Automated Traffic Engineering in SDWAN: Beyond Reinforcement Learning

Libin Liu[†§] Li Chen[‡] Hong Xu[§] Hua Shao[*]
[†]Tencent [‡]Huawei [§]City University of Hong Kong [*]Tsinghua University
lesbinliu@tencent.com, Chen.li7@huawei.com, henry.xu@cityu.edu.hk, shaoh18@mails.tsinghua.edu.cn

***Abstract*—Traffic engineering (TE) is a critical and difficult problem that involves assigning traffic with various requirements to paths with different constraints. Recently, machine learning algorithms, especially deep neural networks (DNN), are applied to TE, yet they all assume that the network is a black box, limiting them to only model-free reinforcement learning (RL) algorithms. In this paper, we introduce *differentiable programming* to TE, and show that the network environment can be sufficiently modeled for TE optimization. Specifically, we design a fully-differentiable network environment, ∂NE, that can be directly integrated into any DNN models. With ∂NE, we can differentiate with respect to control parameters, and directly evaluate gradients between actions and states to facilitate gradient descent based training of DNN models. We show with a proof-of-concept prototype that ∂NE accelerates DNN training for TE by 228$\times$ and achieves higher scalability compared to existing network simulators. Most importantly, dNE opens up the possibility to apply arbitrary deep learning models to TE beyond RL.**

I. Introduction

For cloud service providers with global presence, the wide area network (WAN) is one of the most important pieces of infrastructure, connecting its global datacenters and guaranteeing the delivery of many planet-scale applications. WANs are privately owned and usually centrally controlled following the software defined networking paradigm, and are referred to SDWANs hereafter. Traffic in SDWANs is typically of high volume and still rapidly growing [7], [9]. Therefore despite much prior work, traffic engineering (TE) continues to draw attention from both academia and industry as an important means to increase performance and reduce costs.

TE is inherently difficult as it involves mapping traffic with different demands and priorities to network paths with varying constraints and costs. As an offline problem, TE can be cast as a mixed integer programming problem which is NP-hard in general [13]. In practice, TE systems need to make decisions online, responding to various network events such as link down or flapping, which adds to the difficulty of the problem. Inspired by the prominent success of reinforcement learning (RL) with deep neural networks (DNN) in complex online decision making tasks [26], DNN-based RL (DRL) algorithms have also been introduced to some traffic optimization tasks in networking, such as adaptive bitrate control in video streaming [19], traffic scheduling [3], and TE [28], [34].

Specifically, for TE, Stampa et al. [28] and Xu et al. [34] both adopt actor-critic based DRL algorithms to solve TE in SDWAN and are our closest related work. They use model-free algorithms and rely on simulated network environments (ns3 [24] and OMNet++ [31], respectively) to train their models. These simulated environments are poorly suited for deep learning because they are not natively differentiable and cannot be trained by classical gradient based optimization methods. Thus the authors resort to approximating the state-action or action-value functions using some differentiable DNN models, which introduces a host of problems. DNN as a non-linear function approximator is known to be unstable or even diverge during training [8], [20], which prolongs the training time for DRL based TE solutions. Large DRL models also require significant tuning efforts especially with techniques like experience replay and target network to improve its robustness [16].

Why are we stuck with RL in the first place? We observe that these DRL approaches share the same underlying assumption that, the network as a whole cannot be explicitly modeled, and thus must be treated as a black box with model-free DRL algorithms. While this is valid in some cases where the environment is stochastic (e.g. flows in datacenter networks [3]) or uncontrollable (e.g. video streaming over Internet [19]), we believe that the SDWAN environment is well-understood and can be adequately modeled for the purpose of TE. In a SDWAN with a logically centralized controller, the traffic demands and tunnel-path assignments can unambiguously determine the next state of the network, i.e. available bandwidth for different traffic classes on each link, as well as the TE metrics, such as latency (hop count), path length, and link utilization.

We build ∂NE, a fully-differentiable network environment, as a constructive proof of our argument above. ∂NE takes as input a set of matrices describing the network topology, tunnel traffic demands, and tunnel-path assignments, computes the resulting network state, and obtains user-defined performance metrics such as the maximum link utilization. More interestingly, leveraging the automatic differentiation capability of many deep learning frameworks [33], ∂NE directly outputs the gradients for the parameters of the DNN model used to solve TE with respect to the TE performance. Essentially ∂NE allows the DNN model for TE to be trained end-to-end with standard gradient based methods that are widely used in deep learning.

The key benefit of ∂NE is that, by following differentiable programming principles [27], it enables rapid adoption of recent and future advances in machine learning for TE in SDWAN. We implement ∂NE as a differentiable "layer" using

popular deep learning frameworks, so that it can be easily integrated into any deep learning models. Both model-based and model-free DRL algorithms can directly use ∂NE as the environment and apply actions to it, without the need to approximate the dynamics of the network as if it is a black box. In addition, ∂NE supports arbitrary deep learning models beyond RL, such as recurrent neural networks (RNN) [25], differentiable neural computers [6], etc., thereby greatly expanding the solution space with many new approaches and promising potential gains. As a concrete example, we implement a simple RNN-based TE algorithm in our PyTorch-based ∂NE prototype, and experiments show that it achieves 7.0% higher throughput and 89.8% lower congestion loss than state-of-the-art DRL proposal for TE [34] in §IV.

Lastly, we concede that, in practice, various random events in the network may not be properly modeled with ∂NE, such as random packet drops and transient congestion [18]. Yet we believe that, even for DRL models that intend to deal with such dynamics, ∂NE is still helpful as a pre-deployment training platform thanks to its unique characteristics in differentiability, training speed, and scalability in contrast to existing network simulators [12], [24], [31].

We encourage the community to think beyond RL and explore new design space with ∂NE in applying machine learning techniques to TE and possibly other resource allocation problems in networking.

II. Background & Motivation

We start by providing background on TE and then presenting the motivation for our design of ∂NE.

A. Background on SDWAN TE

SDWAN [7], [9] leverages a logically centralized controller to make TE decisions with a global view. A SDWAN TE algorithm produces a solution that maps tunnels to the paths in the network, while maintaining network-wide objectives, e.g. load balancing and high link utilization. Following the characterization by Kumar et al. [13], a TE algorithm makes two choices: (1) determine the paths to use between all ingress-egress pairs, and (2) determine the rates to allocate traffic demands on the paths. Any algorithm in TE systems can be analyzed with respect to these choices.

Recently, due to the success of deep learning in solving complex online control problems [3], DNN-based RL algorithms are introduced to TE in WAN. These algorithms are either model-based [2], or model-free [28], [34]. For model-based algorithms, they are online algorithms trying to fill (and then exploit) a table containing <state, action, reward> tuples. Therefore, they have sample efficiency problem [8], leading to poor performance in unseen network states. For model-free algorithms [28], [34], they both use off-policy, actor-critic, deterministic policy gradient algorithm [16] that interacts with the network environment. In particular, Stampa et al. [28] does not explore the full capability of SDN (e.g. optimizing per-tunnel placement directly). In terms of rate allocation, for DRL-TE [34], it is unclear if DRL has advantages over traditional optimization approaches, e.g. linear programing [7], [9].

B. Motivation

The above DNN-based TE approaches all assume that the network cannot be modeled and must be treated as a black-box. This assumption has two implications:

- It limits them to only RL algorithms, and have to rely on the RL agents to learn and approximate the dynamics of the system, such as state transition, action-value function, and state-value function. Therefore, they are subjected to limitations of RL algorithms, e.g. sample inefficiency [8] and variance [20]. On the other hand, in many tasks, other DNN-based methods consistently outperform RL ones, such as trajectory optimization in robotics control [30], and Monte Carlo Tree Search in Atari games [21]. We believe adopting the "black-box" assumption prevents us from other promising DNN-based algorithms in TE.
- This assumption also forces them to rely on network simulators (e.g. OMNet++ and ns3) or real environments, thus limiting the training speed. DNN models require a large amount of training samples. Thus, it is slow to train DNN models on a traditional discrete event simulator, and convergence for DRL models is hard to achieve. Relying on simulators also limits the scalability. As we will show in §IV-C, the time between applying an action to getting a feedback is usually >200ms on a traditional simulator with more than 100 nodes in the network.

We believe, particularly for SDWAN TE, network is not a black box: the current network state and a set of tunnel-path assignments can unambiguously determine the next state of the network. TE metrics, such as latency, path length, link utilization, can also be directly calculated (§III). What is lacking, however, is a mechanism to efficiently optimize the DNN models in algorithms. We find that some topics, such as physics engines in robotics [4] and ray tracing [14], are in the same situation where the environment can be specified explicitly. Differentiable programming as we notice has been tremendously useful for them [4], [14], [15].

C. Differentiable Programming & ∂NE

Differentiable programming is a new programming paradigm in which the software is programmed by assembling networks of parameterized functional blocks. Due to such construction, the software can be differentiated throughout, usually via automatic differentiation (AD) frameworks [1], [32], [33]. This allows direct gradient-based training and optimization of parameters in the program, often via gradient descent. Software following this paradigm is thus data-driven, and can be trained end-to-end with input-output tuples, or optimized with respect to an objective function.

Following this paradigm, we design ∂NE to be a fully-differentiable network environment, which can be used as a "layer" embedded directly into a deep learning model. A DNN based TE model can then be trained with accurate gradients for current control parameters in the DNN. Both

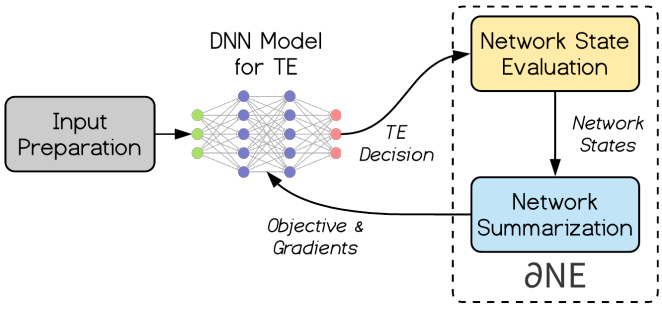

Fig. 1: ∂NE Design

model-based and model-free DRL algorithms can directly use ∂NE as an environment. It is also important to clarify that, DRL algorithms using ∂NE are not necessarily "model-based", because a model-based DRL agent still needs to learn the transition dynamics of the environment on its own. With ∂NE, there is no longer a need to learn such dynamics: we can simply run the entire simulation of current actions on ∂NE, obtain corresponding performance results, as well as the gradients for training improvements.

Most importantly, ∂NE can readily support arbitrary deep learning models beyond RL, such as recurrent neural networks [25], differentiable neural computers [6], etc. In other words, the true value of ∂NE is that it allows rapid adoption of recent and future advances in machine learning beyond RL in SDWAN TE.

III. ∂NE Design & Implementation

We now present the design of ∂NE and a preliminary prototype based on PyTorch.

A. Design & Work Flow

∂NE is designed as a "layer" that can be readily incorporated in any deep learning models. Inside the ∂NE layer, as shown in Fig. 1, there are two stages: network evaluation, and network summarization. The entire work flow is as follows:

1) We provide a set of input preparation utilities for the developer of the TE model to gather the necessary inputs, including tunnel requirements, and network description (physical connectivity and link bandwidth).
2) The TE model provides a TE decision, which is obtained by the Network State Evaluation stage. It computes the network state given the input.
3) Then, in the Network Summarization stage, user-defined performance objectives (maximum link utilization, average latency/hop-count, etc.) are calculated given the network state.
4) Using AD, the gradient with respect to the current control parameters of the TE model is obtained given the current summarization of the network, and then used for training with gradient descent methods.

B. Network Model

We proceed to describe our network model in ∂NE.

First we consider the traffic. We use T to denote the set of tunnels in the network. A tunnel in this paper is defined as the aggregated traffic of a particular traffic class between an ingress-egress router pair. We note that our definition of tunnel is usually referred as a "demand" or "flow" in some literature [7], [13], [17], [18], wherein a tunnel actually refers to one established path or a set of paths between the ingress and egress routers. Thus they only perform rate allocation on these established paths. Our choice of definition is different, because we want to model both the path finding process and the rate allocation process in TE.

In the current design ∂NE considers six types of requirements for tunnels expressed as a set of matrices:

- Source S: a $|T|\times 1$ vector where $S[t]$ contains the index of the source vertex of the tunnel $t\in T$.
- Destination D: a $|T|\times 1$ vector where $D[t]$ contains the index of the destination vertex of the tunnel $t\in T$.
- Bandwidth requirements B: a $|T|\times 1$ vector where $b[t](\geq 0)$ is the bandwidth demand of tunnel $t\in T$.
- Tunnel class C: a $|T|\times|P|$ matrix where P is the set of priorities (or traffic classes) enabled in the network. Here $C[t][p]$ is 1 if the priority for tunnel t is p, and 0 otherwise. Each tunnel can only have one priority ($\sum_p C[t][p]=1$). We also assume strict priority queueing on all routers.
- Latency requirements L: a $|T|\times 1$ vector where $L[t](\geq 0)$ denotes the maximum total latency acceptable to tunnel $t\in T$.
- Cost requirements Z: a $|T|\times 1$ vector where $Z[t](\geq 0)$ denotes the maximum total cost acceptable to tunnel $t\in T$.

Next we consider the network. The network is a directed graph $G=(E,V)$, where $E=\{e\}$ is the set of directed edges and $V=\{v\}$ the set of vertices. The network is characterized by the following attributes:

- Link capacity N: a $|E|\times 1$ vector where $N[e]$ contains the capacity of an edge e which can be either constant or dynamic.
- Measured latency M: a $|E|\times 1$ vector where $M[e]$ denotes the measured latency of an edge e which can be constant or dynamic.
- Link cost K: a $|E|\times 1$ vector where $K[e]$ contains the IGP metric of edge e in the network.

The TE algorithm in ∂NE aims to solve a tunnel placement problem and produce a tunnel-path assignment decision A as a $|T|\times|E|$ matrix, where $A[t][e]$ is either 0 or 1, denoting whether tunnel t takes edge e in its path(s).

The Network State Evaluation stage checks if all the input matrices follow the above description.

C. Network Summarization

In the network summarization stage, user-defined objective functions are computed given network descriptions (N,M,K), tunnel requirements (S,D,B,C,L,Z), and tunnel assignment (A). We have implemented a few frequently used objectives and constraints in TE, as follows:

Latency functions:

- `invalid_lat`(A,M,L): Latency constraint validation function tests if the sum of latencies along a tunnel's path

exceeds the tunnel's latency requirement. It returns 0 if all latency constraints are met.

- `max_lat`(A,M) obtains the maximum latency of all tunnels, and `avg_lat`(A,M) obtains the average.

Bandwidth functions:

- `invalid_bw`(A,B,N): Bandwidth constraint validation function tests if the sum of bandwidth requirements on every edge exceeds its capacity.
- `max_bw`(A,B,N) obtains the maximum bandwidth of all tunnels, and `avg_bw`(A,B,N) obtains the average.

Cost functions:

- `invalid_cost`(A,Z,K): Cost constraint validation function tests if the sum of costs along a tunnel's path exceeds the tunnel's cost budget requirement.
- `max_cost`(A,Z) obtains the maximum cost among all tunnels, and `avg_cost`(A,Z) obtains the average.

Using these functions, we can readily define common TE objectives such as minimization of maximum link utilization, minimization of average latency, etc. Users can also define new objectives by combining these functions or by adding new functions to the network summarization stage. However, to ensure that the user-defined functions and objectives are differentiable, we stipulate that the functions should only use inputs and outputs from the previous two stages. With the help of AD tools [29], we can obtain gradients with respect to the control parameters directly.

D. Implementation

We implemented a simple prototype of ∂NE with PyTorch [22] leveraging its AD support [23]. Following the design above, ∂NE can also be implemented using other frameworks [1], [33]. The current version of ∂NE is implemented as a layer that can be embedded into any neural networks on PyTorch. To use ∂NE, a model should initialize a ∂NE instance as one of its layers, supply the tensors in ∂NE's network model (§III-B) with values, and provide (or define) objective functions in the network summarization stage. Then it can be trained and optimized as any other models on PyTorch.

IV. Evaluation

In this section we evaluate ∂NE using experiments. We are mainly interested in answering three questions: (1) Can ∂NE easily enable DNN-based TE algorithms beyond RL? (2) Can ∂NE accelerate training of DNN-based TE algorithms? (3) Can ∂NE scale to support large SDWAN networks?

Summary of results:

- **Usability**. Deep learning models using PyTorch can incorporate ∂NE natively. To show this, we implement the DRL-TE [34] model, and a novel LSTM-based [5] model, LSTM-TE, on ∂NE. We find ∂NE easy to use: it takes only ∼1 day to implement the LSTM-TE, and ∼3 hours to implement DRL-TE.
- **Performance**. Our implementation of LSTM-TE outperforms state-of-the-art DRL-TE [34] on the Abilene topology for both throughput (7.0% higher) and congestion loss (89.8% lower). LSTM-TE also shows comparable performance to TE algorithms using multipath routing, such as K-Shortest-Path + Multi-Commodity-Flow used in SWAN [7].
- **Training speed**. Compared to traditional network simulators, e.g. OMNet++, ∂NE achieves much faster training (>228×).
- **Scalability**. Traditional network simulators become slower with larger network sizes. We show that network size does not affect the evaluation and training speed of ∂NE.

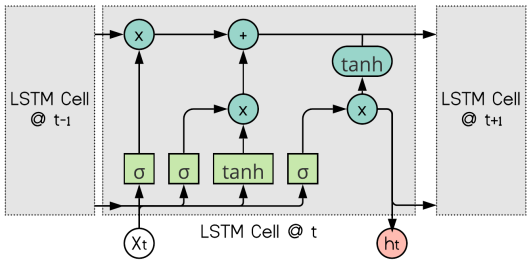

Fig. 2: Long-Short-Term-Memory Network

A. Usability of ∂NE

We demonstrate ∂NE's usability by implementing DNN-based models for TE. First, we propose and implement a new DNN-based model for TE called Long-Short-Term-Memeory TE (LSTM-TE) that does not use RL methods. Although LSTMs achieve superior performance in many problem domains [10], to the best of our knowledge, they have not been used in TE tasks. We describe the model in the following.

LSTM Cell. For each element in the input sequence, an LSTM cell (Fig. 2) computes the following:

$$
\begin{aligned}
i_t &= \sigma(W_{ii}x_t + b_{ii} + W_{hi}h_{(t-1)} + b_{hi}),\\
f_t &= \sigma(W_{if}x_t + b_{if} + W_{hf}h_{(t-1)} + b_{hf}),\\
g_t &= \tanh(W_{ig}x_t + b_{ig} + W_{hg}h_{(t-1)} + b_{hg}),\\
o_t &= \sigma(W_{io}x_t + b_{io} + W_{ho}h_{(t-1)} + b_{ho}),\\
c_t &= f_t \star c_{(t-1)} + i_t \star g_t,\\
h_t &= o_t \star \tanh(c_t).
\end{aligned}
$$

Here h_t is the hidden state at time t, c_t is the cell state at time t, x_t is the input at time t, $h_{(t-1)}$ is the hidden state of the layer at time $t-1$ or the initial hidden state at time 0, and i_t,f_t,g_t,o_t are the input, forget, cell, and output gates, respectively. $W_{...}$ are the weights of the respective DNNs in the rectangle nodes in the computation graph in Fig. 2, and $b_{...}$ are their respective biases. σ is the sigmoid function, and $\star$ is the Hadamard product.

Model Input. We concatenate the following vectors to form an input vector X_t for tunnel t:

- Tunnel source, a $|V|\times 1$ vector with one-hot encoding.
- Tunnel destination, a $|V|\times 1$ vector with one-hot encoding.
- Tunnel bandwidth requirement, a scalar.
- Tunnel class, a $|P|\times 1$ vector with one-hot encoding, where P is the set of traffic classes.
- Available k shortest paths for the tunnel t which is a $k\times|E|$ vector. In our current design, we set k=4.

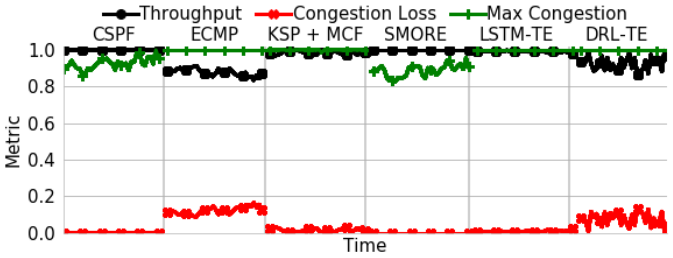

Fig. 3: LSTM-TE's performance on the Abilene topology over 35-hour time-series traffic demand matrices against traditional TE schemes. For each traffic demand matrix, the average network load is 0.8.

The first four vectors are obtained by transforming the tunnel requirements S,D,B,C defined in §III-B.

Objective Function. We use minimization of maximum link utilization as the objective function. The loss function for training is defined as the difference between the objective function values of two consecutive epochs.

LSTM-TE Operations. In LSTM-TE, we consider the network environment as a discrete-time dynamic system. An LSTM cell is integrated with ∂NE following the steps outlined in §III-A. Specifically, at the beginning of each training epoch, for an input containing $|T|$ tunnels, we feed the LSTM cell with a vector containing information of one tunnel in one time step. After all the tunnels are processed in $|T|$ time steps, we obtain the final hidden state as the output A of the current training epoch, which is a $|T|\times|E|$ matrix (assigning tunnels to edges). We then feed A into ∂NE to perform the network evaluation and summarization stages. Finally, we obtain the objective function from ∂NE, compute the loss, and perform back-propagation to update the weights $W_{...}$ in the cell. This finishes one epoch of training.

Summary. Overall, implementing a new DNN-model is easy on ∂NE. It takes only one day to complete the LSTM-TE. We also implemented the DRL-TE algorithm following [34], and replace its OMNet++ environment with ∂NE. This takes only ∼3 hours.

B. Performance of LSTM-TE

We proceed to look at the performance of LSTM-TE.

Training LSTM-TE. During the training phase, we randomly generate traffic demands with various average link loads (0.2, 0.4, 0.6, and 0.8) for all tunnels on the Abilene topology, and for each load level we run 1000 epochs.

Schemes Compared. To demonstrate the efficiency of LSTM-TE, we compare it against a set of existing TE solutions, including CSPF, ECMP, KSP+MCF, SMORE [13], and DRL-TE which we implemented in §IV-A following [34].

Settings. Except for DRL-TE and LSTM-TE, we run all other TE schemes on YATES [12] on the Abilene topology using the traffic demand matrices provided in YATES's data repository [11] with the average network load scaled to 0.8. We run DRL-TE and LSTM-TE on top of ∂NE with identical traffic demand matrices and topology. For all schemes, we set the path budget k to be 4, which is the same as in B4 [9] and SMORE [13].

Results. Fig. 3 shows throughput normalized to total demand, maximum congestion (fractional link utilization), and congestion loss (normalized traffic dropped due to congestion) over 35 hours of traffic traces. We observe that LSTM-TE performs better than ECMP and KSP+MCF in terms of all metrics and is essentially on par with the best non-DNN schemes, CSPF and SMORE, in terms of throughput and congestion loss. Compared to ECMP, LSTM-TE improves throughput by 13.1% and reduces congestion loss by 93.9% on average. For KSP+MCF, the gains are 0.8% and 51.8%, respectively. We can also see that it performs worse than CSPF and SMORE in terms of maximum congestion. This is because CSPF and SMORE select the paths and subsequently distribute the traffic over the corresponding paths, while current ∂NE design only selects a single path based on the tunnel demands, without performing any rate adaptation. We consider multipath support and rate adaptation as future work for ∂NE and LSTM-TE. Despite the limitation of single path routing, LSTM-TE still outperforms DRL-TE on throughput and congestion loss (7.0% and 89.8% gains on average). This demonstrates the value of ∂NE: we can go beyond RL and readily reap the benefits of arbitrary DNN-based algorithms.

C. Training Speed

We investigate the training speed of the TE simulator defined as the response time between taking an TE action and making an observation of the effect. A faster training speed is beneficial, due to 1) faster rate of experience generation for control tasks (e.g. state-action-reward triples for DRL algorithms), which potentially reduces variance and accelerates convergence; and 2) faster iteration time for optimization tasks (e.g. RNN-like model above), which reduces total training time. We compare ∂NE against a traditional network simulator used in [28], OMNet++ (v5.1) [31]. We build a random topology of 100 nodes and 500 1Gbps links on both ∂NE and OMNet++. We create 100 tunnels with 10Kbps bandwidth requests, and place them in the network as evenly as possible. We then change placement of a random tunnel, and observe the change of bandwidth usages in corresponding links. We repeat the procedure 1000 times and measure the time between setting a new tunnel placement and observing the change. We find that, for OMNet++ the average (p99) latency is 228.3ms (1.594s). But for ∂NE, the change is effective almost immediately (p99 latency is 0.977ms), because only one matrix multiplication is needed. This shows that ∂NE is much more responsive than traditional network simulators, and can achieve $>228\times$ lower training time.

D. Scalability

Scalability is a huge concern especially for conventional discrete event simulators. The simulations become much slower as the network size increases. We repeat the above experiment on OMNet++ with the network scaled from 100 to 1000 nodes, and number of edges to 5000. We find that the average time

between setting a new placement and observing the change increases from 228.3ms to 29.1s. Such a long delay implies that, for a moderately sized SDWAN (1000 routers), each training iteration takes almost half a minute. Considering that many deep learning models are trained for thousands of iterations, the total solution time using traditional network simulator is intolerable. ∂NE, on the other hand, is insensitive to network size (number of nodes/edges/tunnels) because it simply needs to alter one or more dimensions of the tensors in the network model. We observe that scaling from 500 to 5000 edges makes almost no difference on the time of a matrix multiplication operation ($<$1ms for network evaluation) or AD ($<$1ms for training). This indicates that ∂NE can scale to much larger networks while maintaining the same evaluation and training speed.

V. Concluding Remarks

Recent DNN-based TE algorithms invariably assume that the network is a black box. Yet we believe that the network model for TE can be clearly specified. We build ∂NE, a fully-differentiable network environment, as a constructive proof. Our evaluations show that ∂NE accelerates training speed of DNN models for TE by $228\times$ and achieves better scalability compared to existing network simulators. More importantly, ∂NE opens up the possibility to apply any DNN-based models to TE in SDWAN beyond RL, as shown in our implementation of LSTM-TE on ∂NE. We also believe *differentiable programming* is also suitable for other domains, such as resource provisioning and cluster scheduling.

References

[1] "TensorFlow," https://www.tensorflow.org/.

[2] J. A. Boyan and M. L. Littman, "Packet routing in dynamically changing networks: A reinforcement learning approach," in *Advances in neural information processing systems*, 1994, pp. 671–678.

[3] L. Chen, J. Lingys, K. Chen, and F. Liu, "AuTO: scaling deep reinforcement learning for datacenter-scale automatic traffic optimization," in *ACM SIGCOMM*, 2018.

[4] J. Degrave, M. Hermans, J. Dambre *et al.*, "A differentiable physics engine for deep learning in robotics," *Frontiers in neurorobotics*, vol. 13, 2019.

[5] F. A. Gers, J. Schmidhuber, and F. Cummins, "Learning to forget: Continual prediction with LSTM," 1999.

[6] A. Graves, G. Wayne, M. Reynolds, T. Harley, I. Danihelka, A. Grabska-Barwińska, S. G. Colmenarejo, E. Grefenstette, T. Ramalho, J. Agapiou *et al.*, "Hybrid computing using a neural network with dynamic external memory," *Nature*, vol. 538, no. 7626, p. 471, 2016.

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

[8] A. Irpan, "Deep Reinforcement Learning Doesn't Work Yet," https://www.alexirpan.com/2018/02/14/rl-hard.html, 2018.

[9] S. Jain, A. Kumar, S. Mandal, J. Ong, L. Poutievski, A. Singh, S. Venkata, J. Wanderer, J. Zhou, M. Zhu, J. Zolla, U. Hölzle, S. Stuart, and A. Vahdat, "B4: Experience with a Globally Deployed Software Defined WAN," in *ACM SIGCOMM*, 2013.

[10] A. Karpathy, "The Unreasonable Effectiveness of Recurrent Neural Networks," http://karpathy.github.io/2015/05/21/rnn-effectiveness/, May 2015, (Accessed on 06/28/2019).

[12] P. Kumar, C. Yu, Y. Yuan, N. Foster, R. Kleinberg, and R. Soulé, "YATES: Rapid Prototyping for Traffic Engineering Systems," in *ACM SOSR*, 2018.

[11] P. Kumar, C. Yu, Y. Yuan, N. Foster, R. Kleinberg, and R. Soulé, "Yates' Traffic Demands," https://github.com/cornell-netlab/yates/tree/master/data/demands.

[13] P. Kumar, Y. Yuan, C. Yu, N. Foster, R. Kleinberg, P. Lapukhov, C. Lin Lim, and R. Soulé, "Semi-Oblivious Traffic Engineering: The Road Not Taken," in *USENIX NSDI*, 2018.

[14] T.-M. Li, M. Aittala, F. Durand, and J. Lehtinen, "Differentiable monte carlo ray tracing through edge sampling," in *SIGGRAPH Asia 2018 Technical Papers*. ACM, 2018, p. 222.

[15] T.-M. Li, M. Gharbi, A. Adams, F. Durand, and J. Ragan-Kelley, "Differentiable programming for image processing and deep learning in Halide," *ACM Transactions on Graphics (TOG)*, vol. 37, no. 4, p. 139, 2018.

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

[17] H. Liu, X. Wu, M. Zhang, L. Yuan, R. Wattenhofer, and D. Maltz, "zUpdate: Updating Data Center Networks With Zero Loss," in *ACM SIGCOMM*, 2013.

[18] H. H. Liu, S. Kandula, R. Mahajan, M. Zhang, and D. Gelernter, "Traffic Engineering with Forward Fault Correction," in *Proc. ACM SIGCOMM*, 2014.

[19] H. Mao, R. Netravali, and M. Alizadeh, "Neural Adaptive Video Streaming with Pensieve," in *SIGCOMM*. ACM, 2017.

[20] H. Mao, S. B. Venkatakrishnan, M. Schwarzkopf, and M. Alizadeh, "Variance Reduction for Reinforcement Learning in Input-Driven Environments," *arXiv preprint arXiv:1807.02264*, 2018.

[21] J. Oh, X. Guo, H. Lee, R. L. Lewis, and S. Singh, "Action-conditional video prediction using deep networks in atari games," in *Advances in neural information processing systems*, 2015, pp. 2863–2871.

[22] A. Paszke, S. Gross, S. Chintala, and G. Chanan, "Pytorch," 2017.

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

[24] G. F. Riley and T. R. Henderson, "The ns-3 network simulator," in *Modeling and tools for network simulation*. Springer, 2010, pp. 15–34.

[25] A. Sherstinsky, "Fundamentals of Recurrent Neural Network (RNN) and Long Short-Term Memory (LSTM) Network," *arXiv preprint arXiv:1808.03314*, 2018.

[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] Skymind.ai, "A Beginner's Guide to Differentiable Programming — Skymind," https://skymind.ai/wiki/differentiableprogramming, (Accessed on 06/28/2019).

[28] G. Stampa, M. Arias, D. Sanchez-Charles, V. Muntés-Mulero, and A. Cabellos, "A deep-reinforcement learning approach for software-defined networking routing optimization," *arXiv preprint arXiv:1709.07080*, 2017.

[29] TensorFlow, "Automatic differentiation and gradient tape," https://www.tensorflow.org/tutorials/eager/automatic_differentiation, 2019, (Accessed on 04/13/2019).

[30] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2012, pp. 5026–5033.

[31] A. Varga, "OMNeT++," in *Modeling and tools for network simulation*. Springer, 2010, pp. 35–59.

[32] F. Wang, J. Decker, X. Wu, G. Essertel, and T. Rompf, "Backpropagation with Callbacks: Foundations for Efficient and Expressive Differentiable Programming," in *NeurIPS*, 2018.

[33] Wikipedia, "Automatic differentiation," https://en.wikipedia.org/wiki/Automatic_differentiation, 2019, (Accessed on 04/13/2019).

[34] Z. Xu, J. Tang, J. Meng, W. Zhang, Y. Wang, C. H. Liu, and D. Yang, "Experience-driven networking: A deep reinforcement learning based approach," in *IEEE INFOCOM*, 2018.