Learning Traffic Encoding Matrices for Delay-Aware Traffic Engineering in SD-WANs

Majid Ghaderi[*], Wenjie Liu[†], Shihan Xiao[†], and Fenglin Li[†]
[*]University of Calgary, Email: mghaderi@ucalgary.ca
[†]Huawei Technologies, Emails: {liuwenjie, xiaoshihan, lifenglin}@huawei.com

***Abstract*—This paper introduces Traffic Encoding Matrices (TEMs) as an alternative to traditional Traffic Matrices (TMs) for representing network traffic demands. While TMs only capture average demand information, TEMs are designed to capture distributional demand information by learning representations whose variations capture most of the structure of the distribution of demands. We present a practical approach based on off-the-shelf neural autoencoders to efficiently construct TEMs at the edge of the network. We then present the design and evaluation of NeuroTE, a DRL-based framework for delay-aware traffic engineering in SD-WANs using TEMs. Using real traffic traces, we present experimental results to demonstrate the advantages of using TEMs instead of TMs in traffic engineering. Our results show that, when traffic demands are dynamic, TEM-based control leads to: 1) improved network performance, and 2) faster convergence to the optimal solution compared to TM-based control using exactly the same DRL control algorithm.**

I. Introduction

Motivation. Traffic engineering (TE) is a fundamental problem in network control, and has been extensively studied in the literature [1]–[4], [4]–[8]. In abstract form, TE can be framed as a combinatorial optimization problem: given a capacitated network and a set of flows, find an assignment of flows to network paths in order to optimize an objective such as minimizing the maximum link utilization (MLU). Each flow is specified by its origin, destination and traffic demand. The set of flows can be succinctly represented by a traffic matrix (TM) [9], whose elements specify the average traffic demand between pairs of origin-destination nodes in the network. When multi-path routing is supported in the network, which is a common scenario, the TE problem can be solved for a fixed TM in polynomial time, *e.g.*, using linear programming [10].

In real-world, however, traffic demands are dynamic and fluctuate over time. To respond to changes in traffic demands, in adaptive TE, flow assignments are dynamically updated over time. Specifically, the assignment of flows changes in discrete control epochs. At the beginning of each epoch, a flow assignment is computed for that epoch to accommodate a nominal traffic matrix that is predicted for that epoch [11]–[13]. Traditionally, optimization-based TE has been concerned with optimizing for performance objectives that can be mathematically represented in such a way that the resulting optimization problem can be solved efficiently. For example, minimizing MLU is widely considered in the TE literature, as link utilization is linearly proportional to the average traffic demand that is readily provided by a TM. While performance objectives such as MLU capture a desired network configuration from the perspective of network operators, *i.e.*, distributing traffic across the network, they may not map directly to application QoS requirements. Specifically, a growing portion of network traffic is generated by applications that require very low end-to-end delay to work efficiently [1], [3], [6], [14]–[16]. Despite this, incorporating complex objectives such as end-to-end delay in optimization-based TE is not feasible. First, there is no analytically tractable model of end-to-end delay for a general network without making simplifying assumptions about the network and traffic (*e.g.*, Poisson arrival process). Second, modeling delay generally requires higher order traffic statistics such as variance (*e.g.*, average delay in a $G/G/1$ queue) which are not captured in a TM. Recall that a TM only contains information about the *average* traffic demand over a control epoch. Such a basic representation fails to capture higher order distributional information about traffic demands that potentially affect the performance objective. *This is a fundamental limitation of all TM-based approaches for traffic engineering in dynamic networks.*

Clearly, if distributional information about demands could be provided and incorporated into the traffic engineering model, it would lead to more accurate flow assignments and consequently improved network performance. Unfortunately, most legacy network devices can only provide coarse-grained flow counters (*e.g.*, number of bytes passing through a port) over a slow polling cycle [17]. As such, an average TM is all that can be extracted from the network, but even that has its own challenges [9]. Naturally, the question then is how can we extract distributional demand information for use in traffic engineering? Our motivation for this work is based on two recent trends in the networking area: 1) extension of software-defined networking (SDN) architecture into the wide-area network (WAN), referred to as SD-WAN [18], and 2) emergence of high-performance programmable switches with line rate processing at data plane [19], [20], thanks to their ability to offload packet processing to onboard accelerators such as GPUs and DPUs. The SD-WAN architecture enables centralized traffic engineering, while programmable switches allow traffic processing at the edge of the network without requiring any modification at the core.

Our Work. In this work, we present the design and evaluation of NeuroTE, an ML-based framework for traffic engineering in SD-WANs that can optimize for any objective, such as end-to-end delay, that can be inferred from network measurements. NeuroTE addresses two key challenges: ($\mathbf{Q}_1$) How to extract and encode distributional demand information in an online manner when traffic demands are dynamic and fluctuate over time? ($\mathbf{Q}_2$) How to utilize such dynamic distributional information about demands in traffic engineering specially when the network objective includes complex performance metrics such as end-to-end delay.

The NeuroTE approach to address ($\mathbf{Q}_1$) is the automatic extraction and encoding of distributional demand information at the edge of the network using deep neural autoencoders. NeuroTE applies an end-to-end approach to train the autoencoders with respect to the objective being optimized. The autoencoders are trained to generate *encoding vectors* whose variations capture most of the structure of the distribution of demands. Using the learned encoding vectors, we construct a *traffic encoding matrix* (TEM), whose elements specify the traffic encoding vectors of every flow in the network. To address ($\mathbf{Q}_2$), the NeuroTE approach is based on reinforcement learning (RL) [21], which can be naturally integrated with traffic encoders in an end-to-end control framework. Specifically, we will design a TE algorithm based on the deep reinforcement learning (DRL) approach [22], which compared to traditional RL, is more scalable and can deal with continuous state spaces, making it suitable for TEM-based traffic engineering. While NeuroTE can optimize for any objective that can be inferred from the network, for concreteness and to demonstrate the utility of our framework, we define the TE objective as minimizing the *maximum end-to-end delay* experienced by any flow in the network.

Contributions. While the idea of using DRL in TE has been studied in the literature (see, *e.g.*, [23]), our main contribution in this work is an end-to-end learning framework for TE. While existing DRL-based approaches are capable of optimizing for a richer set of performance metrics, e.g., delay, they are designed within the constraints of traditional approaches in the sense that they only replace the model-based controller with a DRL agent. The input to the DRL controller is the legacy traffic matrix. If we provide the DRL agent with distributional traffic information, the agent will learn faster (i.e., converge faster) and make better decisions (i.e., improved performance), as it has more detailed information about the environment with which it interacts. The NeuroTE framework demonstrates how such information can be learned in the form of a traffic encoding matrix at the edge of the network, and how it can be used for TE in a centralized DRL algorithm.

To this end, our contributions can be summarized as follows:

- We present the design and evaluation of NeuroTE, a new ML-based approach for delay-aware traffic engineering in SD-WANs with dynamic traffic demands.
- We propose the concept of traffic encoding matrices, and design a general technique for constructing traffic encodings based on off-the-shelf neural autoencoders.
- We design a DRL-based controller based on Soft Actor-Critic algorithm [24] that we customize with a novel prioritization technique for enhanced training performance.
- We conduct experiments using real WAN traffic traces to study the performance of NeuroTE and compare it with existing approaches. In particular, we compare NeuroTE with a DRL-based TE approach that uses TMs instead of TEMs [3] to show the benefits of using TEMs.

Organization. We formally define the TE problem in Section III and present NeuroTE in Section IV. Evaluation results are presented in Section V. Related works are reviewed in Section VI, while Section VII concludes the paper.

II. Deep Reinforcement Learning

Reinforcement learning (RL) is the area of machine learning that deals with sequential decision-making [21]. A standard RL setup consists of an agent that interacts with an environment in discrete decision epochs. The environment state transitions are stochastic and assumed to be governed by a Markov process. At each decision epoch t, the agent observes some environment state s_t and takes an action a_t. Following the action, the state of the environment transitions to s_{t+1} and the agent receives a scalar reward $r_t = r(s_t, a_t)$. The agent's behavior is defined by a policy $\pi(a_t|s_t)$, which maps states to a probability distribution over the actions. Define the return $R_t = \sum_{k=t}^{\infty} \gamma^{k-t} r(s_k, a_k)$ as the total discounted future reward at epoch t, where $\gamma \in [0, 1]$ is called the discount factor. The goal in reinforcement learning is to learn a policy that maximizes the expected return over some distribution of the initial states, as defined by

$$\pi^* = \arg\max_{\pi} \mathbb{E}_{\pi}\left[R_0\right] . \tag{1}$$

The action-value function $Q^{\pi}(s, a)$ is used in many reinforcement learning algorithms to learn the optimal policy. It describes the expected return starting from state s, taking the action a and following policy π afterwards

$$Q^{\pi}(s, a) = \mathbb{E}_{\pi}\left[R_t | s_t = s, a_t = a\right] . \tag{2}$$

Specifically, if the optimal action-value function, defined as $Q^*(s, a) = \max_{\pi} Q^{\pi}(s, a)$, is known, then in any given state s, the optimal action $a^*(s)$ can be found by solving for $a^*(s) = \arg\max_a Q^*(s, a)$. One of the early breakthroughs in reinforcement learning was the development of an algorithm known as *Q-learning*. In Q-learning, the optimal action-value function is directly learned using the Bellman equation:

$$Q(s, a) = \mathbb{E}\big[r(s_t, a_t) + \gamma \max_{a'} Q(s_{t+1}, a') \big| s_t = s, a_t = a\big] . \tag{3}$$

In most practical problems, the number of possible (state, action) pairs is too large to store the action-value function in a lookup table. It is therefore common to use a function approximator [21] to represent the action-value function as $Q_{\theta}(s, a)$, where θ denotes the parameter set of the approximator. While many forms of function approximators can be used, in deep reinforcement learning (DRL), deep neural networks are used for function approximation. The seminal work kick-starting the revolution in DRL was the development of the Deep Q-Network (DQN) algorithm that could learn to play Atari video games directly from pixel images [22]. In DQN, the deep neural network approximating the action-value function is trained by minimizing the loss function, defined as

$$L(\theta) = \mathbb{E}\left[\big(y_t - Q_{\theta}(s_t, a_t)\big)^2\right] , \tag{4}$$

where $y_t = r(s_t, a_t) + \gamma \max_{a'} Q_\theta(s_{t+1}, a')$ is the *target* value for training the neural network. While using neural networks as function approximators was not new, it was generally believed that learning value functions using non-linear function approximators (such as neural networks) was difficult and unstable [25]. Two effective techniques were proposed in DQN to improve learning stability: 1) the network is trained with samples from a replay buffer to minimize correlations between samples, and 2) the network is trained with a separate target network to give consistent targets during updates. Although DQN can be applied to problems with high-dimensional state space, it can only handle low-dimensional discrete action spaces. Many practical control problems such as traffic engineering have continuous and high-dimensional action spaces. It is not straightforward to apply DQN to continuous control problems since it relies on finding the action that maximizes the action-value function, which requires solving a non-trivial optimization problem at each epoch. Instead, the actor-critic approach [26] has often been used for continuous control. For example, the DDPG algorithm [27] maintains a parameterized actor function $\pi_\phi(\cdot|s)$ and a parameterized critic function $Q_\theta(s, a)$, where both actor and critic are implemented using deep neural networks.

III. Delay-Aware Traffic Engineering

In this section, we describe the problem of delay-aware traffic engineering in SD-WAN that is considered in our work.

A. Network Model

We consider a general model of an SD-WAN as depicted in Fig. 1. In this model, the network consists of a set of edge switches that are inter-connected via a set of paths. The WAN interconnect may span multiple autonomous systems in which case the physical composition of paths may not be known to the traffic engineering controller, *e.g.*, some links could be implemented as MPLS tunnels whose substrate topology would be unknown. We use the term "flow" to refer to the traffic from one edge switch to another edge switch. Let $\mathscr{F} = \{f_1, \ldots, f_K\}$ denote the set of flows (*i.e.*, switch pair traffic) in the network. Each flow f_i can be routed over a set of paths $\mathscr{P}_i$. While this can result in packet-level reordering, we do not model its effects, assuming existing techniques can be applied to resolve any reordering (*e.g.*, [28]). This model is widely considered in the traffic engineering literature [1]–[8].

B. Network Objective

One major objective of traffic engineering is to minimize congestion in the network by optimally distributing traffic over different network paths. While network congestion manifests itself in increased packet loss and delay, it is difficult to model such metrics mathematically as needed in optimization-based TE approaches. Thus, most existing approaches try to indirectly minimize congestion by controlling the utilization of links in the network, which can be mathematically represented by a closed-form expression. In contrast, one of the strengths of NeuroTE is its ability to optimize for any performance metric, such as end-to-end delay, that can be inferred from network measurements. Specifically, we define the network objective as *minimizing the maximum delay experienced by any flow* during an epoch. The goal is to reduce the severity of network delay spikes, which are detrimental to the performance of delay-sensitive applications, but notoriously difficult to prevent due to the dynamic nature of network traffic.

C. Problem Statement

The problem of delay-aware traffic engineering is to determine an optimal flow assignment for a given set of flows to minimize the *Maximum Flow Delay* (MFD). Let w_{ij} denote the fraction of flow f_i traffic that is routed on path $P_j \in \mathscr{P}_i$. The goal is to determine a *flow assignment*, *i.e.*, the set of w_{ij}'s for each flow based on a given traffic encoding matrix $\mathbf{T}$. The computed flow fractions are used for routing traffic during the current control epoch. In particular, if the rate of traffic of flow f_i is r_i, then the edge switch routes $w_{ij} r_i$ amount of flow f_i traffic over path P_j. Notice that r_i is a random variable and changes with time, but w_{ij}'s remain fixed during each control epoch. Let $\mathbf{w} = (w_{ij}, P_j \in \mathscr{P}_i, f_i \in \mathscr{F})$ denote a flow assignment. The problem can be formally described as the following optimization problem:

Delay-Aware TE Problem:

$$\min_{\mathbf{w}} \quad \mathrm{MFD}(\mathbf{w}, \mathbf{T}) \tag{5a}$$

$$\text{subject to} \quad \sum_{P_j \in \mathscr{P}_i} w_{ij} = 1, \quad \text{for all } f_i \in \mathscr{F}_i \tag{5b}$$

$$w_{ij} \geq 0, \quad \text{for all } P_j \in \mathscr{P}_i, f_i \in \mathscr{F} \tag{5c}$$

where, $\mathbf{T}$ denotes the estimated TEM for the epoch, and the objective function $\mathrm{MFD}(\mathbf{w}, \mathbf{T})$ denotes the maximum flow delay for a given flow assignment $\mathbf{w}$ and traffic encoding matrix $\mathbf{T}$,

$$\mathrm{MFD}(\mathbf{w}, \mathbf{T}) = \max_{f_i \in \mathscr{F}} \max_{P_j \in \mathscr{P}_i} d_{ij}(\mathbf{w}, \mathbf{T})\,. \tag{6}$$

In this definition, $d_{ij}(\mathbf{w}, \mathbf{T})$ denotes the maximum *one-way delay* of flow f_i on its path P_j over the entire control epoch. Thus, MFD captures the maximum one-way delay among all flows in a control epoch.

It is worth mentioning that optimizing for such an end-to-end network objective in model-based approaches is quite challenging as there is no accurate and yet tractable mathematical model for end-to-end delay under generic traffic demands. Most available results based on queuing theory rely on very specific patterns of traffic (*e.g.*, Poisson) and can often be solved for average delay only. In NeuroTE, however, the objective is directly inferred from the network. In our design, edge nodes periodically send feedback to the TE controller, which uses these feedbacks to infer MFD in each epoch.

D. Model Generality

We believe that our framework can be applied to other control problems that rely on traffic demands to optimize network performance. For example, in network utility maximization (NUM) framework for resource allocation [29],

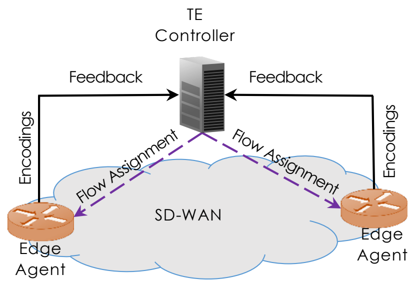


Fig. 1: NeuroTE conceptual architecture.

the objective is to maximize the total network utility. By carefully modeling the utility of each flow, a broad range of performance objectives in terms of network efficiency and fairness can be achieved [3], [30]. However, not only the NUM framework requires an accurate mathematical representation of network utility, but also the utility function must posses certain properties (*e.g.*, a concave function of the rate). In contrast, the NeuroTE framework does not impose any such restrictions as it infers the network utility from network feedback.

IV. NeuroTE Design

In this section, we present the design of NeuroTE for delay-aware TE, and discuss how traffic encoding matrices are constructed and utilized for decision making in NeuroTE.

A. Architecture

The high-level architecture of NeuroTE is depicted in Fig. 1. The main components of the architecture as well as their operations are described below.

Control Loop. The system works in control epochs. At the end of each epoch, edge agents send their local traffic encoding matrices and network feedback to the controller. The controller computes a flow assignment for the subsequent control epoch and sends it to edge agents to enforce in that epoch. The length of the control epoch should be chosen to achieve a balance between performance and control overhead. The system forms a closed control loop, and as such the length of the control epoch is lower bounded by the latency of the control loop. This includes time required to compute traffic encodings and collect feedback at edge, round-trip-time between edge agents and the controller, and the time it takes for the controller to compute a flow assignment.

TE Controller. The controller implements the DRL algorithm for computing flow assignments. While a distributed algorithm can be considered (as in [31]), the centralized model nicely fits the architecture of SD-WAN [1], [2]. It also results in a control architecture with lower complexity and faster control loop feedback. The design of the DRL algorithm is described in detail in Section IV-C

Edge Agents. The edge agents have three main functionalities: 1) to sample local traffic and generate traffic encoding vectors, 2) to split local flows according to the flow assignment for the current control epoch, and 3) to send feedback about the network objective (*i.e.*, maximum flow delay) to the controller. The rate of traffic sampling is dictated by the capabilities of the edge switch hardware. The sampling itself involves computing the rate of traffic over the sampling interval.

Delay Measurement. There are a number of software and hardware-based solutions that can be used to accurately measure one-way delay in a network [32]–[34]. While our design is independent of the specific protocol used, a passive measurement approach that utilizes the programmability of edge switches to timestamp data packets at their origin is a low-overhead solution that can be adopted in NeuroTE.

Flow Splitting. As commonly considered in literature on traffic engineering (*e.g.*, [1]–[8]), we assume packet level flow splitting. Consider a flow assignment $\mathbf{w}$, where w_{ij} denotes the fraction of flow f_i that should be sent over path $P_j \in \mathscr{P}_i$. The flow fraction w_{ij} is interpreted as the probability with which each packet of flow f_i is transmitted over path P_j.

B. Traffic Encoding

Design. We use a neural autoencoder [35] to extract and encode traffic demand information. The autoencoder attempts to characterize the unknown distribution from which the traffic demands are sampled through learning a set of features whose variations capture most of the structure of the distribution. The autoencoder is composed of two stages, an encoder and a decoder. The encoder and decoder networks, denoted by f and g, are trained such that the encoder outputs a learned representation $\mathbf{e} = f(\mathbf{x})$ of an input $\mathbf{x}$, while the decoder outputs $g(f(\mathbf{x})) \approx \mathbf{x}$. This is achieved by penalizing the model according to the reconstruction error such that the model can learn the most important features of the input data and how to best reconstruct the original data from the encoded state $\mathbf{e}$, which we call the *encoding vector*. For each flow f_i, a separate encoding vector $\mathbf{e}_i$ is computed by the autoencoder. The input to the encoder is a series of traffic samples. Specifically, the edge agent measures per-flow[1] traffic demand periodically and keeps a window of measurements of size τ. The encoding vector for flow f_i in next epoch is computed based on the input vector $\mathbf{x}_i = [x_i(t-1), \dots, x_i(t-\tau)]$, where $x_i(t-n)$ denotes the sampled traffic rate of flow i in the n-th previous sampling time, where the current time is denoted by t.

Training. Recall that encoders run on edge nodes. This raises several questions with respect to training of the encoders. First, *how many encoders should be trained?* Second, *how to train encoders end-to-end with the DRL algorithm?* To answer the first question, we note that a flow in NeuroTE represents aggregate traffic between an origin and destination node. As such, we hypothesis that different flows would have similar statistical properties. This allows us to apply transfer learning techniques to train only a single encoder model and then use it for encodings on all edge nodes, which significantly reduces the training time and complexity compared to training multiple flow-specific models. While a single encoder is trained using offline traffic traces, each edge agent runs its own encoder based on the single trained model. To answer the second

[1]Recall that we defined a flow as the traffic between an origin-destination pair and not a single 5-tuple IP flow.

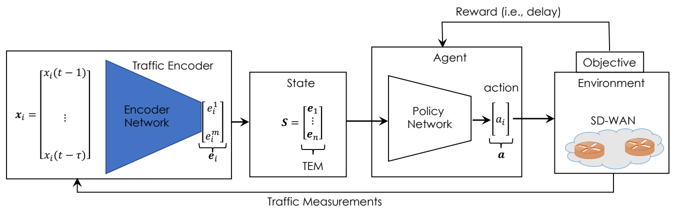

Reward (i.e., delay)
Traffic Encoder
$x_i = \begin{bmatrix} x_i(t-1) \\ \vdots \\ x_i(t-\tau) \end{bmatrix}$
Encoder Network
$\underbrace{\begin{bmatrix} e_i^1 \\ e_i^m \end{bmatrix}}_{e_i}$
State
$S = \underbrace{\begin{bmatrix} e_1 \\ \vdots \\ e_n \end{bmatrix}}_{\text{TEM}}$
Agent
Policy Network
action
$\underbrace{\begin{bmatrix} a_i \end{bmatrix}}_{a}$
Objective
Environment
SD-WAN
Traffic Measurements

Fig. 2: NeuroTE end-to-end design.

question, we can backpropagate the gradient information to edge agents for online tuning of the encoder. An advantage of our design is that the encoders and the controller are based on DNNs, which are end-to-end differentiable for training.

Representation vs Prediction. We emphasize that our goal is to show the importance of capturing distributional demand information. As such, we have decided to use a basic autoencoder architecture in our design and train it only with respect to reconstruction error, as opposed to, for example, prediction error. However, we expect to see improved performance by using more advanced architectures such as those based on long short-term memory (LSTM) [36] encoder and decoders, which are more suitable for sequence learning.

C. Controller Design

The controller in NeuroTE is implemented as a DRL agent. Fig. 2 shows the end-to-end design of NeuroTE corresponding to the conceptual architecture depicted in Fig. 1.

DRL Environment. First, we describe the design of the state space, action apace and reward function.

- *State Space:* The state of the environment is composed of the traffic encoding vectors of flows in the network. Formally, the state is represented as the vector $\mathbf{S} = [e_1, \dots, e_n]$, where $e_i \in \mathbb{R}^m$ denotes the encoding vector of flow f_i. Note that this defines a continuous state space in $\mathbb{R}^m$. The encoding vectors $\mathbf{e}_i$ are provided to the algorithm in a traffic encoding matrix for each epoch.
- *Action Space:* Each action is represented as a vector $\mathbf{a} = [a_1, \dots, a_k]$, where $a_i \in [0, 1]$ is the score assigned to path P_i. The action space is defined over $[0, 1]^k$, where $k = \sum_{f_i \in \mathscr{F}} |\mathscr{P}_i|$ denotes the number of paths in the network. The flow assignment weights $\mathbf{w}$ are computed from the action vector $\mathbf{a}$ as:
$$w_{ij} = \frac{a_{ij}}{\sum_{P_j \in \mathscr{P}_i} a_{ij}}, \quad \forall P_j \in \mathscr{P}_i, f_i \in \mathscr{F}\ . \tag{7}$$
- *Reward:* The objective of a DRL agent is to maximize the expected cumulative reward. The objective of NeuroTE is to minimize MFD. Thus, we define reward as the negative of MFD. Formally, we have $r = -\text{MFD}(\mathbf{w}, \mathbf{T})$, where $\mathbf{w}$ is computed from the action taken by the agent.

DRL Agent. In our current design, we use a state-of-the-art DRL algorithm called *soft actor-critic* (SAC) [24]. However, we modify the original SAC to enhance its performance in NeuroTE by designing a customized method for experience replay, as discussed in this section. SAC is an off-policy algorithm based on maximum entropy reinforcement learning. The experimental results show that SAC consistently outperforms other DRL algorithms for continuous-action benchmarks, both in terms of learning speed and robustness [24]. Having said that, the design of NeuroTE is *independent* of the specific DRL algorithm applied. In other words, SAC can be easily replaced with any other DRL algorithm that can deal with continuous state and action spaces.

In SAC, the agent receives a bonus reward at each epoch proportional to the entropy of the policy at that epoch, which changes the RL problem to

$$\pi^* = \arg\max_{\pi} \mathbb{E}\Big[\textstyle\sum_{t=0}^{\infty} \gamma^t \big(r(s_t, a_t) + \alpha H(\pi(\cdot|s_t))\big)\Big], \tag{8}$$

where the hyper-parameter $\alpha \in [0, 1]$ balances exploitation and exploration, and affects the stochasticity of the optimal policy. SAC concurrently learns a policy π_ϕ and two Q-functions Q_{θ_1} and Q_{θ_2}. The Q-functions are learned by regressing to a single shared target. To avoid stability issues, the algorithm uses two target networks $Q_{\bar{\theta}_1}$ and $Q_{\bar{\theta}_2}$, which are obtained as exponentially moving averages of the Q-network parameters over the course of training. As with other DRL algorithms that use a deep neural network to approximate $Q(s, a)$, SAC stores observed transitions in a replay buffer $\mathscr{D}$ and uses experience replay when updating the Q-networks with respect to the following loss function

$$L(\theta_j, \mathscr{D}) = \mathop{\mathbb{E}}_{(s,a,r,s') \sim \mathscr{D}} \Big[\big(y(r, s') - Q_{\theta_j}(s, a)\big)^2\Big], \tag{9}$$

where, $j = 1, 2$, and the target $y(r, s')$ is given by

$$y(r, s') = r + \gamma\Big(\min_{j=1,2} Q_{\bar{\theta}_j}(s', a') - \alpha \log \pi_\phi(a'|s')\Big), \tag{10}$$

where a' is sampled from the current policy $\pi_\phi(\cdot|s')$. The policy should choose the action that maximizes the expected future return plus expected future entropy in each state, which results in maximizing the following function

$$J_\pi(\phi) = \mathop{\mathbb{E}}_{\substack{s \sim \mathscr{D} \\ \xi \sim \mathscr{N}}} \Big[\min_{j=1,2} Q_{\theta_j}(s, f_\phi(\xi, s)) - \alpha \log \pi_\phi(f_\phi(\xi, s)|s)\Big], \tag{11}$$

where $f_\phi(\xi, s)$ is a deterministic function obtained by reparametrizing the policy using a neural network transformation $a = f_\phi(\xi, s)$, where ξ is an input noise vector, sampled from a fixed distribution such as a spherical Gaussian. Finally, the optimal policy can be computed by performing gradient ascent over $J_\pi(\phi)$ with respect to policy parameter set ϕ.

Prioritized SAC. During training, model parameters are updated in several iterations, where in each iteration the parameters are updated using a mini-batch of data drawn from replay buffer $\mathscr{D}$. When drawing a mini-batch, a straightforward approach is to sample uniformly from the buffer, as in the original SAC algorithm. However, it has been shown that using a method called *prioritized experience replay*, which assigns a priority to each experience, results in significant improvement over uniform sampling [37]. This method has been used with several DRL algorithms including DQN [37], DDPG [38] and SAC [39]. In particular, the method proposed for SAC [39] considers the TD-error as the measure of priority, similar to the prioritization method proposed for DQN [37]. The TD-error of Q_{θ_j} for transition i sampled from $\mathscr{D}$ is given by

$$\delta_{ij} = y(r_i, s'_i) - Q_{\theta_j}(s_i, a_i)\,. \quad (12)$$

However, SAC is an actor-critic algorithm where in each update iteration, both actor and critic parameters are updated using transitions sampled from the replay buffer. The method proposed in [38] uses both TD-error and gradient of policy, but their method is specific to DDPG. We extend this method to enable prioritized experience replay in SAC. To our knowledge, this is the first paper that considers both critic and actor for prioritized experience replay in SAC. To update the policy network, a gradient step is taken with respect to $J_\pi(\phi)$. We can approximate the gradient of $J_\pi(\phi)$ with

$$\begin{aligned}\hat{\nabla} J_\pi(\phi) &= -\nabla_\phi \log \pi_\phi(a|s) \\ &\quad + \big(\nabla_a Q_\theta(s,a) - \alpha \nabla_a \log \pi_\phi(a|s)\big) \nabla_\phi f_\phi(\xi, s), \quad (13)\end{aligned}$$

where, we define $Q_\theta(s,a) = \min_{j=1,2} Q_{\theta_j}(s,a)$. In prioritized experience replay, the probability of sampling a particular transition i is proportional to its priority,

$$P(i) = p_i^\beta / \textstyle\sum_{k\in\mathscr{D}} p_k^\beta, \quad (14)$$

where $p_i > 0$ denotes the priority of transition i. The exponent β determines how much prioritization is used. In particular, setting $\beta = 0$ yields uniform sampling. Since SAC has two Q-networks, we define the average TD-error of transition i as

$$\delta_i = \frac{1}{2}(|\delta_{i1}| + |\delta_{i2}|), \quad (15)$$

and then compute the priority of transition i as

$$p_i = \delta_i + \lambda \big|\nabla_a Q_\theta(s_i, a_i) - \alpha \nabla_a \log \pi_\phi(a_i|s_i)\big| + \epsilon, \quad (16)$$

where, ϵ is a small positive constant to ensure all transitions are sampled with some probability and the second term represents the loss applied to the actor. Parameter $\lambda \in [0,1]$ is used to weight the contributions of each component. To account for the bias introduced due to prioritized sampling, updates to the network are weighted with importance sampling weight [37]. For each transition i, we first compute its importance sampling weights $\omega_i = \big(\frac{1}{|\mathscr{D}|}\frac{1}{P(i)}\big)^\eta$. Then, during the update of Q_{θ_j} network, we use $\omega_i\delta_{ij}$ instead of δ_{ij}. For stability reasons, we normalize weights by $1/\max_i \omega_i$ so that they only scale the update downwards. Also, as suggested in [37], we linearly anneal η from its initial value η_0 to 1 (at the end of learning). Typical suggested values for these parameters are as: $\beta = 0.6$, $\epsilon = 0.01$, and $\eta_0 = 0.4$. We also set $\lambda = 0.5$.

V. Evaluations

In this section, we present evaluation results using real traffic traces to demonstrate the benefits of using TEMs instead of TMs in traffic engineering. We also study the impact of alternative design choices on the performance of NeuroTE.

A. Methodology

We developed a flow-level network simulator and used the PyTorch platform [40] to implement neural networks and DRL algorithms. We measure link utilizations from the simulator and then use a neural network to map utilizations to delays. The neural network is trained based on real measurements obtained from our internal testbed, but otherwise is unknown

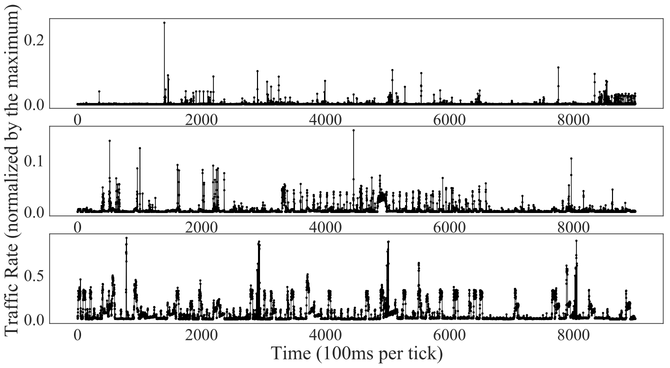


Fig. 3: Sample traffic traces used in evaluations. The figures show traffic rates for 3 randomly chosen flows out of 20 flows.

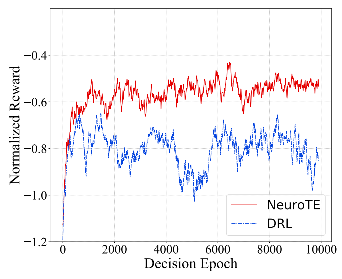
to traffic engineering algorithms. This provides a fairly accurate simulation platform for estimating end-to-end delay using captured traffic traces. All the neural network models are trained on a single server with an Intel Xeon Gold 6252N 2.30GHz CPU with 128GB memory and a Tesla P40 GPU.

Network Settings. We use two well-known network topologies for evaluations, namely the NSFNET (with 13 nodes) and ARPANET (with 29 nodes), both available in the Internet topology zoo [41]. For each topology, we assign traffic demands to $|\mathscr{F}| = 20$ randomly-selected origin-destination pairs. For each origin-destination pair, we use 3 shortest paths for routing traffic. The capacity of each link is set to 1 Gbps.

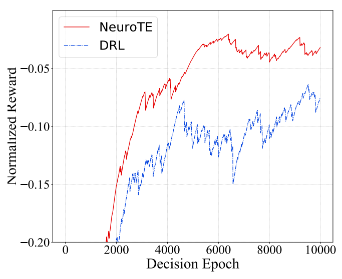
Traffic Traces. We use traffic traces collected from the WIDE backbone network [42]. Specifically, we use 15-minute traffic traces collected on April 12, 2017 to train all algorithms and then test them on traffic traces collected at the same time on the following day (*i.e.*, April 13, 2017). To generate traffic matrices, we randomly mapped the IP address of packets in the traffic traces to node indices in the simulated network topologies. The same origin-destination pairs with the same mapping are used consistently for evaluating different algorithms. To show the overall variability in our traffic traces, we have plotted the time series of traffic rates for three sample flows (*i.e.*, origin-destination pairs) in Fig. 3. As can be seen from the figure, there is significant variability in traffic demands in these traces, both within each flow and across different flows. We use these traces to compute traffic matrices over time by averaging flow rates over sampling time intervals. Each traffic matrix is computed over a time interval of 100 ms. By default, each control epoch is set to contain $K = 100$ traffic matrices.

Performance Metrics. We compare different approaches with respect to their: 1) training performance and 2) testing performance. For training performance, we are primarily considering the convergence of DRL algorithms and the quality of the state that the algorithm converges to. For testing performance, we consider the MFD achieved under each approach. Specifically, we plot the CDF of MFD to show the tail behavior of end-to-end delay with each TE approach. All CDF plots are in *logarithmic scale* to better show the differences.

Model Parameters. For the autoencoder, we have a three-layer fully-connected neural network as the encoder, which

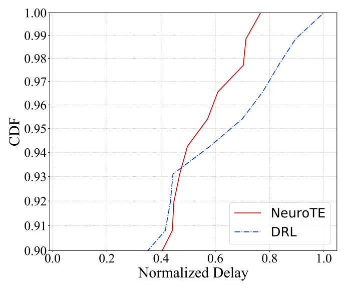


(a) Training curve in NSFNET.

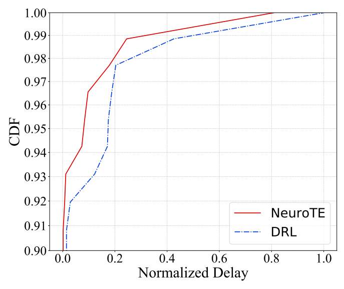


(b) Training curve in ARPANET.

Fig. 4: Convergence behavior of NeuroTE and DRL during training.



(a) MFD distribution in NSFNET.



(b) MFD distribution in ARPANET.

Fig. 5: Distribution of end-to-end delay under NeuroTE and DRL.

includes K input neurons, $2K$ and K neurons in the first and second layer, respectively, and utilizes the ReLU activation function [43]. The size of the output of the encoder (*i.e.*, the *encoding vector*) is set to $H = 50$, by default. The decoder is a three-layer neural network with the inverted structure of the encoder. After pre-training the autoencoder, the encoding vectors are fed into the DRL model. We use a two-layer neural network to serve as the actor network in the DRL model, which contains 256 neurons for both the first and second layer, with ReLU for activation. The output layer uses the Softmax activation function [35] to normalize the output actions. We use a two-layer neural network to model the critic network, which includes 256 neurons for both the first and second layer with the ReLU activation function. For the TM-based approaches, we use the mean of K sampled TMs over the past control epoch to construct the traffic matrix, and share the exact same DRL model as that of NeuroTE. The hyper parameters of the SAC algorithm include the learning rate as 0.0003, the batch size as 256, the discount factor for rewards as 0.99, and the entropy coefficient as 0.2. The reward is set as the negative of MFD, as discussed earlier, and is normalized by the maximum MFD in the figures.

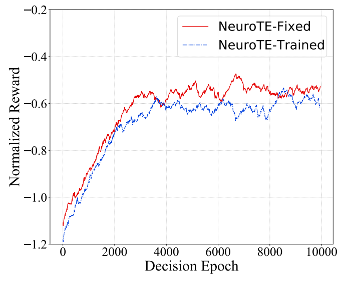
Baseline Approach. In addition to NeuroTE, we have also implemented an existing DRL-based approach for traffic engineering recently proposed in [3]. This approach uses the DDPG algorithm for continuous control. To have a fair comparison, we modify their approach so that it uses the exact same DRL algorithm used in NeuroTE. We refer to this modified approach as DRL, which represents the state-of-the-art in DRL-based traffic engineering. Extensive evaluation results presented in [3] show that the DRL-based approach outperforms other approaches based on Shortest Path, Load Balanced, and Network Utility Maximization approaches (please refer to [3] for details). Given space limitations, here we only present results comparing NeuroTE and DRL. *Any improvement achieved by NeuroTE compared to DRL are solely due to utilizing TEMs instead of TMs, as the control algorithm itself is exactly the same in both approaches.*

B. Results and Discussion

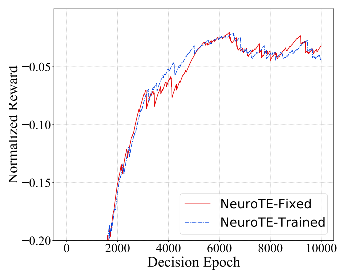
Convergence Benchmark. Figs. 4(a) and 4(b) show the training performance of NeuroTE and DRL in NSFNET and ARPANET topologies, respectively. Recall that ARPANET is a larger topology, as such the total reward will be higher in this topology because typical delay values will be smaller. From the figures, we see that not only NeuroTE converges to a stable state faster than DRL, but also the final converged state in NeuroTE results in a higher total reward compared to that of DRL. The faster convergence of NeuroTE is due to the ability of TEMs to construct better representation of network traffic compared to TMs. While there could be significant short-term variability in traffic demands, the distribution of demand does not change substantially over short periods of time. This stability of the space state allows the SAC algorithm to converge faster in NeuroTE.

Performance Benchmark. Figs. 5(a) and 5(b) show the performance of NeuroTE and DRL during the testing phase. This figure depicts the CDF of the normalized MFD achieved under each approach in logarithmic scale. Notice that smaller MFD values indicate smaller maximum flow delay, and thus better network performance. Again, we observe that using TEMs, as in NeuroTE, outperforms the TM-based approach for majority of end-to-end delay values. Only in the smaller NSFNET topology, DRL performs slightly better for small values of delay. We think with a larger network for the autoencoder part of the model, this gap could be closed. Nevertheless, we see that the maximum delay in NSFNET could be higher by almost 25% under DRL compared to NeuroTE.

How Beneficial is Dynamic Encoder Training? In NeuroTE, by default, the autoencoder is continuously tuned at runtime with respect to the reward achieved by the DRL algorithm, in an end-to-end manner. Obviously, this adds to the complexity of the system as well as the overall control overhead, as gradient information needs to be backpropagated to the encoder model at the edge of the network. In this experiment, our goal is to verify if indeed such an end-to-end continuous training is beneficial for our traffic engineering problem, at least in the setup that we have considered. Recall that the autoencoder is always pre-trained first, whether we tune it at runtime or not. The results are presented in Fig. 6. The Fixed encoder refers to the one that is only pre-trained, while Trained refers to the one that is updated at runtime. We can observe from this figure that there is no significant improvement in performance by using end-to-end training. We believe that this merely indicates that the DRL algorithm is not able to convey a sufficiently sensitive signal to the encoder for training. It may be possible to properly regularize the DRL objective to generate more useful signals for the encoder.

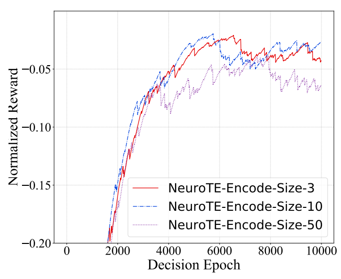


(a) Training curve in NSFNET.

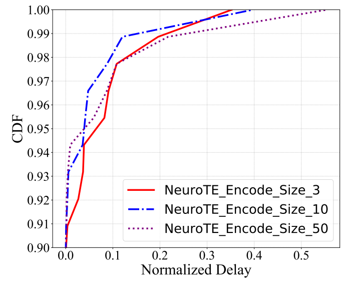


(b) Training curve in ARPANET.

Fig. 6: Effect of dynamic encoder training.



(a) Training curve in ARPANET.



(b) MFD distribution in ARPANET.

Fig. 7: Effect of different encoding vector sizes.

What is a Good size for Encoding Vectors? Fig. 7 shows the performance of NeuroTE with different encoding vector sizes for the Trained encoder on ARPANET topology. Results for Fixed trainer and NSFNET topology showed a similar trend, and thus omitted to save space. Interestingly, we only see marginal change in the performance by changing the size of the encoding vectors. Moreover, we can see that the best performance is achieved with a medium-sized encoding vector. Specifically, we see that setting the encoding vector size to 10 achieves the best performance, while setting the encoding vector to small or large values, namely 3 and 50, leads to lower performance. This indicates that a proper encoding vector size can achieve the best performance rather than naively using the largest encoding vector size, which results in higher computational and communication overhead. Recall that the encoding vectors are computed by the edge agents and then transmitted to the controller. Therefore, using smaller encoding vectors is desirable to minimize the communication overhead in the control plane of the network. Moreover, larger encoding vectors mean a larger state space for the DRL algorithm, which slows down the training.

Summary. Our experiments show that not only NeuroTE achieves better performance compared to DRL, thanks to using traffic encoding matrices, but also it is reasonably robust to various design choices. In particular, these results confirm that a traffic encoder that has been trained offline can be reliably used at runtime without requiring dynamic updates.

VI. Related Work

There is a large body of literature on traffic engineering and ML-based network control. Here, we only review some representative works that are most relevant to our work.

Optimization-based TE. Existing optimization-based TE approaches can be broadly classified into adaptive and online approaches. Adaptive approaches work in epochs and typically use past measurements to estimate a TM for the next epoch [1], [5]. These approaches have an open loop control and thus optimize for performance metrics that can be accurately described mathematically. In online approaches, traffic is monitored in real-time and the control algorithm reacts to instantaneous traffic changes [8]. Such approaches have a closed loop control and thus if network dynamics are slow enough compared to network RTT, for example in a datacenter [44]–[46], they can converge to an optimal network configuration. However, in a WAN environment with long RTT, these approaches can experience a large transient penalty [7].

ML-based TE. Machine learning has been successfully applied to a variety of problems in the networking area (see [47] and references therein). Recent works on the application of machine learning in network control can be found in [48]–[50]. The most relevant works to ours are [3], [15], [16], which apply DRL to TE. Specifically, Xu *et al.* [3] apply the DDPG algorithm [27] to the problem of delay-sensitive TE. Using simulations, they show that a DRL-based approach can outperform optimization-based approaches when the performance objective is the average end-to-end delay. Zhang *et al.* [16] use a DRL algorithm to identify the so-called critical flows and then selectively reroute them to balance link utilization in an SDN network. However, the actual TE controller in their work is based on a linear programming model. The authors in [15] consider TE with end-to-end delay, but their focus is on distributed TE based on multi-agent RL. In contrast, our work considers an SDN network where centralized control allows a simpler controller algorithm design. Orthogonal to our work on TE, there are also several recent works on using DRL for congestion control on end systems [49], [51]–[53].

All these works, however, rely on a traditional traffic matrix for traffic engineering. In contrast, we propose using traffic encoding matrices in an end-to-end learning architecture. A traffic encoding matrix provides a better representation of distributional demand information, which allows the DRL-based controller to achieve better performance in terms of training convergence and end-to-end delay.

VII. Conclusion

In this paper, we presented the design and evaluation of NeuroTE, a general DRL-based framework for traffic engineering. The design of NeuroTE is based on employing machine learning for network control in an end-to-end manner; not just for decision making but also for learning traffic demand representations. Using real traffic traces, we showed that NeuroTE results in improved network performance and faster convergence compared to existing TM-based DRL approaches. We believe that the core idea of NeuroTE, *i.e.*, end-to-end learning of demand representation and decision making, has a lot of promise and can be applied to other network control problems such as dynamic resource allocation problems. A worthwhile future work is to study NeuroTE over larger network topologies and with more diverse traffic patterns.

References

[1] C.-Y. Hong, S. Kandula, R. Mahajan, M. Zhang, V. Gill, M. Nanduri, and R. Wattenhofer, "Achieving high utilization with software-driven WAN," in *Proc. ACM SIGCOMM*, Aug. 2013.

[2] S. Jain *et al.*, "B4: Experience with a globally deployed software defined WAN," in *Proc. ACM SIGCOMM*, Aug. 2013.

[3] Z. Xu *et al.*, "Experience-driven networking: A deep reinforcement learning based approach," in *Proc. IEEE INFOCOM*, Apr. 2018.

[4] P. Kumar *et al.*, "Semi-oblivious traffic engineering: The road not taken," in *Proc. USENIX NSDI*, Apr. 2018.

[5] C. Zhang, Y. Liu, W. Gong, J. Kurose, R. Moll, and D. Towsley, "On optimal routing with multiple traffic matrices," in *Proc. IEEE INFOCOM*, Apr. 2005.

[6] V. Jalaparti, I. Bliznets, S. Kandula, B. Lucier, and I. Menache, "Dynamic pricing and traffic engineering for timely inter-datacenter transfers," in *Proc. ACM SIGCOMM*, Aug. 2016.

[7] H. Wang, H. Xie, L. Qiu, Y. R. Yang, Y. Zhang, and A. Greenberg, "COPE: Traffic engineering in dynamic networks," in *Proc. ACM SIGCOMM*, Aug. 2006.

[8] A. Elwalid, C. Jin, S. Low, and I. Widjaja, "MATE: MPLS adaptive traffic engineering," in *Proc. IEEE INFOCOM*, Apr. 2001.

[9] A. Medina, N. Taft, K. Salamatian, S. Bhattacharyya, and C. Diot, "Traffic matrix estimation: Existing techniques and new directions," in *Proc. ACM SIGCOMM*, Aug. 2002.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, *Introduction to Algorithms*, 3rd ed. MIT Press and McGraw–Hill, 2009.

[11] A. Bayati, V. Asghari, K. Nguyen, and M. Cheriet, "Gaussian process regression based traffic modeling and prediction in high-speed networks," in *Proc. IEEE GLOBECOM*, Dec. 2016.

[12] N. Ramakrishnan and T. Soni, "Network traffic prediction using recurrent neural networks," in *Proc. IEEE ICMLA*, Dec. 2018.

[13] A. Azzouni and G. Pujolle, "NeuTM: A neural network-based framework for traffic matrix prediction in SDN," in *Proc. IEEE NOMS*, Apr. 2018.

[14] M. Dolati and M. Ghaderi, "Achieving high utilization with software-driven WAN," in *Proc. IEEE IWQoS*, Aug. 2019.

[15] P. W. Pinyarash Pinyoanuntapong, Minwoo Lee, "Delay-optimal traffic engineering through multi-agent reinforcement learning," in *Proc. IEEE INFOCOM, NI Workshop*, 2019.

[16] J. Zhang, M. Ye, Z. Guo, C.-Y. Yen, and H. J. Chao, "CFR-RL: Traffic engineering with reinforcement learning in SDN," *IEEE J. Sel. Areas Commun.*, vol. 10, no. 38, 2020.

[17] A. Tootoonchian, M. Ghobadi, and Y. Ganjali, "OpenTM: Traffic matrix estimator for OpenFlow networks," in *Proc. PAM*, Apr. 2010.

[18] What is SD-WAN?, accessed Oct. 14, 2021. [Online]. Available: https://www.cisco.com/c/en_ca/solutions/enterprise-networks/sd-wan/what-is-sd-wan.html

[19] CloudEngine 16800, accessed Oct. 14, 2021. [Online]. Available: http://e.huawei.com/topic/cloud-engine2019/en/index.html

[20] NVIDIA BlueFiled Data Processing Unit, accessed Oct. 14, 2021. [Online]. Available: https://www.nvidia.com/en-us/networking/products/data-processing-unit/

[21] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*, 3rd ed. MIT Press, 2018.

[22] V. Minh *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, 2015.

[23] Y. Xiao, J. Liu, J. Wu, and N. Ansari, "Leveraging deep reinforcement learning for traffic engineering: A survey," *IEEE Commun. Surveys Tuts.*, vol. 4, no. 23, 2021.

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

[25] N. C. Luong, D. T. Hoang, S. Gong, D. Niyato, P. Wang, Y.-C. Liang, and D. I. Kim, "Applications of deep reinforcement learning in communications and networking: A survey," *IEEE Commun. Surveys Tuts.*, vol. 21, no. 4, 2019.

[26] S. Levine, "Actor-critic algorithms," accessed Oct. 14, 2021. [Online]. Available: http://rail.eecs.berkeley.edu/deeprlcourse-fa17/f17docs/lecture_5_actor_critic_pdf

[27] 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," in *Proc. ICLR*, May 2016.

[28] C. Raiciu *et al.*, "How hard can it be? Designing and implementing a deployable multipath TCP," in *Proc. NSENIX NDSI*, Apr. 2012.

[29] X. Lin and N. Shroff, "Utility maximization for communication networks with multipath routing," *IEEE Trans. Autom. Control*, vol. 5, no. 51, 2006.

[30] K. Winstein and H. Balakrishnan, "TCP ex Machina: Computer-generated congestion control," in *Proc. ACM SIGCOMM*, Aug. 2013.

[31] R. Lowe, Y. Wu, A. Tamar, J. Harb, P. Abbeel, and I. Mordatch, "Multi-agent actor-critic for mixed cooperative-competitive environments," in *Proc. ACM NIPS*, Dec. 2017.

[32] L. D. Vito, S. Rapuano, and L. Tomaciello, "One-way delay measurement: State of the art," *IEEE Trans. Instrum. Meas.*, vol. 12, no. 57, 2008.

[33] Accedian Networks, "One-way delay measurement techniques," accessed Oct. 14, 2021. [Online]. Available: https://accedian.com/wp-content/uploads/2015/05/One-WayDelayMeasurementTechniques-AccedianWhitePaper.pdf

[34] Viavi Solutions, "One-way delay and jitter measurement," accessed Oct. 14, 2021. [Online]. Available: https://www.viavisolutions.com/ja-jp/literature/one-way-delay-jitter-measurement-application-notes-en.pdf

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

[36] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, 1997.

[37] T. Schaul, J. Quan, I. Antonoglou, and D. Silver, "Prioritized experience replay," in *Proc. ICLR*, Oct. 2016.

[38] Y. Hou, L. Liu, Q. Wei, X. Xu, and C. Chen, "A novel DDPG method with prioritized experience replay," in *Proc. IEEE SMC*, Oct. 2017.

[39] C. Wang and K. Ross, "Boosting soft actor-critic: Emphasizing recent experience without forgetting the past," Jun. 2019, accessed Oct. 14, 2021. [Online]. Available: https://arxiv.org/pdf/1906.04009.pdf

[40] PyTorch. [Online]. Available: https://pytorch.org/

[41] S. Knight, H. Nguyen, N. Falkner, R. Bowden, and M. Roughan, "The internet topology zoo," *IEEE J. Sel. Areas Commun.*, vol. 29, no. 9, 2011.

[42] *WIDE Traffic Traces*, 2018, http://mawi.wide.ad.jp/mawi/.

[43] X. Glorot, A. Bordes, and Y. Bengio, "Deep sparse rectifier neural networks," in *Proc. AISTATS*, Apr. 2011.

[44] M. Alizadeh *et al.*, "CONGA: Distributed congestion-aware load balancing for datacenters," in *Proc. ACM SIGCOMM*, Aug. 2014.

[45] R. Mittal *et al.*, "TIMELY: RTT-based congestion control for the datacenter," in *Proc. ACM SIGCOMM*, Aug. 2015.

[46] K. Nagaraj, D. Bharadia, H. Mao, S. Chinchali, M. Alizadeh, and S. Katti, "NUMFabric: Fast and flexible bandwidth allocation in datacenters," in *Proc. ACM SIGCOMM*, Aug. 2016.

[47] M. Wang, Y. Cui, X. Wang, S. Xiao, and J. Jiang, "Machine learning for networking: Workflow, advances and opportunities," *IEEE Netw.*, vol. 32, no. 2, 2018.

[48] H. Mao, M. Alizadeh, I. Menachey, and S. Kandula, "Resource management with deep reinforcement learning," in *Proc. ACM HotNets*, Nov. 2016.

[49] Z. Xu, J. Tang, C. Yin, Y. Wang, and G. Xue, "Experience-driven congestion control: When multi-path TCP meets deep reinforcement learning," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 6, 2019.

[50] H. Mao, M. Schwarzkopf, S. B. Venkatakrishnan, Z. Meng, and M. Alizadeh, "Learning scheduling algorithms for data processing clusters," in *Proc. ACM SIGCOMM*, Aug. 2019.

[51] N. Jay, N. H. Rotman, B. Godfrey, M. Schapira, and A. Tamar, "A deep reinforcement learning perspective on Internet congestion control," in *Proc. ICML*, 2019.

[52] S. Abbasloo, C.-Y. Yen, and H. J. Chao, "Classic meets modern: a pragmatic learning-based congestion control for the Internet," in *Proc. ACM SIGCOMM*, 2020.

[53] H. Jianga, Q. Li, Y. Jiang, G. Shen, R. Sinnotte, C. Tian, and M. Xu, "When machine learning meets congestion control: A survey and comparison," *Computer Networks*, vol. 192, 2021.