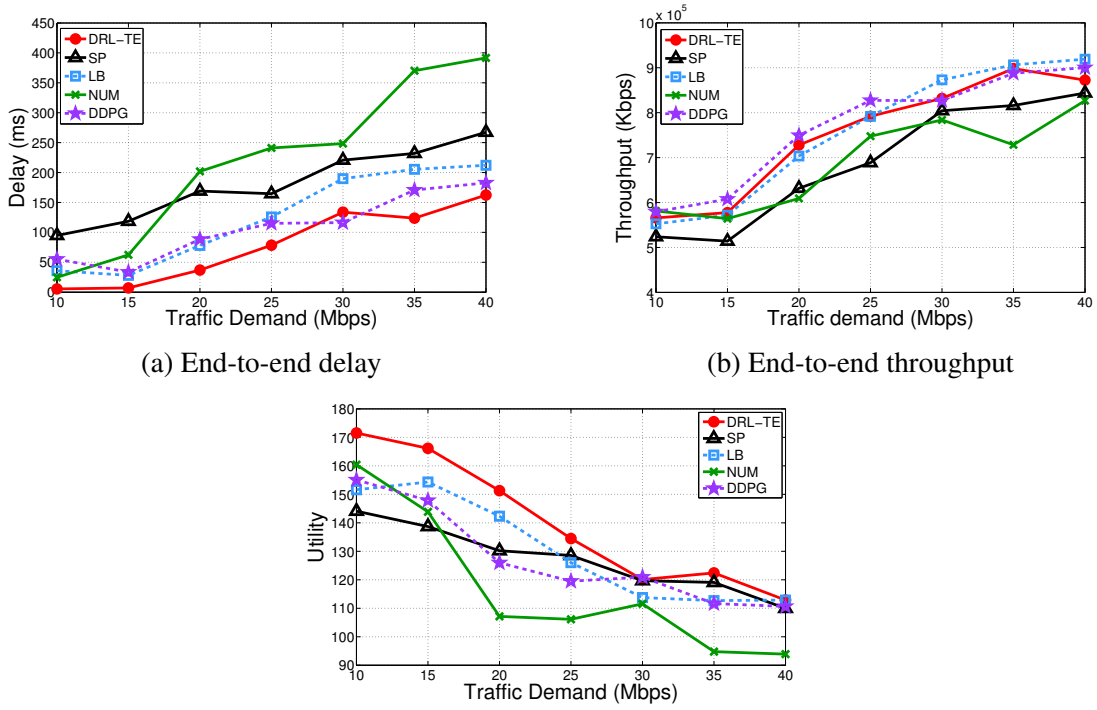


(b) End-to-end throughput



(c) Network utility

Fig. 3.1: Performance of all the methods over the NSFNET topology



(a) End-to-end delay

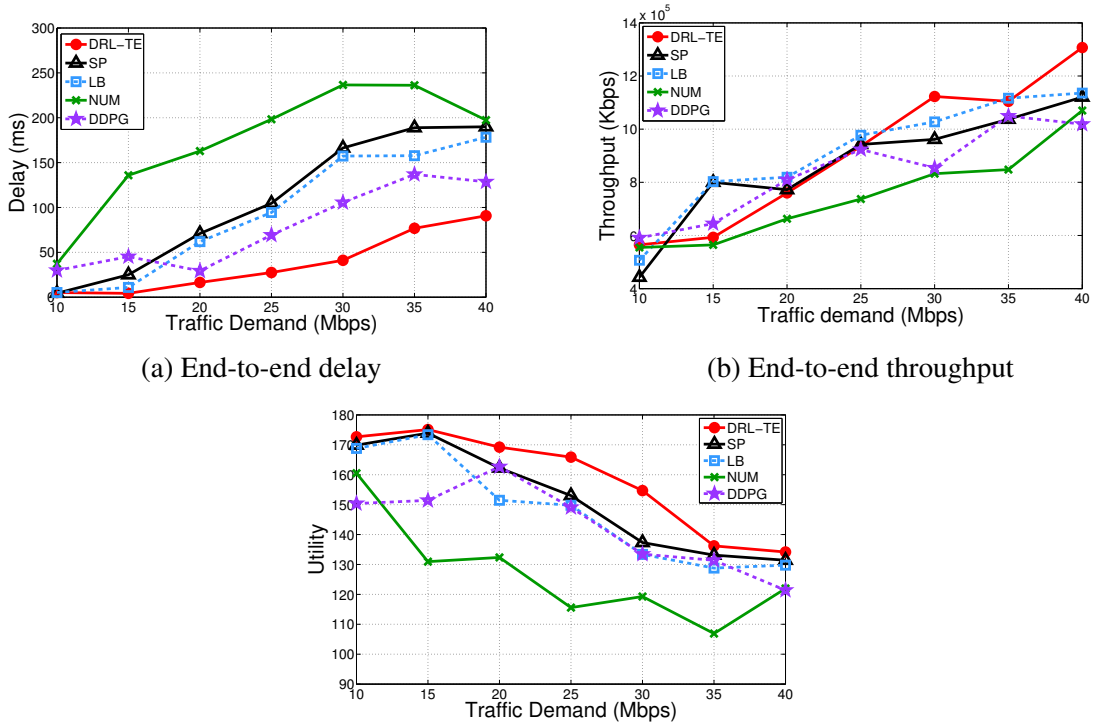
(b) End-to-end throughput

(c) network utility

Fig. 3.2: Performance of all the methods over the ARPANET topology

online learning procedure in terms of the reward. For illustration and comparison purposes, we normalized and smoothed the reward values using a commonly-used method $(r - r_{\min})/(r_{\max} - r_{\min})$ (where r is the actual reward, $r_{\min}$ and $r_{\max}$ are the minimum and maximum rewards during online learning respectively) and the well-known forward-backward filtering algorithm [38] respectively. We present the corresponding simulation results in Fig. 3.4. Note that for these results, the corresponding traffic demand was generated using window $[10, 30]$Mbps. We can make the following observations from these results.

1) From Figs. 3.1a, 3.2a and 3.3a, we can see that compared to all the four baseline methods, DRL-TE significantly reduces end-to-end delay on all the three topologies. For example, on the NSF topology, when the traffic load is medium (i.e., traffic demand window is $[10, 30]$Mbps), DRL-TE significantly reduces the end-to-end delay by 51.6%, 28.6%, 74.6% and 50.0% respectively, compared to SP, LB, NUM and DDPG. Overall, DRL-TE



(a) End-to-end delay

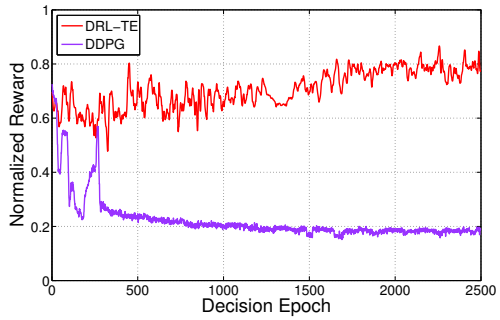
(b) End-to-end throughput

(c) Network utility

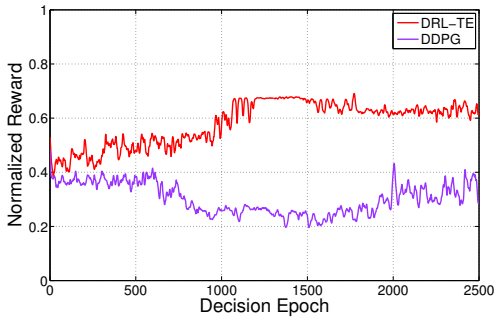
Fig. 3.3: Performance of all the methods over the random topology

achieves an average reduction of 55.4%, 47.1%, 70.5% and 44.2% respectively. Compared to throughput, end-to-end delay is harder to deal with since as discussed above, it lacks accurate mathematical models that can well capture its characteristics and runtime dynamics. It is not surprising to see NUM leads to fairly poor performance since it fails to explicitly address end-to-end delay and its design is based on the assumption that network state is fairly stable or slowly changes, which may not be true; while simple solutions such as SP and LB offers expected performance since intuitively, the shortest paths and load balancing (which can avoid congestions) can help reduce delay. DRL-TE unarguably delivers superior performance with regards to end-to-end delay because it keeps learning runtime dynamics and making wise decisions to move to the optimal with the help of DNNs.

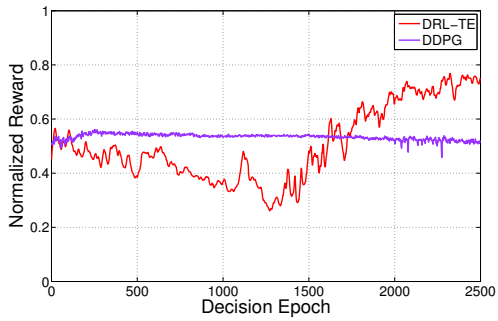
2) Even though the objective (reward function) of DRL-TE is not to simply maximize end-to-end throughput, it still delivers satisfying performance, as shown in Figs. 3.1b, 3.2b and 3.3b. Compared to all the other methods, DRL-TE leads to consistently higher



(a) NSFNET topology



(b) ARPANET topology



(c) Random topology

Fig. 3.4: Reward over the three network topologies during online learning

throughput on the NSFNST topology. On both the ARPANET and random topologies, the throughput values given by DRL-TE are comparable to those given by LB (load balancing is supposed to yield high throughout), but still higher than those offered by SP and NUM.

3) As expected, we can see from Figs. 3.1c, 3.2c and 3.3c that DRL-TE outperforms all the other methods in terms of the total utility because its reward function is set to maximizing it. On average, DRL-TE outperforms SP, LB, NUM and DDPG by 7.7%, 9.1%, 26.4% and 12.6% respectively.

4) From Figs. 3.1-3.3, we can observe no matter which method is used and no matter which network topology is chosen, the throughput and delay basically go up with the traffic demand; while the total utility generally go down. This is easy to understand because the higher the traffic load, usually the higher the throughput, but the higher the delay due to longer waiting time or even congestion, which brings down the total utility. Moreover, the throughput does not increase monotonically, when the network becomes saturated, higher

traffic demands may even lead to poorer throughput due to congestion and packet losses. We also notice that DRL-TE is robust to changes of traffic load and network topology since it performs consistently better than all the other methods across all the traffic demand settings and all the topologies.

5) In addition, we can also observe from Figs. 3.1-3.3 that DDPG does not work very well on these topologies. For example, compared to SP and LB, it performs generally worse in terms of the total utility, even though it provides slightly better end-to-end delay. To further explain why DRL-TE works better than DDPG, we also show how the reward value changes during online learning over the three network topologies in Fig. 3.4. Clearly, over all these network topologies, DRL-TE quickly (within just a couple of thousands of decision epoches) reaches a good solution (that gives a high reward); while DDPG seems to be stuck at local optimal solutions with lower reward values. Particularly, on the random topology, we can only see minor improvement on the first few hundred epoches, then it fails to find better solutions (actions) to improve the reward. These results clearly justify the effectiveness of the proposed new techniques including TE-aware exploration and the actor-critic-based prioritized experience replay.

3.5 Summary

In this chapter, we proposed to use a novel experience-driven approach for resource allocation in communication networks, which can learn to well control a communication network from its experience rather than an accurate mathematical model. Specifically, we presented a novel and highly effective DRL-based control framework, DRL-TE, to solve the TE problem. The proposed framework enables experience-driven control by jointly learning network dynamics, and make decisions under the guidance of two DNNs, actor and critic networks. Moreover, we proposed two new techniques, TE-aware exploration and actor-critic-based prioritized experience replay, to optimize the general DRL frame-

work particularly for TE. We implemented DRL-TE in ns-3, and conducted a comprehensive simulation study to evaluate its performance on two well-known network topologies, NSFNET and APRANET, and a random topology. Extensive simulation results have shown that 1) compared to several widely-used baseline methods, DRL-TE significantly reduces end-to-end delay and consistently improves the total utility, while offering better or comparable throughput; 2) DRL-TE is robust to network changes; and 3) DRL-TE consistently outperforms DDPG, which, however, does not offer satisfying performance.

CHAPTER 4
TRANSFER LEARNING FOR EXPERIENCE-DRIVEN NETWORKING

4.1 Overview

Deep Reinforcement Learning (DRL) has been shown to be a useful technique for enabling experience-driven networking due to its capabilities of supporting model-free control, dealing with highly dynamic time-variant environments, and handling a sophisticated state space, as demonstrated by the last chapter. However, another important problem for experience-driven networking should be carefully addressed: when network configurations are changed, how to train a new DRL agent for a new network environment. It is quite common that configurations of a communication network are changed over time. For example, if a load or delay sensitive routing metric [95] is used, then the routing path between a pair of source and destination nodes may be changed over time. Moreover, the topology of a network may be changed over time due to node/link failures or manual re-configurations conducted by a network administrator.

Intuitively, there are three straightforward solutions: 1) keeping using the existing (likely a well-trained) agent; 2) training a new agent from scratch for the new environ-

ment; and 3) fine-tuning the existing agent for the new environment. The first solution may not work well due to lack of knowledge or necessary training for the new environment. Some DRL agents may be able to survive minor changes (e.g. traffic load changes at certain nodes) but could not deal with significant re-configurations, such as changes of routing paths or even the whole topology as mentioned above, which has been shown by our experimental results presented later. The second solution may suffer from a long training time and unpredictable performance, which are common issues of DRL-based solutions. The third solution seems reasonable but is actually ineffective because it ignores the knowledge captured by the existing agent when training the new agent. As we may realize, when a human learns a new skill (e.g., snowboarding), his/her old knowledge and experience (especially those related to similar skills, e.g., ski) may play a key role. Thus, we believe both old knowledge and new experience are important for learning to adapt to a new environment. Hence, both the existing agent and newly collected samples should be effectively leveraged for training the new agent.

Even though the proposed DRL-TE works well for Traffic Engineering (TE) for a given network and path sets, it may lead to serious performance degradation if we keep using it whenever the topology and/or path sets are changed. As mentioned above, because they lack necessary training or ignore the knowledge captured by the existing agent for the new environment, the straightforward solutions may not work well for the TE problem in the context of experience-driven networking. In this chapter, we present an Actor-Critic-based Transfer learning framework for the TE problem using policy distillation, which we call ACT-TE. One of the critical insights of ACT-TE is that the critic network of an existing DRL agent could provide knowledge for a new DRL agent in a new environment since they belong to the same task domain. Instead of only randomly exploring solutions at the early stage of training like most of the existing works [10, 11], we used the critic network of an existing well-trained DRL agent to guide the training of the new agent. Furthermore, we applied the prioritized replay buffer [12] to filter out more knowledgeable transitions

that help further improve the exploration efficiency.

Specifically, the main contributions of this chapter are summarized in the following:

- We propose a novel Actor-Critic-based Transfer learning framework, ACT-TE, for the TE problem using policy distillation. The basic idea and design of ACT-TE can also be used to other networking problems.
- We implement ACT-TE using ns-3 and perform a comprehensive evaluation over three representative network topologies in terms of throughput, delay and utility.
- Extensive packet-level simulation results demonstrate the effectiveness and superiority of ACT-TE over the above three straightforward baselines and several widely-used traditional methods. More importantly, these results confirm that transfer learning and policy distillation are the right choice for dealing with re-configurations in the context of experience-driven networking.

To the best of our knowledge, we are the first to tackle the network re-configuration problem using transfer learning in the context of experience-driven networking.

4.2 Problem Statement

For representation completeness, we briefly introduce the TE problem and the transfer learning setting in this section. Note that the similar TE definition is also used in Chapter 3.

We consider a general network G with M end-to-end communication sessions, each of which consists of a set of candidate paths $\mathbf{P}$ for a pair of source and destination nodes. The policy of a DRL agent is to find a set of split ratios $\boldsymbol{w}$ for each candidate path on each communication session. Specifically, for a communication session m with $|\mathbf{P}_m|$ candidate paths, a split ratio $w_{m,n}$ specifies the probability of a packet to be transmitted by path n. Note that the sum of all split ratios within the same session is 1, e.g. $\sum_{n=1}^{|\mathbf{P}_m|} w_{m,n} = 1$.

We apply a widely used α-fairness model $U_\alpha(x) = \frac{x^{1-\alpha}}{1-\alpha}$ as the utility function to evaluate the Quality-of-Service (QoS) of the TE problem [117, 134]. The objective of the TE problem is to maximize the total utility function $U(\cdot)$ over all communication sessions, which is defined as:

$$\sum_{m=1}^{M} U(x_m, y_m) = \sum_{m=1}^{M} U_{\alpha_1}(x_m) - \kappa \cdot U_{\alpha_2}(y_m), \tag{4.1}$$

where x_m and y_m are end-to-end throughput and delay of session m, respectively, and κ is used to balance the relative importance of throughput and delay.

In this thesis, our main focus is the transfer learning on network re-configurations in the TE problem. Suppose we already have a well-trained DRL agent $\mathcal{Q}$ on network G, the goal of the transfer learning is that with the guidance of the existing agent $\mathcal{Q}$, a new DRL agent can quickly and effectively adapt to a new network $\hat{G}$, whose network configuration is significantly different from G.

When extending our framework to other networking problems, such as congestion control or resource allocation, we only need to replace the definition of state space, action space, and reward function. Then we can apply the proposed framework for transfer learning, i.e., applying a well-trained DRL agent to help the training of the new DRL agent in a new environment.

4.3 Actor-Critic-based Transfer Learning Framework

We first design the state space, action space, and reward function for the TE problem described above.

STATE: the state of a DRL agent at decision epoch t consists of two parts: 1) the path-side state statistics, including all candidate paths of each communication session. Formally, the path-side state statistics for a path n of the session m can be represented by $\mathcal{A}_m^n = (h_m^n, l_m^n, j_m^n, o_m^n)$, in which four variables indicate the end-to-end path throughput, average

packet delay, packet jitter and packet delivery ratio, respectively; 2) the session-side state statistics, including all communication sessions. Formally, the session-side state statistics for a communication session m, $\mathcal{B}_m = (x_m, y_m, d_m)$, in which three variables indicate the end-to-end session throughput, average packet delay and sending rate, respectively. Overall, the state is defined as $\mathbf{s} = [\mathcal{A}_m^n, \mathcal{B}_m], \forall\ m \in \{1 \dots M\}, \forall\ n \in \{1 \dots |\mathbf{P}_m|\}$. Note that both path-side and session-side run-time statistical information can help build the knowledge of network traffic patterns, which is especially useful when the network configuration is changed.

ACTION: the action indicates a set of split ratios for all the communication sessions, which is defined as $\mathbf{a} = [w_m^n], \forall\ m \in \{1 \dots M\}, \forall\ n \in \{1 \dots |\mathbf{P}_m|\}$. As mentioned above, the sum of a set of split ratios for a communication session m is to 1, i.e., $\sum_{n=1}^{|\mathbf{P}_m|} w_{m,n} = 1$.

REWARD: the reward r is defined as the total utility value of all communication sessions, $r = \sum_{m=1}^{M} U(x_m, y_m)$.

In a network, each communication session consists of multiple paths, thus we included the well-known key statistics from both path-side and session-side, such as the end-to-end path throughput and the average session throughput. These statistics can reflect the state of the network. We indeed have also considered other information to the state space (such as historical statistics from past decision epochs) but observed that they did not lead to noticeable performance gains in our experiments (and may increase the computation overhead). As demonstrated by our experiments, the chosen statistics are sufficiently good for characterizing the system state and action, and the DRL agent can achieve a satisfying performance. The action is quite straightforward that determines the probability of a packet to be transmitted by one path, and we used the α-fairness model in the reward function that could well balance the efficiency and fairness among communication sessions. The same definition is also used by previous work [11].

To tackle the continuous action space, we propose to use the actor-critic-based DRL algorithm, specifically, the DDPG algorithm, as the experience-driven networking method to

solve the TE problem as described in section 4.2. Suppose there already exists a DRL agent $\mathcal{Q}$, whose critic network has been well-trained by the DDPG algorithm. In this chapter, we mainly focus on the training of a new DRL agent when the network is re-configured, In other words, on a new network environment, we need to train a new actor network $\pi(\cdot)$ and a new critic network $Q(\cdot)$ with weights $\boldsymbol{\theta}_\pi$ and $\boldsymbol{\theta}_Q$, respectively. For simplicity, we use the term *agent* or *DRL agent* to indicate the new DRL agent.

As mentioned above, it is quite common that a deployment network configuration is different from the one used to train the DRL agent, or the configuration of a communication network is changed over time. Therefore, to quickly adapt to the network environment with a new configuration, an efficient and effective transfer method is necessary. However, as analyzed above and confirmed by our simulation results in Section 6.4, some straightforward transfer methods, like directly applying an existing well-trained DRL agent to a new network environment or directly fine-tuning an existing DRL agent in the new network environment, do not work well for TE for the following reasons: 1) since a DRL agent learns the control policy from its past experiences, a brand new network configuration, which, without exploring new experience, may cause confusion to the agent; 2) the DDPG method only uses a plain replay buffer to store and sample transitions, which fails to distinguish more informative transitions and does not fully take the knowledge of the existing DRL agent into consideration. To solve the transfer problem mentioned above, we propose ACT-TE, an Actor-Critic-based Transfer learning framework for TE. ACT-TE takes an existing well-trained DRL agent $\mathcal{Q}$ and a prioritized experience replay buffer $\mathcal{B}$ as input, to boost the training of a new DRL agent in the network environment with a new configuration. The framework is shown in Fig. 4.1. ACT-TE leverages new experience (i.e., collected with priorities from the prioritized experience replay buffer $\mathcal{B}$) and old knowledge (i.e., distilled from an existing well-trained DRL agent $\mathcal{Q}$) to compute the boosted TD-error ξ. The boosted TD-error is then used to update the critic network and the priority of the selected transition.

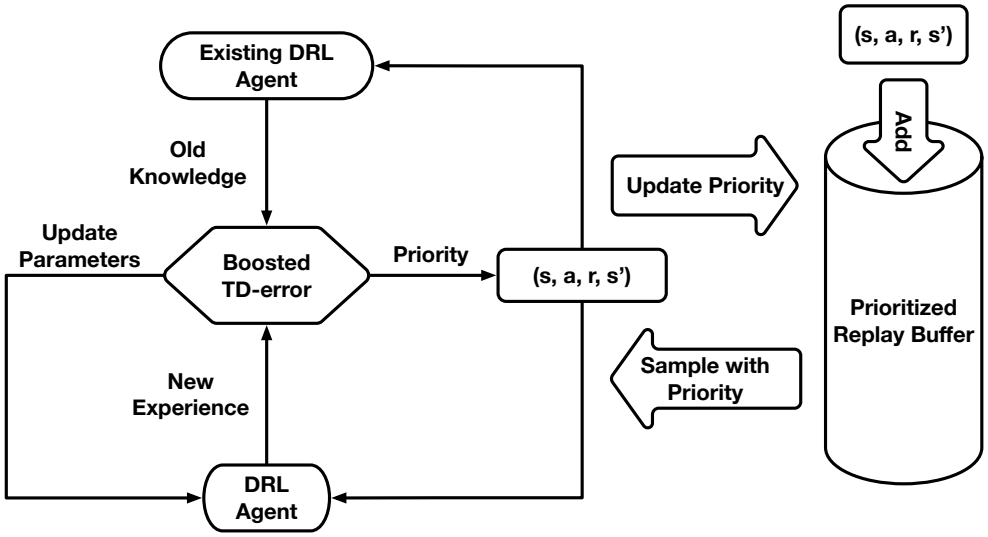


Fig. 4.1: The ACT-TE framework

As mentioned in Chapter 2.2, the critic network is a value function, which specifies the expected discounted cumulative reward (Q-value) of a state-action pair. In other words, the critic network discriminates how *good* or *poor* an action is for a given state. For an existing well-trained DRL agent in its learning environment, the critic network can precisely estimate the Q-value and provide guidance to train the actor network. Because of the generalization ability of DNNs, the critic network can still provide feedback to the actor network even though that state-action pair has not been seen before. In our TE problem, when the configuration of network G is changed, the critic network may not work well under the new network environment $\hat{G}$, especially lacking of the training with new transitions collected from the new network environment. However, compared with a new agent with randomly initialized critic network, the critic network from $\mathcal{Q}$ is still more informative at the beginning of several training epochs. Similar to [106], in ACT-TE, we use the critic network of an existing DRL agent $\mathcal{Q}$ to distill previous knowledge and transfer it to a new DRL agent. Note that for simplicity, we use $\mathcal{Q}$ to represent the critic network of the DRL agent as well. The boosted TD-error ξ for training the critic network is defined as:

$$\xi_i = r_i + \gamma\Phi_i - Q(\mathbf{s}_i, \mathbf{a}_i), \tag{4.2}$$

where the definition of ξ_i is different from the original DDPG algorithm, which only considers the Q-value from critic network for the state $\mathbf{s}_{i+1}$ and action $\pi(\mathbf{s}_{i+1})$ pair. ξ_i is defined as the combination of Q-values from both critic network and existing DRL agent $\mathcal{Q}$:

$$\Phi_i = \epsilon \cdot \mathcal{Q}(\mathbf{s}_{i+1}, \pi'(\mathbf{s}_{i+1})) + (1-\epsilon) \cdot Q'(\mathbf{s}_{i+1}, \pi'(\mathbf{s}_{i+1})), \tag{4.3}$$

where ϵ is a decay variable to balance the relative importance of $Q'(\cdot)$ and $\mathcal{Q}$. Note that ϵ decreases with the training epochs, expressing the fact that with more and more new transitions are used to train the new DRL agent, we trust more in current critic network than the previous one from $\mathcal{Q}$. The $\pi'(\cdot)$ and $Q'(\cdot)$ are the target networks of actor and critic, which are introduced by [81] to improve the learning stability. The target networks clone the structures of the actor and critic networks, their weights $\boldsymbol{\theta}_{\pi'}$ and $\boldsymbol{\theta}_{Q'}$ are initialized with their counterparts and slowly updated following their counterparts:

$$\boldsymbol{\theta}_{Q'} \leftarrow \sigma\boldsymbol{\theta}_{Q} + (1-\sigma)\boldsymbol{\theta}_{Q'} \tag{4.4}$$

$$\boldsymbol{\theta}_{\pi'} \leftarrow \sigma\boldsymbol{\theta}_{\pi} + (1-\sigma)\boldsymbol{\theta}_{\pi'} \tag{4.5}$$

The same as Chapter 3, we extend the prioritized experience replay [108] into DDPG by taking into consideration the gradient (Q gradient) from the critic network. The priority of each transition is defined as the combination of TD-error and Q gradient,

$$p_t = \eta \cdot (|\delta_t| + \phi) + (1-\eta) \cdot \overline{|\nabla_{\mathbf{a}} Q|}, \tag{4.6}$$

where η is a trade-off hyper-parameter. $\overline{|\nabla_{\mathbf{a}} Q|}$ is the average of the absolute value of Q gradient. ϕ is a small constant to avoid edge cases when TD-error is zero. The probability of a transition i to be selected is defined as:

$$P(i) = \frac{p_i^{\tau_0}}{\sum_j p_j^{\tau_0}}, \tag{4.7}$$

where p_i is the priority of transition i, and τ_0 is a scaling factor to determine how much prioritization to use. When $\tau_0 = 0$, the prioritized sampling becomes uniform sampling, i.e., sampling transition without priorities. The denominator indicates the sum of all priorities (with scaling factor) of transitions in the replay buffer.

Due to the prioritized sampling manner, the prioritized replay buffer imports bias to the training of parameters. In order to correct the training bias, Importance-Sampling (IS) weigh ρ [108] is introduced as:

$$\rho_i = \Big(\frac{1}{|\mathcal{B}| \cdot P(i)}\Big)^{\tau_1}, \tag{4.8}$$

where τ_1 is a linear annealing weight from its initial value to 1, which is used to avoid very large updating in the training. Moreover, IS weight ρ_i is usually normalized over a mini-batch $\rho_i = \rho_i / \max_j \rho_j$ for better learning stability.

Overall, the weights of critic network $\boldsymbol{\theta}_Q$ can be updated by the boosted TD-error ξ_i with IS weight ρ_i:

$$\boldsymbol{\theta}_Q \leftarrow \boldsymbol{\theta}_Q - \beta^Q \cdot \rho_i \cdot \nabla_{\boldsymbol{\theta}_Q} \xi_i^2, \tag{4.9}$$

and the weights of actor network can be updated by:

$$\boldsymbol{\theta}_\pi \leftarrow \boldsymbol{\theta}_\pi - \beta^\pi \cdot \nabla_{\mathbf{a}} Q(\mathbf{s}_t, \pi(\mathbf{s}_t)) \cdot \nabla_{\boldsymbol{\theta}_\pi} \pi(\mathbf{s}_t), \tag{4.10}$$

where β^Q and β^π are the learning rates of actor and critic network, respectively.

We formally present ACT-TE as Algorithm 2. In order to train a new agent, we need an existing DRL agent $\mathcal{Q}$ as input to our framework. Before the training, we need to initialize all the weights of DNNs. The weights $\boldsymbol{\theta}_\pi$ of actor network $\pi(\cdot)$ and the weights $\boldsymbol{\theta}_Q$ of critic network $Q(\cdot)$ are randomly initialized. The weights $\boldsymbol{\theta}_{\pi'}$ and $\boldsymbol{\theta}_{Q'}$ of target actor network $\pi'(\cdot)$ and target critic network $Q'(\cdot)$ are initialized with the weights of their counterparts, $\boldsymbol{\theta}_{\pi'} \leftarrow \boldsymbol{\theta}_\pi$ and $\boldsymbol{\theta}_{Q'} \leftarrow \boldsymbol{\theta}_Q$. We can apply an exploration method during the training to select

actions (line 3), e.g., the Ornstein-Uhlenbeck processing or the TE-aware exploration as proposed in the Chapter 3. The transition samples are first stored into the replay buffer $\mathcal{B}$ with maximal priorities and then are sampled with priorities (line 6 and line 8). The next state $\mathbf{s}_{i+1}$ is obtained from the sampled transition, and the action $\mathbf{a}_{i+1}$ is derived from the target actor network $\pi'(\cdot)$ (line 9). The boosted TD-error then can be computed (line 10 and line 11) and used to update the transition priority (line 13). With the boosted TD-error and computed IS weight (line 12), the weights of critic network $\boldsymbol{\theta}_Q$ and actor network $\boldsymbol{\theta}_\pi$ can be updated (line 15), and the weights of target networks can be updated accordingly (line 16).

In our implementation, we split the input (i.e., state **s** and action **a**) for the critic network into three feedforward paths: 1) the path-side statistics of input state **s** are filtered by a 1-D convolutional neural network with 32 1x3 filters and followed by a 1-D maxpooling layer; 2) the session-side statistics of input state **s** are connected to a fully-connected (FC) layer with 64 neurons; 3) the input action **a** to the critic network is directly connected to an FC layer with 256 neurons. The outputs of above three paths are concatenated together and then connected to a hidden FC layer with 256 neurons. The actor network has the same structure design with critic network to extract features from the state input. The hidden FC layer of the actor network is slightly different from the critic network. The output path of the path-side statistics and the output path of the session-side statistics are directly connected to two FC hidden layers, with 256 neurons each. All the layers use *ReLu* as activation function except the output layer of actor and critic, which are *linear* and *sigmoid*, respectively. Note that we also introduce batch-normalization layer and l-2 regularization for the purpose of stabilizing training. The prioritized replay buffer is implemented by sum-tree for efficient sampling, which is the same as [108]. The number of batch size K is 32, the reward discount factor γ is set to 0.99, the learning rate β^π and β^Q are 0.0001 and 0.001, respectively, the updating rate σ for target networks is 0.001. In addition, we set $\tau_0 = 0.1$, $\tau_1 = 0.9$, $\eta = 0.9$.

Algorithm 2: ATC-TE

Input: Existing DRL agent $\mathcal{Q}$, prioritized replay buffer $\mathcal{B}$, maximum training epoch T, mini-batch size K;
while epoch t $< T$ **do**
  Select an action $\mathbf{a}_t$ with exploration method;
  Execute action $\mathbf{a}_t$ and get reward r_t;
  Observe next state $\mathbf{s}_{t+1}$;
  Store transition sample $(\mathbf{s}_t, \mathbf{a}_t, r_t, \mathbf{s}_{t+1})$ into $\mathcal{B}$ with the maximal priority $p_t = p_{max}$;
  for i = 1 **to** K **do**
    Sample a transition $(\mathbf{s}_i, \mathbf{a}_i, r_i, \mathbf{s}_{i+1})$ from $\mathcal{B}$ with probability computed by Equ. (4.7);
    Estimate the action for state $\mathbf{s}_{i+1}$: $\mathbf{a}_{i+1} \leftarrow \pi'(\mathbf{s}_{i+1})$;
    Obtain estimated Q-value for the pair ($\mathbf{s}_{i+1}$, $\mathbf{a}_{i+1}$) from $Q'(\cdot)$ and existing agent $\mathcal{Q}$: $\Phi \leftarrow \epsilon \cdot Q'(\mathbf{s}_{i+1}, \mathbf{a}_{i+1}) + (1-\epsilon) \cdot \mathcal{Q}(\mathbf{s}_{i+1}, \mathbf{a}_{i+1})$;
    Compute the boosted TD-error by Equ. (4.2);
    Compute the IS weight by Equ. (4.8);
    Update the priority of transition i by Equ. (4.6);
  end for
  Update $\boldsymbol{\theta}_Q$ and $\boldsymbol{\theta}_\pi$ by Equ. (4.9) and Equ. (4.10) respectively;
  Update $\boldsymbol{\theta}_{Q'}$ and $\boldsymbol{\theta}_{\pi'}$ by Equ. (4.4) and Equ. (4.5) respectively;
end while

4.4 Performance Evaluation

To evaluate the performance of ACT-TE, we conducted extensive simulations using the packet-level network simulator ns-3 [87]. The DNN models (i.e., the actor network, the critic network, and their corresponding target networks) were implemented by Tensorflow [1]. Due to the light weight design of our DNN models, we directly ran the simulation and trained the DNN models on a regular desktop with Intel Quad-Core 2.6 Ghz and 8 GB memory. In the rest of this section, we detail our simulation and analyze the simulation results.

In our simulation, we evaluated ACT-TE on two well-studied network topologies, Advance Research Projects Agency Network (ARPANET [3]) and NSF Network (NSFNET [88]). Moreover, we applied a widely-used network topology generator BRITE [80] to generate random network topologies with 20 nodes and 100 links. We randomly generated 10 end-to-end communication sessions (i.e., $M = 10$) for each network topology and randomly

selected top-3 shortest paths for each communication session. The link capacity is set to 100Mbps and the propagation delay is set to 1ms. Note that the traffic demand for each communication session is not fixed. Instead, it follows a Possion distribution with its mean value ranging from 20Mbps to 100Mbps. We set the decision epoch to 100ms.

To investigate the transfer learning performance of ACT-TE on the TE problem, we generated a pair of configurations for each topology: one will be used to train a DRL agent as the existing agent $\mathcal{Q}$, referred to as the old configuration; the other one will be used to evaluate the transfer learning algorithms, referred to as the new configuration. We proposed three simulation scenarios with different types of network configuration changes:

- For NSFNET, keeping the same network topology and the communication sessions (i.e., with the same source and destination nodes), but changing the candidate path set for each session.
- For ARPANET, keeping the same network topology, but changing the communication sessions with different source and destination nodes (their candidate path set is changed accordingly).
- For random topologies, changing the entire network topology, in the meanwhile, changing the communication sessions and the candidate path set.

For comparison, we used three widely-used traditional methods as TE solutions:

- **Random Bandwidth Allocation (RND)**: Each communication session randomly determines the split ratios of its candidate paths at each decision epoch.
- **Load balance (LB)**: Each communication session evenly distributes the traffic load on its candidate paths.
- **Network Utility Maximization (NUM)**: Each communication session obtains the split ratios of its candidate paths by solving a convex programming problem with an optimization objective to maximize the network utility.

Besides, we also compared our ACT-TE with three straightforward learning-based transfer methods:

- **Directly Load (LOAD)**: Directly applying the existing DRL agent to make decisions without any further training in the new network environment.
- **Fine-tune (TUNE)**: Applying the existing DRL agent, but fine-tune with new experience from the new network environment.
- **Train from scratch (SCR)**: Training a new DRL agent from scratch without any old knowledge.

In the simulation, for each pair of configurations, we first trained a DRL agent in the network environment with the old configuration until it was converged (i.e., the agent can make *good* decisions in that environment). Then we changed the network environment with the new network environment and evaluated the performance of ACT-TE and other baseline methods. For a fair comparison, we limited the number of training epochs to 2,000 in the new network environment for ACT-TE, TUNE and SCR.

We compared three commonly-used network performance metrics for evaluation, including the total end-to-end throughput, the average end-to-end packet delay, and the average utility value within one decision epoch. First we show the performance of a well-trained DRL agent in the old network environment on a random generated topology. The result is shown in Fig. 4.3 (the DRL agent is marked as DRL). As expected, the well-trained DRL agent obviously outperforms other traditional baseline methods in terms of utility value. This result illustrates that the agent can work well on the old configuration. Note that we have not performed any transfer learning here. This well-trained agent will be used as an input to our ACT-TE. We have the same observations in NSFNET and ARPANET, i.e., a well-trained DRL agent in the old network environment works well for the TE problem. We omit other results on the old configurations for simplicity. The simulation results of transfer learning are shown in Figs. 4.2, 4.4 and 4.5, each of which shows one of the network

changing case on its corresponding network topology. The results are presented by error bars, which stand for the mean value $\pm$ standard deviation over all the decision periods. By summarizing these results, we can make the following observations:

1) Although the learning goal of the DRL agent (reward) is not to directly maximize the average throughput or minimize the average delay, ACT-TE still can find feasible solutions that significantly reduce end-to-end delay while preserving a high throughput, as shown in Figs. 4.2, 4.4 and 4.5. For instance, on the randomly generated topology, ACT-TE reduces the average end-to-end delay by 24.6%, 31.2% and 20.7% compared to LB, NUM and RND, respectively. The reason is that these traditional TE solutions hardly take run-time statistics of the network environment into consideration. The results confirm that the

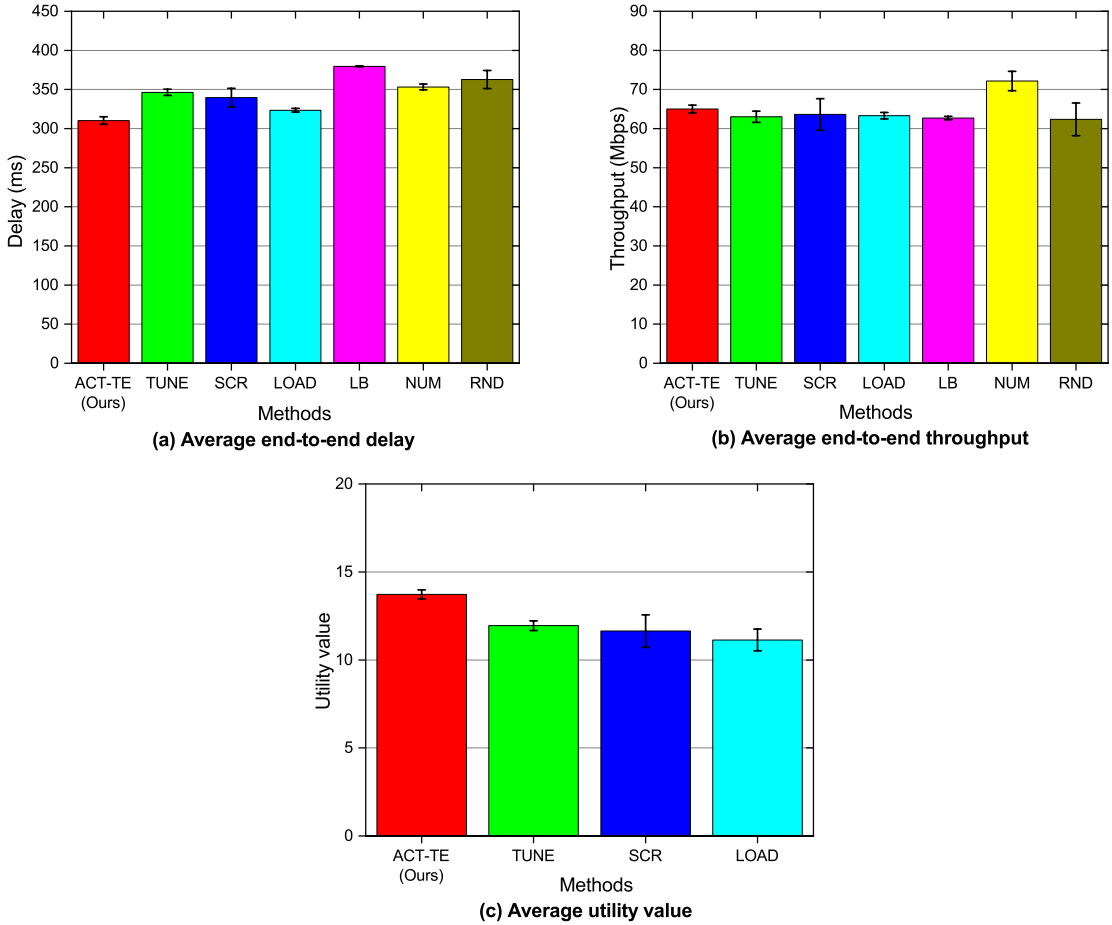


Fig. 4.2: Performance of all the methods in NSFNET

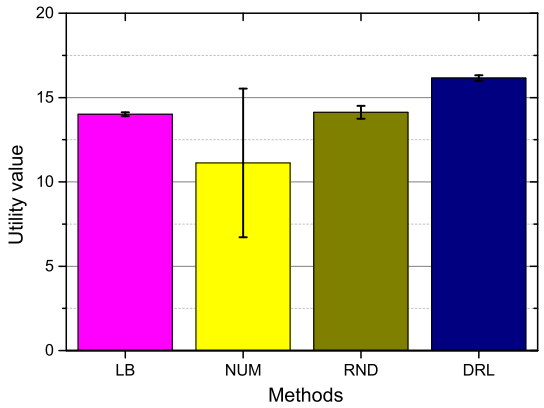


Fig. 4.3: Performance of the well-trained DRL model on the random topology

DRL and policy distillation are the right choices for dealing with the TE problem and reconfigurations in the context of experience-driven networking.

2) As expected, we can see from Figs. 3(c), 4(c) and 5(c) that ACT-TE outperforms all other three straightforward learning-based transfer methods in terms of utility value. Although the learning goal of SCR and TUNE is the same as ACT-TE, training with the same number of epochs, ACT-TE can effectively and quickly find solutions to solve the TE problem in a new network environment. For example, in NSFNET, ACT-TE achieves 14.9%, 17.9% and 23.3% more utility values compared to TUNE, SCR and LOAD, respectively. It is interesting to see that TUNE does not work well in all scenarios. We suspect this is due to the mechanism of DNNs and its stochastic gradient descent updating manner. A well-trained neural network is easier to get stuck on the part of the state and action space when facing an unkown environment. That is why the DNNs of DRL are required to be randomly initialized at the beginning of training. However, in our framework, instead of directly mimicking the weights of DNNs of the existing DRL agent, we utilized the knowledge of the existing agent in a softer manner, i.e., learning from its Q-value in a decayed way. With more transitions are collected during the training, we trusted more on the judgment of the new agent, thus avoiding getting stuck on a local optimal.

3) From Figs. 4.2(c), 4.4(c) and 4.5(c), we can see that LOAD shows the worst per-

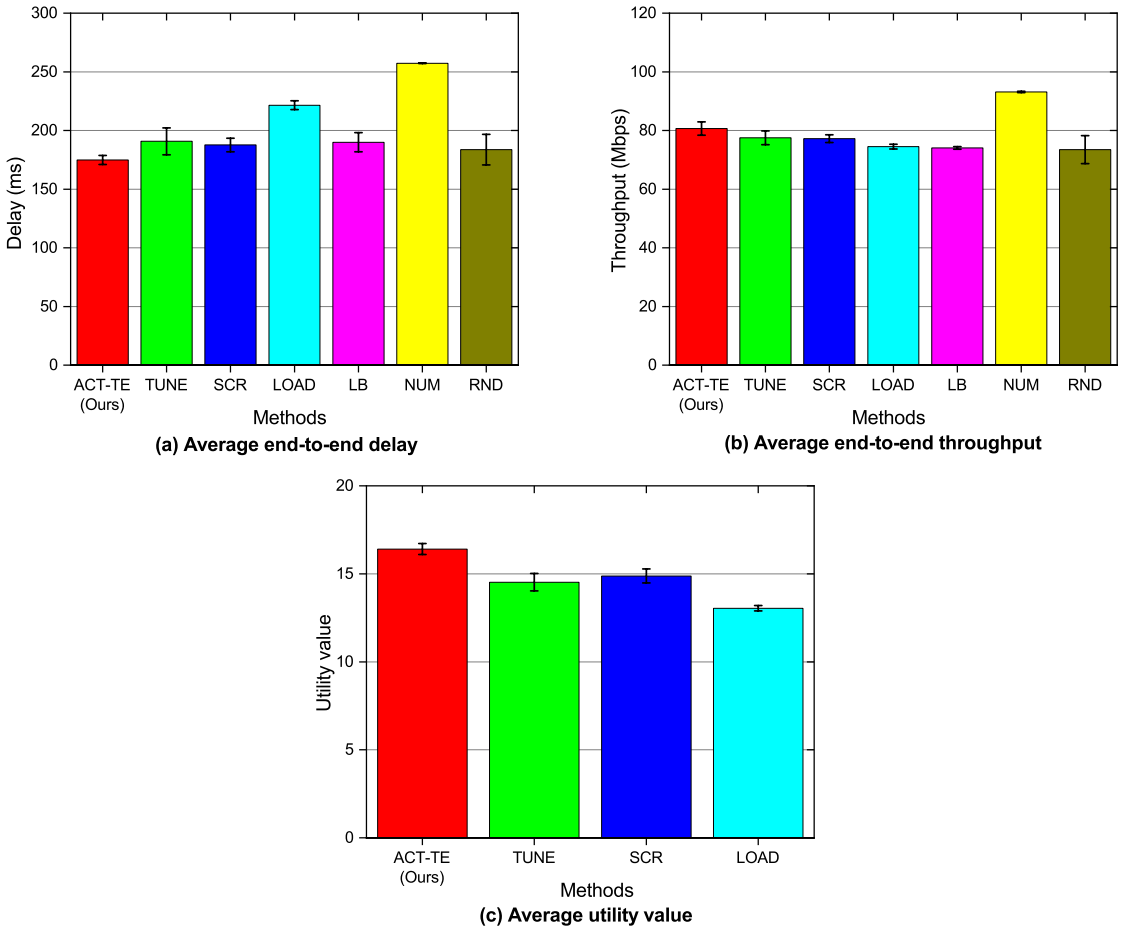


Fig. 4.4: Performance of all the methods in ARPANET

formance among three learning-based transfer methods in terms of utility value. It is not surprising, since LOAD directly applies the existing DRL agent to the new network environment without any further training with new experience. ACT-TE, TUNE and SCR, instead, utilize new transitions collected from the new network environment to improve their policies. In particular, from Figs. 4.4(a) and 4.4(b), we can find that without any training in the new network environment, LOAD even performs much worse than traditional TE methods (i.e., the end-to-end delay of LOAD is much higher than LB but their average throughput are almost the same). The above two observations emphasize that new experience are necessary for the DRL agent to adapt to the new network environment.

4) From Figs. 4.3 and 4.5, we can find that although an existing DRL agent can perform

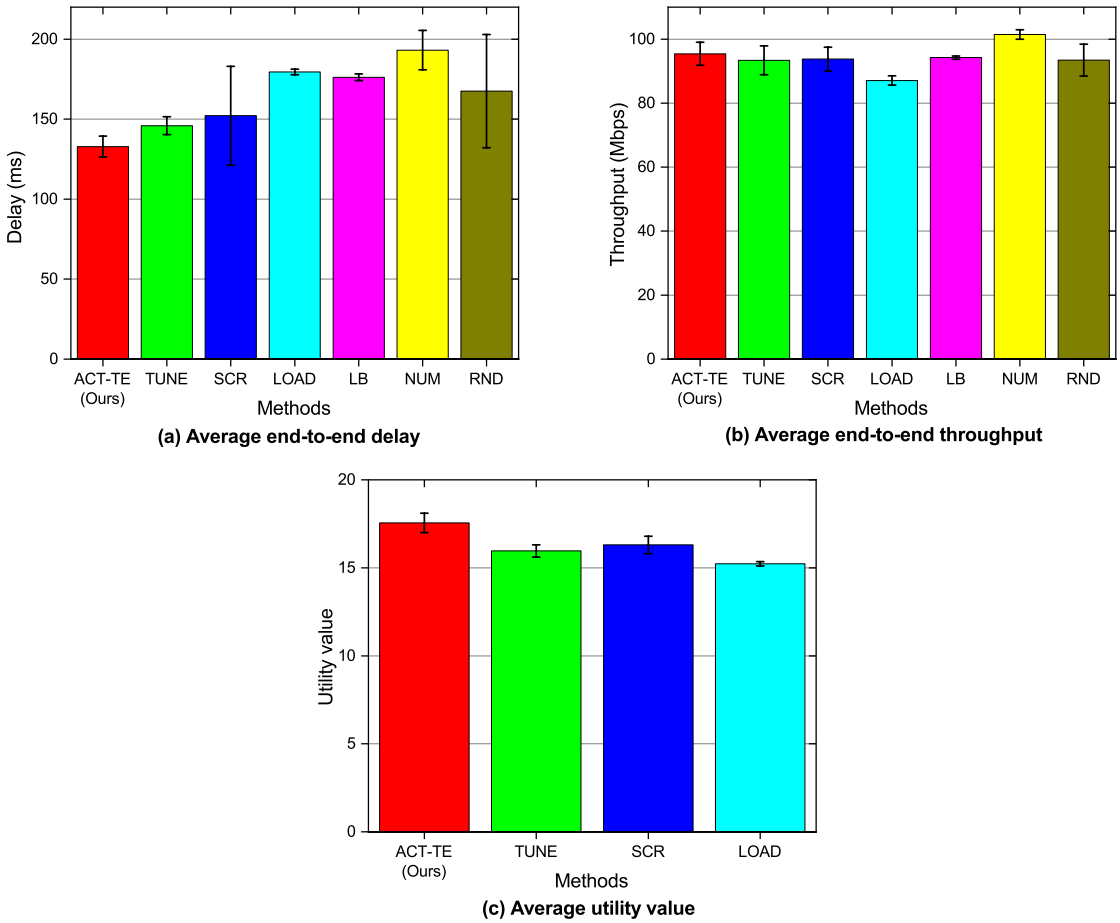
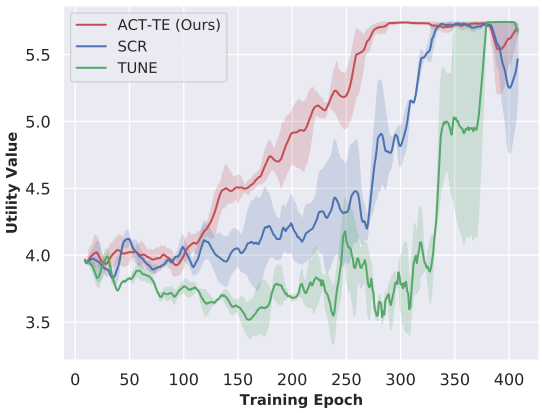


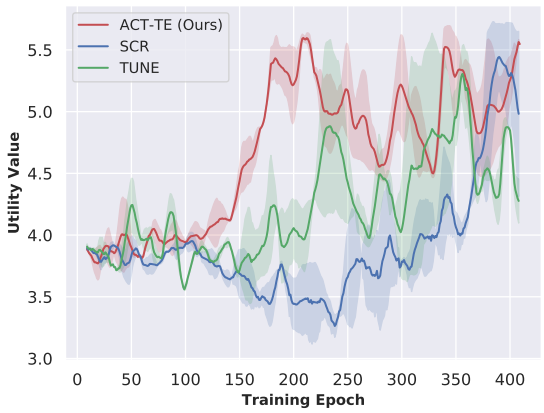
Fig. 4.5: Performance of all the methods on the random topologies generated by BRITE

well in the old network environment, if we keep using it when the network configuration is changed (the entire network topology is changed in this scenario), the DRL agent suffers from serious performance degradation, it even fails to beat the traditional baseline methods, like LB or RND, in terms of delay and throughput on the new network environment. The results confirm that new experience are important to a DRL agent in the TE problem.

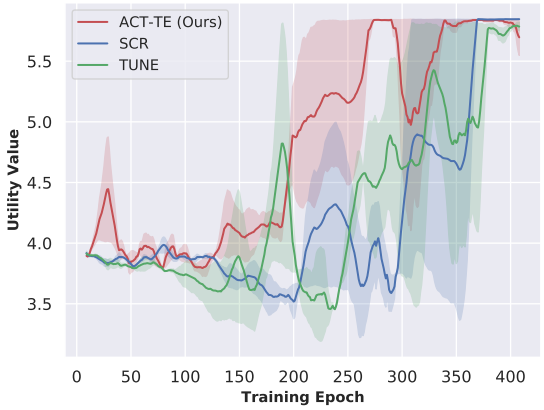
5) According to Figs. 4.2(c), 4.4(c) and 4.5(c), we can observe that compared with SCR, ACT-TE achieves better learning performance in terms of utility value. Note that SCR only utilizes new experience for training, which is collected from the new network environment. However, ACT-TE not only uses new experience, but also utilizes old knowledge from the existing DRL agent. This is the key reason why ACT-TE can boost the training of new



(a) Scenario 1: changing the candidate path set



(b) Scenario 2: changing the communication session



(c) Scenario 3: changing the entire network topology

Fig. 4.6: Performance of experiments. The results are averaged over three runs and the shadow represent a standard deviation.

DRL agent and achieve better performance.

6) As shown in Figs. 4.2(c), 4.4(c) and 4.5(c), compared with TUNE that directly fine-tunes the weights of an existing DRL agent in the new network environment, our ACT-TE takes the advantage of old knowledge captured by the existing agent. Thus ACT-TE is more effective. In particular, fine-tuning the existing agent TUNE in the scenario ARPANET and NSFNET even leads to lower utilities value when compared with SRC. Overall, the observations well verify that the policy distillation design of ACT-TE is effective.

7) According to Figs. 4.2(b), 4.4(b) and 4.5(b), we can see that NUM achieves the highest throughput compared with other methods over all the network topologies. However, since the the objective function of NUM is only to maximize the throughput util-

ity, NUM fails to find feasible solutions that can balance the performance on the network throughput and delay. In contrast, since the learning goal (reward) consists of the throughput and delay, the learning-based methods can find a trade-off between those two metrics. In particular, ACT-TE outperforms all other methods expect NUM in terms of throughput, in the meanwhile, ACT-TE finds solutions with the lowest delay.

8) In addition, we can also observe from Figs. 4.2, 4.4 and 4.5 that no matter how to change the network configuration, ACT-TE can effectively adapt to the new environment. From the smaller change on the path sets to the larger change on the communication sessions to the change on the entire topology, the simulation results confirm that our ACT-TE steadily achieves the best performance with the help of old knowledge and new experience.

Furthermore, we added three more test scenarios to illustrate the convergence of our framework. In these scenarios, instead of limiting the training epochs like previous settings, we trained each learning-based transfer method until convergence. The network topology is randomly generated, which consists of 10 nodes and 20 links. We generated 2 communication sessions and 3 candidate paths for each session. We ran simulations with all three aforementioned network configuration changes. The corresponding results are shown in Fig. 4.6. As we can see, ACT-TE could improve the training convergence speed when moving to new environments. For example, in Fig. 4.6a, when changing the candidate path set for each session, ACT-TE converges faster to adapt to the new configuration. When training with 200 epochs, ACT-TE could achieve 30% and 53% more average utility values compared to SCR and TUNE, respectively. Moreover, we can observe that our framework is stable on all three re-configuration scenarios since the variance between different runs of ACT-TE is relatively small.

4.5 Summary

In this chapter, we proposed to solve a practical and fundamental problem for experience-driven networking: when network configurations are changed, how to train a new DRL agent to quickly adapt to the new environment. Specifically, we presented ACT-TE, which uses the policy distillation to effectively train a new DRL agent to solve the TE problem in new network environments. With the help of both old knowledge (i.e., distilled from the existing agent) and new experience (i.e., collected from the newly collected samples), ACT-TE can achieve the best performance compared with traditional TE methods and straightforward learning-based transfer methods. Based on three widely-used network topologies, NSFNET, ARPANET and randomly generated topology using BRITE, we designed three different re-configuration scenarios accordingly and conducted extensive packet-level simulations using ns-3. The simulation results show that 1) the existing well-trained DRL agents do not work well in new network environments; and 2) ACT-TE significantly outperforms both two straightforward methods (training from scratch and fine-tuning based on an existing DRL agent) and several widely-used traditional methods in terms of network utility, throughput and delay.

CHAPTER 5
EXPERIENCE-DRIVEN CONGESTION CONTROL

5.1 Overview

In this chapter, we consider to design an experience-driven congestion control framework for Multi-Path TCP (MPTCP). MPTCP [27] was designed to make use of multiple network interfaces (e.g., Ethernet, WiFi and 4G/LTE) to improve end-to-end bandwidth and robustness, and has already become a widely-used standard protocol. MPTCP allows to split a single TCP flow into multiple sub-flows across multiple paths. It has attracted lots of attention from both industry and academia due to its potential on significant throughput improvements, which are highly desired for some emerging applications that demand high end-to-end bandwidth.

Current TCP's congestion control does not perform well on lossy and high Round Trip Time (RTT) links, especially on Multi-Path TCP (MPTCP) [21], which has been proposed as a mechanism for transparently supporting multiple connections to the application layer. Moreover, most congestion control algorithms, including those designed particularly for MPTCP, pre-define some packet-level events as response signals, and specify a fixed con-

trol policy with one or multiple rules for different cases. For example, halving the congestion window when a packet loss is detected [52]; adjusting the congestion window by a certain amount based on the changing rate of RTTs [13]. Such congestion control algorithms may not work well in a complex and highly-dynamic network, in which many factors (such as random loss, a large range of RTTs, lossy links, rate reshaping at gateways or middleboxes, etc.) may affect its performance since it looks impossible to pre-define the best or even a good rule for each possible case that may occur at runtime. Note that traditional congestion control algorithms can be considered as heuristic algorithms for the corresponding optimization problems, which likely lead to suboptimal (instead of optimal) solutions. Hence, they can be categorized as optimization-based methods described above.

In this chapter, we present a Deep Reinforcement Learning (DRL) based control framework, DRL-CC (DRL for Congestion Control). DRL-CC utilizes DRL to learn to take best actions according to runtime states without relying on any accurate mathematical model or any pre-defined control policy.

The novelty of our design is to utilize a Recurrent Neural Network (RNN), Long Short-Term Memory (LSTM [51]), under a DRL framework for learning a representation for all active MPTCP flows and dealing with their dynamics. We, for the first time, integrate the LSTM-based representation learning network into an actor-critic framework called DDPG (Deep Deterministic Policy Gradient [71]) for continuous (congestion) control, and leverages the emerging deterministic policy gradient [115] to train critic, actor and LSTM networks in an end-to-end manner.

We implemented DRL-CC based on the MPTCP implementation in the Linux kernel. We conducted extensive real experiments to evaluate its performance. Specifically, we compared DRL-CC with the well-known congestion control algorithms proposed particularly for MPTCP, including LIA [101], BALIA [96], OLIA [64] and wVegas [136], which have all been implemented in the Linux Kernel [91]. Second, we tested our method under different settings and cases, such as different link bandwidths, link delays, packet loss

ratios, etc. In addition, we conducted our performance evaluation in terms of goodput, fairness, robustness and TCP-friendliness. The experimental results well justify the effectiveness and superiority of DRL-CC.

To the best of our knowledge, we are the first to address congestion control in MPTCP using emerging DRL. In addition, the end-to-end trainable model integrating LSTM, actor and critic networks is novel and has not been used in the context of DRL. Moreover, it can handle a variable input size. We believe such a design may have a significant impact on future research along this line since it can be applied to many other system control problems with a time-varying input size, e.g., for routing in mobile ad-hoc networks, connection requests may come and go at any time as well.

5.2 DRL-Based Congestion Control Framework

First of all, we give an overview for DRL-CC, which is illustrated in Fig. 5.1. The key idea behind our design is to utilize a single (rather than multiple independent) DRL agent to perform congestion control for all active MPTCP flows on an end host to maximize the overall utility (defined by a utility function). As mentioned above, “all the active MPTCP flows" only refer to those whose sending host is the one where DRL-CC is running. To realize this idea, we design the architecture of DRL-CC (we will use DRL-CC and DRL-CC agent interchangeably in the following), which consists of the following components:

- Representation Network (Section 5.2.1): It leverages LSTM to learn a *representation* of current states of all active MPTCP and TCP flows in a sequence learning manner.
- Actor-Critic (Section 5.2.2): It trains an actor network and a critic network along with the LSTM-based representation network in an end-to-end manner and derives an action for congestion control of a MPTCP flow based on the learned representation and the state of the target flow.

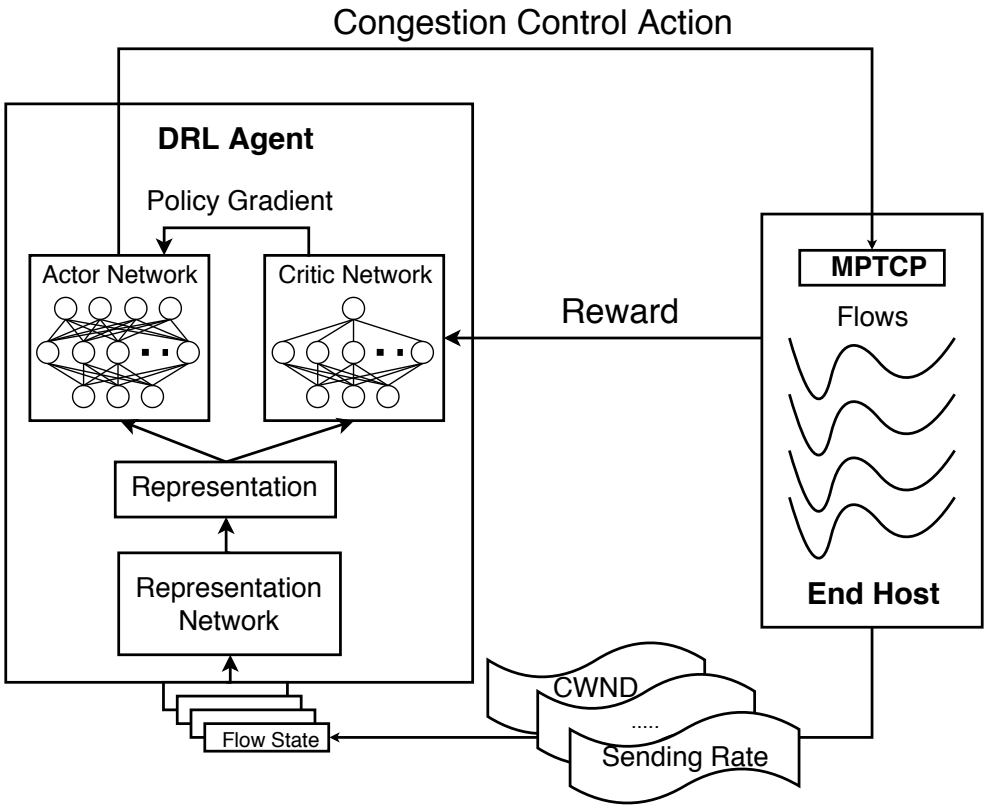


Fig. 5.1: The architecture of DRL-CC

Next, we describe the state, action and reward of DRL-CC:

STATE: The state of a flow i at epoch t $\mathbf{s}_t^i = [\mathbf{s}_t^{1,1}, \cdots, \mathbf{s}_t^{i,k}, \cdots, \mathbf{s}_t^{N,K_i}]$, and $\mathbf{s}_t = [b_t^{i,k}, g_t^{i,k}, d_t^{i,k}, v_t^{i,k}, w_t^{i,k}]$, where $\mathbf{s}_t^{i,k}$ is the state of subflow k of flow i at t; $b_t^{i,k}$, $g_t^{i,k}$, $d_t^{i,k}$, $v_t^{i,k}$ and $w_t^{i,k}$ are the corresponding sending rate, goodput, average RTT, the mean deviation of RTTs and the congestion window size respectively; and N is the total number of both TCP and MPTCP flows, and K_i is the number subflows of flow i. If flow i is a TCP flow, then $K_i = 1$; and if flow i is a MPTCP flow, then $K_i \geq 1$. Then the state at epoch t, $\mathbf{s}_t = [\mathbf{s}_t^1, \cdots, \mathbf{s}_t^i, \cdots, \mathbf{s}_t^N]$. Here *goodput* can be considered as effective throughput, which only counts those successfully received packets. We select these key parameters into the state because they may have a significant impact on the end-to-end performance and have been considered in the design of some related works [134]. During our testing, we found that adding more parameters into the state does not necessarily result in noticeable performance improvement, which, however, undoubtfully increases data collection overhead. Note that

the values of these parameters are all measured during the past epoch $(t-1)$. In order to well address interference and fairness on an end host, we consider all the flows (including both regular TCP and MPTPC) when designing the state space. Certainly, if the flow is a (regular) TCP flow, then there is only one subflow (i.e., $K_i = 1$).

ACTION: An action at epoch t $\mathbf{a}_t = [x_t^1, \cdots, x_t^k, \cdots, x_t^K]$, where x_k specifies how much change needs to be made to the congestion window of subflow k of the target MPTCP flow. The positive, negative and 0 values lead to increasing, reducing and staying at the same congestion window size respectively. Note that at each epoch t, DRL-CC only takes an action on one (target) MPTCP flow.

REWARD: the reward at epoch t, $r_t = \sum_i^N U(i,t)$, where $U(i,t)$ gives the utility of active MPTCP/TCP flow i . Note that the proposed framework is not restricted to any particular utility function. Many different functions (such as throughput, delay, α-fairness [117]) can be used here to calculate the network utility. This reward should be designed according to real needs from upper-layer applications. In our implementation, we chose a widely-used utility function $U(i,t) = \log g_t^i$ [134], where g_t^i is the average goodput of MPTCP flow i during the past epoch. It is known that maximizing this utility function leads to proportional fairness, which is considered to achieve a good tradeoff between goodput and fairness. Moreover, the reward takes into account both TCP and MPTCP flows for the sake of TCP-friendliness.

In short, DRL-CC works as follows. The DRL-CC agent interacts with the end host by collecting the above runtime state information $\mathbf{s}_t$ at each epoch t. The agent is periodically queried by each MPTCP flow and there is only one querying flow at each epoch t (i.e., target flow). At each epoch t, the agent derives an action using the actor and critic networks according to the representation learned by the LSTM-based network and the state of the target flow. Then it deploys the action via the MPTCP implementation (in the OS kernel) to the target flow.

5.2.1 Representation Network

The representation network takes as input the states of all active TCP and MPTCP flows (i.e., $\mathbf{s}_t$) at each decision epoch t and generates a representation (i.e., a vector with a smaller size), which is then used by the actor-critic method (Section 5.2.2) for deriving actions. As mentioned above, the main difficulty is to deal with the situation in which the number of flows may change over time. Most DNN (such as a feed-forward neural network) need to have a fixed input size. A straightforward way to use a feed-forward neural network here is to zero-pad the input if the actual number of flows is smaller its input size. We tested this solution via experiments and found that it is ineffective, especially for the cases where the number of flows is much smaller. Similarly, if the number of flows is larger, then we have to exclude some flows, which obviously lead to poor representation learning too. We decide to choose LSTM [51] to serve this purpose, which can have a variable input size (length).

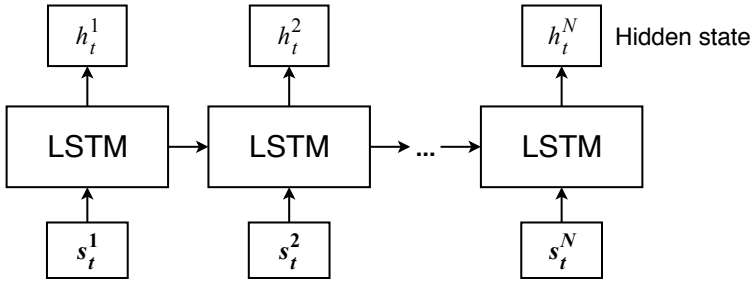


Fig. 5.2: The representation network

As illustrated in Fig. 5.2, the states of flows are fed into LSTM one by one (one at each step) and the representation is learned in a sequence learning manner [118] such that the last hidden state $\mathbf{h}_t^N$ is returned as the representation. For simplicity, we denote this representation for epoch t by $\mathbf{h}_t$ (rather than $\mathbf{h}_t^N$). It is worth mentioning that this LSTM-based representation network can be trained together with the actor and critic networks using back propagation in an end-to-end manner, which is discussed in the next section. This is very important since end-to-end training likely leads to better performance than

training each part of a model separately.

5.2.2 DRL-CC Framework

At each decision epoch t, the representation $\mathbf{h}_t$ (learned by the LSTM-based network described above) is concatenated with the state of the target MPTCP flow and fed into the actor-critic method as input. Then the actor-critic method leverages the actor and critic networks to derive an action, which specifies how to adjust the congestion window size for each subflow of the target MPTCP flow. As mentioned above, the network utility will be calculated used as a reward signal to optimize the decision policy.

We formally present the DRL-CC framework in Algorithm 3. First the algorithm randomly initialize all parameters $\boldsymbol{\theta^R}$ of representation network $R(\cdot)$; $\boldsymbol{\theta^\pi}$ of actor network $\pi(\cdot)$; and $\boldsymbol{\theta^Q}$ of critic network $Q(\cdot)$. The target networks are used here to improve the learning stability. Target networks $R'(\cdot)$, $\pi'(\cdot)$ and $Q'(\cdot)$ clone the structures of their counterparts, whose parameters are initialized using their counterparts (line 2) and slowly updated using a control parameter τ (line 19). τ is usually set to a very small value such that these target networks are only slightly updated in this step. In our implementation, we set $\tau = 0.001$. This DRL agent will run as a daemon process, waiting for queries from MPTCP flows. So the main body of this algorithm includes a dead loop, where $\mathbf{e}_t$ is the state of the querying (i.e., target) MPTCP flow.

Since all the parameters of the DNNs are randomly initialized, in the early stage of training, the DRL agent cannot totally rely on the action derived from the actor network. An inexperienced DRL agent needs to explore sufficiently with random transition samples to gain necessary good and bad experience, and eventually learns a good (hopefully the best) control policy. Similar as in [71], we apply an Ornstein-Uhlenbeck process to add some random noise to a derived action for efficient and effective exploration in this continuous control task.

The representation of all active flows $\mathbf{h}_t$ is derived from the representation network

Algorithm 3: DRL-CC

1: Randomly initialize representation network $R(\cdot)$, actor network $\pi(\cdot)$ and critic network $Q(\cdot)$, with parameters $\boldsymbol{\theta^R}$, $\boldsymbol{\theta^\pi}$ and $\boldsymbol{\theta^Q}$ respectively;
2: Initialize target networks $R'(\cdot)$, $\pi'(\cdot)$ and $Q'(\cdot)$ with parameters $\boldsymbol{\theta^{R'}} := \boldsymbol{\theta^R}$, $\boldsymbol{\theta^{\pi'}} := \boldsymbol{\theta^\pi}$, $\boldsymbol{\theta^{Q'}} := \boldsymbol{\theta^Q}$;
3: Initialize replay buffer $\mathbf{B}$;
4: Initialize Ornstein-Uhlenbeck process $\mathcal{O}$ for exploration;
5: **while** (TRUE) **do**
6: Derive hidden state $\mathbf{h}_t$ from the representation network $R(\mathbf{s}_t)$;
7: Derive an action $\overline{\mathbf{a}_t}$ from the actor network $\pi(\mathbf{e}_t, \mathbf{h}_t)$;
8: Apply the random process $\mathcal{O}$ to generate an action $\mathbf{a}_t$ based on $\overline{\mathbf{a}_t}$;
9: Execute action $\mathbf{a}_t$ and observe the reward r_t;
10: Store transition sample $(\mathbf{s}_t, \mathbf{e}_t, \mathbf{a}_t, r_t, \mathbf{s}_{t+1}, \mathbf{e}_{t+1})$ into replay buffer $\mathbf{B}$;
/**Training the three networks**/
11: Sample H transitions $(\mathbf{s}_j, \mathbf{e}_j, \mathbf{a}_j, r_j, \mathbf{s}_{j+1}, \mathbf{e}_{j+1})$ from $\mathbf{B}$;
12: Obtain representation $\mathbf{h}_{j+1}$ from $R'(\mathbf{s}_{j+1})$;
13: Compute target value for the critic network $Q(\cdot)$: $y_j := r_j + \gamma \cdot Q'(\mathbf{e}_{j+1}, \mathbf{h}_{j+1}, \pi'(\mathbf{e}_{j+1}, \mathbf{h}_{j+1}))$;
14: Update the parameters of the critic network by minimizing the loss: $\frac{1}{H}\sum_{j=1}^{H}(y_j - Q(\mathbf{e}_j, \mathbf{h}_j, \mathbf{a}_j))^2$;
15: Compute the policy gradient from the critic network: $\nabla_{\mathbf{a}} Q(\mathbf{e}, \mathbf{h}, \mathbf{a})|_{\mathbf{a}=\pi(\mathbf{e}_j,\mathbf{h}_j),\mathbf{h}=R(\mathbf{s}_j),\mathbf{e}=\mathbf{e}_j}$;
16: Update the parameters of the actor network using the sampled policy gradients: $\frac{1}{H}\sum_{j=1}^{H} \nabla_{\mathbf{a}} Q(\mathbf{e}, \mathbf{h}, \mathbf{a}) \cdot \nabla_{\boldsymbol{\theta^\pi}} \pi(\mathbf{e}, \mathbf{h})|_{\mathbf{e}=\mathbf{e}_j,\mathbf{h}=R(\mathbf{s}_j)}$;
17: Compute the policy gradient from the actor network: $\nabla_{\mathbf{h}} \pi(\mathbf{e}, \mathbf{h})|_{\mathbf{e}=\mathbf{e}_j,\mathbf{h}=R(\mathbf{s}_j)}$;
18: Update the parameters of the representation network using the sampled policy gradient: $\frac{1}{H}\sum_{j=1}^{H} \nabla_{\mathbf{a}} Q(\mathbf{e}, \mathbf{h}, \mathbf{a}) \cdot \nabla_{\mathbf{h}} \pi(\mathbf{e}, \mathbf{h}) \cdot \nabla_{\boldsymbol{\theta}^R} R(\mathbf{s})|_{\mathbf{s}=\mathbf{s}_j}$;
/**Updating the target networks**/
19: Update the parameters of the corresponding target networks: $\boldsymbol{\theta}^{R'} := \tau\boldsymbol{\theta}^R + (1-\tau)\boldsymbol{\theta}^{R'}$; $\boldsymbol{\theta}^{Q'} := \tau\boldsymbol{\theta}^Q + (1-\tau)\boldsymbol{\theta}^{Q'}$; $\boldsymbol{\theta}^{\pi'} := \tau\boldsymbol{\theta}^\pi + (1-\tau)\boldsymbol{\theta}^{\pi'}$;
20: **end while**

$R(\cdot)$ (line 6), and the action for the target MPTCP flow is derived from the actor network $\pi(\cdot)$ (line 7). Experience relay has also been utilized here to improve learning stability. Transition samples are first stored into a replay buffer $\mathbf{B}$ (line 10), and then randomly sampled to a mini-batch of H samples (line 11) for training the representation, critic and actor networks. As introduced above, the critic network is basically a DQN. Hence, the parameters of the critic network $\boldsymbol{\theta}^Q$ are updated by minimizing the commonly-used squared

error loss (line 14), where the target value y_i is evaluated by applying the Bellman equation (line 13). The parameters of the representation and actor networks $\boldsymbol{\theta}^R$ and $\boldsymbol{\theta}^\pi$ are updated together with the sampled (i.e., an average over the H samples) policy gradients using the chain rule defined in Equation (2.7) (lines 15–18). From this training process, we can see that the proposed neural network model (including the representation, critic and actor networks) is end-to-end trainable.

In our implementation, the representation network is a single-layer LSTM unit. The actor network is a fully-connected feed-forward neural network with 2 hidden layers, which includes 128 neurons in both layers. The Rectified Linear function is used for activation in hidden layers and the hyperbolic tangent function is used for activation in the output layer. The critic network has the same structure as the actor network except the output layer, which has only one linear neuron. In our implementation, the actor and critic networks are trained by the Adam optimizer [63], whose learning rates are set to 0.0001 and 0.001 respectively. The discount factor is set to $\gamma = 0.90$. To simplify the neural network implementation, we leveraged TFLearn [125], which provides a higher-level API to TensorFlow, to construct the above three neural networks.

5.2.3 Implementation of DRL-CC

We implemented the DRL-CC framework on Ubuntu 16.04. We chose to use the MPTCP v0.92 [91], which is a Linux kernel implementation of MPTCP and was built based on the Linux Kernel long-term support release v4.4.x. The available resource of a kernel program is strictly limited: even the floating point calculation is not allowed in the kernel. A DRL agent, however, may need to do lots of complex mathematical calculations (e,g, computing the gradients) for both forward passes and back propagations in the DNN training and inference. Thus, it is impossible to run the DRL agent in the kernel. We implemented the proposed DRL agent as a user-space process using Tensorflow [1]. The DRL agent runs as a daemon process, which is always kept active and waits for the MPTCP flow queries.

Every flow reserves a memory space in the kernel for their subflows, and every subflow can fetch their congestion window size from its memory space. Whenever a MPTCP flow queries, the DRL agent derives an action and deploys it by updating the corresponding congestion window size for each subflow through the MPTCP implementation.

In order to be compatible to current MPTCP implementation, we implemented the proposed DRL-CC agent as a pluggable program following the Linux's specification for congestion control. First, we specified the congestion handler interface *tcp_congestion_ops*, which is a structure of function call pointers. Then we implemented a callback function *mptcp_drl_cong_avoid*, which will be called by each subflow every time an acknowledgment packet is received. Using this function, subflows can keep observing and updating their congestion window sizes.

In addition, before the online-testing, we trained the DRL-CC agent for over $50,000$ epochs (i.e., $50,000$ transition samples) in an offline manner, using iPerf3 [58] to continuously generate packets to keep the network always busy in the test environment, which produced sufficient transition samples for training. Due to different link delays, packet loss rates and bottleneck bandwidth settings, the offline training time varies from an hour to several hours. For example, in the setting of the bandwidth $b_1 = b_2 = 8$Mbps, the delay $d_1 = d_2 = 200$ms and the packet loss rate $p_1 = p_2 = 0.5\%$, it took 2.5 hours to complete the offline training process. Once it was taken online, it immediately became ready for use without any setup latency. Note that offline training only needs to be done once and no additional offline training is needed if the agent is rebooted. As mentioned above, even though we used DNNs for inference in our implementation, each of which, however, has only 2 hidden layers. According to our testing, the online inference time is really short, about 0.5ms, which causes negligible overhead for online decision making. Just as many other RL agents, re-training needs to be performed for DRL-CC when the network environment changes (e.g., from a low-bandwidth and high-delay network to a high-bandwidth and low-delay network). This is because sufficient transition samples need to be collected

to update the DNN of the agent such that it can gain enough experience for the new environment to make good decisions when similar situations occur. However, what is the the best way to re-train a trained agent for a new environment is a fairly big research topic and is out of the scope of this proposal, which will be studied in our future work.

5.3 Performance Evaluation

We conducted a comprehensive empirical study for performance evaluation under various test scenarios. In this section, we describe the settings of our test environment, test scenarios, and then present and analyze the corresponding results.

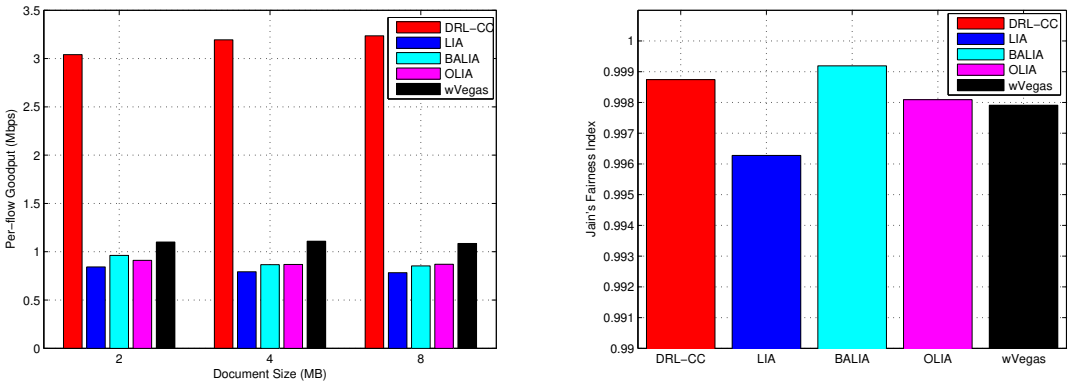
We compared DRL-CC with a few baselines, including LIA [101], BALIA [96], OLIA [64] and wVegas [136], which are all well-known congestion control algorithms proposed particularly for MPTCP. We used their implementation in MPTCP v0.92 [91] for our experiments.

We set up a test environment in our lab for our experiments. The test environment consists of 2 laptops as client and server separately, both running Ubuntu Linux 16.04LTS. Due to the light weight of our design, there is no need for any special device (such as GPU) for training. We found that we could easily run and train the DRL-CC agent on a regular laptop, which has an Intel i7-3630QM CPU and 4GB memory. Two nodes are connected with a Gigabit switch. The server and client have two and one Gigabit Ethernet interfaces respectively, which created two different communication links (i.e., single-link paths) for our testing.

In the test environment, each MPTCP flow includes two subflows, which is the most common setting for MPTCP in practise and has also been used for testing in related works [21, 96]. Similar as in [21], we controlled some key parameters of the communication links in the test environment, such as delay, bandwidth and loss rate using netem [84], which can emulate the communication properties of a wide area network for testing network proto-

cols. We considered a wide range of settings in our experiments: the link delay was set to range from 50ms to 400ms, the packet loss rate was set to range from 0.5% to 4%, and the bottleneck bandwidth varied from 2Mbps to 16Mbps. In our experiments, all the data packets were captured by tcpdump [124] and analyzed by wireshark [133].

We introduce our test scenarios and present the corresponding experimental results. In the first four test scenarios, we evaluated the performance of DRL-CC in a relatively steady environment. Specifically, 5 MPTCP flows (each with 2 subflows) were established between the server and the client and kept active through each experiment. The MPTCP data traffic was generated by retrieving a binary document from a simple HTTP server. The document size ranged from 2MB to 8MB. The goodput was calculated by diving the document size by the elapsed download time. Each number presented in the following figures is the *average goodput per MPTCP flow*.



(a) Average per-flow goodput (b) Jain's fairness index

Fig. 5.3: Scenario 1: $b_1 = b_2 = 8$Mbps, $d_1 = d_2 = 100$ms and $p_1 = p_2 = 2\%$

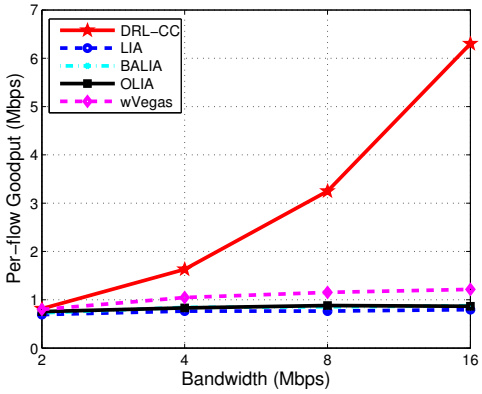
Scenario 1: In this scenario, we show how the document size affects the goodput. We set the bandwidth $b_1 = b_2 = 8$Mbps, the delay $d_1 = d_2 = 100$ms and the packet loss rate $p_1 = p_2 = 2\%$. We used the documents with different sizes: 2M, 4M and 8M. The corresponding results are presented in Fig. 5.3. First, we can see that DRL-CC significantly outperforms all the other methods in terms of goodput. For example, when the document

size is 8M, DRL-CC outperforms LIA, BALIA, OLIA, wVegas by 313%, 279%, 272%, 198% respectively. Moreover, since there are two links (paths) between the server and the client and each of them has a bandwidth of 8Mbps, the total end-to-end bandwidth is 16Mbps. There are 5 MPTCP flows and each of them obtains an average goodput of 3.2Mbps (if DRL-CC is used), which means that DRL-CC makes full use of all available bandwidth. In addition, we show the Jain's fairness index (calculated over all MPTCP flows) given by each algorithm in Fig. 5.3b. We can see that all the algorithms achieve very good fairness since the corresponding indices are all close to 1. Hence, compared to the baselines, DRL-CC leads to much higher goodput without sacrificing fairness. This is mainly due to the way how we define the reward (Section 5.1), particularly the utility function, which usually leads to a good tradeoff between goodput and fairness.

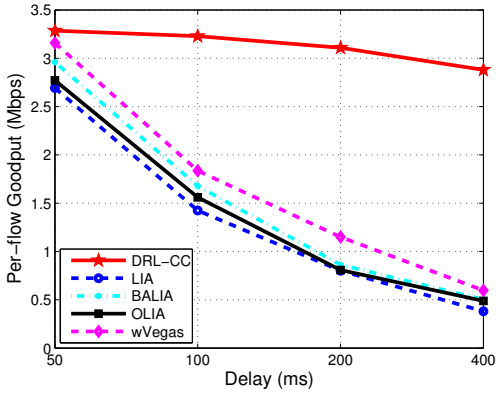
Since all the methods have a similar behavior with different document sizes, and in order to have a relatively long testing time, we used the 8M document in the following scenarios. We present the results corresponding to the next three scenarios in Fig. 5.4. In addition, we found all the algorithms led to similarly good fairness in the other scenarios. Due to space limitation, we omit the corresponding results and figures.

Scenario 2: In this scenario, we fixed the delay $d_1 = d_2 = 100$ms and the packet loss rate $p_1 = p_2 = 2\%$, we aimed to show the performance of all these methods with different bandwidths by setting the bandwidth b_1/b_2 to 2M, 4M, 8M and 16M in different experiments respectively. The corresponding results are presented in Fig. 5.4a. We can see when the bandwidth is small (i.e. 2Mbps), the goodputs of all the methods are fairly low and almost the same. When the bandwidth is increased, DRL-CC leads to sharp improvements on goodput, which are much more significant than those given by the other methods. Particularly, when the bandwith is 8Mbps, DRL-CC leads to $325\%, 280\%, 269\%$ and 181% improvements over the baselines respectively.

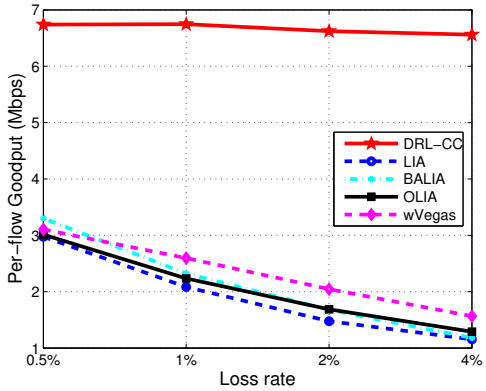
Scenario 3: This scenario was designed to show how the goodputs given by these methods vary with the delay. We set the bandwidth $b_1 = b_2 = 8$Mbps and the packet



(a) Scenario 2: Average per-flow goodput VS. bandwidth with $d_1 = d_2 = 100$ms and $p_1 = p_2 = 2\%$



(b) Scenario 3: Average per-flow goodput VS. delay with $b_1 = b_2 = 8$Mbps and $p_1 = p_2 = 0.5\%$



(c) Scenario 4: Average per-flow goodput VS. loss rate with $b_1 = b_2 = 16$Mbps and $d_1 = d_2 = 50$ms

Fig. 5.4: Performance of all the methods over different settings

loss rate $p_1 = p_2 = 0.5\%$, the delay d_1/d_2 was changed from 20ms all the way to 400ms. The results are shown in Fig. 5.4b. Similar as in the last scenario, when the delay is small (i.e. 50ms), the goodputs of all the methods are fairly high and close. When the delay is increased, the goodputs given by all the methods drop as expected. However, DRL-CC only experiences pretty minor degradation on goodput; while the drops of the other methods are much more substantial. Particularly, when the delay is 400ms, DRL-CC offers $656\%, 473\%, 489\%$ and 382% improvements over the baselines respectively.

Scenario 4: We designed this scenario to see how the packet loss rate affects the good-

puts given by all the methods. We set the bandwidth $b_1 = b_2 = 16$Mbps and the delay $d_1 = d_2 = 50$ms. The packet loss rate was set to 0.5%, 1%, 2% and 4% in different experiments respectively. The corresponding results are shown in Fig. 5.4c. Similar as in Scenario 2, when the packet loss rate is increased, DRL-CC maintains fairly statable performance without a sharp degradation on goodput. However, the goodputs given by all the baselines drop dramatically with the packet loss rate. Particularly, in the case of lossy links with a loss rate of 4%, DRC-CC outperforms these baselines by $468\%, 456\%, 409\%$ and 319% respectively.

In summary, first of all, this set of scenarios and experiments well justify the superiority of DRL-CC on goodput. Particularly, we observe that DRL-CC significantly outperform the baselines in those cases with high bandwidth, long delay and high packet loss rate. Through good training, DRL-CC can find that making better use of available bandwidth leads to much higher goodput. So it always tries to increase congestion window sizes quickly and aggressively when detecting more available bandwidth. However, the other methods behave much more conservatively in this case since they don't have a mechanism that can explicitly and quickly utilize available bandwidth. In addition, DRL-CC is more suitable for tough network environments (e.g., lossy wireless networks) with a high delay or packet loss rate. As mentioned before, most existing methods follow pre-defined policies to control congestion windows, which are usually too conservative, i.e, reducing or significantly reducing window sizes once detecting long RTTs or packet losses but opening congestion windows back up slowly. This certainly leads to low goodput. However, in these cases, DRL-CC usually makes a few attempts and quickly figures out the best ways to set up window sizes without being too conservative or aggressive. Therefore, we can observe DRL-CC brings much more improvements in these tough cases. Last but not the least, as mentioned above, DRL-CC features a joint congestion control over all active MPTCP flows, which is expected to deliver superior performance over those baselines that perform congestion control for flows independently.

Next we introduce two scenarios, in which we tested DRL-CC in a more dynamic and complicated environment. For example, we changed the number of MPTCP flows or even the number of subflows over time. Note that these situations may occur in practice, e.g. a user may open and close a website frequently over time, which leads to establishments and terminations of multiple MPTCP flows. The number of subflows of a MPTCP may also change due to the network state fluctuations, e.g. stepping away from a WiFi hotspot may cause the loss of the corresponding link on a mobile phone. Note that those baselines are not supposed to have any problem dealing with such dynamics since they all manage individual flows separately.

Most settings in Scenarios 5 and 6 are the same as the last few scenarios. We used those key parameters as follows: the bandwidth $b_1 = b_2 = 8$Mbps, the delay $d_1 = d_2 = 100$ms and the packet loss rate $p_1 = p_2 = 2\%$. In the following scenarios, rather than requesting a file from server, we directly used iPerf3 [58] to continuously generate packets to keep the network busy.

Scenario 5: In this scenario, we tested DRL-CC's capability of dealing with the case with dynamic establishments and terminations of MPTCP flows. During a testing period of 150 seconds, establishments of MPTCP flows followed a Poisson process where the lambda was set to 10; and each flow lasted for 30 seconds. The average (over time) total goodputs of all MPTCP flows are shown in Fig. 5.5. We can see that DRL-CC is robust to such a highly-dynamic environment. Compared to the baselines, DRL-CC can still achieve 382%, 351%, 336%, 257% improvements on total goodput.

Scenario 6: As the number of subflows of a MPTCP flow may be changed during the running time, we considered the scenario where one of two subflows suddenly disappeared. Specifically, a total of 5 MPTCP flows were established in the beginning. During a testing period of 200 seconds, one of the subflows of each flow was closed via shutting down a network interface at 60s. The corresponding results are shown in Fig. 5.6. Similar as in the last scenario, DRL-CC can deliver robust performance in this dynamic case. Specifically,

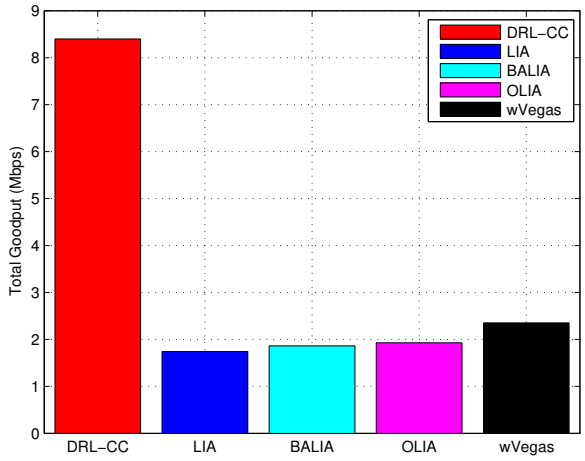


Fig. 5.5: Scenario 5: Average total goodput in the case with dynamic establishments and terminations of MPTCP flows

DRL-CC outperforms those baselines by 193%, 178%, 186%, 204% respectively. In order to show the behavior of flows and the performance of DRL-CC over time, we plot Fig. 5.7. We can see that DRL-CC experiences a sharp drop right at the subflow termination time 60s. However, we observe that its total average goodput stabilizes at 8Mbps (maximum possible after the termination), which shows that DRL can quickly adjust itself to the single network interface setting at 60s, and utilize the rest of available bandwidth.

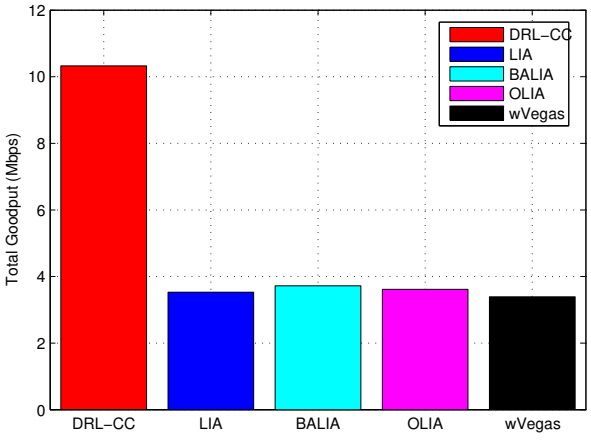


Fig. 5.6: Scenario 6: Average total goodput in the case with dynamic terminations of MPTCP subflows

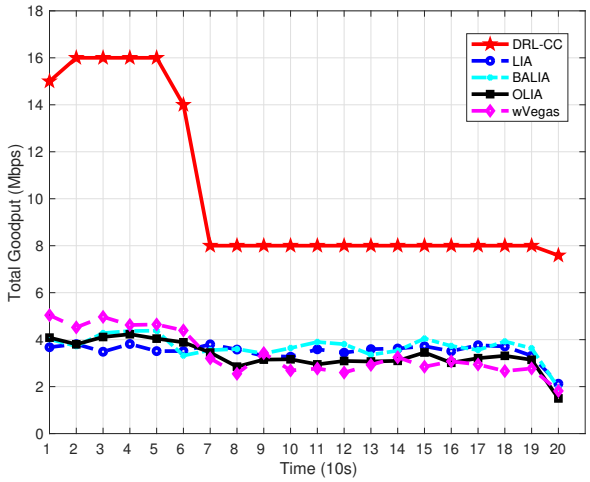


Fig. 5.7: Scenario 6: Average total goodput over time in the case with dynamic terminations of MPTCP subflows

In summary, we conclude that DRL-CC is robust to highly-dynamic network environments. As mentioned above, our design features an LSTM-based network that can learn an effective representation of all active flows. Unlike feed-forward neural networks or CNNs (commonly used in DRL), our model can well handle a variable input size (i.e., the cases with dynamic establishments and terminations of flows and subflows). We actually observed that DRC-CC was able to adjust its control policy quickly and properly whenever there was change during these experiments, which ensures good and stable overall performance. Our results have confirmed the effectiveness and robustness of our design.

Another important property of MPTCP is its friendliness to (regular) TCP flows. If there simultaneously co-exists both MPTCP and TCP flows in a network, MPTCP should not bring goodput improvements for its own flows at the cost of those TCP flows. It is quite common to have both TCP and MPTCP flows in a network since some servers/clients may not support MPTCP.

Scenario 7: We designed this scenario to evaluate the goodputs of all active flows given by all these congestion control methods in a MPTCP and TCP co-existing environment. In this scenario, there were a total of 5 MPTCP flows (that used two different links (path) for communications as described above), and 5 regular single-path TCP flows that competed

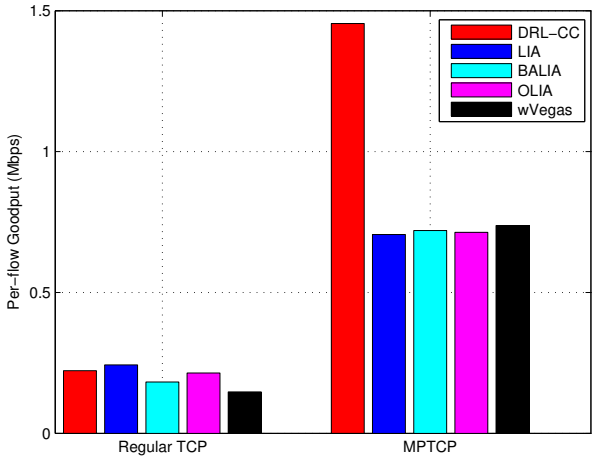


Fig. 5.8: Scenario 7: Per-flow goodputs of regular TCP and MPTCP flows

for one of MPTCP's links. Those key parameters were set as follows: $b_1 = b_2 = 8$Mbps, $d_1 = d_2 = 100$ms and $p_1 = p_2 = 2\%$ We measured the average per-flow goodput for both TCP and MPTCP, and presented the results on Fig. 5.8. We can see that if DRL-CC is used, the goodput of a TCP flow is quite similar as those given by the baselines; however, the corresponding MPTCP flows have a much higher goodput. Specifically, compared to the best baseline LIA, the per-flow TCP goodput corresponding to the use of DRL-CC is slightly lower but the corresponding MPTCP goodput is 106% higher. Moreover, DRL-CC offers higher goodputs for both TCP and MPTCP flows than all the other baselines. This is also mainly due to the way how we define the reward (Section 5.1), particularly the utility function, which takes into account both TCP and MPTCP flows. This observation confirms that DRL-CC is TCP-friendly.

In addition, we conducted an additional experiment in a practical wireless environment, in which a laptop was equipped with two WiFi network interfaces and there existed asymmetric links. Most of other settings are the same as those in the above scenarios, and there were 5 MPTCP flows in total.

Scenario 8: In this scenario, we aimed to demonstrate how DRL-CC performs in a practical wireless environment with asymmetric links. The bandwidth and delay were lim-

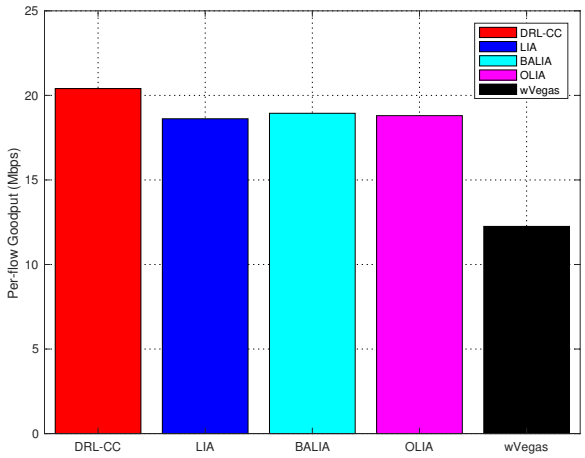


Fig. 5.9: Scenario 8: Average per-flow goodput in the case with asymmetric wireless links

ited to $b_1 = 100$Mbps and $b_2 = 10$Mbps; and $d_1 = 5$ms and $d_2 = 2$ms respectively. We measured the average per-flow goodput for MPTCP flows, and presented the corresponding results in Fig. 5.9.

Since both the delay and the packet loss ratio are fairly low in this scenario, the performance gaps between different methods are relatively smaller compared to other scenarios but they are still noticeable. Specifically, we can see that DRL-CC still outperforms all the other baseline methods in terms of goodput, by 9.6%, 7.7%, 8.51% and 66.5% on average respectively. Thus, we can conclude that DRL-CC is able to make good use of available bandwidth and perform consistently well under different network conditions such as those with asymmetric characteristics.

5.4 Summary

In this chapter, we presented the design, implementation and evaluation of a DRL-based framework, DRL-CC, for congestion control in MPTCP. DRL-CC utilizes a single agent to dynamically and jointly perform congestion control for all active MPTCP flows on an end host with the objective of maximizing the overall utility (such as goodput). DRL-CC

features a novel end-to-end trainable DNN model for action inference, which consists of a flexible LSTM-based representation network, a critic network and an actor network. This neural network architecture can be used to learn an effective representation of all active TCP and MPTCP flows to enable the above joint control, and deal with network dynamics with time-varying flows. We implemented DRL-CC based on the MPTCP implementation in the Linux kernel. We conducted a comprehensive empirical study to evaluate the performance of DRL-CC under seven different scenarios. The experimental results have well justify its effectiveness and superiority over a few well-known MPTCP congestion control algorithms (including LIA, OLIA, BALIA and wVegas) in all of these scenarios in terms of goodput and fairness; its robustness to highly-dynamic environments with time-varying flows; as well as its friendliness to the regular TCP.

CHAPTER 6
EXPERIENCE-DRIVEN CONTROL FOR MOBILE AND EDGE COMPUTING

6.1 Overview

Over the past few years, Deep Neural Networks (DNNs) have become the *de facto* approach for a variety of tasks (such as image classification, face recognition, object detection, action recognition, machine translation, etc) in multiple domains including computer vision and Natural Language Processing (NLP). With the emergence of more and more powerful chipsets and hardware and the rise of Artificial Intelligence of Things (AIoT), there is a growing need for bringing DNN models to empower mobile and edge devices with intelligence such that they can support attractive AI applications, such as flower recognition on a smartphone, voice-activated digital assistant, Driver Monitoring System (DMS) and Advanced Driver Assistance System (ADAS), in a real-time or near real-time manner.

However, deploying large DNN models onto resource-limited devices is quite challenging since most commonly-used DNNs have very complex architectures with a huge number of layers and parameters (e.g. ResNet152 [43] and Faster-RCNN [104]) and thus are known to be computationally intensive [113] and slow (inference with most DNNs cannot be real-

time or near real-time even with a powerful GPU server); while most mobile and edge devices have very limited computing power and resources. When a user runs a DNN model on a mobile device, its CPU undertakes all inference workload by default [126, 99] without the help of GPU or other available hardware, which is obviously not efficient. In a cloud with powerful GPU servers, it is a common practice to simply offload all DNN models to GPUs for high-throughput and low-latency processing. However, computational hardware on a mobile/edge device usually has much less computing power than those GPUs designed particularly for DNNs in a cloud. Moreover, mobile/edge GPU may have to undertake other major tasks (such as graphics); while its cloud counterpart is usually used only for DNNs. Hence, it may not always be wise to distribute all DNN inference workload to GPUs on a mobile/edge device. In addition to latency and throughput, energy efficiency has always been another major concern for mobile and edge computing since most mobile devices and some edge devices are battery-powered with limited energy resources. Moreover, as mentioned above, AI applications with DNNs are computationally intensive thus energy hungry. So energy efficiency should be another design goal, which, however, may further complicate the scheduling problem.

In this chapter, we aim to develop a novel experience-driven control framework for resource co-scheduling, which jointly schedules multiple DNN models over heterogeneous hardware with the objective of achieving low-latency, high-throughput and energy-efficient inference. First of all, we perform a preliminary empirical study to gain some insights about the right direction for DNN scheduling over heterogeneous hardware by running some simple experiments on a Google's Android-based Pixel smartphone. We make several interesting findings from our preliminary study: 1) The current practice utilizing single hardware (a CPU or GPU) for DNN inference is inefficient; and a better way is to make the CPU and GPU work concurrently by co-scheduling tasks on both of them. 2) A straightforward scheduling method with a pre-defined fixed policy, e.g., round-robin, is neither high-throughput friendly nor energy-efficient; hence, designing a low-latency, high-throughput

and energy-efficient co-scheduling algorithm for DNN inference is quite challenging.

Motivated by these findings, we design and implement a novel online Co-Scheduling framework based on deep REinforcement Learning (DRL), which we call COSREL. First of all, COSREL achieves significant speedup over the commonly-used methods by efficiently utilizing all heterogeneous computational hardware. Second, COSREL leverages emerging DRL to make dynamic and wise online scheduling decisions for DNN models based on system runtime state. Third, COSREL is capable of making a good tradeoff among latency, throughput and energy efficiency. Last but not the least, COSREL makes no change to given DNN models, and thus preserves their accuracies. In addition, training a DRL agent for on-device inference is challenging (see Section 6.3.3). We propose a novel device-server co-training algorithm, which makes a device and a server work collaboratively and efficiently to train the DRL agent of COSREL. We summarize our contributions in the following:

1. We conduct a preliminary empirical study for inference with a simple DNN model on an off-the-shelf smartphone, and make several interesting findings, which can serve as a guidance for the design of an efficient co-scheduling algorithm.

2. We present the design and implementation of a novel co-scheduling framework, COSREL, which has several desirable features. Moreover, we propose a novel device-server co-training algorithm to train its DRL agent.

3. We well justify the effectiveness and superiority of COSREL by extensive experiments on off-the-shelf mobile devices with widely-used DNN models.

6.2 Preliminary Study

In this section, through a preliminary study, we discuss the problems of the current practice, which undertakes all inference tasks on single hardware, and the challenges associated with

online co-scheduling on mobile and edge devices with heterogeneous hardware.

In a typical AI application scenario (e.g., image classification), a developer first designs a DNN model for this application, trains the model in servers with training data, and then converts the model to fit into mobile/edge devices using an on-device inference framework (such as Tensorflow Lite [126]). When deploying the DNN model to devices, the current practice is to specify a particular hardware, CPU (by default) or GPU (if available), to execute the DNN inference. However, this kind of single-hardware solution does not take advantage of heterogeneous computational resources. To utilize all available computational resources to support DNN inference, we can design a straightforward scheduling solution, e.g., evenly distributing inference tasks to different hardware in a round-robin manner, which, however, is shown to be inefficient. We conducted some experiments on an off-the-shelf smartphone, Google's Pixel 2 XL [97], which runs Android 10 on Qualcomm Snapdragon 835 CPU and Adreno 540 GPU. We used a simple DNN model for continuous image classification on this device, which mainly consists of few convolutional layers and fully-connected layers. The basic information of this DNN model is summarized in Table 6.1, where #Cov is the number of convolutional layers with four $3 \times 3 \times 32$ and two $3 \times 3 \times 64$ filters, respectively, #MaxP is the number of max pooling layers, and the #FC is the number of fully connected layers with 1,000 neurons each.

#Input	#Cov	#MaxP	#FC	#Output
$244 \times 244 \times 3$	6	3	2	1000

Table 6.1: The basic information of a simple DNN

In our experiments, images kept arriving at a rate of 25 Frames Per Second (FPS), which is a typical sampling rate of a smartphone camera. For a given observation period, our goal is to complete as many inference tasks as possible and in the meanwhile, maintain a low inference latency. Here, each inference *task* refers to the process of computing the output from an input image using a given DNN. At the beginning of each decision epoch, the scheduler computes an assignment for all the following inference tasks that will arrive

within this epoch. To show the performance of different scheduling methods, we used the following three metrics for comparisons: 1) *throughput*: the number of completed tasks during an observation period; 2) *inference latency (or simply latency)*: the elapsed time from the arrival of a task to the end of the inference; and 3) *energy efficiency index*: the average amount of energy consumption per task. We calculated the throughput and the average inference latency of completed tasks. We measured the energy drop of the smartphone battery within a certain period using the Android API *BatteryManage* and computed the corresponding energy efficiency index. The same measurement method has been used in [73]. Similar energy efficiency metrics have also been used in the context of cloud computing [62] and in the context of wireless communications [86, 77].

In our experiments, we evaluated three scheduling methods: running all inference tasks only on the CPU (labeled as CPU), only on the GPU (labeled as GPU), and a straightforward round-robin-based co-scheduling method (labeled as Round-Robin). We set the duration of each decision epoch to 200ms and the duration of an observation period to 10s in our experiments. We ran the experiments for 100 observation periods and present the average values in terms of the three metrics in Figures 6.1, 6.2, and 6.3. We make several interesting findings from these results:

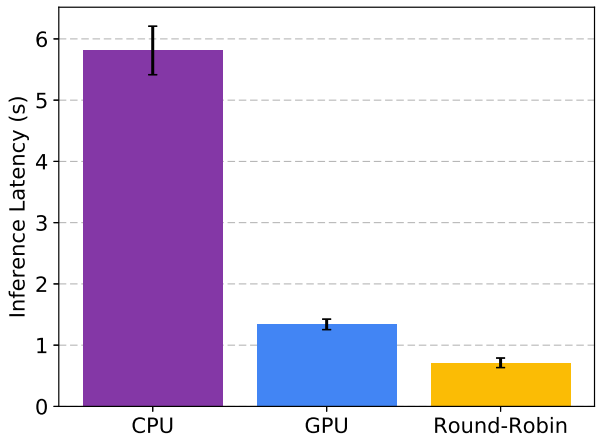


Fig. 6.1: Average inference latency

Finding 1: Co-scheduling DNN tasks over all computational hardware can significantly reduce the inference latency. The current practice usually leverages single hardware, CPU or GPU, for inference, which turns out to be inefficient due to under-utilization of available computational resources. As we can see from Figure 6.1, by fully leveraging all computational resources on the device by co-scheduling tasks, even a simple straightforward round-robin co-scheduling algorithm significantly reduces the average inference latency. Specifically, compared to the CPU-only and GPU-only methods, the round-robin method significantly reduces the average inference latency by 87.7% and 46.9%, respectively. This is because it distributes the workload to both the CPU and GPU, which improves resource utilization and thus reduces latency. Note that the GPU-only solution yields unsatisfactory latency because as mentioned above, mobile GPU has much less computing power than those GPUs designed particularly for DNN inference in a cloud; and moreover, GPU is a batch-processing hardware, which usually leads to high throughput but long latency. This finding leads us to believe that co-scheduling tasks over all computational hardware can significantly reduce the inference latency.

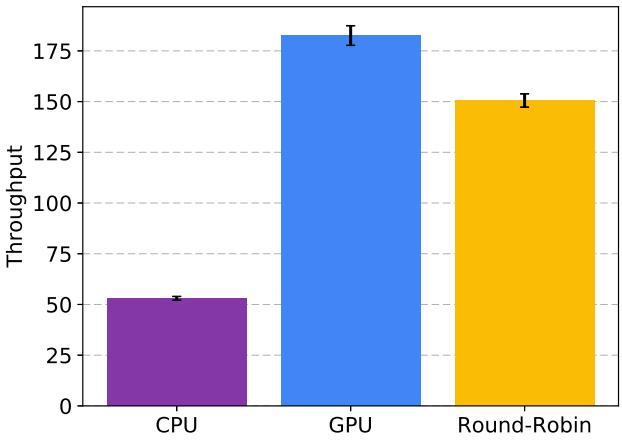


Fig. 6.2: Throughput

Finding 2: A straightforward co-scheduling method does not yield satisfying throughput. Even though it has been demonstrated that task co-scheduling can utilize both the CPU

and GPU to reduce DNN inference latency, a straightforward method, such as Round-Robin, does not yield satisfying throughput, which is shown in Figure 6.2. Specifically, compared to the GPU-only method, the round-robin algorithm produces 17.5% less throughput on average. This is because heterogeneous hardware usually results in different inference times, distributing too much workload to low-speed hardware (e.g., CPU) may hurt the overall performance. Straightforward scheduling methods, such as Round-Robin, usually follow a pre-defined fixed policy and ignore system state at runtime, which likely leads to unsatisfactory performance too. Hence, we can learn that it is necessary to carefully design an intelligent scheduling algorithm, which can fully utilize all computational resources, and make wise online scheduling decisions based on runtime system state with the objective of achieving low-latency and high-throughput inference.

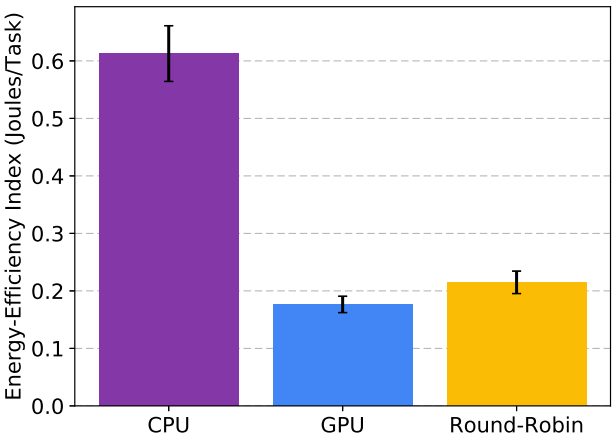


Fig. 6.3: Energy efficiency index

Finding 3: Achieving energy efficiency is non-trivial. As we can see from Figure 6.3, the straightforward co-scheduling method, Round-Robin, cannot well balance the performance and energy consumption, and thus leads to 21.6% more energy consumption per task, compared to the GPU-only method. The CPU-only solution is not energy-efficient either. This is because even though it may lead to a low energy dropping rate, it tends to spend a long time to process DNN inference tasks, which is not efficient in terms of energy

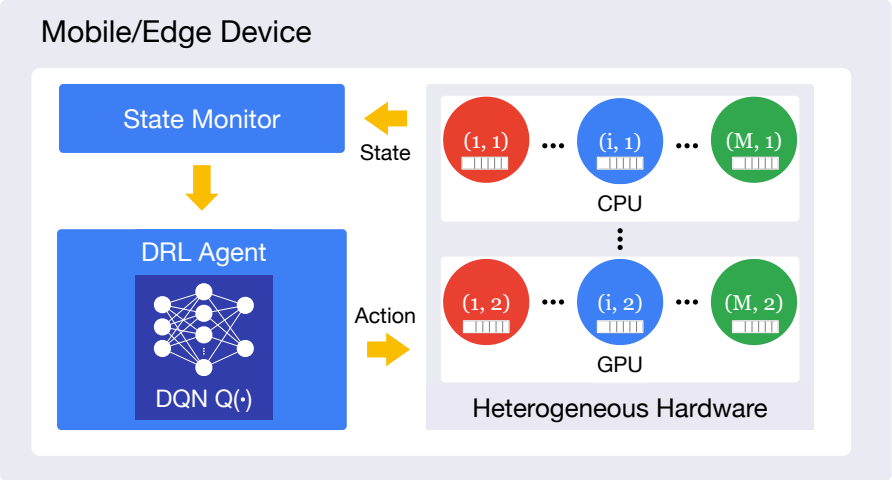


Fig. 6.4: The architecture of COSREL

efficiency index. Therefore, when designing a co-scheduling algorithm for DNN inference, we should carefully and explicitly address energy efficiency in our design.

6.3 Design and Implementation

In this section, we first present an overview of COSREL and then describe the details of its design and implementation.

6.3.1 Overview

Motivated by the findings described above, we propose a novel online co-scheduling framework called COSREL. Figure 6.4 illustrates the architecture of COSREL, which consists of two components:

1. *State Monitor*: It periodically collects runtime state of the system, and reports it to the DRL agent for decision making.

2. *DRL Agent*: It is the core of COSREL, which takes the runtime state as input and applies a DRL-based algorithm to compute a co-scheduling solution.

COSREL works as follows: given a well-trained DRL agent, at each decision epoch, based on the runtime state received from the state monitor, it derives a co-scheduling solution and deploys it to the system. We will discuss how to train the DRL agent in Section 6.3.3. The desirable features of COSREL is summarized in the following:

1. *Full Utilization of Heterogeneous Hardware:* COSREL fully utilizes all computational resources on heterogeneous hardware to support on-device inference of DNN models.

2. *DRL-based Online Co-Scheduling:* Based on DRL, COSREL makes dynamic and wise online co-scheduling decisions with consideration for system state at runtime.

3. *Good Tradeoff among Latency, Throughput and Energy Efficiency:* COSREL can achieve low-latency, high-throughput and energy-efficient DNN inference by setting the reward of its DRL agent properly.

4. *User/Model Transparency:* COSREL is transparent to users and DNN models, i.e., it makes no change to given DNN models and thus preserves their accuracies.

5. *Complementariness to existing DL frameworks and Model Compression Techniques:* As mentioned above, COSREL can work together with any existing DL framework and/or any model compression technique to further accelerate DNN inference.

We summarize major notations in Table 6.2 for quick reference.

6.3.2 DRL-based Co-Scheduling

In this section, we present the design and implementation of the proposed DRL agent. We consider the following co-scheduling problem: given a set of DNN models $\mathcal{M}$ and a set of computational hardware $\mathcal{N}$, the co-scheduling problem seeks a co-scheduling solution, which assigns each DNN model to one of the computational hardware. Formally, the co-scheduling solution is given by a $M \times N$ matrix $\mathbf{a}$, where $M = |\mathcal{M}|$, $N = |\mathcal{N}|$, and

Table 6.2: Major Notations

Notation	Description
$\mathcal{M}$	A set of DNN models
M	The number of DNN models
$\mathcal{N}$	A set of computational hardware
N	The number of computational hardware
$\mathbf{s}_t$, $\mathbf{a}_t$, r_t	The state, action and reward at decision epoch t
x_{ij}	The throughput of DNN model i on hardware j
y_{ij}	The average inference latency of DNN model i on hardware j
$Q(\cdot)$	The deep Q-Network (DQN)
$Q'(\cdot)$, $Q''(\cdot)$	The DQN clone and its target network
$\boldsymbol{\theta}$, $\boldsymbol{\theta}'$, $\boldsymbol{\theta}''$	The sets of weights of $Q(\cdot)$, $Q'(\cdot)$ and $Q''(\cdot)$

each element $a_{ij} = 1$ denotes that DNN model i is assigned to computational hardware j; otherwise, $a_{ij} = 0$. Note that in a real system, each DNN model is loaded to all hardware in advance. At each decision epoch, *assigning a DNN model i to hardware j means that the inference tasks associated with DNN model i are inserted into the queue corresponding to model-hardware pair (i, j) for processing.* At runtime, we run a thread for each model-hardware pair (i, j), which keeps picking inference tasks from the corresponding queue and running them on hardware i. All these threads run concurrently in parallel. In the example illustrated by Figure 6.4, $N = 2$ and hardware 1 and 2 are the CPU and GPU respectively; each circle represents a thread corresponding to a model-hardware pair (i, j), which maintains a queue with tasks (of DNN model i) assigned to hardware j.

In COSREL, we propose to employ a DRL agent to solve the above scheduling problem. At each decision epoch t, the DRL agent observes system state $\mathbf{s}_t$; then based on its current control policy $\pi(\cdot)$ and state $\mathbf{s}_t$, the DRL agent computes a scheduling solution $\mathbf{a}_t$. In COSREL, we apply a Deep Q-Network (DQN) [81] $Q(\cdot)$ to deriving the control policy. Basically, DQN (i.e., $Q(\cdot)$) is a DNN that takes the current system state and an action (i.e., a scheduling solution) as input and outputs a continuous value, which is called Q-value.

The Q-value can be considered as a score which tells how well to take an action $\mathbf{a}$ at state $\mathbf{s}_t$. The control policy π is defined as selecting action $\mathbf{a}_t$ with the highest Q-value at state $\mathbf{s}_t$, i.e., $\pi(\mathbf{s}_t) : \mathbf{a}_t := \text{argmax}_{\mathbf{a}} Q(\mathbf{s}_t, \mathbf{a})$. We present the definitions of state and action for the DRL agent in the following.

State: the state of the DRL agent is basically the runtime statistics, which are collected periodically from the device. In our design, the state consists of the following three features: 1) the length of the task queue of each model-hardware pair; 2) the inference time of each model on every computational hardware; 3) the resource usage (in percentage) of each computational hardware.

Note that as mentioned above, each inference *task* refers to the process of computing the output from input (e.g., an image) using a given DNN. So the length of a task queue is the number of tasks waiting in that queue for being processed by the corresponding hardware. We only keep necessary features from the runtime statistics in our design such that we will not introduce significant overhead to the DRL agent. We have tried different combinations of the available features and found that the above three features are sufficient for capturing the essence of the system state at runtime.

Action: An action of the DRL agent is rather straightforward, which is defined as a co-scheduling solution $\mathbf{a}$, which is a $M \times N$ matrix. Each of its element $\mathbf{a}_{ij} = \{0, 1\}$ specifies if model i is *assigned* to hardware j as described above.

Reward: COSREL is so flexible that it can accommodate different application-specific and/or device-specific needs by setting its reward function properly. There are usually two cases: the first case emphasizes performance (in terms of latency and throughput); and the second case cares about both performance and energy efficiency.

Specifically, for those real-time AI applications (such as object detection, object tracking, etc) or for those edge devices with continuous power supply, the major concern of the design is certainly performance (in terms of latency and throughput). In this case, we need to maximize the system performance by jointly addressing both latency and throughput in

the reward function. In our design, instead of directly maximizing throughout (which likely leads to significant unfairness) or minimizing latency, we choose to follow the widely-used α-fairness model [117, 134] to address both latency and throughput, and the reward is defined accordingly as:

$$r = \sum_{i=1}^{M} \sum_{j=1}^{N} (U(x_{ij}) - \rho \cdot U(y_{ij})), \tag{6.1}$$

where $U(\cdot)$ is a utility function, x_{ij} and y_{ij} are the throughput and the average inference latency of DNN model i on the hardware j during the last decision epoch, respectively; and ρ is a scaling factor used to balance their relative importance and $\rho := 0.1$ in our implementation. According to the α-fairness model, the utility function is $U_\alpha(x) = (\frac{x^{1-\alpha}}{1-\alpha})$. For $\alpha > 0$, $U_\alpha(x)$ increases monotonically with x. α can be used to tradeoff fairness and performance. In our design, we choose to set $\alpha := 1$, then $U(x) = \log x$, which is considered to make a good tradeoff between performance and fairness thus has been widely used for resource allocation.

In addition, for those mobile devices (such as smartphones, pads and wearable devices) that are usually battery-powered, energy consumption is one of major concerns and should play a key role in the reward function. In this case, we can define another reward function, which includes both performance-related factors (i.e., throughput and latency) and energy consumption. Formally, the reward is defined as:

$$r = \sum_{i=1}^{M} \sum_{j=1}^{N} (U(x_{ij}) - \rho \cdot U(y_{ij}) - \sigma \cdot e) \tag{6.2}$$

where e is the energy consumption since the last sampling; and σ is a scaling factor and $\sigma := 0.01$ in our implementation.

Implementation Details: We implemented COSREL on Android 10 [6], which is one of the most popular OSs for mobile and edge devices. To collect the runtime statistics of the system, including necessary information for state and reward, we developed a state

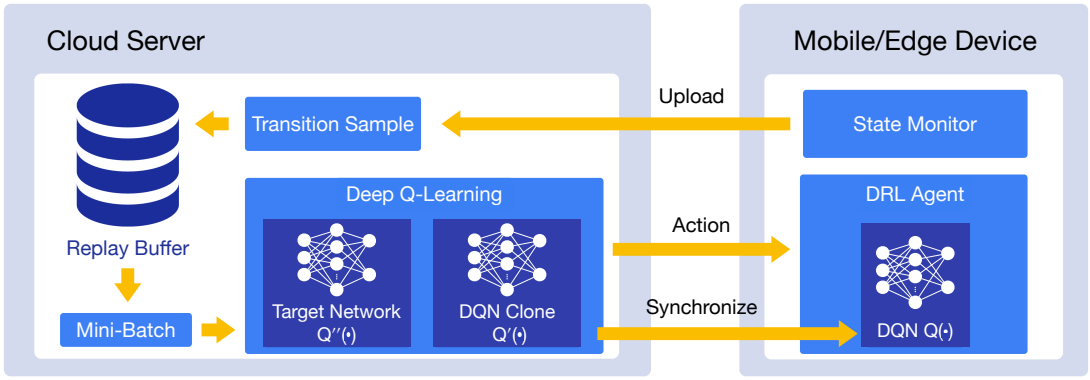


Fig. 6.5: Device-server co-training

monitor, which runs as daemon process in the Android system, periodically broadcasting those information. The DRL agent registers the broadcasting and receives the information from the state monitor. The information will be used as the input to the DQN in the DRL agent for decision making and to form transition samples for training (which will be discussed next). We used Tensorflow Lite [126] for our implementation, which allows us to perform DNN inference using different types of computational hardware on the Android-based device, including CPU and GPU. Tensorflow Lite uses the $Interpreter$ class to warp all the functions needed by DNN inference, including configuring the hardware, loading DNN models, executing operations with input data, and accessing results. It uses a CPU to support inference of DNN models by default. To perform inference on a GPU, we need to select the $GpuDelegate$ option when initializing the $Interpreter$. To fully leverage all the computational resources, we warped each DNN model as a separate thread, which maintains a task queue and multiple $Interpreter$ (each of them is configured for certain hardware). At runtime, DNN inference tasks are submitted to the task queue with input data, and our DRL agent tells which hardware are used to perform the corresponding DNN inference. COSREL keeps running all threads to process the corresponding tasks from their task queues.

6.3.3 Device-Server Co-Training

A common practice for using a DNN model (e.g., a Convolution Neural Network (CNN)) on a mobile/edge device is to train the model on a (or multiple) server with given training data, and then deploy it on the device only for inference. However, this approach does not work for a DRL agent, i.e., it cannot be well trained only on a server in an offline manner, since data (i.e., transition samples) for training the agent need to be collected continuously via interactions with the mobile/edge device. On the other hand, it is also difficult to train a DRL agent only on a mobile/edge device. As we all know, complex mathematical operations (e.g., calculating gradients and backpropagation) are needed to be performed during the training process, which, however, are not supported by most mobile DL frameworks such as TensorFlow Lite and Pytorch Mobile. Moreover, training a DRL agent only on a mobile/edge device may take a long time to converge due to its very limited computational resources.

Hence, we propose to let the server and mobile/edge device to work collaboratively and efficiently to train the DRL agent, which we call *device-server co-training*. During the co-training process, the on-device DRL agent periodically sends transition samples to the server, where the training algorithm performs backpropagations and updates the DQN of the DRL agent.

For the completeness of the presentation, we formally present the device-server co-training algorithm as Algorithm 4. According to this algorithm, we have an on-device DRL agent with its DQN $Q(\cdot)$ on the device; and a clone $Q'(\cdot)$ of the on-device DQN and its corresponding target network $Q''(\cdot)$ on the server. First the algorithm randomly initializes the weights of $Q(\cdot)$, and copies its weights to its clone $Q'(\cdot)$ and the target network $Q''(\cdot)$ (Lines 1 and 2). We choose a random action as the starting point. After executing the scheduling action, the state transits to $\mathbf{s}_{t+1}$, and the agent observes the reward r_t from the system. Then the DRL agent sends transition sample $(\mathbf{s}_t, \mathbf{a}_t, r_t, \mathbf{s}_{t+1})$ to the server for updating the DQN clone $Q'(\cdot)$(on the server).

Algorithm 4: Device-Server Co-Training

Input: The number of training epochs T, the size of a mini-batch K, the exploration ratio ϵ, replay buffer $\mathcal{B}$, DQN $Q(\cdot)$ with weights $\boldsymbol{\theta}$, its clone $Q'(\cdot)$ with weights $\boldsymbol{\theta}'$, and its target network $Q''(\cdot)$ with weights $\boldsymbol{\theta}''$.

1 Randomly initialize the weights $\boldsymbol{\theta}$ of DQN $Q(\cdot)$;
2 $\boldsymbol{\theta}' := \boldsymbol{\theta}$; $\boldsymbol{\theta}'' := \boldsymbol{\theta}$;
3 **foreach** *observation period* **do**
4 Send a random action $\mathbf{a}_1$ to the DRL agent;
5 $t := 1$;
6 **while** *decision epoch* $t < T$ **do**
  /**Execution on the device**/
7 Receive action $\mathbf{a}_t$ from the server;
8 Execute the action and observe the reward r_t;
9 Receive system state $\mathbf{s}_{t+1}$ from the state monitor;
10 Send transition sample $(\mathbf{s}_t, \mathbf{a}_t, r_t, \mathbf{s}_{t+1})$ to the server;
  /**Training on the server**/
11 Receive the transition sample and store it into replay buffer $\mathcal{B}$;
12 Sample K transition samples $(\mathbf{s}_i, \mathbf{a}_i, r_i, \mathbf{s}_{i+1})$ from $\mathcal{B}$;
13 Compute the target value for each transition sample $y_i := r_i + \gamma \text{max}_{\mathbf{a}} Q''(\mathbf{s}_{i+1}, \mathbf{a})$;
14 Update the weights $\boldsymbol{\theta}'$ of the DQN clone with the loss function $\mathcal{L} = \frac{1}{K}\sum_{i=1}^{K}(y_i - Q'(\mathbf{s}_i, \mathbf{a}_i))^2$;
15 Update the weights of target network $\boldsymbol{\theta}'' := \tau\boldsymbol{\theta}' + (1-\tau)\boldsymbol{\theta}''$;
16 Select an action with the ϵ-greedy policy

$$\mathbf{a}_{t+1} := \begin{cases} \text{a random action} & \text{with } \epsilon \text{ probability;} \\ \textit{argmax}_a Q'(\mathbf{s}_{t+1}, \mathbf{a}) & \text{otherwise;} \end{cases}$$

17 Send action $\mathbf{a}_{t+1}$ to the DRL agent;
18 **end**
19 **end**
20 Synchronize DQN $Q(\cdot)$ with its clone $Q'(\cdot)$: $\boldsymbol{\theta} := \boldsymbol{\theta}'$;

On the server, whenever receiving a transition sample from the device, the server stores it into its replay buffer. Then the algorithm samples a mini-batch of transition samples from the replay buffer and then calculates the target value (Line 13). The loss function used for training is defined as the mean square error of the current output of DQN and the target value (Line 14). The DQN clone $Q'(\cdot)$ can then be updated using any training algorithm (such as Stochastic Gradient Descent (SGD)) on the server. Note that the learning objective

here is to find the best policy π that maximizes the cumulated discounted reward over a long decision period. We apply a soft-update to slowly updating the target network $Q''(\cdot)$ with a scaling factor τ (Line 15) and $\tau := 0.01$ in the implementation. The target network and experience replay buffer here are both used to improve learning stability [81, 71]. Similar as in [81], we apply the ϵ-greedy policy for exploration (Line 16). Specifically, we take an random action with ϵ probability, otherwise, we take an action derived by the control policy π. ϵ decays as the training progresses. Note that we choose an action using the ϵ-greedy policy on the server and send it to the device for execution because the server has the most updated DQN $Q'(\cdot)$. To minimize the communication overhead, instead of synchronizing $Q(\cdot)$ with $Q'(\cdot)$ in every step, we do it only once in the end (Line 20). In every step, we only need to send the action generated using the ϵ-greedy policy to the DRL agent on the device for execution. Since the DRL agent does not need the target network to derive an action, it is deployed only on the server. The interactions between the device and the server are only needed during the training process. Once the DRL agent is well trained, it makes scheduling decisions based only on its own DQN $Q(\cdot)$.

Implementation Details: We implemented the server-side training algorithm presented above as a training server using Flask [26] and PyTorch [100] and deploy it on a desktop computer. As mentioned above, the DRL agent and the device-side training algorithm were implemented using TensorFlow Lite. The training server also works as a web server, waiting for HTTP requests from the DRL agent on the device. At the beginning of each decision epoch, DRL agent constructs a POST request, fills in a transition sample in the JSON format, and sends it to the server. Upon receiving the request, the server parses the JSON string and stores the transition sample into its replay buffer.

In our implementation, we chose a two-layer fully-connect-ed neural network (with ReLu as the activation function) to serve as the DQN, which includes 256 and 128 neurons in the first and second layer respectively. We applied the Adam [63] algorithm to updating the weights of DQN, and set the learning rate to 0.005.

6.4 Performance Evaluation

In this section, we first present the common experimental setup and then introduce the three testing scenarios. After that, we present the experimental results and the corresponding analysis.

6.4.1 Experimental Setup

We performed extensive experiments on an off-the-shelf device, Google Pixel 2 XL running Android 10, which has Qualcomm Snapdragon 835 CPU and Adreno 540 GPU [97]. For device-server co-training, the server side program ran on a desktop with Ubuntu 18.04, Intel quad 2.6GHz CPU and 32 GB RAM. Similar as in our preliminary study, we set a decision epoch to 200ms and an observation period to 10s in all the experiments. To evaluate the performance of different scheduling algorithms, we used average inference latency, throughput and energy efficiency index as the metrics, which have all been introduced in Section 6.2.

For a comprehensive evaluation, we compared COSREL with three widely-used baselines:

1. **CPU-only (labeled as "CPU"):** This method only uses CPU for the inference of DNN models.

2. **GPU-only (labeled as "GPU"):** This method only uses GPU for the inference of DNN models.

3. **Round-Robin (labeled as "Round-Robin"):** This method evenly distributes DNN inference tasks to CPU and GPU in a round-robin manner.

4. **Basic COSREL (labeled as "COSREL-P"):** This is COSREL, whose DRL agent was trained using the basic reward function (Equation (6.1)).

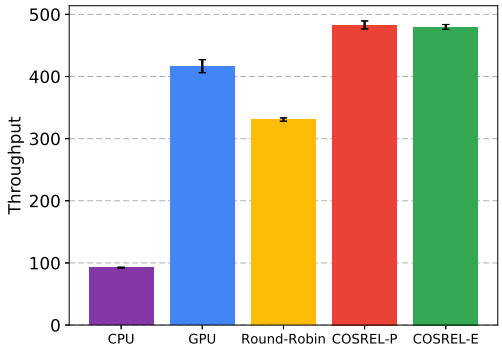
5. **Energy-efficient COSREL (labeled as "COSREL-E"):** This is COSREL, whose DRL agent was trained using the energy-efficient reward function (Equation (6.2)).

6.4.2 Experimental Results and Analysis

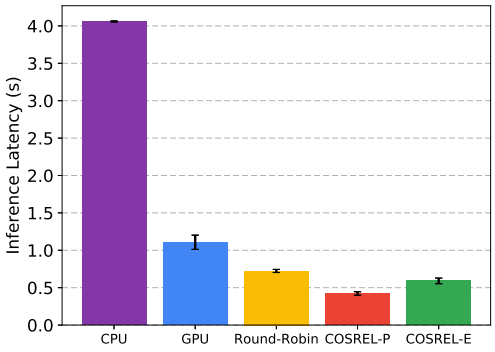
To evaluate the performance of COSREL, we designed three different testing scenarios. In the first scenario, we deployed a single DNN model on the device to evaluate whether COSREL can fully utilize all computational hardware to speed up the DNN inference. In the second scenario, we deployed two different DNN models on the device, aiming to evaluate whether COSREL can effectively and efficiently utilize heterogeneous computational resources to support multiple DNN models. In the third scenario, we had two real applications, facial expression recognition and facial landmarks detection, which consist of three correlated DNN models. We want to evaluate if COSREL can still work well in such a complex environment with multiple correlated DNN models and heterogeneous computational hardware. In all these three scenarios, we trained the DRL agent for $20,000$ decision epochs. When completing the training, we tested COSREL for 100 observation periods. Then we calculated the averages of throughput, average inference latency, energy efficiency index, and their corresponding standard deviations (shown by error bars in Figures 6.6, 6.8, and 6.9).

Scenario 1: In this scenario, we deployed a widely-used DNN model, i.e., MobileNet [55] to the smartphone. We set the arrival rate of images to 53FPS. From the results shown in Figure 6.6, we can make the following observations:

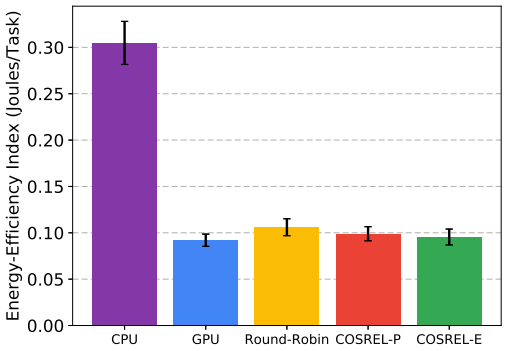
1) It is not surprising to see that the CPU-only method always has the worst performance. This is because CPU is designed for general-purpose computing, but the inference of DNN models needs a lot of floating-point vector/matrix calculations. During the experiment, when the CPU-only method was used, we observed that a long queueing delay dominated the inference latency. Hence, compared to a GPU, it usually takes more time for a CPU to execute the inference of DNN models. Specifically, compared to the CPU-only



(a) Throughput



(b) Average inference latency



(c) Energy efficiency index

Fig. 6.6: Scenario 1: the performance of different methods with a single DNN model

method, the GPU-only method reduces the average inference latency by 72.72% and boosts the throughput by over 3.5x on average.

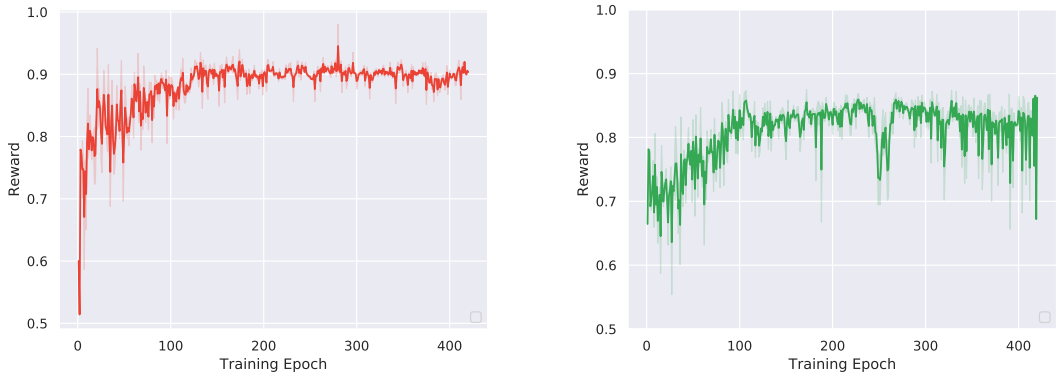
2) Although GPU can accelerate the inference of DNN models, the GPU-only method, however, is not efficient either. When too many inference tasks are assigned to the GPU, the length of its queue increases and then the inference latency rises sharply. By co-scheduling tasks on both the CPU and GPU, we can fully utilize all the computational resources, and avoid abusing the GPU. As we can see from Figure 6.6(b), even a simple straightforward co-scheduling method, Round-Robin, can help reduce the average inference latency. Specifically, the round-robin method reduces the average inference latency by 82.18% and 34.65% respectively on average, compared to the CPU-only method and the GPU-only method.

3) Although heterogeneous hardware can potentially speed up DNN inference, without a carefully-designed co-scheduling algorithm, we may fail to harness the real power of co-scheduling and parallel computing. As we can see from Figure 6.6(a), the round-robin method delivers lower throughput than the GPU-only method. This observation confirms that it is necessary to design an intelligent co-scheduling algorithm, which can fully utilize all computational resources based on the system state at runtime.

4) Although the goal of COSREL-P is not directly set to minimize the average inference latency or to maximize the throughput, COSREL-P still delivers satisfying performance in terms of both metrics. Specifically, compared to the CPU-only and GPU-only, and round-robin methods, COSREL-P significantly improves the throughput by 4.2x, 15.92%, and 46.06% respectively on average; and significantly reduces the average inference latency by 89.61%, 61.89%, and 41.68% respectively. These results well justify the effectiveness and superiority of COSREL.

5) As we can see from Figure 6.6(c), both COSREL-P and COSREL-E perform well in terms of energy efficiency. Even though energy efficiency is not directly addressed in its reward function, COSREL-P can still offer satisfying performance. Specifically, compared to the CPU-only and round-robin methods, COSREL-P leads to 67.52% and 6.63% less energy consumption per task respectively on average. As energy is explicitly addressed in its reward function, COSREL-E achieves better energy efficiency than COSREL-P with almost the same throughput and a slight increase on average inference latency. These results illustrate that our proposed framework is flexible and can be tuned to accommodate different needs. If energy consumption is the major concern (e.g., battery-powered mobile devices), we can use COSREL-E. Otherwise, if the performance has the highest priority, we can choose COSREL-P. Note that in this simple scenario, the GPU-only method offers slightly better energy efficiency than COSREL. However, in the next two scenarios (that are more realistic and complicated), COSREL leads to noticeable improvements over the GPU-only method in terms of energy efficiency index, which can be observed from

Figures 6.8 and 6.9.

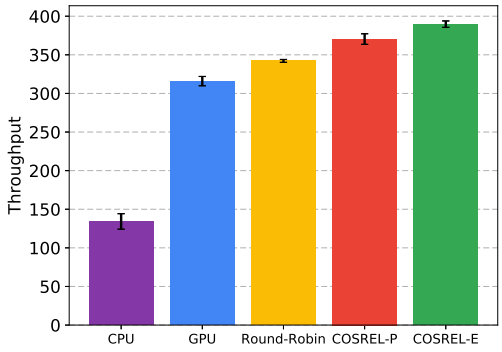


(a) The reward of COSREL-P (b) The reward of COSREL-E

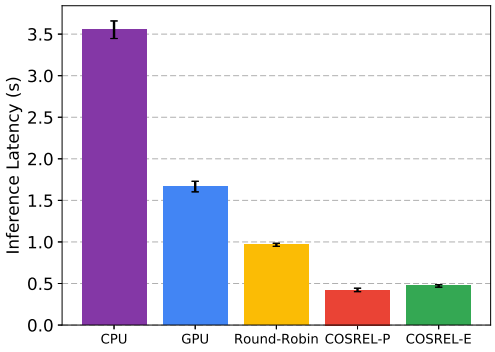
Fig. 6.7: Scenario 1: the reward of COSREL during the device-server co-training

6) We show how the reward of the DRL agent improves during the training in Figure 6.7. Note that we normalized the reward values for better presentation. At the very beginning, the DRL agent has a high probability to randomly explore possible actions instead of taking actions derived by the DQN. Therefore, we observed large fluctuations on the reward. As the training progresses, the DRL agent gradually finds good solutions that lead to larger rewards. Finally, the reward converges to relatively high values, which indicates that the DRL agent is well-trained and ready for online deployment. Note that although the DRL agent stabilizes after a certain period, there are still small fluctuations on the reward. This is because the system is highly dynamic. Specifically, the inference time of each DNN model and the workload on each hardware are highly time-variant, which may affect the decision making of COSREL. However, the fluctuations are quite insignificant, which indicates that COSREL can consistently produce good co-scheduling solutions once it stabilizes.

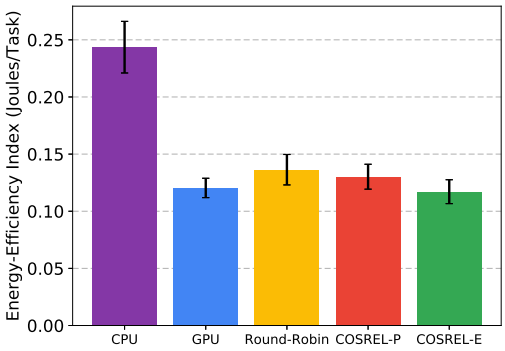
Scenario 2: It is quite common to have multiple DNN models on a device for supporting different DL applications. In this scenario, we deployed two different widely-used DNN models (i.e., MobileNet [55] and SqueezeNet [57]) on the smartphone. Since they ran on the same device, they competed for the computational resources in the hardware.



(a) Throughput



(b) Average inference latency

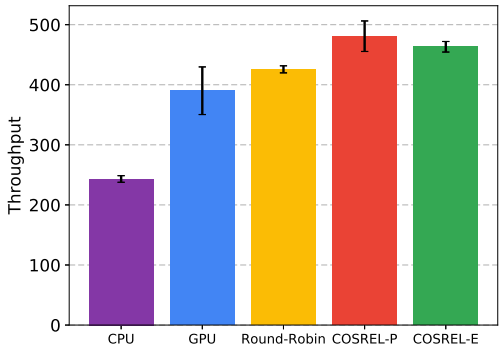


(c) Energy efficiency index

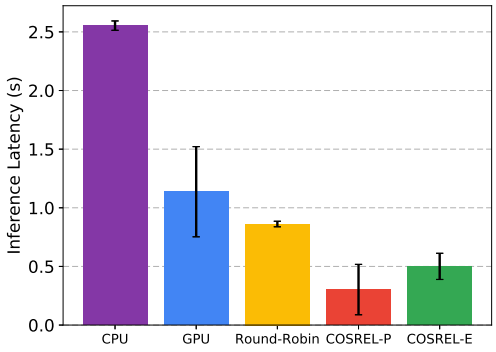
Fig. 6.8: Scenario 2: the performance of different methods with multiple DNN models

We set the arrival rate of images to 25FPS in this scenario. The corresponding results are presented in Figure 6.8. As we can see, in this scenario, both COSREL-P and COSREL-E deliver much better performance than all the baselines in terms of both throughput and latency. Specifically, on average, compared to the CPU-only, GPU-only and round-robin methods, COSREL-P improves the throughput by 1.8x, 17.24% and 8.27% respectively; and it reduces the average inference latency by 88.10%, 74.63% and 56.22%. As for energy consumption, COSREL-E turns out to be the best solution. Specifically, compared to the three baselines and COSREL-P, COSREL-E consumes 51.91%, 2.71%, 14.10%, 10.05% less energy per task respectively on average. These results illustrate that COSREL is capable of handling the co-scheduling problem in the case of multiple DNN models.

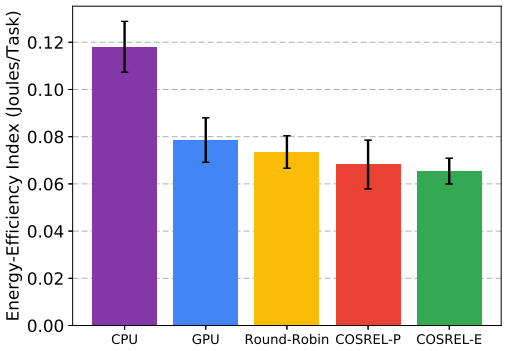
Scenario 3: In this scenario, we tested the performance of COSREL in a more compli-



(a) Throughput



(b) Average inference latency



(c) Energy efficiency index

Fig. 6.9: Scenario 3: the performance of different methods with multiple correlated DNN models

cated case with three correlated models. Specifically, we conducted the experiments with two practical applications: facial expression recognition [122] and facial landmarks detection [121]. In the facial expression recognition application, a face detection DNN model first takes an image as input and detects the human face shown in the image. Then a facial expression recognition model is applied to classifying the detected face into 7 categories of expressions, including angry, disgust, fear, happy, sad, surprise and neutral. In the facial landmarks detection application, a face detection model is first applied to detecting the human face from an input image. Then a facial landmarks detection model is used to detect 5 key points from the detected face, including the positions of left and right eyes, nose, left corner and right corner of the mouth. Obviously, two applications share a common face de-

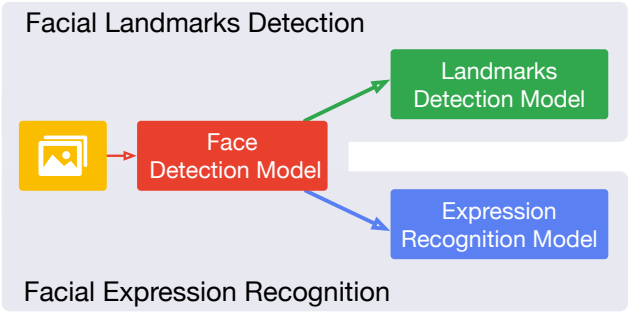


Fig. 6.10: The structure of the two correlated AI applications

tection model. We chose MobileNet-SSD [72]) as the backbone network for face detection to support these two applications. The structure of the models used in this scenario is illustrated in Figure 6.10. Through the model decomposition and layer-wise model sharing, we constructed a processing pipeline, avoiding duplicate inference with the backbone network and thus improving the overall efficiency for inference. We set the arrival rate of images to 25FPS in this scenario. From the results shown in Figure 6.9, we can make the following observations:

1) As expected, COSREL-P offers the best performance in terms of both throughput and latency in this scenario. Specifically, on average, COSREL-P significantly improves the throughput by 97.74%, 23.26%, 12.95% respectively and significantly reduces the average inference latency by 88.14%, 73.38%, 64.87% respectively over the three baselines. These results well justify the superiority of COSREL-P in terms of throughput and latency.

2) Even though COSREL-P does not explicitly consider energy efficiency, it still outperforms all the baseline methods. For example, COSREL-P uses 42.24% less energy per task than the CPU-only method on average. By sacrificing throughput and latency a little bit, COSREL-E further improves the energy efficiency by 4.09% over COSREL-P. Note that COSREL-E still outperforms all the baselines in terms of throughput and latency. Specifically, COSREL-E improves the average throughput by 90.50%, 18.75%, 8.82% respectively and reduces the average inference latency by 80.40%, 55.99%, 41.92% respec-

tively.

These results demonstrate that both COSREL-P and COSREL-E work very well in the complicated case with multiple correlated DNN models.

6.5 Summary

In this chapter, we presented COSREL to fully leverage computational resources on heterogeneous hardware using DRL to effectively and efficiently support concurrent inference of multiple DNN models on a mobile or edge device. COSREL has several desirable features, including full utilization of heterogeneous hardware, DRL-based online co-scheduling, good tradeoff among latency, throughput and energy efficiency, user/model transparency, and complementariness to existing DL frameworks and model compression techniques. We also proposed a novel and efficient device-server co-training algorithm for COSREL. We implemented COSREL on an off-the-shelf Android smartphone, and conducted extensive experiments with various testing scenarios and widely-used DNN models to compare it with three commonly-used baselines. It has been shown by the experimental results that 1) COSREL consistently and significantly outperforms all the baselines in terms of both throughput and latency; and 2) COSREL is generally superior to all the baselines in terms of energy efficiency.

CHAPTER 7
CONCLUSIONS AND FUTURE PLAN

The major goal of this thesis is not to propose yet another protocol for traffic engineering or congestion control but to promote the experience-driven design philosophy for networking and computing. Despite emerging Deep Reinforcement Learning (DRL) seems a promising technique for enabling experience-driven control, there are still many barriers and challenges to its application. While we have discussed and solved some of them in the thesis, many pristine and interesting research directions still lie ahead.

As aforementioned, DRL relies on DNNs to extract features from the state and derive control actions. Therefore, designing a proper DNN structure is critical to the DRL agent. For example, in Chapter 3, since the state space is relatively straightforward, we only apply a common Fully-Connected Neural Network (FCNN) with 2 hidden layers to serve as the actor and critic networks (to make decisions). However, FCNN has a fixed input size, thus it can not deal with dynamic number of flows in MPTCP. Therefore, to handle multiple flows that may come and go at any time, in Chapter 5, we proposed a more complicated LSTM-based representation learning network to extract features. The results well demonstrated its robustness to highly-dynamic environments with time-varying flows. With the development of deep learning technologies, more and more powerful DNNs are presented. For example, *Graph Neural Network (GNN)* [107, 149] has been proposed to directly op-

erate on the graph structure. GNN retains a state that can represent information from its neighborhood with arbitrary depth, which, we believe, is more suitable for networking problems, like TE. It is worth investigating how to integrate these advanced DNNs with DRL to further improve the control performance.

One of the key advantages of experience-driven control is to automatically adapt and optimize for various of system dynamics. In practice, however, the DRL agent usually suffers from performance degradation when coping with unknown (untrained) environments. In Chapter 4, we proposed to leverage transfer learning to rapidly learn a new control policy in a new environment with both old knowledge and new experience, which was shown to be more effective and efficient than training from scratch or straightforwardly fine-tuning. Some recent work [25, 36, 75] proposed *meta reinforcement learning*, which aims to learn a policy that can adapt to a new environment with as little data as possible. More extremely, [30, 49] proposed *zero-shot learning*, which enables learning an adaptive policy even without pre-training in the new environment. Meta learning and zero-shot learning provide good future research directions for the environment adaptation problem.

It is also well known that DRL suffers from sample inefficient, i.e., requiring lots of transition samples for training. Moreover, in some practical systems, deploying an action and observing the state changes are quite slow, thus it may take a long time to interact with the environment and collect enough transition samples. Although in this thesis, we mostly focus on proposing new techniques to improving the training efficiency, e.g., TE-aware exploration and actor-critic-based prioritized experience replay in Chapter 3, we can consider this problem from other new perspectives. For example, *Batch Reinforcement Learning (batch RL)* [5, 29, 66] has been proposed to learn from a fixed dataset without further interactions with the environment. To utilize existing large datasets for networking and computing applications, the ability of batch RL algorithms to learn offline from these datasets has an enormous potential impact in shaping the way we build experience-driven control frameworks for the future.

REFERENCES

[1] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen *et al.*, Tensorflow: Large-scale machine learning on heterogeneous distributed systems, *arXiv preprint*, 2016, arXiv:1603.04467.

[2] Adreno: en.wikipedia.org/wiki/Adreno

[3] ARPANET, *https://en.wikipedia.org/wiki/ARPANET*

[4] S. Agarwal, M. Kodialam, and TV. Lakshman, Traffic engineering in software defined networks, *IEEE INFOCOM'2013*, pp. 2211–2219.

[5] R. Agarwal, D. Schuurmans, and M, Norouzi, An optimistic perspective on offline reinforcement learning, *ICML'2020*, pp. 104–114.

[6] Android 10: android.com/android-10/

[7] R. K. Ando and T. Zhang, A high-performance semi-supervised learning method for text chunking, *ACL'2005*, pp. 1–9.

[8] S. Bai, W. Zhang, G. Xue, J. Tang, and C. Wang, DEAR: Delay-bounded energy-constrained adaptive routing in wireless sensor networks, *IEEE INFOCOM'2012*, pp. 1593–1601.

[9] R. Banner, I. Hubara, E. Hoffer, and D. Soudry, Scalable methods for 8-bit training of neural networks, *NeurIPS'2018*, pp. 5145–5153.

[10] D. Bega, M. Gramaglia, M. Fiore, A. Banchs, and X. Costa-Perez, Deepcog: Optimizing resource provisioning in network slicing with AI-based capacity forecasting, *IEEE Journal on Selected Areas in Communications*, Vol. 38, No. 2, 2019, pp. 361–376.

[11] J. Blitzer, R. McDonald, and F. Pereira, Domain adaptation with structural correspondence learning, *EMNLP'2006*, pp. 120–128.

[12] E. V. Bonilla, K. M. Chai, and C. Williams, Multi-task gaussian process prediction, *NeurIPS'2008*, pp. 153–160.

[13] L. L Brakmo and L. L Peterson, TCP Vegas: End to end congestion avoidance on a global Internet, *IEEE Journal on selected Areas in communications*, Vol. 13, No. 8, 1995, pp. 1465–1480.

[14] J. Q. Candela, M. Sugiyama, A. Schwaighofer, and N. D. Lawrence, Dataset shift in machine learning. *The MIT Press*, 2009.

[15] G. Chen, W. Choi, X. Yu, T. Han, and M. Chandraker, Learning efficient object detection models with knowledge distillation, *NeurIPS'2017*, pp. 742–751.

[16] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, and *et al.*, {TVM}: an automated end-to-end optimizing compiler for deep learning, *arXiv preprint*, 2015, arXiv:1512.01274.

[17] W. M. Czarnecki, R. Pascanu, S. Osindero, S. M. Jayakumar, G. Swirszcz, and M. Jaderberg, Distilling policy distillation, *arXiv preprint*, 2019, arXiv:1902.02186.

[18] W. Dai, G.-R. Xue, Q. Yang, and Y. Yu, Transferring naive bayes classifiers for text classification, *AAAI'2007*, pp. 540–545.

[19] W. Dai, G.-R. Xue, Q. Yang, and Y. Yu, Co-clustering based classification for out-of-domain documents, *ACM SIGKDD'2007*, pp. 210–219.

[20] M. Dong, Q. Li, D. Zarchy, P. B Godfrey, and M. Schapira, PCC: Re-architecting congestion control for consistent high performance, *USENIX NSDI'2015*, 2015.

[21] P. Dong, J. Wang, J. Huang and H. Wang, and G. Min, Performance enhancement of multipath TCP for wireless communications with multiple radio interfaces, *IEEE Transactions on Communications*, Vol. 64, No. 8, 2016, pp. 3456–3466.

[22] E. Einhorn and A. Mitschele-Thiel, RLTE: reinforcement learning for traffic-engineering, *IFIP AIMS'2008*, pp. 120–133.

[23] B. Fang, X. Zeng, and M. Zhang, NestDNN: resource-aware multi-tenant on-device deep learning for continuous mobile vision, *ACM MobiCom'2018*, pp. 115–127.

[24] S. Ferlin, O. Alay, T. Dreibholz, D. A Hayes, D. A. Hayes and M. Welzl, Revisiting congestion control for multipath TCP with shared bottleneck detection, *IEEE INFOCOM'2016*, 2016.

[25] C. Finn, P. Abbeel, and S. Levine, Model-agnostic meta-learning for fast adaptation of deep networks. *ICML'2017*, pp. 1126–1135.

[26] Flask: palletsprojects.com/p/flask/

[27] A. Ford, C. Raiciu, M. Handley and O. Bonaventure, TCP extensions for multipath operation with multiple addresses, *IETF RFC 6824*, 2013.

[28] M. Fortunato, M. G. Azar, B. Piot, J. Menick, I. Osband, A. Graves, V. Mnih, R. Munos, D. Hassabis, O. Pietquin, C. Blundell, and S. Legg, Noisy networks for exploration, *arXiv preprint*, 2017, arXiv:1706.10295.

[29] S. Fujimoto, D. Meger, and D. Precup, Off-policy deep reinforcement learning without exploration, *ICML'2019*, pp. 2052–2062.

[30] S. Genc, and S. Mallya, S. Bodapati, T. Sun, and Y. Tao, Zero-Shot reinforcement learning with deep attention convolutional neural networks, *arXiv preprint*, 2020, arXiv:2001.00605.

[31] P. Georgiev, N. Lane, K. Rachuri, and C. Mascolo, LEO: scheduling sensor inference algorithms across heterogeneous mobile processors and network resources, *ACM MobiCom'2016*, pp. 320–333.

[32] I. Goodfellow, Y. Bengio, and A. Courville, Deep learning, *MIT Press*, 2016.

[33] S. Gu, T. Lillicrap, I. Sutskever, and S. Levine, Continuous deep Q-Learning with model-based acceleration, *ICML'2016*, pp. 2829–2838.

[34] S. Gu, T. Lillicrap, Z. Ghahramani, R. Turner and S. Levine, Q-prop: Sample-efficient policy gradient with an off-policy critic, *arXiv: 1611.02247*, 2016.

[35] P. Gupta and P. R. Kumar, The capacity of wireless network, *IEEE Transactions on Information Theory*, Vol. 46, No. 2, 2000, pp. 388–404.

[36] A. Gupta, R. Mendonca, Y. Liu, P. Abbeel, and S. Levine, Meta-reinforcement learning of structured exploration strategies, *NeurIPS'2018*, pp. 5302–5311.

[37] Gurobi Optimizer, *http://www.gurobi.com/*

[38] F. Gustafsson, Determining the initial states in forward-backward filtering, *IEEE Transactions on Signal Processing*, Vol. 44, No. 4, 1996, pp. 988–992.

[39] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, Soft actor-critic: Offpolicy maximum entropy deep reinforcement learning with a stochastic actor, *arXiv preprint*, 2018, arXiv:1801.01290.

[40] S. Han, J. Pool, J. Tran, and W. Dally, Learning both weights and connections for efficient neural network, *NeurIPS'2015*, pp. 1135–1143.

[41] S. Han, H. Mao, and W. Dally, Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding, *ICLR'2016*.

[42] S. Han, H. Shen, M. Philipose, S. Agarwal, A. Wolman, and A. Krishnamurthy, MCDNN: an execution framework for deep neural networks on resource-constrained devices, *ACM MobiSys'2016*.

[43] K. He, X. Zhang, S. Ren, and J. Sun, Deep residual learning for image recognition, *IEEE CVPR'2016*, pp. 770–778.

[44] Y. He, X. Zhang, and J. Sun, Channel pruning for accelerating very deep neural networks, *IEEE ICCV'2017*, pp. 1389–1397.

[45] Y. He, J. Lin, Z. Liu, H. Wang, L. Li, and S. Han, AMC: AutoML for model compression and acceleration on mobile devices, *ECCV'2018*, pp. 784–800.

[46] M. Hessel, J. Modayil, H. Van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver, Rainbow: Combining improvements in deep reinforcement learning, *AAAI'2018*, pp. 3215–3222.

[47] S. Hassayoun, J. Iyengar, and D. Ros, Dynamic window coupling for multipath congestion control, *IEEE ICNP'2011*, 2011, pp. 341-352.

[48] H. v. Hasselt, A. Guez, and D. Silver, Deep reinforcement learning with double Q-learning, *AAAI'2016*, pp. 2094–2100.

[49] I. Higgins, A. Pal, A. Rusu, L. Matthey, C. Burgess, A. Pritzel, M. Botvinick, C. Blundell, and A. Lerchner, DARLA: improving zero-shot transfer in reinforcement learning, *ICML'2017*, pp. 1480–1490.

[50] G. Hinton, O. Vinyals, and J. Dean, Distilling the knowledge in a neural network, *arXiv preprint*, 2015, arXiv:1503.02531.

[51] S. Hochreiter and J. Schmidhuber, Long Short-Term Memory, *Neural Computation*, Vol. 9, No. 8, 1997, pp. 1735–1780.

[52] J. C. Hoe, Improving the start-up behavior of a congestion control scheme for TCP, *ACM SIGCOMM'1996*, Vol 26, No. 4, 1996, pp. 270–280.

[53] M. Honda, Y. Nishida, L. Eggert, P. Sarolahti, and H. Tokuda, Multipath congestion control for shared bottleneck, *PFLDNeT Workshop*, 2009.

[54] L. Hou and J. Kwok, Loss-aware weight quantization of deep networks, *ICLR'2018*.

[55] A. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, MobileNets: efficient convolutional neural networks for mobile vision applications, *arXiv preprint*, 2017, arXiv:1704.04861

[56] X. Huang, T. Yuan, G. Qiao, and Y. Ren, Deep reinforcement learning for multimedia traffic control in software defined networking, *IEEE Network*, vol. 32, no. 6, 2018, pp. 35–41.

[57] F. Iandola, S. Han, M. Moskewicz, K. Ashraf, W. Dally, and K. Keutzer, SqueezeNet: alexNet-level accuracy with 50x fewer parameters and< 0.5 MB model size, *arXiv preprint*, 2016, arXiv:1602.07360

[58] iPerf3: *https://iperf.fr/*

[59] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. HÃűlzle, S. Stuart and A. Vahdat B4: Experience with a globally-deployed software defined WAN, *ACM SIGCOMM'2013*, pp. 3–14.

[60] J. Jiang and C. Zhai, Instance weighting for domain adaptation in nlp, *ACL'2007*, pp. 264–271.

[61] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, Neurosurgeon: collaborative intelligence between the cloud and mobile edge, *ACM SIGARCH'2017*, Vol. 45, No. 1, 2017, pp. 615–629.

[62] G. Katsaros, J. Subirats, J. Fitó, J. Guitart, P. Gilet, and D. Espling, A service framework for energy-aware monitoring and VM management in clouds, *Future Generation Computer Systems*, Vol. 29, No. 8, 2013, pp. 2077–2091.

[63] D. Kingma and J. Ba, Adam: a method for stochastic optimization, *ICLR'2015*.

[64] R. Khalili, N. Gast, M. Popovic, U. Upadhyay and J. Le Boudec, MPTCP is not Pareto-optimal: performance issues and a possible solution, *ACM CoNEXT'2012*, 2012.

[65] N. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar, DeepX: a software accelerator for low-power deep learning inference on mobile devices, *ACM/IEEE IPSN'2016*, pp. 1–12.

[66] S. Lange, T. Gabel, and M. Riedmiller, Batch reinforcement learning, *Reinforcement Learning, Springer*, 2012, pp. 45–73.

[67] N. D. Lawrence and J. C. Platt, Learning to learn with the informative vector machine, *ICML'2004*, pp. 65.

[68] T. A. Le, C. Hong, M. Razzaque, S. Lee and H. Jung, ecMTCP: an energy-aware congestion control algorithm for multipath TCP, *IEEE communications letters*, Vol. 16, No. 2, 2012, pp, 275–277.

[69] Y. Li, A. Papachristodoulou, M. Chiang, and A. R. Calderbank, Congestion control and its stability in networks with delay sensitive traffic, *Computer Networks*, Vol. 55, No. 1, 2011, pp. 20–32.

[70] J. Li, Z. Zhao, R. Li, and H. Zhang, Ai-based two-stage intrusion detection for software defined iot networks, *IEEE Internet of Things Journal*, Vol. 6, No. 2, 2018, pp. 2093–2102.

[71] 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, *ICLR'2016*.

[72] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C. Fu, and A. Berg, SSD: single shot multibox detector, *ECCV'2016*, pp. 21–37.

[73] L. Liu, S. Isaacman, A. Bhattacharjee, and U. Kremer, POSTER: exploiting approximations for energy/quality tradeoffs in service-based applications, *IEEE International Conference on Parallel Architectures and Compilation Techniques (PACT)*, 2017, pp. 156–157.

[74] S. Liu, Y. Lin, Z. Zhou, K. Nan, H. Liu, and J. Du, On-demand deep model compression for mobile devices: a usage-driven model selection framework, *ACM MobiSys'2018*, pp. 389–400.

[75] H. Liu, R. Socher, and C. Xiong, Taming maml: Efficient unbiased meta-reinforcement learning, *ICML'2019*, pp. 4061–4071.

[76] S. H Low and D. E. Lapsley, Optimization flow control. I. Basic algorithm and convergence, *IEEE/ACM Transactions on networking*, Vol. 518, No. 6, 1999, pp. 861–874.

[77] X. Lu, P. Wang, D. Niyato, D. Kim, and Z. Han, Wireless networks with RF energy harvesting: a contemporary survey, *IEEE Communications Surveys & Tutorials*, Vol. 17, No. 2, 2014, pp. 757–789.

[78] H. Mao, M. Alizadeh, I. Menache, and S. Kandula, Resource management with deep reinforcement learning, *HotNet'2016*, pp. 50–56.

[79] H. Mao, R. Netravali, and M. Alizadeh, Neural adaptive video streaming with pensieve, *ACM SIGCOMM'2017*, pp. 197–210.

[80] A. Medina, A. Lakhina, I. Matta and J. Byers, BRITE: An approach to universal topology generation, *IEEE MASCOTS'2011*, pp 346–353.

[81] 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*, Vol. 518, No. 7540, 2015, pp. 529–533.

[82] V. Mnih, *et al.*, Asynchronous methods for deep reinforcement learning, *ICML'2016*, pp. 1928–1937.

[83] X. Jiang, H. Wang, Y. Chen, and *et al.*, MNN: a universal and efficient inference engine, *arXiv preprint*, 2020, arXiv:2002.12418.

[84] netem: *https://wiki.linuxfoundation.org/networking/netem*

[85] Neural Processor: en.wikichip.org/wiki/neural_processor

[86] D. Ng, E. Lo, and R. Schober, Wireless information and power transfer: energy efficiency optimization in OFDMA systems, *IEEE Transactions on Wireless Communications*, Vol. 12, No. 12, 2013, pp. 6352–6370.

[87] ns-3, *https://www.nsnam.org/*

[88] NSFNET, *https://en.wikipedia.org/wiki/National_Science_Foundation_Network*

[89] NVIDIA-T4: nvidia.com/en-us/data-center/tesla-t4/

[90] Open Shortest Path First (OSPF), *https://en.wikipedia.org/wiki/Open_Shortest_Path_First*

[91] C. Paasch, S. Barre, *et al.*, Multipath TCP in the Linux Kernel, *https://www.multipath-tcp.org*

[92] F. Paganini, Z. Wang, J. C Doyle and S. H Low, Congestion control for high performance, stability, and fairness in general networks, *IEEE/ACM Transactions on Networking (ToN)*, Vol. 13, No. 1, 2005, pp. 43–56.

[93] D. P. Palomar and M. Chiang, A tutorial on decomposition methods for network utility maximization, *IEEE Journal on Selected Areas in Communications*, Vol. 24, No. 8, 2006, pp. 1439–1451.

[94] S. J. Pan and Q. Yang, A survey on transfer learning, *IEEE Transactions on knowledge and data engineering*, vol. 22, no. 10, 2009, pp. 1345–1359.

[95] L. L. Peterson and B. S. Davie, Computer networks: a systems approach, *Elsevier*, 2007.

[96] Q. Peng, A. Walid, J. Hwang and S. H Low, Multipath TCP: analysis, design, and implementation, *IEEE/ACM Transactions on Networking*, Vol. 24, No. 1, 2016, pp. 596–609.

[97] Google Pixel 2 XL: en.wikipedia.org/wiki/Pixel_2

[98] A. Polino, R. Pascanu, and D. Alistarh, Model compression via distillation and quantization, *ICLR'2018*.

[99] PyTorch Mobile: pytorch.org/mobile/

[100] PyTorch: pytorch.org/

[101] C. Raiciu, M. Handley, and D. Wischik, Coupled congestion control for multipath transport protocols, *RFC 6356*, 2011.

[102] C. Raiciu, C. Paasch, S. Barre, A. Ford, M. Honda, F. Duchene, O. Bonaventure, and M. Handley, How hard can it be? designing and implementing a deployable multipath TCP, *USENIX NSDI'2012*, pp. 399–412.

[103] L. Rao, X. Liu, K. Kang, W. Liu, L. Liu and Y. Chen, Optimal joint multi-path routing and sampling rates assignment for real-time wireless sensor networks, *IEEE ICC'2011*, pp. 1–5.

[104] S. Ren, K. He, R. Girshick, and J. Sun, Faster R-CNN: towards real-time object detection with region proposal networks, *NeurIPS'2015*, pp. 91–99.

[105] A. Romero, N. Ballas, S. Kahou, A. Chassang, C. Gatta, and Y. Bengio, Fitnets: hints for thin deep nets, *ICLR'2014*.

[106] A. A. Rusu, S. G. Colmenarejo, C. Gulcehre, G. Desjardins, J. Kirkpatrick, R. Pascanu, V. Mnih, K. Kavukcuoglu, and R. Hadsell, Policy distillation, *arXiv preprint*, 2015, arXiv:1511.06295.

[107] F. Scarselli, M. Gori, A. Tsoi, M. Hagenbuchner, G. Monfardini, The graph neural network model, *IEEE Transactions on Neural Networks*, No. 1, Vol. 20, 2008, pp. 61–80.

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

[109] A. Schwaighofer, V. Tresp, and K. Yu, Learning gaussian process kernels via hierarchical bayes, *NeurIPS'2005*, pp. 1209–1216.

[110] M. Shaio, S. Tan, K. Hwang, and C. Wu, A reinforcement learning approach to congestion control of high-speed multimedia networks, *Cybernetics and Systems: An International Journal*, Vol. 36, No. 2, 2005, pp. 181–202.

[111] R. Z. Shen, Valiant Load-Balancing: building networks that can support all traffic matrices, *Algorithms for Next Generation Networks*, 2010.

[112] X. Shen, J. Gao, W. Wu, K. Lyu, M. Li, W. Zhuang, X. Li, and J. Rao, Ai-assisted network-slicing based next-generation wireless networks, *IEEE Open Journal of Vehicular Technology*, Vol. 1, 2020, pp. 45–66.

[113] S. Shi, Q. Wang, P. Xu, and X. Chu, Benchmarking state-of-the-art deep learning software tools, *IEEE International Conference on Cloud Computing and Big Data*, 2016, pp.99-104.

[114] J. F. Shortle, J. M. Thompson, D. Gross, and C. M. Harris, Fundamentals of queueing theory (5th Edition), *Wiley*, 2018.

[115] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, Deterministic policy gradient algorithms, *ICML'2014*, pp. 387–395.

[116] D. Silver, A. Huang, C. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctoc, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, M. Leach, K. Kavukcuoglu, T. Graepel, and D. Hassabis, Mastering the game of Go with deep neural networks and tree search, *Nature*, Vol. 529, No. 7587, 2016, pp. 484–489.

[117] R. Srikant, The mathematics of Internet congestion control, *Springer Science & Business Media*, 2012.

[118] I. Sutskever, O. Vinyals, and Q. V Le, Sequence to sequence learning with neural networks, *NeurIPS'2014*, pp. 3104–3112.

[119] R. S Sutton, D A McAllester, S. P Singhand, and Y. Mansour, Policy gradient methods for reinforcement learning with function approximation, *NeurIPS'2000*, pp. 1057–1063.

[120] R. S. Sutton and A. G. Barto, Reinforcement learning: an introduction (2nd Edition), *MIT press*, 2018.

[121] Y. Sun, X. Wang, and X. Tang, Deep convolutional network cascade for facial point detection, *IEEE CVPR'2013*, pp. 3476–3483.

[122] Y. Tang, Deep learning using linear support vector machines, *arXiv preprint*, 2013, arXiv:1306.0239.

[123] G. Tesauro, N. K. Jong, R. Das, and M. N. Bennani, A hybrid reinforcement learning approach to autonomic resource allocation, *IEEE International Conference on Autonomic Computing*, 2006, pp. 65–73.

[124] Tcpdump: *https://www.tcpdump.org/*

[125] TFLearn: *http://tflearn.org/*

[126] Tensorflow Lite: *http://tensorflow.org/lite*

[127] P. Thulasiraman, J. Chen and X. Shen, Multipath routing and max-min fair QoS provisioning under interference constraints in wireless multihop networks, *IEEE Transactions on Parallel and Distributed systems*, Vol. 22, No. 5, 2011, pp. 716–728.

[128] D. Vengerov, A reinforcement learning approach to dynamic resource allocation, *Engineering Applications of Artificial Intelligence*, Vol. 20, No. 3, 2007, pp. 383–390.

[129] Z. Wang, T. Schaul, M. Hessel, H. Van, M. Lanctot and N. De Freitas, Dueling network architectures for deep reinforcement learning, *ICML'2016*, pp. 1995–2003.

[130] J. Wang, J. Tang, Z. Xu, Y. Wang, G. Xue, X. Zhang, and D. Yang, Spatiotemporal modeling and prediction in cellular networks: A big data enabled deep learning approach, *IEEE INFOCOM'2017*, pp. 1–9.

[131] K. Weiss, T. M. Khoshgoftaar, and D. Wang, A survey of transfer learning, *Journal of Big data*, Vol. 3, No. 1, 2016, pp. 9.

[132] R. J. Williams, Simple statistical gradient-following algorithms for connectionist reinforcement learning, *Machine learning*, Vol. 8, No. 3–4, 1992, pp. 229–256.

[133] wireshark: available from https://www.wireshark.org/

[134] K. Winstein and H. Balakrishnan, Tcp ex machina: Computer-generated congestion control, *ACM SIGCOMM'2013*, pp. 123–134.

[135] D. Xu, M. Chiang and J. Rexford, Link-state routing with hop-by-hop forwarding can achieve optimal traffic engineering, *IEEE/ACM Transactions on networking*, Vol. 19, No. 6, 2011, pp. 1717–1730.

[136] M. Xu, Y. Cao and E. Dong, Delay-based Congestion Control for Multipath TCP, *IEEE ICNP'2012*, pp. 1–10.

[137] M. Xu, F. Qian, M. Zhu, F. Huang, S. Pushp, and X. Liu, DeepWear: adaptive local offloading for on-wearable deep learning, *IEEE Transactions on Mobile Computing*, Vol. 19, No. 2, 2019, pp. 314–330.

[138] M. Yang, S. Wang, J. Bakita, T. Vu, F. Smith, J. Anderson, and J. Frahm, Re-thinking CNN frameworks for time-sensitive autonomous-driving applications: addressing an industrial challenge, *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2019, pp. 305–317.

[139] S. Yao, Y. Zhao, A. Zhang, L. Su, and T. Abdelzaher, DeepIoT: compressing deep neural network structures for sensing systems with a compressor-critic framework, *ACM SenSys'2017*, pp. 1–14.

[140] S. Zagoruyko and N. Komodakis, Paying more attention to attention: improving the performance of convolutional neural networks via attention transfer, *ICLR'2017*.

[141] Y. Zaki, T. Pötsch, J. Chen, L. Subramanian, & C. Görg, Adaptive congestion control for unpredictable cellular networks, *ACM SIGCOMM'2015*, Vol. 45, No. 4, 2015, pp. 509–522.

[142] A. Zappone, M. Di Renzo, and M. Debbah, Wireless networks design in the era of deep learning: Model-based, AI-based, or both? *IEEE Transactions on Communications*, Vol. 67, No. 10, 2019, pp. 7331–7376.

[143] L. Zeng, E. Li, Z. Zhou, and X. Chen, Boomerang: on-demand cooperative deep neural network inference for edge intelligence on the industrial internet of things, *IEEE Network*, Vol. 33, No. 5, 2019, pp. 96–103.

[144] T. Zhang, S. Ye, K. Zhang, J. Tang, W. Wen, M. Fardad, and Y. Wang, A systematic DNN weight pruning framework using alternating direction method of multipliers, *ECCV'18*, pp. 184–199.

[145] J. Zhao, J. Liu, and H. Wang and C. Xu, Multipath TCP for datacenters: From energy efficiency perspective, *IEEE INFOCOM'2017*, pp. 1–9.

[146] Z. Zhao, Z. Jiang, N. Ling, X. Shuai, and G. Xing, ECRT: an edge computing system for real-time image-based object tracking, *ACM SenSys'2018*, pp. 394–395.

[147] P. Zhou, L. Cheng, and D. O. Wu, Shortest path routing in unknown environments: is the adaptive optimal strategy available? *IEEE SECON'2016*, pp. 1–9.

[148] H. Zhou, S. Bateni, and C. Liu, S^3DNN: supervised streaming and scheduling for GPU-accelerated real-time DNN workloads, *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, 2018, pp. 190–201.

[149] J. Zhou, G. Cui, Z. Zhang, C. Yang, Z. Liu, L. Wang, C. Li, and M. Sun, Graph neural networks: A review of methods and applications, *arXiv preprint*, 2018, arXiv:1812.08434.

Zhiyuan Xu

126 Jamesville Avenue, apt G5, Syracuse, NY, USA

(+1) 315-278-1389 | zxu105@syr.edu | xuzhiyuan1528.github.io

Education

Syracuse University

Ph.D. Candidate, Department of Electrical Engineering and Computer Science — *Aug. 2015 - May 2021 (anticipated)*

Dalhousie University

Exchange Student, Department of Computer Science — *Jan. 2015 - May 2015*

National Taiwan University of Science and Technology

Exchange Student, Department of Computer Science and Information Engineering — *Feb. 2013 - Jun. 2013*

University of Electronic Science and Technology of China

B.S., Department of Computer Engineering — *Sept. 2011 - Jun. 2015*

Work Experience

Syracuse University — *Syracuse, USA*

Research Assistant, *advisor: Prof. Jian Tang* — *Aug. 2015 - Present*

- Proposed a two-step Deep Reinforcement Learning (DRL)-based resource allocation framework in cloud radio access networks
- Proposed Traffic Engineering (TE)-aware exploration and actor-critic-based prioritized experience replay for DRL-based TE
- Proposed an actor-critic-based DRL framework for scheduling with a large discrete action space in distributed computing systems
- Proposed an LSTM-based representation learning network for DRL-based congestion control in MultiPath TCP
- Proposed an actor-critic-based transfer learning framework for experience-driven networking
- Proposed a knowledge transfer based multi-task DRL framework for continuous control tasks
- Proposed a co-scheduling framework for DNN models on mobile and edge devices with heterogeneous hardware
- Implemented the proposed frameworks in ns-3, Linux Kernel, Apache Storm, and Android with Python, C++, and Java

Publications

Total Google Scholar Citations: 900+

[1] **Z. Xu**, K. Wu, Z. Che, J. Tang, and J. Ye
Multi-Task deep reinforcement learning with knowledge transfer for continuous control
*Conference on Neural Information Processing Systems (**NeurIPS**)*, 2020 (AR: 20.1%)

[2] **Z. Xu**, D. Yang, J. Tang, Y. Tang, T. Yuan, Y. Wang, and G. Xue
An actor-critic-based transfer learning framework for experience-driven networking
*IEEE/ACM Transactions on Networking (**TON**)*, Vol. 29, No. 1, pp. 360–371, 2020 (IF: 3.3)

[3] **Z. Xu**, J. Tang, C. Yin, Y. Wang, G. Xue, J. Wang, and M. C. Gursoy
ReCARL: resource allocation in cloud RANs with deep reinforcement learning
*IEEE Transactions on Mobile Computing (**TMC**)*, 2020 (IF: 5.1, Accepted)

[4] **Z. Xu**, J. Tang, C. Yin, Y. Wang, G. Xue
Experience-driven congestion control: when multi-path TCP meets deep reinforcement learning
*IEEE Journal on Selected Areas in Communications (**JSAC**)*, Vol. 37, No. 6, pp. 1325–1336, 2019 (IF: 11.4)

[5] **Z. Xu**, J. Tang, J. Meng, W. Zhang, Y. Wang, C. Liu, and D. Yang
Experience-driven networking: a deep reinforcement learning based approach
*IEEE International Conference on Computer Communications (**INFOCOM**)*, pp. 871–1879, 2018 (AR: 19.2%)

[6] **Z. Xu**, Y. Wang and J. Tang, J. Wang, and M. C. Gursoy
A deep reinforcement learning based framework for power-efficient resource allocation in cloud RANs
*IEEE International Conference on Communications (**ICC**)*, pp. 1–6, 2017

[7] T. Li, **Z. Xu**, J. Tang, and Y. Wang
Model-free control for distributed stream data processing using deep reinforcement learning
*International Conference on Very Large Data Bases (**VLDB**)*, vol. Vol. 11, No. 6, pp. 705–718, 2018 (AR: 21.0%)

[8] N. Liu, X. Ma, **Z. Xu**, Y. Wang, J. Tang, and J. Ye
AutoCompress: an automatic DNN structured pruning framework for ultra-high compression rates
*Association for the Advancement of Artificial Intelligence (**AAAI**)*, 2020 (AR: 20.6%)

[9] C. Yin, J. Tang, **Z. Xu**, Y. Wang
Memory augmented deep recurrent neural network for video question answering
*IEEE Transactions on Neural Networks and Learning Systems (**TNNLS**)*, 31, 9, pp. 3159–3167, 2020 (IF: 8.7)

[10] J. Xu, J. Tang, **Z. Xu**, C. Yin, K. Kwiat, and C. Kamhoua
A deep recurrent neural network based predictive control framework for reliable distributed stream data processing
*International Parallel & Distributed Processing Symposium (**IPDPS**)*, pp. 262–272, 2019 (AR: 27.7%)

[11] N. Liu, Y. Liu, B. Logan, **Z. Xu**, J. Tang, and Y. Wang
Learning the dynamic treatment regimes from medical registry data through deep Q-network
Scientific Reports, Vol. 9, No. 1, p. 1495, 2019 (IF: 3.9)

[12] B. Zhang, C. Liu, J. Tang, **Z. Xu**, J. Ma, and W. Wang
Learning-based energy-efficient data collection by unmanned vehicles in smart cities
*IEEE Transactions on Industrial Informatics (**TII**)*, Vol. 14, No. 4, pp. 1666–1676, 2018 (IF: 9.1)

[13] J. Wang, J. Tang, **Z. Xu**, Y. Wang, G. Xue, X. Zhang, and D. Yang
Spatiotemporal modeling and prediction in cellular networks: a big data enabled deep learning approach
*IEEE International Conference on Computer Communications (**INFOCOM**)*, pp. 1323–1331, 2017 (AR: 20.9%)

[14] N. Liu, Z. Li, J. Xu, **Z. Xu**, S. Lin, Q. Qiu, J. Tang, and Y. Wang
A hierarchical framework of cloud resource allocation and power management using deep reinforcement learning
*International Conference on Distributed Computing Systems (**ICDCS**)*, pp. 372–382, 2017 (AR: 16.9%)

[15] Y. Liu, B. Logan, N. Liu, **Z. Xu**, J. Tang, and Y. Wang
Deep reinforcement learning for dynamic treatment regimes on medical registry data
*IEEE International Conference on Healthcare Informatics (**ICHI**)*, pp. 380–385, 2017

Awards & Honors

2018 **Best-in-Session Presentation Award**, IEEE INFOCOM 2018
2018 **Student Travel Grant Award**, IEEE INFOCOM 2018
2015 **Outstanding Undergraduate Students**, UESTC
2014 **Excellent College Students**, China Computer Federation (CCF)
2014 **The Most Excellent App Award**, China Mobile Innovation and Entrepreneurship Application Contest
2013 **National Encouragement Scholarship**, Nationwide

Skills

Research Interests Deep Learning, Deep Reinforcement Learning, Mobile/Edge Computing, Wireless Communications
Full-stack DL Model Design, Model Training/Testing, Model Compression/Quantization, Model Mobile Deployment
DL/DRL tools Tensorflow, Keras, PyTorch, Tensorflow Lite, NCNN
Platforms Linux Kernel, Embedded System, Android, React
Programming Languages Python, Java, C++, JavaScript, LaTex
Chinese, English

Service

Conference & Journal Review

- IEEE Conference on Computer Vision and Pattern Recognition (**CVPR**), 2020
- AAAI Conference on Artificial Intelligence (**AAAI**), 2020
- ACM Computing Surveys (**CSUR**), 2020
- IEEE/ACM Transactions on Networking (**TON**), 2020
- IEEE Transactions on Neural Networks and Learning Systems (**TNNLS**), 2020
- IEEE Journal on Selected Areas in Communications (**JSAC**), 2019, 2020
- IEEE Transactions on Mobile Computing (**TMC**), 2019, 2020
- IEEE Transactions on Wireless Communications (**TWC**), 2019, 2020
- IEEE Transactions on Network and Service Management (**TNSM**), 2020
- IEEE Transactions on Industrial Electronics (**TIE**), 2020
- IEEE Transactions on Emerging Topics in Computational Intelligence (**TETCI**), 2020
- Journal of Network and Systems Management (**JNSM**), 2020
- Computers and Electrical Engineering, 2020
- Neurocomputing, 2019, 2020
- IEEE Access, 2019, 2020
- IEEE Communications Letter, 2018, 2020
- ACM SIGKDD Conference on Knowledge Discovery and Data Mining (**KDD**), 2019
- IEEE/ACM International Symposium on Quality of Service (**IWQoS**), 2019
- IEEE Transactions on Vehicular Technology (**TVT**), 2019
- Journal of Parallel and Distributed Computing (**JPDC**), 2019