IEEE INFOCOM 2022 - IEEE Conference on Computer Communications | 978-1-6654-5822-1/22/$31.00 ©2022 IEEE | DOI: 10.1109/INFOCOM48880.2022.9796784

Learning Buffer Management Policies for Shared Memory Switches

Mowei Wang*, Sijiang Huang*, Yong Cui*§, Wendong Wang†, Zhenhua Liu‡
*Tsinghua University, China
†Beijing University of Posts and Telecommunications, China
‡Huawei Technologies, China

***Abstract*—Today's network switches often use on-chip shared memory to improve buffer efficiency and absorb bursty traffic. Current buffer management practices usually rely on simple heuristics and have unrealistic assumptions about the traffic pattern, since developing a buffer management policy suited for every scenario is infeasible. We show that modern machine learning techniques can be of essential help to learn efficient policies automatically.**

In this paper, we propose Neural Dynamic Threshold (NDT) that uses deep reinforcement learning (RL) to learn buffer management policies without human instructions except for a high-level objective. To tackle the high complexity and scale of the buffer management problem, we develop two domain-specific techniques upon off-the-shelf deep RL solutions. First, we design a scalable RL model by leveraging the permutation symmetry of the switch ports. Second, we use a two-level control mechanism to achieve efficient training and decision-making. The buffer allocation is directly controlled by a low-level heuristic during the decision interval, while the RL agent only decides the high-level control factor according to the traffic density. Testbed and simulation experiments demonstrate that NDT generalizes well and outperforms hand-tuned heuristic policies even on workloads for which it was not explicitly trained.

I. Introduction

In the modern Internet, network devices, such as switches and routers, rely on a packet buffer to absorb burst traffic and avoid packet loss. Insufficient buffer results in reduced throughput [1], thereby impairing the network quality of service [2]. Most switches today use the on-chip shared buffering pool to increase the buffer utilization [3]–[5], instead of the private memory that is exclusive for a single port. The shared buffer is dynamically allocated to different ports according to specific buffer management policies [4].

In the last several decades, many buffer management policies [6]–[15] have been proposed to regulate buffer allocation. Among them, the Dynamic Threshold (DT) policy [7] is widely used by manufacturers (e.g., Broadcom [4]), which restricts the queue length via a dynamic threshold that is proportional to the current unoccupied buffer space. Several variants of DT are also designed for priority queues [10], [13], different traffic patterns [6], [14], [15], and with more decision information [8]. These policies often aim to maximize link utilization, minimize packet loss and buffer utilization, while maintain fairness across different ports.

§Yong Cui (cuiyong@tsinghua.edu.cn) is the corresponding author.

Although these policies have achieved great success, they often rely on heuristics that prioritize generality and easy implementation over the pursuit of ideal performance on different traffic workloads. The traffic could arrive at the switch with time-varying spatial patterns (e.g., combinations of incast with different degrees). Previous theoretical analysis [9] shows that the optimal buffer policy depends on the incoming traffic rate of *all* ports, and the corresponding buffer allocation for each port should be different to match its traffic status. However, designing such a policy to obtain the potential benefit is not easy. Massive information (e.g., detailed stats of each port) and control choices (e.g., different buffer allocation schemes for each port) can increase decision complexity and introduce various corner cases. Hence, existing policies turn to simple heuristics, forfeiting the ability to generalize in a wide range of traffic patterns, even with painfully fine-tuned parameters.

We advocate leveraging modern machine learning techniques to help sidestep this problem. Recently learning-based schemes have shown great promise in the networking domain [16]–[23]. Switch vendors also develop new switches equipped with AI-chips [24], which makes it possible to apply neural networks to switch and unleashes the potential of machine learning to the switch design field.

In this paper, we propose Neural Dynamic Threshold, a learning-based buffer management solution for shared memory switches. NDT is built on modern deep reinforcement learning techniques that can automatically learn highly-efficient policies from experience given only a high-level objective, e.g., minimizing average flow completion time (FCT). NDT expresses its management policy as a deep neural network (NN) since it can extract features from the high-dimensional input information without handcrafted engineering. The management policy is trained by interacting with simulated environments, where it allocates the buffer, observes the reward signal, and gradually improves its control policy. However, it is non-trivial to directly apply off-the-shelf RL algorithms to the buffer management problem. To make it feasible, we develop a scalable neural network model, a decision trigger mechanism, and an action encoding scheme with domain knowledge.

First, compared to traditional RL applications (e.g., video games [25], [26]), buffer management problem has a larger problem size in both the state space (i.e., the number of potential traffic conditions that the buffer management policy faces) and the action space (i.e., the number of possible choice combinations). This is because buffer management policy must scale to tens to hundreds of ports and decide among potentially thousands of configurations per port (e.g., different sizes of the allowed buffer occupancy).

To solve this problem, we designed a scalable neural network architecture that leverages a permutation-equivariant neural network [27]–[29] to process port state information and make buffer management decisions. Our neural network reuses several small building blocks for each port, thus it can scale to an arbitrary number of ports while maintaining the model size unchanged. This neural network design substantially reduces the complexity of the model and the requirement of the training data, which is the key to efficient learning.

Second, the learned buffer management policy should make decisions with high efficiency. The conventional decision trigger mechanism can not directly extend to learning-based solutions due to hardware constraints. Existing policies usually work at the packet level, i.e., every time a packet enqueues or dequeues, which leaves only hundreds of nanoseconds for decision making. This inference time budget is too tight for learning-based solutions deployed on the current, even the most powerful, AI inference chips [30](§IV-B).

To solve this problem, we design a novel trigger mechanism that invokes decision-making according to the traffic density, which greatly enlarges the decision interval. However, the naive action encoding scheme (i.e., directly deciding the buffer size for each port) is not compatible with this trigger mechanism since this fixed buffer decision can not handle the traffic dynamics during the decision interval. To tackle this problem, we leverage existing domain knowledge and encode the control factor of existing heuristics as the action. Specifically, the buffer allocation is directly controlled by the selected heuristic during the decision interval while NDT only makes decisions for the control factor when triggered. This two-level control mechanism not only enlarges the effective time of the action but also greatly narrows the exploration space of NDT, enabling fast training since NDT can learn from a high starting point.

We evaluate NDT in both simulation and a DPDK-based switch prototype. The simulation results demonstrate that NDT outperforms all the existing policies under realistic traffic workloads. Especially, NDT outperforms DT over 90% of the time and reduces average FCT by up to 28%. Testbed experiments show that NDT trained in a simulated environment can generalize well to the real-world switch prototype and the traffic workload unseen during training. In summary, we make the following key contributions:

- First RL-based buffer management framework for shared memory switches that can learn directly from the performance metrics (§III).
- A scalable neural network architecture that can process switch features of an arbitrary number of ports, leveraging the permutation symmetry of the switch ports (§IV-A).
- A set of mechanisms to achieve efficient decision making including an action encoding scheme with domain knowledge and a novel decision trigger mechanism (§IV-B).
- A prototype implementation and evaluation of NDT in both simulation and a DPDK-based testbed (§V-§VI).

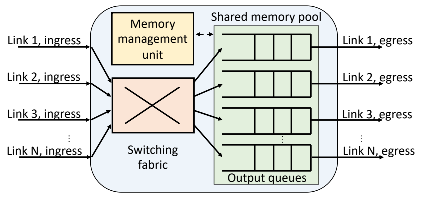


Figure 1. The architecture of the shared memory switch.

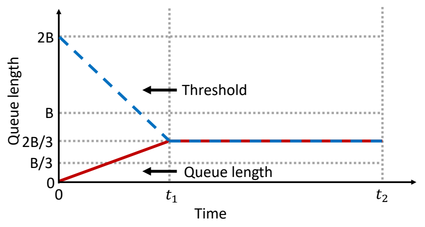


Figure 2. An example of queue length and buffer threshold evolution under DT policy.

II. Background

Shared memory switch. As shown in Figure 1, the shared memory switch is often modeled as an output-queued switch where packets arriving from input ports immediately cross the switching fabric and enter a queue at the output port. The queue will gradually build up when multiple packets from different input ports are destined to the same output port. Unlike traditional switches using private memory for each port, the shared memory switch uses the on-chip shared memory, which could be shared across multiple output queues. The concrete buffer allocation is controlled by buffer management policies implemented in the memory management unit.

Buffer management policies. Many buffer management policies [6]–[15] have been proposed to optimize throughput, buffer utilization, and fairness. However, it is not easy to make trade-offs between efficiency and fairness. To achieve efficiency, the buffer should be used whenever possible to reduce packet loss. Nevertheless, this may raise the fairness problem that a small number of ports occupy most of the shared buffer, leaving other ports starving.

These solutions can be generally divided into two categories: preemptive policies and non-preemptive policies. For preemptive policies [9], [12], packets that are already in the memory can be overwritten by a newly arrived packet if the buffer is full. The preemptive policies are proved to be optimal for some scenarios [9], [12] whereas it is difficult to implement [6], [7]. Thus in the rest of this paper, we only consider the non-preemptive setting. For non-preemptive policies [6]–[8], [10], [11], [13], the arrived packet can only be dropped before entering the queue, which means packets in memory will never be dropped. Among the non-preemptive policies, the Dynamic Threshold [7] policy has been widely used by switch vendors [3], [4].

Dynamic threshold policies. DT is a threshold-based policy, in which the queue lengths of all ports are constrained by

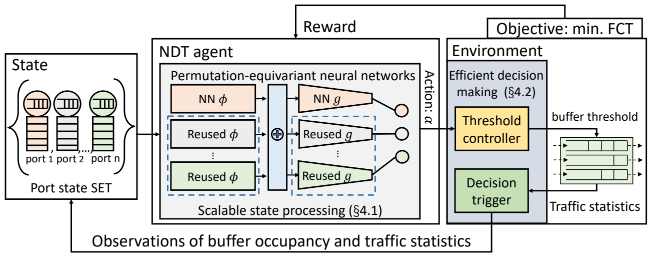


Figure 3. In NDT's RL framework, a buffer management agent observes the port state to decide a buffer allocation action on the switch environment and receives a reward based on a high-level objective. The agent uses a permutation-equivariant neural network to extract features from the port state set and outputs action for each port.

a threshold, which is proportional to the current amount of unused buffer space. Specifically, let $q_i(t)$ be the queue length (or buffer occupancy) of the port i and B be the total shared buffer size, then the threshold $T(t)$ is given by:

$$T(t) = \alpha \cdot (B - \sum_i q_i(t)) \tag{1}$$

where α is a fixed control factor shared by all ports. A packet arriving in port i at time t will get dropped if $q_i(t) > T(t)$.

To better understand the mechanism of DT, Figure 2 shows an example time series of queue length and buffer threshold evolution [6]. Assume that the switch buffer is empty at time $t = 0$, then $\sum_i q_i(t) = q_k(t)$. Here let $\alpha = 2$, then $T(t) = 2 \cdot (B - q_k(t))$. At time $t = 0$, $q_k(0) = 0$ and $T(0) = 2B$, thus $q_k(0) < T(0)$. Packets are allowed to enter into the buffer, and $q_k(t)$ will increase until $q_k(t) = T(t) = 2B/3$. Once $T = q_k$, the port is not allowed to occupy additional buffer and the queue length will not increase any longer. DT trades off the efficiency for fairness by always reserving a fraction of buffer to avoid starving newly overloaded ports at the cost of potential packets dropping from burst traffic even when there is free buffer space. This policy fits well to the uniform traffic pattern but can not generalize to skew and bursty traffic.

Based on the classic DT scheme, in recent years, researchers propose several variants to cope with specific traffic patterns. Enhanced Dynamic Threshold (EDT) policy [6], as a typical one, is designed for bursty traffic and allows to temporarily relax the threshold of the port identified with micro-burst traffic. However, the method to recognize the micro-burst traffic also relies on heuristic, which has many tunable parameters requiring expert knowledge and significant effort to devise. These solutions rely on heuristics that use the fixed control rule or have explicit assumptions of the traffic load, thus can not generalize to a broad range of traffic patterns.

III. Overview

NDT provides a new option for buffer management by using data-driven techniques and automatically learns traffic-specific buffer management policies. NDT does not require human guidance and explicit modeling of system or traffic patterns, except a high-level objective (e.g., minimizing the average FCT). Specifically, NDT is built based on deep reinforcement learning. Figure 3 shows the high-level framework of NDT.

Table I
Notations used throughout this paper.

Entity	Symbol	Entity	Symbol
total buffer size	B	port	i
link line-rate	C	port state vector	x_i
total number of ports	N	port embedding	e_i
port queue length	q_i	device embedding	$\bar{e}$
port threshold	T_i	port summary embedding	z_i
port control factor	α_i	non-linear functions	ϕ, f, g, w
port packets enqueue	v^i_{eq}	port packets dropping	v^i_{dp}
port packets dequeue	v^i_{dq}	port packets excess	v^i_{ex}

NDT represents its buffer management policy as an agent that uses a neural network to make decisions. The NDT agent is trained using RL algorithms by interacting with the offline (simulated) environment. When step k is triggered at time step t_k, the agent observes the port state information s_k, chooses a buffer allocation action a_k. After executing the action, the state of the environment transitions to s_{k+1} and the agent receives the reward r_k. The reward is set based on NDT's high-level objective and can be used as a signal to improve the policy. In the following, we will detail our key design ideas. Table I defines our notations.

State as a set. In NDT, the state $s = \{x_1, ..., x_N\}$ is a set, where each element is a per-port state vector x_i of port i and N is the total port number. Each port state consists of five attributes: *(i)* the port buffer occupancy, *(ii)* bytes of packet enqueue since last decision, *(iii)* bytes of packet dropping since last decision, *(iv)* the last action, and *(v)* the time interval since last decision. We choose these features in an attempt to include all the information necessary for decision-making from three perspectives: port buffer status, traffic statistics, and previous decisions. NDT can easily support additional state information.

Policy as a scalable neural network. To achieve scalability, NDT leverages the permutation symmetry of the switch ports underlying the buffer management problem. Vanilla neural network designs can have a large scale and must be trained multiple times to learn identical spatial traffic patterns, resulting in a substantial increase in the amount of training data and training time. NDT uses a permutation-equivariant neural network [27]–[29] as its policy, and that can scale to an arbitrary number of ports since it reuses several building blocks (e.g., small NN) for port state processing and decision making. The permutation-equivariant NN can extract features in a permutation-invariant way with the help of certain permutation-invariant operations (e.g., sum or max) while keeping the order information for the corresponding decision making. The details will be introduced in §IV-A.

Action with efficient decision making. The design space of NDT's action encoding is limited by the capability of the supporting hardware. To achieve efficient decision making, NDT encodes its action with domain knowledge, and uses a novel decision trigger mechanism that can enlarge the decision interval and reduce the decision frequency. Specifically, NDT uses a two-level control mechanism, where the agent only decides the control factor α when triggered while the actual buffer allocation is directly controlled by the dynamic

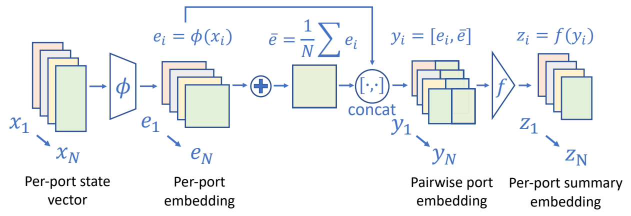


Figure 4. The architecture of permutation-equivariant neural networks. Each color corresponds to a port.

threshold policy during the decision interval. The design detail of NDT's action encoding scheme will be elaborated in §IV-B.

Reward enabling end-to-end objectives. With the expressive reward function, NDT natively supports optimizing the end-to-end performance metric. For example, if the objective is to minimize the average FCT, NDT sets the reward as $r_k = -(t_k - t_{k-1})F_k$ for the action a_k, where F_k is the number of active flows during the decision interval $[t_{k-1}, t_k)$. The goal of the RL algorithm is to maximize the expected cumulative reward $\mathbb{E}[-\sum_{k=1}^{T}(t_k - t_{k-1})F_k]$, where T is the total number of actions in a training episode. However, a flow could come and leave during a decision interval, whereupon in practice reward r_k is computed by summing the active time of all active flows during this interval. This objective minimizes the active number of flows throughout the entire process, and consequently, it can effectively minimize the average FCT. Note that the information of flow active time is only needed during the offline training process. After training, the NDT agent works alone without the guidance of the reward.

NDT also supports other high-level objectives by changing the concrete reward function accordingly. For example, if the objective is to minimize the total packet loss, the reward function can be expressed as $r_k = -\sum_i l_i$, where l_i is the number of packet loss of port i during the decision interval.

RL algorithm on continuous action space. In NDT, we model the action in continuous action space and choose Soft Actor-Critic (SAC) [31], [32] as our learning algorithm. SAC is a state-of-the-art RL algorithm and has successfully applied to real-world learning problems (e.g., robot control [32]), proven to be sample efficient and robust. Please refer to [32] for the details.

IV. Design

This section describes the design details of NDT, including the scalable state processing (§IV-A) and the efficient decision making (§IV-B).

A. Scalable state processing

When invoked, the NDT agent takes state information of all ports as input and decides buffer allocation among all ports. To build the agent, a naive solution is to build a large neural network that takes a flat feature vector containing all the port state information as input. However, this approach can not scale to process information of an arbitrary number of ports since the input size of the neural network is fixed. The resulted complexity of the neural network will increase with the number of ports, rendering the NN difficult to train.

NDT tries to solve this problem based on a crucial observation that the switch ports are *permutation symmetric*. Specifically, we find that the control action of each port only depends on the port state information itself and is not related to the order of the port. For example, considering a many-to-one traffic incast scenario, only the degree of the incast matters while it is inconsequential whether the "one" port is port i or port j. Considering this permutation symmetry of switch ports, all port state information can be treated as a whole as an unordered *set*. Then the state space can be dramatically reduced since different permutations of the same combination of port traffic status can be treated equivalently.

Invariant model. To leverage this property, one choice is to adopt the permutation-invariant neural network architecture (e.g., Deep sets [27], [29]), which is designed for sets. It uses some operations or functions g that are permutation invariant to the order of input elements in the set $\mathbf{x} = \{x_1, x_2, ..., x_M\}$, i.e., for any permutation π :

$$g(\{x_1, ..., x_M\}) = g(\{x_{\pi(1)}, ..., x_{\pi(M)}\}) \tag{2}$$

These operations include "mean", "max", "min", etc. Although it sounds promising, this approach can not be directly applied to NDT since the order information has been discarded so that the output (i.e., the action) can not be mapped to a corresponding input element (i.e., the port state).

Equivariant model. All things considered, NDT expresses its policy as permutation-equivariant neural networks [27], [28], a variant of permutation-invariant neural networks. Its goal is to learn a function g that **permutes the corresponding outputs when the input elements are permuted**, i.e., for any permutation π :

$$g([x_{\pi(1)}, ..., x_{\pi(M)}]) = [g_{\pi(1)}(\mathbf{x}), ..., g_{\pi(M)}(\mathbf{x})] \tag{3}$$

It embeds the port state information in embedding vectors, keeping the order information, and outputs the actions for each port. The architecture of our permutation-equivariant neural network is shown in Figure 4.

The embedding takes the set of the port state as input, and outputs three types of embeddings step by step: 1) port embeddings, which capture information from the heterogeneous attributes of per-port state vectors; 2) device embeddings, which capture the information of all ports, i.e., the switch device; and 3) port summary embeddings, which aggregate information from the port and device embeddings. Note that the stored information in these embeddings is automatically learned from the end-to-end training without manual feature engineering. NDT's permutation-equivariant neural network is scalable because it reuses several building blocks to get the above embeddings. These building blocks are implemented as small NNs that operate on low-dimensional inputs.

Port embedding. Given the port state vector x_i, NDT first build a port embedding $e_i = \phi(x_i)$. The result e_i is a vector (e.g., in $\mathbb{R}^{16}$) that captures information from heterogeneous attributes (see § III) of the port state vector. In addition, the port state vector can also be time series that carry information of multiple decision steps. The non-linear function $\phi(\cdot)$ could

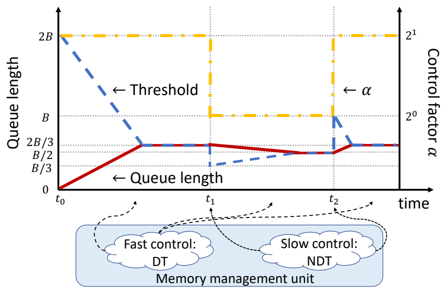


Figure 5. NDT's two-level control mechanism: an example. During decision interval the buffer threshold of each port is directly controlled by the DT policy with α_i at the packet-level, while the control factor α_i of each port is adjusted by the NDT agent when the trigger condition is satisfied.

be implemented as a recurrent neural network (RNN) to capture the temporal information.

Device and summary embedding. To leverage the permutation symmetry, we can take a permutation-invariant operation as the core building block. Based on that, since we need to make a decision for each port (i.e., the set element) instead of the whole device (i.e., the set), we need to maintain the order information throughout the processing procedure.

NDT achieves this with two steps. First, we create device embedding $\bar{e}$ by adopting the permutation-invariant operation on port embeddings, i.e., $\bar{e} = \frac{1}{N}\sum_i e_i$. Then we compute the port summary embeddings as:

$$z_i = f([e_i, \bar{e}]) = f([e_i, \frac{1}{N}\sum_i e_i]) \tag{4}$$

where e_i is the preprocessed port embeddings and f is a non-linear transformation implemented as a (small) neural network. In this process, each port embedding e_i is marked by device embeddings $\bar{e}$, which results in a port summary embedding z_i. In other words, each e_i customizes the whole set of port embeddings for itself to keep the order information. In this way, all z_is are permutation equivariant while containing the information of all ports.

These summary embeddings can be then used to get the actions. The continuous action a_i for each port is often modeled following a normal distribution $\mathcal{N}(\mu_i, \sigma_i)$ and the mean μ_i and the standard deviation σ_i is transformed from z_i with the function g and w, i.e., $\mu_i = g(z_i)$ and $\sigma_i = w(z_i)$. To make sure the whole procedure is end-to-end differentiable, the continuous action is derived with a reparameterization trick [32]: $a_i = \mu_i + \epsilon \cdot \sigma_i$, where the ϵ is a noise sampled from the standard normal distribution. NDT reuses the non-linear transformations ϕ, f, g, and w for all ports.

B. Efficient decision making

To enable efficient decision making, two design questions need to be answered: 1) *"what action should be taken?"* and 2) *"when should the action be taken?"*.

A naive strawman solution is to directly decide the allowed buffer size for each port at packet-level (i.e., every time a packet enqueues or dequeues, as what the DT policy does). Although this solution enables full expressiveness of the whole action space, it puts a high requirement of inference speed on the agent. For example, it only takes 120ns to dequeue a 1500B packet under a 100Gbps link. The time budget could be even tighter if packets arrive concurrently at multiple ports. On the other hand, even the most powerful AI inference chip can only take inference every 12us [30]. Even if the learned policy can be multiple times lighter than the resnet-50 [33] benchmark used to measure the AI chip, the chip still can not sustain such an inference speed requirement.

One option is to invoke the agent at a fixed interval longer than the maximum inference time. However, determining the time interval poses a new challenge. If the interval is too long, then when the traffic pattern changes or burst traffic arrives during the decision interval, the fixed buffer size assigned by the last decision becomes a lock that may cause severe packet dropping. If the interval is too small, making decisions in high frequency places a heavy burden on the training process, because many decisions are unnecessary and do not contribute to the overall outcome.

To solve this problem, NDT uses a two-level control mechanism (Figure 5), which consists of an action encoding scheme with domain knowledge and a novel trigger mechanism that can enlarge the decision interval. Specifically, NDT only decides a high-level control factor when the trigger (i.e., the slow control) activates, whereas the buffer allocation is directly controlled by a specified heuristic algorithm that acts at the packet level during the decision period (i.e., the fast control).

Action encoding with domain knowledge. To enlarge the action effective time, NDT turns to the domain knowledge and takes the DT policy [7] (§II) as the underlying buffer management mechanism. We will first recall the vanilla DT policy and then illustrate how NDT incorporates it.

The vanilla DT policy dynamically adjusts the queue threshold according to Equation 1 with the control factor α that is **shared** by all ports. With the queue building up, the threshold gradually decreases and finally converge, and a certain proportion of buffer is left unused for the light-load port whose queue length is below the threshold. When the queue length q_i equals or exceeds the threshold T_i, newly arrived packets will be dropped.

In NDT, we encode the control factor α_i of each port i as the action. Each port i uses an **independent** α_i and its queue length is restricted by the threshold T_i computed with the port α_i according to the equation 1. If α_i is a power of 2, the threshold computation can be easily implemented with only a shift register [7]. Thus we only consider the values of α_i in form of 2^m in this paper, which leads to a smaller action space. To get the α_i, we quantize the continuous output of the permutation-equivalent neural network and map it to a predefined set of α_i values (§VI).

In summary, the buffer is directly managed by the DT policy at the packet level while the control factor α_i for each port is adaptively controlled by the NDT agent. In this way, during the decision interval, the adaptation and robustness of the buffer management are ensured by the property of DT. Besides, embedding the domain knowledge of DT into NDT highly narrows the exploration space of NDT enabling fast training since NDT learns from a high starting point.

Cumulative-event trigger. To reduce the influence of high decision frequency, NDT adopts a novel cumulative-event trigger mechanism that invokes the agent every several packet events. Here, the packet event includes packet enqueue, packet dequeue, and packet dropping. The key idea is to trigger the agent only when necessary so the reduction on the decision number will not cause much performance degradation.

Specifically, the decision making is aligned to the traffic density and the NDT agent is dynamically triggered when the following condition is satisfied:

$$v^i_{ex} = v^i_{eq} + v^i_{dp} - v^i_{dq} \geq \beta \tag{5}$$

where v^i_{eq}, v^i_{dp}, and v^i_{dq} are the volume of packets enqueued, dropped and dequeued of port i since the last decision, respectively. β is a sensitivity hyperparameter. Zooming in on this mechanism, each packet arrival can only make a contribution to either v^i_{eq} when the queue length q_i is below the threshold T_i or v^i_{dp} when equals or exceeds. Hence, the v^i_{eq} and v^i_{dp} together describe the volume of incoming traffic, and v^i_{ex} reflects the traffic volume that exceeds the port capacity.

When the incoming rate matches the line rate, NDT does not need to make decisions (i.e., $v^i_{ex} = 0$). Note that we do not allow v^i_{ex} to be negative so that the packet dequeue v^i_{dp} counting for the queue draining process will not influence future traffic, which occurs when packets have been cumulated in queue during last decision interval. When the volume of incoming traffic reaches β (i.e., $v^i_{ex} \geq \beta$), either massive queue building up or severe packet dropping occurs so that we invoke the agent to adjust the control factor to match the current traffic status. We set $\beta = B/N$ as the fair share of the total buffer size among all ports since this reflects the expected buffer usage when facing the uniform traffic pattern.

In addition, the agent can also be correctly triggered with moderate traffic that only slightly exceeds the line rate but lasts for a while. This is because the left-side items in equation 5 can cumulate in time while β is a constant so that normally a small excess of the rate will finally trigger the agent. However, in practice, we find that the traffic rate could fluctuate around the line rate and v^i_{ex} always below β. In this case, the packet dropping could be unbounded, which could be a loophole for the whole system. To ensure robustness, we add a "safeguard" [34] to the proposed trigger mechanism. We force the agent to invoke when the packet dropping during the decision interval is larger than a "safe line", i.e., $v^i_{dp} \geq \gamma$, where γ is a hyperparameter that controls the maximum tolerance of the packet loss during a decision interval. We set γ as the same value as β for simplicity.

In summary, when the rate of incoming traffic of a port exceeds the line rate to the extend represented by the parameter β, the agent will adjust the α_is for all ports. After the trigger, these v^i_{eq}, v^i_{dp}, and v^i_{dq} of all ports will reset, which means the agent will only trigger by the busiest port and will not be re-triggered too soon by another port. This is reasonable since the α_i of other ports have also been adjusted recently by the last trigger. We only use the "safeguard" trigger as the last resort and expect it will never be triggered. To prevent the frequent "safeguard" trigger, we use a separate counter for it to measure the packet dropping and only trigger it once until the normal trigger is invoked.

V. Implementation

We first describe NDT's training infrastructure. Since existing commodity switches do not provide the interface to modify in-built buffer management mechanisms, we implement NDT on a software switch based on Intel DPDK [35], which follows the methodology of previous works [6], [36], [37].

A. Training infrastructure

Figure 6 shows the training infrastructure of NDT, which includes the NDT agent, the packet-level simulated environment, and the interface between them.

NDT Agent. We build the NDT agent with Python and Tensorflow [38]. For each observation, NDT constructs the per-port state x_i as an 8-step time-series of past observations. After normalization, x_i is passed to the permutation-equivariant neural network. The transformation functions ϕ consists of two parts. First, it passes each dimension of x_i to a 1D convolution layer (CNN) with 16 filters, each of size 3 with stride 1. Then it aggregates the results with fully-connected neural networks with 64 hidden units. The transformation functions f is also implemented as a fully connected neural network with 32 hidden units. The functions g and w are responsible for output and thus have only 1 output neuron. We apply the tanh activation function on the final output a to enforce its range within $[-1, 1]$ and map it to corresponding control factor $\alpha \in \{2^{-2}, 2^{-1}, ..., 2^5\}$. Since these neural networks are reused for all ports, NDT's model is lightweight and consists of only 47,650 parameters (~230KB) in total.

To speed up training, NDT spawns multiple learning agents in parallel. Each learning worker is configured to experience an environment with different traffic traces. The workers continually send their experiences (i.e., {state, action, reward} tuples) to a central agent, which aggregates them to generate a single buffer management policy. For each sequence of tuples that it receives, the central agent uses the SAC algorithm [32] to compute gradients and perform gradient descent steps. The central agent then updates the neural network parameters and pushes out the new model to the worker and the test agent. The test agent will validate the current model for model selection.

Packet-level simulator. We leverage ns-3 simulator [39] to build our simulated environment. The flow generator creates flows according to the input traffic trace that describes the source port, destination port, start time and flow size of each flow. By default, ns-3 does not support shared memory architecture. We implement it by enabling an output queue to get the queue length in all other ports of the same switch with a static member of C++ class.

Communication Interface. To enable communications between the python-based agent and the C++-based ns-3 simulator, we leverage ns3-gym [40] as the middleware interface.

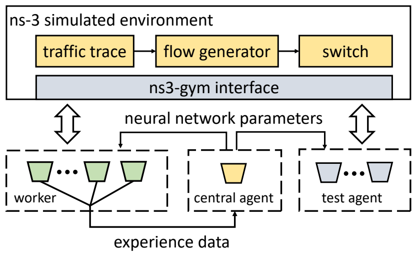


Figure 6. The training infrastructure of NDT including the NDT agent, the ns-3 simulator, and the ns3-gym interface.

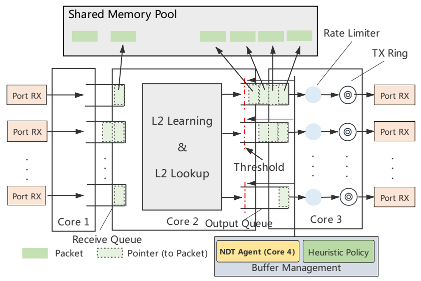


Figure 7. The architecture of DPDK-based switch prototype.

B. Testbed

We build a prototype switch with a server equipped with multiple NICs and implement NDT on top of it using Intel DPDK [35] to deliver data directly in user space (Figure 7). The basic L2 switch references the implementation in [6].

Receive, forward and send. The packet is firstly retrieved from the receiving port, then forwarded to the output queue, and finally transferred to the transmit ring (TX ring) buffer. To ensure the packets will not cumulate in the TX ring, we add a rate limiter to the output queue implemented as a leaky bucket so that the buffer occupancy in the output queue can correctly reflect the real buffer occupancy. To achieve line-rate sending/receiving, each RX/TX module is implemented on an individual core. The forwarding module runs on a separate core, and a hash table in address-port mapping is used to reduce the lookup delay. Note that the CPU processing rate is much higher than the line rate, thus most packets will be in the output queues rather than receive queues.

Shared memory pool. Packet transmission is through pointers rather than processing real data packets. In particular, after a packet enters the switch, it is placed in a pre-allocated shared memory pool, and a pointer to this packet returns. When sending a data packet, we put the pointer into the TX ring, and the DPDK driver will take the real data packet from the shared memory pool according to the pointer before sending the packet to the network.

Buffer management with NDT. Each output queue has a threshold. The packets will be discarded if the queue length exceeds the threshold or the total queue length of all ports exceeds the entire buffer size. The baseline buffer management policy is implemented by controlling the threshold of each port. The NDT model is loaded using Tensorflow's C library, and the inference is done using a separate core. First, the core always monitors the detailed statistics of the output queue to determine whether the trigger condition is satisfied (§IV-B). Then a circular queue is used to assemble the time-series state data. Finally, the state is transferred to the model which is pre-loaded in the memory, and the model returns the decision. The switch will use the returned control factor to calculate the threshold of each port (§IV-B) until the next trigger.

VI. Evaluation

In this section, we use large-scale simulation and our DPDK-based testbed to evaluate NDT.

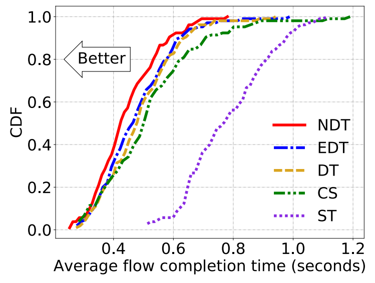
A. Methodology

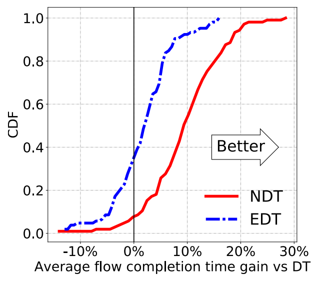
Training with simulation. We consider a 16-port shared-memory switch with 1MB buffer and 1Gbps port line speed [6], [41]. We train the NDT agent with the realistic heavy-tailed web search workload [5]. The training dataset consists of more than a hundred traffic traces, where each includes hundred to thousand of flows. The flows are generated with exponentially distributed inter-arrival time configured with the target load 50%. The source and destination of each flow are arbitrarily chosen with a random in-cast ratio ranging from 1 to 15. For each traffic trace, we control the number of active ports and randomly assign the incast degree to them to increase the representativity of the training dataset. The objective of the NDT agent is to minimize the average FCT. In both the simulation and the testbed, we use the same model.

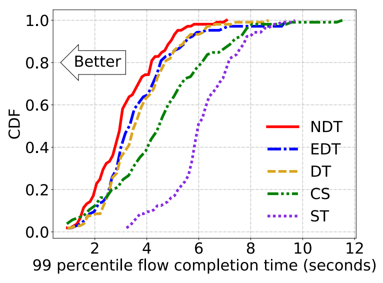
Schemes. We compare NDT with four baseline algorithms: 1) The Static Threshold (ST) policy, which does not allow buffer sharing and sets the buffer threshold for each port with the fair split of the total buffer size, i.e., B/N. This policy is also referred to as the complete partition policy; 2) The Complete Sharing (CS) policy, which does not restrict the buffer usage for any port; 3) The DT [7] policy, which set the shared buffer threshold for all ports as the α times unoccupied buffer size. We set $\alpha = 1$ according to the author's suggestion [7]; 4) The Enhanced Dynamic Threshold (EDT) [6] policy, which is built based on DT and allow the port with micro-burst traffic temporarily relax the threshold constraint. We set $\alpha = 1$ and other parameters according to the suggested parameter settings in §4.4 of [6].

B. NDT vs. Existing heuristics in simulation

We first compare the performance of NDT with that of baseline heuristics. The test traffic traces are with the same generation process as the training dataset but unseen during training. We measure the average FCT of each trace as the performance metric. Figure 8(a) shows a cumulative distribution of the average FCT over 105 test traffic traces. First, ST performs worst among all the baseline algorithms. This is because ST does not share the private port buffer with other ports, which validates the efficiency of the shared buffer architecture. NDT outperforms all baseline algorithms. NDT can achieve this performance gain since it uses a more expressive policy that allows each port to use different control factors and learns how to adapt to the traffic pattern in a fine-grained way. NDT automatically learns this policy through end-to-end RL learning, while the best-performing baseline algorithm EDT required careful tuning.


(a) CDF of average FCT.


(b) CDF of FCT gains over DT.


(c) CDF of 99 percentile FCT.

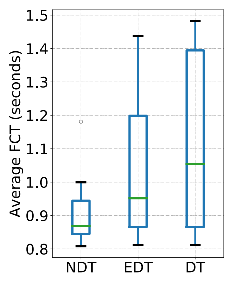
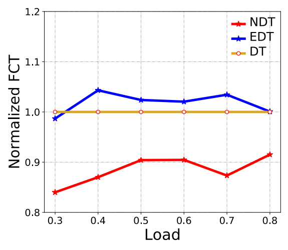

(d) 96-port environment.

Figure 8. NDT's learned buffer management policy achieves 8%–43% lower average FCT than baseline algorithms. The learned model efficiently scales to a larger-scale environment, which shows the scalability of NDT's permutation-equivariant neural networks.


(a) Web serach workload.

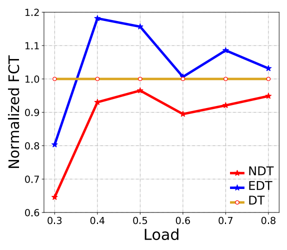

(b) Data mining workload.

Figure 9. NDT outperforms DT and EDT in a real DPDK-based switch prototype and could generalize to different workloads.

Specifically, NDT outperforms DT for more than 90% traces and achieves up to about 28% improvement (Figure 8(b)). NDT also outperforms EDT by a large margin. When considering the tail performance shown in Figure 8(c), NDT improves the 99 percentile FCT by 18% on average over DT while EDT only performs comparably with DT.

We also conduct an experiment on a larger-scale switch environment with the traffic of a 95-to-1 incast competing with elephant flows in other ports [5], which is not included in the training set. The switch has 96 ports with 10Gbps line rate and 6MB buffer. Note that the NDT model is not explicitly trained in this environment. The results shown in figure 8(d) confirm that NDT can efficiently scale to a large-scale environment. This is because the permutation-equivariant neural network reuses the building blocks for an arbitrary number of ports and captures the critical information among all port state pairs.

C. Generalization in a switch testbed

We built a testbed in which 4 host servers are connected to an emulating switch with four ports. The server that emulates the switch equips a 28-core Intel Xeon E5-2660 2.00GHz CPU, 32GB of memory, a 2TB hard disk, and four Intel 82599ES 10-Gigabit Ethernet NICs. The operating system is Ubuntu 16.04 LTS GNU/Linux kernel 4.15.0. The emulating switch has a 256KB shared buffer and uses a rate limiter to limit the sending rate of each port to 1Gbps. Other host servers equip with a 4-core Intel i3-3220 3.30GHz CPU and an Intel 82599ES 10-Gigabit Ethernet NIC. We use an empirical traffic generator [5], [42] to generate the traffic patterns to emulate the simple client/server application. Except for the web search [5] workload, we also use data mining [43] workloads in this experiment, which is not included in the training dataset. We measure the average FCT of the best-performing baselines (i.e., DT and EDT) over different loads. We repeat the experiment 50 times for each load and report the average FCT. The experiment lasts 40 minutes for each baseline and over 43,000 flows are generated in total.

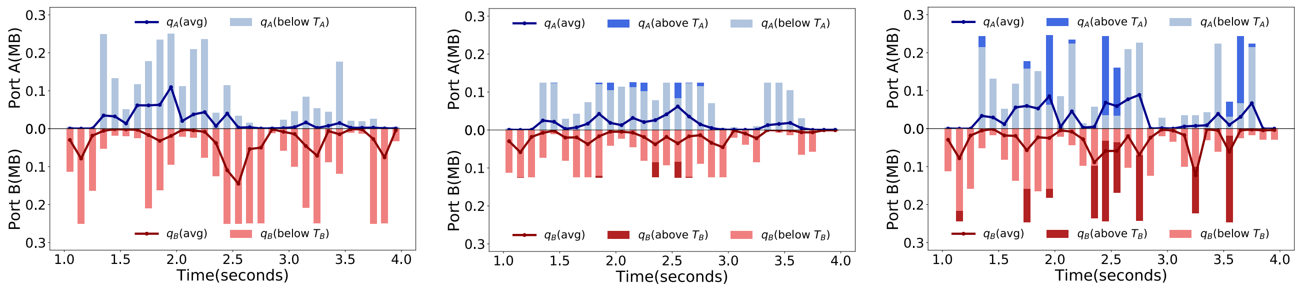
Figure 9(a) shows the average FCT of each policy normalized by that of DT under different traffic loads. NDT outperforms DT and achieves up to 16% improvement when the load is low, which shows that NDT can generalize well from the simulated environment to a real-world switch prototype. The NDT model evaluated here is solely trained using the web search workload. However, even facing the unseen traffic workload, NDT also generalizes well and outperforms other schemes (Figure 9(b)).

D. NDT deep dive

Learned policy. We illustrate the learned policy by NDT with a 4-to-2 incast example, where the two active destination port is marked as A and B. Figure 10 shows the evolution of the queue length q_i and its interaction with threshold T_i of CS, DT, and NDT[1]. There are three key observations from the results. First, CS performs poorly since it does not control the competition among different ports. For example, during 2.5s to 2.8s in Figure 10(a), port B occupies most of the buffer which starves the port A resulting in a near-zero queue length. Second, DT suffers from its fixed control factor so that it can not fully utilize the buffer (Figure 10(b)). Finally, NDT's performance gain comes from the initiative drain of the heavy-load queue as shown in Figure 10(c). NDT will firstly relax the threshold to absorb the burst as much as possible (like CS) but decrease the threshold after absorbing the burst by setting a small α, which reduces the long-term buffer occupation and improve the buffer cycle efficiency. Besides, the threshold of other light-load ports is set with almost no constraint to allow burst absorbing (not shown).

We also measure the port buffer occupancy when packet drops (Figure 11). We find that CS drops packets when the port either occupies all the buffer or starves, which confirms our finding in Figure 10(a). DT and EDT usually drop packets around the threshold converge point even when there is still an unoccupied buffer. NDT starts to drop packets when the port occupies more than 60% of the total buffer due to its initiative

[1]We omit the detailed behavior of EDT due to the space limitation. Generally, it behaves like DT but temporally relaxes the threshold constraints of the heavy-load port.



(a) CS policy: average FCT 0.99 seconds. (b) DT policy: average FCT 0.93 seconds. (c) NDT policy: average FCT 0.76 seconds.

Figure 10. For clarity, we only show part of this experiment. Each bar represents the queue status of a 0.1s interval. The portion of q above T is depicted in a dark color, while the portion of q below T is depicted in tint color. We plot with the maximum queue length and the minimum threshold of each interval. We also report the average queue length during the interval as the dark line.

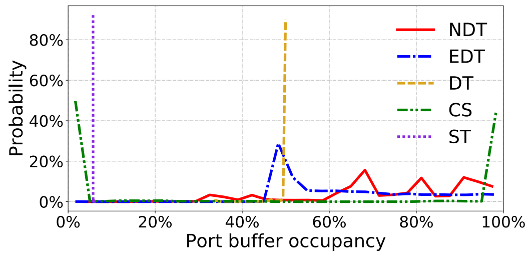
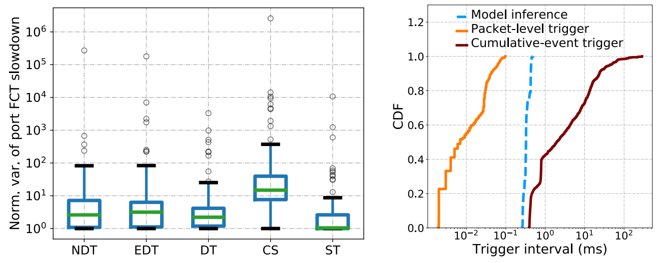


Figure 11. Port buffer occupancy when packet drops.



(a) FCT fairness. (b) Inference on testbed.

Figure 12. Micro benchmarks.

drain behavior. Packet dropping of NDT can also sometimes occur with low buffer occupancy since NDT can not always achieve a perfect reservation of future traffic.

FCT fairness. Considering the intra- and inter-port competition between flows, we use a new FCT fairness metric to measure the fairness among switch ports, i.e. $\mathrm{VAR}(\{\frac{FCT^i}{FCT^i_{isolated}}\})$. Here, FCT_i is the average FCT of flows that are destined to port i under a normal experiment while $FCT^i_{isolated}$ is the average FCT of flows that are destined to port i under an isolated environment where there is no competition between ports. We refer to this fraction as "port FCT slowdown". Finally, we measure the variance of port FCT slowdowns over all ports as the FCT fairness metric (lower is better). The intuition behind this metric is that a fair policy will fairly share the performance degradation between different ports when the buffer resource is limited while independent of their absolute performance. As shown in Figure 12(a), ST performs the best while CS is totally unfair, which validates the effectiveness of our metric. NDT performs comparably with EDT since both of them strive to achieve high efficiency by temporarily sacrificing the fairness among different ports. By doing this tradeoff, NDT achieves better overall performance. NDT could achieve better fairness by adding the fairness constraints to the reward function and we leave this as our future work.

Inference performance. Figure 12(b) shows cumulative distributions of the trigger interval of NDT's cumulative-event trigger mechanism (in red) and the naive per-packet trigger mechanism (in orange) in our DPDK-based testbed (§V-B). We also report the model inference time (in blue) of NDT. The average inference delay of NDT is less than 0.4ms, while the interval between NDT's triggers is typically on the scale of several to tens milliseconds. In all the cases, the inference time is shorter than the trigger interval of NDT while it is larger than the per-packet ones. The trigger intervals of the naive mechanism vary since multiple packet events (e.g., enqueue or dequeue) could occur concurrently at different ports.

Thus NDT's trigger mechanism greatly enlarges the decision interval, reduces the trigger frequency, and loosens the burden of the AI chips. Note that the inference is conducted with the CPU on our emulated switch server. The inference time can be further reduced with speedup techniques, e.g., model compression [44], which provides a potential that NDT could be deployed in switches without AI chips by using the control plane CPU to make the inference. We also measure the trigger interval of two mechanisms in a simulated 100Gbps environment. The per-packet mechanism triggers at the scale of nano- even pico-seconds while NDT's trigger mechanism still works at a larger time scale, i.e., micro- to milli-seconds, which matches the inference speed of AI-chips [30].

VII. Conclusion

In this paper, we show that learning effective buffer management policies automatically using reinforcement learning is feasible. NDT's design principle is consistent with the recent research trend of combining learning-based methods with heuristics [21]–[23]. NDT's learning innovations, such as its permutation-equivariant neural network design, may be applied to other permutation-symmetric networking problems.

Acknowledgment

This work was supported in part by the NSFC Project of China (No. 6213000078 and No. 61872211). We thank Antong Shan, Xiaoyu Cheng, Deping Peng, Linbo Hui for their work on the early version of this paper.

References

[1] G. Appenzeller, I. Keslassy, and N. McKeown, “Sizing router buffers,” *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 4, pp. 281–292, 2004.

[2] M. Alizadeh, T. Edsall, S. Dharmapurikar, R. Vaidyanathan, K. Chu, A. Fingerhut, V. T. Lam, F. Matus, R. Pan, N. Yadav *et al.*, “Conga: Distributed congestion-aware load balancing for datacenters,” in *ACM SIGCOMM*, 2014.

[3] U. Cummings, A. Lines, P. Pelletier, and R. Southworth, “Shared-memory switch fabric architecture,” Oct. 12 2010, uS Patent 7,814,280.

[4] S. Das and R. Sankar, “Broadcom smart-buffer technology in data center switches for cost-effective performance scaling of cloud applications,” 2012.

[5] M. Alizadeh, A. Greenberg, D. A. Maltz, J. Padhye, P. Patel, B. Prabhakar, S. Sengupta, and M. Sridharan, “Data center tcp (dctcp),” in *ACM SIGCOMM*, 2010.

[6] D. Shan, W. Jiang, and F. Ren, “Analyzing and enhancing dynamic threshold policy of data center switches,” *IEEE TPDS*, 2017.

[7] A. K. Choudhury and E. L. Hahne, “Dynamic queue length thresholds for shared-memory packet switches,” *IEEE/ACM Transactions On Networking*, 1998.

[8] A. Kesselman and Y. Mansour, “Harmonic buffer management policy for shared memory switches,” *Theoretical Computer Science*, 2004.

[9] I. Cidon, L. Georgiadis, R. Guerin, and A. Khamisy, “Optimal buffer sharing,” *IEEE Journal on Selected Areas in Communications*, 1995.

[10] E. L. Hahne and A. K. Choudhury, “Dynamic queue length thresholds for multiple loss priorities,” *IEEE/ACM Transactions On Networking*, 2002.

[11] G. Ascia, V. Catania, and D. Panno, “An evolutionary management scheme in high-performance packet switches,” *IEEE/ACM Transactions On Networking*, 2005.

[12] S. X. Wei, E. J. Coyle, and M.-T. Hsiao, “An optimal buffer management policy for high-performance packet switching,” in *IEEE GLOBECOM*, 1991.

[13] M. Apostolaki, L. Vanbever, and M. Ghobadi, “Fab: Toward flow-aware buffer sharing on programmable switches,” in *Online Program: Workshop on Buffer Sizing*, 2019.

[14] H. Yousefi’zadeh and E. A. Jonckheere, “Dynamic neural-based buffer management for queuing systems with self-similar characteristics,” *IEEE Transactions on Neural Networks*, 2005.

[15] S. Huang, M. Wang, and Y. Cui, “Traffic-aware buffer management in shared memory switches,” in *IEEE INFOCOM*, 2021, pp. 1–10.

[16] M. Wang, Y. Cui, X. Wang, S. Xiao, and J. Jiang, “Machine learning for networking: Workflow, advances and opportunities,” *IEEE Network*, 2017.

[17] H. Mao, R. Netravali, and M. Alizadeh, “Neural adaptive video streaming with pensieve,” in *ACM SIGCOMM*, 2017.

[18] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, “Learning scheduling algorithms for data processing clusters,” in *ACM SIGCOMM*, 2019.

[19] M. Wang, Y. Cui, S. Xiao, X. Wang, D. Yang, K. Chen, and J. Zhu, “Neural network meets dcn: Traffic-driven topology adaptation with deep learning,” *ACM SIGMETRICS*, 2018.

[20] L. Zhang, M. Wang, Z. Yang, and Y. Jiang, “Machine learning for internet congestion control: Techniques and challenges,” *IEEE Internet Computing*, 2019.

[21] L. Chen, J. Lingys, K. Chen, and F. Liu, “Auto: Scaling deep reinforcement learning for datacenter-scale automatic traffic optimization,” in *ACM SIGCOMM*, 2018.

[22] H. Zhu, V. Gupta, S. S. Ahuja, Y. Tian, Y. Zhang, and X. Jin, “Network planning with deep reinforcement learning,” in *ACM SIGCOMM*, 2021.

[23] S. Abbasloo, C.-Y. Yen, and H. J. Chao, “Classic meets modern: A pragmatic learning-based congestion control for the internet,” in *ACM SIGCOMM*, 2020.

[24] “A cloudengine switch built for the ai era,” http://e.huawei.com/topic/cloud-engine2019/en/index.html, accessed December 22, 2021.

[25] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, “Human-level control through deep reinforcement learning,” *Nature*, 2015.

[26] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, “Mastering the game of go with deep neural networks and tree search,” *nature*, 2016.

[27] M. Zaheer, S. Kottur, S. Ravanbakhsh, B. Poczos, R. R. Salakhutdinov, and A. J. Smola, “Deep sets,” in *NeurIPS*, 2017.

[28] N. Guttenberg, N. Virgo, O. Witkowski, H. Aoki, and R. Kanai, “Permutation-equivariant neural networks applied to dynamics prediction,” *arXiv preprint arXiv:1612.04530*, 2016.

[29] E. Wagstaff, F. B. Fuchs, M. Engelcke, I. Posner, and M. Osborne, “On the limitations of representing functions on sets,” *ICML*, 2019.

[30] “Announcing hanguang 800: Alibaba’s first ai-inference chip,” https://www.alibabacloud.com/blog/announcing-hanguang-800-alibabas-first-ai-inference-chip_595482, accessed December 22, 2021.

[31] T. Haarnoja, A. Zhou, P. Abbeel, and S. Levine, “Soft actor-critic: Off-policy maximum entropy deep reinforcement learning with a stochastic actor,” in *ICML*, 2018.

[32] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel *et al.*, “Soft actor-critic algorithms and applications,” *arXiv preprint arXiv:1812.05905*, 2018.

[33] K. He, X. Zhang, S. Ren, and J. Sun, “Deep residual learning for image recognition,” in *IEEE CVPR*, 2016.

[34] S. Sen, “How to make decisions (optimally)(keynote),” in *OPODIS*, 2018.

[35] D. Intel, “Data plane development kit,” 2014.

[36] K. Qian, W. Cheng, T. Zhang, and F. Ren, “Gentle flow control: avoiding deadlock in lossless networks,” in *ACM SIGCOMM*, 2019.

[37] W. Cheng, K. Qian, W. Jiang, T. Zhang, and F. Ren, “Re-architecting congestion management in lossless ethernet,” in *USENIX NSDI*, 2020.

[38] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, “Tensorflow: A system for large-scale machine learning,” in *USENIX OSDI*, 2016.

[39] G. F. Riley and T. R. Henderson, “The ns-3 network simulator,” in *Modeling and tools for network simulation*. Springer, 2010.

[40] P. Gawłowicz and A. Zubow, “ns-3 meets openai gym: The playground for machine learning in networking research,” in *Proceedings of the 22nd International ACM Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, 2019.

[41] “Packet buffers.” https://people.ucsc.edu/~warner/buffer.html, accessed December 22, 2021.

[42] “empirical-traffice-gen.” https://github.com/datacenter/empirical-traffic-gen.git, accessed December 22, 2021.

[43] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta, “Vl2: a scalable and flexible data center network,” in *ACM SIGCOMM*, 2009.

[44] Y. Cheng, D. Wang, P. Zhou, and T. Zhang, “A survey of model compression and acceleration for deep neural networks,” *arXiv preprint arXiv:1710.09282*, 2017.