A Multi-agent Reinforcement Learning Perspective on Distributed Traffic Engineering

Nan Geng*†§¶, Tian Lan†, Vaneet Aggarwal‡, Yuan Yang*§¶, Mingwei Xu*§¶
*Department of Computer Science and Technology, Tsinghua University, Beijing, China
†School of Engineering and Applied Science, George Washington University, Washington D.C., USA
‡School of Industrial Engineering and School of Electrical and Computer Engineering, Purdue University, USA
§Beijing National Research Center for Information Science and Technology (BNRist), Beijing, China
¶Peng Cheng Laboratory (PCL), Shenzhen, China
gn16@mails.tsinghua.edu.cn, tlan@gwu.edu, vaneet@purdue.edu, yangyuan_thu@mail.tsinghua.edu.cn, xumw@tsinghua.edu.cn

***Abstract*—Traffic engineering (TE) in multi-region networks is a challenging problem due to the requirement that each region must independently compute its routing decisions based on local observations, yet with the goal of optimizing global TE objectives. Traditional approaches often lack the agility to adapt to changing traffic patterns and thus may suffer hefty performance loss under highly dynamic traffic demands. In this paper, we propose a data-driven framework for multi-region TE problems, which makes novel use of multi-agent deep reinforcement learning. In particular, we propose two reinforcement learning agents for each region, namely T-agents and O-agents, to control the terminal traffic and outgoing traffic, respectively. These distributed agents collect local link utilization statistics within their regions, optimize local routing decisions, and observe the resulting congestion-related reward. To facilitate these agents for optimizing global TE objectives, we tailor the agent design carefully including input, output, and reward functions. The proposed framework is evaluated extensively using real-world network topologies (e.g., Telstra and Google Cloud) and synthetic traffic patterns (e.g., the Gravity model). Numerical results show that comparing with existing protocols and single-agent learning algorithms, our solution can significantly reduce congestion and achieve nearly-optimal performance with both superior scalability and robustness. Throughout our simulations, over 90% of tests limit congestion within 1.2 times the global optimal solution.[1]**

I. Introduction

Traffic Engineering (TE), which aims to optimize traffic routing for network performance and resource utilization, is fundamental to networking research. In practice, large networks are often divided into multiple (logical or physical) regions, e.g., to facilitate decentralized scalable management [1], to satisfy domain/enterprise requirements [2], or in accordance with geographical locations [3] and heterogeneous routing technologies [4]. In such multi-region networks, each region typically makes its own local TE decisions based on regional network observations. While significantly improving the scalability of network management, the use of multiple regions makes it difficult to achieve global TE objectives due to the lack of a joint effort and coordination.

Distributed TE in multi-region networks is a very challenging problem. In contrast to traditional TE focusing on a fully-controlled and fully-observable network (also known as intra-region TE) [5]–[10], existing approaches often rely on simple routing heuristics (e.g., directing outgoing traffic to the closest border routers in hot potato routing [11]) or leveraging distributed optimization techniques to decouple a global TE problem [1] [12] [13]. In particular, iterative algorithms like [1] [12] [13] for distributed TE require adjacent regions to share necessary information (e.g., gradients) through real-time communication, incurring communication overhead and slow convergence. As a result, these algorithms lack the agility to adapt to changing traffic patterns and thus suffer hefty performance loss under highly dynamic traffic demands. Another line of work develops a hierarchical SDN architecture [3] [14], where a set of slave controllers are designated to different regions, and a super controller coordinates these slave controllers globally to compute routing decisions. Besides communication cost, it still requires a controller having full control over the entire network albeit in a hierarchical manner.

In this paper, we propose a data-driven framework based on Deep Reinforcement Learning (Deep RL) for distributed TE in multi-region networks. The proposed framework provides a refreshing perspective to this problem by modeling each network region as an individual learning agent that has only local network information and interacts with other agents to make decisions on the fly for performance optimization. Compared with traditional TE approaches, Deep RL, as one of the leading Machine Learning (ML) techniques, has the potential of solving complex and dynamic control problems. Deep RL algorithms can automatically exploit hidden patterns in training data and continue improving its TE strategy over time. Well-trained Deep RL models can do inference efficiently even for the inputs that never appeared before. Besides, Deep RL models can be trained by interacting with the network environment without requiring labeled data that are usually hard to obtain in real networks [15]. Several recent proposals [15]–[17] have capitalized on these advancements to tackle the crucial and timely challenge of TE. However, We hasten to emphasize that these RL-based approaches only focus on intra-region TE problems within a single, fully-controlled region (regarded as a single agent) and thus cannot be applied to the distributed TE problem in multi-region networks.

[1]This paper was completed when Nan Geng was a visiting student at George Washington University.

978-1-7281-6992-7/20/$31.00 ©2020 IEEE

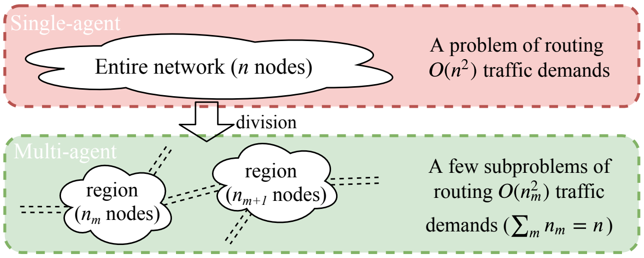

Fig. 1. Multi-agent design has potentially better scalability than single-agent design by dividing a large problem into a few small subproblems.

We argue that investigating multi-agent Deep RL in the context of distributed TE is important in a number of respects: (i) Modeling network regions as individual agents that seek to jointly optimize global TE objectives naturally captures the interplay between them; (ii) Improving network performance through distributed agents has the potential of scaling up as shown in Fig. 1; (iii) More robust TE performance can be obtained for random link failures since small-size regions decrease the probability of having more than one random link failures within a region; and (iv) It provides an exciting new "playground" for multi-agent RL posing unique research challenges. Particularly, we need to come up with new algorithm structures that not only are suitable for implementation in practical networks even under highly dynamic traffic demand and potential link failures, but also facilitate quick convergence and low communication overhead.

A feature of our proposed framework is the design of two separate learning agents for TE in each individual region. This allows us to capitalize on the observation that network traffic in each region can be classified into two categories: terminal traffic that remains in the region and outgoing traffic that traverses multiple regions. Clearly, the routing of terminal traffic directly affects the status of the current region, while the routing of outgoing traffic affects the status of not only the current region but also the other regions, because traffic leaving the region from different border routers has different effects on the other regions. Since providing proper rewards is very important for training Deep RL models and achieving fast convergence, we decided to introduce two types of learning agents, for terminal and outgoing traffics, respectively, and provide them with different reward structures. More precisely, we use a T-agent to observe local network status within the region and to route terminal traffic for optimizing a local TE objective. Another O-agent is designed to solicit reward feedback from neighboring regions and to route outgoing traffic for optimizing a cooperative TE objective. Such a design is also consistent with the design philosophy of existing routing algorithms on border and interior routers. It allows a straightforward implementation in practical networks and strikes a good balance in terms of allowing quick convergence for O-agents and facilitating communication between T-agents. In addition, the separation of T-agent and O-agent also leads to smaller action space in Deep RL.

Unlike most of the existing Deep RL-based approaches [15]–[17] which take real-time traffic matrix (TM) as the input of agents, both T-agent and O-agent in our framework take local link utilizations as the input. These link statistics can be obtained much more easily than real-time TMs. Links with relatively high utilization usually indicate link congestion. Besides, link statistics are also sensitive to link failures. By manually setting the utilization values of broken links to a relatively large value, the agents can adapt to link failures in a way similar to dealing with link congestion.

Two techniques are leveraged to reduce the decision space in the learning algorithm. First, we use the same routing to deliver the traffic demands entering the region from the same ingress router and leaving the region from the same egress router instead of dealing with traffic demands individually. Second, we distinguish elephant flows (with large traffic demand) and mice flows (with small traffic demand). Mice flows below a demand threshold are delivered on statically configured routes to reduce the decision space. For elephant flows, we take splitting ratios of traffic over several pre-computed forwarding paths as the output of agents. Computing and constructing paths in advance is consistent with previous works [5] [9] [17]. Our evaluations show reduced decision space accelerates the convergence speed of the system greatly.

We train T-agents and O-agents using a combined offline and online strategy. In our design, the agents are first trained offline in simulated networks. We adopt an advanced Deep RL algorithm and leverage incremental training to make agents able to accommodate highly heterogeneous traffic patterns and even link failures. We dynamically change traffic in simulated networks in which agents update algorithm parameters incrementally and improve their behaviors continuously. During the online stage, the system can continue to improve with little communication among regions and make near-optimal routing decisions quickly.

We implement our framework using TensorFlow [18] and evaluate its performance using both real-world network topologies (Telstra and Google Cloud) and large-scale synthetic network topologies (with hundreds of nodes). We use the Gravity model [19] to generate highly dynamic traffic including burst demands. Simulation results show that our framework significantly outperforms existing protocols and single-agent learning algorithms and achieves 90-percentile congestion within 1.2 times the optimal for all the simulated topologies. Compared with the single-agent approach [16], our framework achieves 20×-100× the learning speed of [16] and gets at most 72% of congestion reduction in random link failure scenarios.

II. Related Work

Existing TE approaches can be largely classified into traditional model-based routing and data-driven routing.

Traditional model-based routing usually formulates TE problems as an optimization problem and solves the problem through optimization approaches. Most approaches focus on intra-region TE problems [5]–[10]. There are also some approaches proposed for TE in multi-region networks [1], [3], [11]–[14]. The most popular approach is hot potato routing

[11], which, however, usually leads to unexpected congestion. Some approaches [1] [12] [13] solve the TE problem iteratively by exchanging information (e.g., gradients) among regions. However, frequent communication and iterative optimization process lowers the speed of traffic adaption. Some approaches are based on hierarchical SDN architecture [3] [14], which also require full control over the entire network.

Data-driven routing leverages ML techniques to compute routing decisions. Valadarsky et al. [16] use Deep RL to learn the mapping from a sequence of TMs to link weights. These link weights can be used to compute traffic splitting ratios on each router so as to minimize the maximum link utilization. Xu et al. [17] design a Deep RL agent which takes throughputs and average packet delay as the inputs and outputs splitting ratios of traffic over a set of pre-computed paths. However, as described previously, single-agent algorithms face the problem of low scalability and low robustness as networks expand. Lin et al. [20] leverage multi-agent RL and propose a QoS-aware adaptive routing scheme in hierarchical SDN scenarios. Each slave controller computes the routing for the traffic flows in its located region with the help of a super controller. However, global control is not available in many multi-region cases. Besides, flow-level control has been shown to be impractical in high-speed networks [15]. There are also some approaches [21] [22] using supervised learning techniques such as deep belief network (DBN), and graph neural network (GNN), etc. However, labeled data may be unavailable or difficult to measure due to the large size of operational data [15].

Our approach belongs to the second category. We propose a Deep RL-based TE framework for multi-region networks where routing is computed in each region independently and no much overhead is introduced.

III. Problem Statement and Solution Overview

A. Problem Statement

We consider a network consisting of multiple regions. Generally, the entire network can be modelled as a directed graph $G(V, E)$, where V is the node (router/switch) set and E is the edge set. Each edge $e(u, v) \in E$ has a capacity $c_{e(u,v)}$ denoting the maximum traffic amount that can pass the edge from node u to adjacent node v. Note that, edges $e(u, v)$ and $e(v, u)$ can have different capacities. Each region m can be considered as a directed sub-graph $G_m(V_m, E_m)$. We assume that $\bigcup_m V_m = V$ with $V_m \bigcap V_{m'} = \emptyset$ $(m \neq m')$, and that $E_m = \{e(u, v)|u \in V_m\}$. There may be several peering edges connecting two adjacent regions, and the end nodes of these peering edges are border nodes of the regions.

We denote the network traffic as traffic matrix which is a set of traffic demands. Each traffic demand in a TM represents the total traffic amount that needs to be delivered from the source node to the destination node. In a region, demands enter the region from ingress nodes and leave the region from egress nodes. An ingress node can be the source node of some demands or the border node through which some demands come into the region from neighboring regions. An egress node can be the destination node of some demands or the border node through which some demands get out of the region and go to other regions. According to the destination node, we classify the traffic demands in any given region into *terminal demands* and *outgoing demands*. Terminal demands reach their destination nodes in the local region no matter where they come from. In contrast, outgoing demands have destination nodes in other regions no matter where they are from.

The routing task of each region is to properly deliver two kinds of demands from the corresponding ingress nodes to the corresponding egress nodes, i.e., intra-region routing. Particularly, a terminal demand leaves the region from one egress node which is also the destination node. In contrast, an outgoing demand may leave the region from multiple egress nodes (border nodes) since there are usually multiple peering edges connecting the next-hop region and there may be multiple next-hop regions. For inter-region routing, we can use any existing routing algorithms. In this paper, we use the shortest path routing, and our framework can also work with other inter-region routing algorithms.

The goal of TE in multi-region networks is to optimize each region's local routings and achieve a global TE objective, i.e., minimizing the maximum edge cost. The cost of an edge reflects the congestion status of the edge. In particular, the cost of edge $e(u, v)$ is computed by $h(e(u, v)) = \frac{f_{e(u,v)}^{1+\alpha}}{c_{e(u,v)}^{1+\alpha}(1+\alpha)}$, where $f_{e(u,v)}$ is the total traffic amount on edge $e(u, v)$ and $\alpha \geq 0$. The edge cost function $h(\cdot)$ is a general form of cost functions for the scenarios of load balancing [5] [23] or power saving [24]. Our proposed approach can also be extended to other edge congestion-related objectives.

B. Solution Overview

In our design, we take Deep RL agents as the decision makers of routing. Deep RL is a combination of RL and deep neural network (DNN) and is more powerful to tackle complex tasks than RL. Different from supervised learning techniques with external knowledge guidance, a Deep RL agent learns its behavior through interactions with an environment iteratively for a specific objective. At each iteration step, the agent observes the current state of the environment and makes a decision, i.e., an action. Then, the environment evolves transforms from the current state to a new state and returns a reward value to the agent. The reward is a feedback value indicating the quality of the agent's action. The goal of the agent is to learn a policy which is a DNN mapping state to action so as to maximize the discounted cumulative reward [25]. To find a satisfactory policy, Deep RL takes an exploration-exploitation-based method. The agent can take a large-reward action learned so far, which is called action exploitation. The agent can also try a new action for a possibly higher reward, which is called action exploration. A good tradeoff between exploitation and exploration helps the agent "understand" the environment well and learn an optimized policy through enough iterations.

As mentioned previously, the routing of terminal traffic directly affects the status of the current region, while the

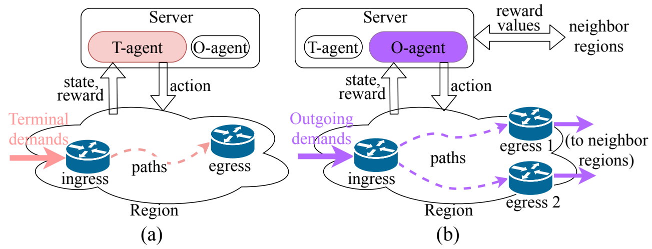

Fig. 2. An illustration of our solution. "state" means edge utilizations of the local region. "action" means traffic splitting ratios over the forwarding paths connecting each pair of ingress and egress node. "reward" means feedbacks which reflect current TE objective values.

routing of outgoing traffic affects the status of not only the current region but also the other regions, because traffic leaving the region from different border routers has different effects on the other regions. In our framework, we propose to use two separate Deep RL agents for traffic engineering in each individual region: T-agent and O-agent. Their definitions are given as follows:

Definition 3.1: T-agent is the Deep RL agent that controls the routing of terminal demands in each region.

Definition 3.2: O-agent is the Deep RL agent that controls the routing of outgoing demands in each region.

Fig. 2 illustrates our proposed solution. For each region, a locally centralized server maintains a T-agent and an O-agent for the region. Each server collects local state information (i.e., edge utilizations) from the corresponding region periodically. Then, the agents take actions (i.e., traffic splitting ratios over pre-computed paths) independently based on the local state information for TE performance improvement. Reward values returned to the T-agents are computed with respect to a local TE objective (i.e., minimizing the maximum edge cost of the local region). While the O-agents get their reward values for a cooperative TE objective by combining the reward values of the T-agents in the local and neighboring regions. The details of link statistics-based state, reduced action space, and congestion-related reward functions will be presented in Section IV-A and IV-B.

To make our framework work well after deployment, we need to train the agents efficiently. Since Deep RL agents learn policies through the exploration-exploitation-based method, online learning from scratch has been widely known to result in poor performance at the beginning [26]. In this paper, we take a combined offline and online strategy. In the offline phase, the agents are trained in a simulated network which has the same topology and capacity settings as the real one. The simulated network will be initialized with offline TMs captured from the real network. We train these agents incrementally using various TMs. After training, the learned DNN parameters will be loaded to online agents for inference (i.e., decision making). During the online stage, the system can continue to improve with little communication among regions and make near-optimal routing decisions quickly. We demonstrate how to train agents in Section IV-C.

IV. Our Proposed Solution

In this section, we describe the development of T-agent and O-agent separately, followed with the training method.

A. T-agent Development

State space. State space is the input of agents, which should capture the key status of the network environment. Most of the existing Deep RL-based approaches take traffic statistics (e.g., TM) as the state input of the agents [15]–[17]. However, it is not easy to obtain real-time TM statistics in multi-region networks which are lack of global view and control. In our design, we use edge statistics, i.e., edge utilizations, which can be measured easily. Edge utilization is determined by both traffic demand and routing decisions. So, it implicitly reflects the traffic demand change, as well as the impact of current actions. Link utilization values provide an indication of which links are congested, so the learning agents can update the current policy accordingly, in order to reduce traffic and congestion on these links.

Formally, we denote $\mathbf{s}_t^{m,T}$ as the state vector of the T-agent in region m at iteration step t. The state vector can be generally expressed as

$$\mathbf{s}_t^{m,T} = \left[\frac{f_e}{c_e}\right]_{e \in E_m},$$

where $\frac{f_e}{c_e}$ is the current utilization of edge e. The size of the state vector $\mathbf{s}_t^{m,T}$ depends on how many edges region m has.

Specifically, the edge utilization being zero indicates that edge failure happens, i.e., the edge is broken down. We manually set the utilization of broken edges to a relatively large value (e.g., 1.0) in $\mathbf{s}_t^{m,T}$, so that the agents can output actions to reduce the traffic amount on the broken edges (just like the process of lightening congestion at bottleneck edges).

Action space. Action space is the output of agents and also routing decisions. Some approaches like [16] decide how to split traffic at every router node. However, such a hop-by-hop method may induce routing inconsistency or even loops if agents make bad actions. So we decide to use routing decisions which distribute traffic demands onto multiple forwarding paths. To simplify the action space and reduce the overhead of path maintenance, we compute and construct K forwarding paths connecting each pair of ingress and egress node in advance, which is consistent with previous works [5] [9] [17]. Thus the routing decision is a set of traffic splitting ratios over these paths. In this paper, we compute forwarding paths using the traffic-oblivious path selection algorithm proposed in [5], which seems to be most suitable for our TE problem (considering the path properties, overhead, etc.). The computed paths have been validated to have some attractive properties such as good load balancing, high diversity, and low stretch. This makes our scheme be able to deal with traffic dynamics and topology changes even when we compute and construct a small number of paths in advance.

The number of paths for an ingress-egress node pair (i.e., K) introduces a tradeoff between potential TE performance and the overhead of Deep RL. While, considering more candidate paths always leads to better path diversity and are more likely to achieve better network performance, it also increases the action space in Deep RL, resulting in higher overhead and slower convergence. In this paper, we set K to three after an empirical study, as the additional benefit of $K > 3$ becomes marginal. Our evaluations show that a small number of carefully selected forwarding paths in each path set are able to balance various TMs well, which is in line with the observations in [5] [9] [17].

As defined previously, T-agent focuses on how to properly deliver terminal demands from their ingress nodes to their egress nodes (also destination nodes) in the local region. Note that there are many terminal demands between the same ingress node and the same egress node. To reduce the action space, we do not consider each terminal demand individually. We design to use the same set of forwarding paths and splitting ratios for the terminal demands between the same ingress node and the same egress node.

Formally, we denote $P^m_{i,j}$ as the set of pre-computed forwarding paths connecting ingress node i and egress node j in region m. Let $\mathbf{a}^{m,T}_t$ be the action vector of the T-agent in region m at iteration step t. The action vector of the T-agent can be expressed as

$$\mathbf{a}^{m,T}_t = \left[a^{i,j}_p\right]_{p \in P^m_{i,j}, i \neq j, i,j \in V_m},$$

where $a^{i,j}_p \in [0, 1]$ means the fraction of traffic amount delivered from ingress node i to egress node j on path p. The terminal demands between the same ingress node i and the same egress node j may have different source nodes but will take the same set of splitting ratios $\{a^{i,j}_p | p \in P^m_{i,j}\}$. Clearly,

$$\sum_{p \in P^m_{i,j}} a^{i,j}_p = 1, \forall i, j \in V_m, i \neq j, \tag{1}$$

which means that all the traffic amount must be delivered from ingress node i to egress node j.

Although the action space has been reduced, it is still potentially very large, i.e., $O(K \cdot |V_m|^2)$. The agents need to do plenty of explorations so that the whole system can converge well. We observe that the aggregated traffic amount of terminal demands between some ingress-egress node pairs is usually very large compared to that between other node pairs. Empirically, the aggregated traffic with large amount is more likely to induce congestion if it is improperly routed. Thus we propose to distinguish elephant flows (the aggregated traffic with large traffic amount) and mice flows (the aggregated traffic with relatively small traffic amount). We consider that the agents only learn and adjust the routing for elephant flows and we just use static routing (e.g., ECMP) for mice flows. Such a design holds the same idea of large flow-based routing in many previous works [7] [27]. Through this mechanism, the action space can be reduced further.

In practice, we can obtain the average aggregated traffic amount of terminal demands between every ingress-egress node pair by analyzing the region's *local* offline traffic traces. Then we sort the traffic amount and take a certain proportion of the aggregated traffic as mice flows. We define *mice flow ratio* ρ which means the proportion of the selected small aggregated traffic with respect to all the aggregated traffic.

Reward. Reward values are computed through a reward function based on current network status, which guides the improvement of agents' policies. A large reward value indicates that the action taken by the agent is good for the TE objective. We denote $r^{m,T}_t$ as the reward value of the T-agent in region m at iteration step t. Recalling that our TE objective is to minimize the maximum edge cost of the whole network, a direct design of the reward function is

$$r^{m,T}_t = -1 \cdot \max_{e \in E}\{h(e)\}, \tag{2}$$

where $h(e)$ calculates the cost of edge e. A small TE objective value means a large reward value. However, the computed reward based on the global objective value cannot correctly evaluate the quality of T-agents' actions. For example, if a T-agent in some region takes an action and causes extremely high edge cost, other T-agents will receive a small reward value even if the actions they took are actually good. Inaccurate rewards will mislead the update of T-agents' DNN parameters.

In our design, we make each T-agent learn its policy for a local objective, i.e., minimizing the maximum edge cost of its located region. Then the reward function can be expressed as

$$r^{m,T}_t = -1 \cdot \max_{e \in E_m}\{h(e)\}. \tag{3}$$

With the reward function in Eq. (3), each T-agent will improve the policy only according to the status of its local region, which, actually, will also benefit the optimization of the global objective. This is because T-agents will adjust the routing of terminal demands to adapt to the routing taken by O-agents and always keep the maximum edge cost in each region small.

B. O-agent Development

State space. In our design, the T-agent and the O-agent of the same region use the identical state input, i.e., edge utilizations of the located region. Formally, we denote $\mathbf{s}^{m,O}_t$ as the state vector of the O-agent in region m at iteration step t. Similar to T-agents, the state vector of O-agents can be expressed as

$$\mathbf{s}^{m,O}_t = \left[\frac{f_e}{c_e}\right]_{e \in E_m}.$$

In practice, the edge statistics can be collected together for the two agents in the same region.

Action space. Similar to T-agents, the action of O-agents also consists of splitting ratios of traffic over a set of pre-computed forwarding paths. Different from T-agents, O-agents aim at how to properly deliver outgoing demands which may leave the region from multiple egress nodes (also border nodes) and enter the next-hop neighboring regions. Therefore, the routing of outgoing demands not only decide how to deliver traffic from the traffic's ingress node to the traffic's egress node but also decide how to balance traffic among multiple egress nodes. To reduce the action space, we take a similar mechanism to T-agent development. Particularly, we use the same set of forwarding paths and splitting ratios for the outgoing demands coming from the same ingress node and going to the same neighboring region.

Formally, we denote $\mathbf{a}_t^{m,O}$ as the action vector of the O-agent in region m at iteration step t. Let $V_{m,m'}$ represent the set of region m's border nodes connecting to the neighboring region m' through peering edges. Let $neighbor(m)$ be the set of neighboring regions of region m. The action vector of O-agent can be expressed as

$$\mathbf{a}_t^{m,O} = \left[a_p^{i,m'}\right]_{p\in P_{i,j}^m, i\neq j, i\in V_m, j\in V_{m,m'}, m'\in neighbor(m)},$$

where $a_p^{i,m'}$ means the splitting ratio of the outgoing demands from the ingress node i to the neighboring region m' on path p. We have

$$\sum_{p\in P_{i,j}^m, j\in V_{m,m'}} a_p^{i,m'} = 1, \forall i \in V_m, \forall m' \in neighbor(m). \quad (4)$$

Any outgoing demands originating from the same ingress node i and going to the same neighboring region m' will take the same set of splitting ratios $\{a_p^{i,m'} | p \in P_{i,j}^m, j \in V_{m,m'}\}$.

We compute forwarding paths connecting each ingress node and each egress node using the same algorithm as the T-agent development. To reduce the overhead of path maintenance and agent training, we choose a limited number of border nodes (also egress nodes) from which outgoing demands can leave the region. Generally, traffic, leaving the region from a border node far way from its ingress node, usually increases the burden of the local region and induces unavoidable path stretch (i.e., long path length). Thus we select the candidate border nodes following the hot potato principle [11], i.e., leaving the region as fast as possible. Particularly, given the outgoing demands from an ingress node to the same neighboring region. We first compute the smallest hop number from the ingress node to each possible border node. Then a few border nodes with a shorter distance to the ingress node than the other border nodes will be selected as the candidate border nodes of the outgoing demands. In our design, we use one forwarding path for each pair of ingress and egress node and three candidate border nodes for delivering outgoing demands.

We do not distinguish elephant and mice flows for O-agent. Recall that outgoing traffic controlled by O-agent can affect the congestion of all regions on its path. Thus, correctly routing a "long" mice flow (that traverses a long forwarding path and has an impact on a large number of links) may even be more crucial than routing a "short" elephant flow. We cannot make the simplification by separating mice and elephants flows for O-agent and ignoring the outgoing mice flows.

Reward. Recall that the routing of outgoing demands affects the traffic distribution of both the local region and the other regions. To reduce congestion of the local region as well as the whole network, the reward value of O-agents should not only reflect the network status of the local region but also reflect the network status of the other regions. Therefore, neither Eq. (2) nor Eq. (3) can be used directly. We observe that the routing of outgoing demands can directly affect the neighboring regions besides the local region. To accurately evaluate the quality of O-agents' actions, the reward values can be calculated based on the status of the local region and the neighboring regions. Therefore, we consider a cooperative TE objective (i.e., minimizing the maximum edge cost of multiple regions) when computing reward values for O-agents. Note that, the T-agent's reward value calculated by Eq. (3) reflects the status of the local region. We propose that adjacent regions can exchange their T-agents' rewards at each iteration step to help compute O-agents' reward values.

Formally, we denote $r_t^{m,O}$ as the reward value returned to the O-agent in region m at iteration step t. The reward value of O-agents can be calculated by

$$r_t^{m,O} = \beta \cdot r_t^{m,T} + (1-\beta) \cdot \mathbf{E}_{m'\in neighbor(m)}[r_t^{m',T}], \quad (5)$$

where β is a non-negative constant and $\beta \in [0, 1]$. In the right hand side of Eq. (5), the first part is the scaled reward value of the T-agent in region m, and the second part is the scaled average reward value of the T-agents in the neighboring regions $m' \in neighbor(m)$. The parameter β is used to adjust the weight of the two parts. With the reward function of Eq. (5) for O-agents and the reward function of Eq. (3) for T-agents, the maximum edge cost of each region can be lowered together.

C. Training Algorithm

We construct a numerical network environment to train the agents. The simulated network has the same topology and the same edge capacity settings as the real target network. Given a TM and the actions of all the agents, the state vectors and the rewards for each agent can be easily calculated. We adopt an advanced Deep RL algorithm and leverage incremental training to make agents able to accommodate highly heterogeneous traffic patterns and even link failures. We show the Deep RL algorithm and incremental training next.

Recall that the purpose of training is to make each agent learn a policy mapping from state vector to action vector so as to maximize the discounted cumulative reward as described in Section III-B. In this paper, we adopt DDPG [25], one of the state-of-the-art Deep RL algorithms, to help the agents learn policies. DDPG supports high dimension state space as well as deterministic and continuous actions, which matches the requirements of our system. In DDPG, there is an *online actor network* and an *online critic network*. The online actor network is the DNN of policy. The online critic network is for approximating the expected discounted cumulative reward called Q-value. Briefly speaking, the online critic network evaluates the behavior of the actor network and helps the actor network update parameters. The online critic network is trained by minimizing the loss function with respect to the evaluated Q-value and a target Q-value. To improve the updating stability, DDPG maintains two separate networks, i.e., a *target actor network* and a *target critic network*, to help estimate the target Q-value. Two target networks have the same DNN structures as the corresponding online networks and are updated slowly from the two online networks.

We formulate and construct the DNNs of DDPG in our design. Since agents update DNN parameters in the same way, we consider a general agent next and remove the index of region (i.e., m) and agent type (i.e., T and O) from the notations for brevity. In a general agent, there is a DDPG instance consisting of four DNNs, i.e., online actor network $\pi(\mathbf{s}_t|\boldsymbol{\theta}^{\pi})$, target actor network $\pi'(\mathbf{s}_t|\boldsymbol{\theta}^{\pi'})$, online critic network $Q(\mathbf{s}_t, \mathbf{a}_t|\boldsymbol{\theta}^{Q})$ and target critic network $Q'(\mathbf{s}_t, \mathbf{a}_t|\boldsymbol{\theta}^{Q'})$ parameterized by $\boldsymbol{\theta}^{\pi}$, $\boldsymbol{\theta}^{\pi'}$, $\boldsymbol{\theta}^{Q}$, and $\boldsymbol{\theta}^{Q'}$, respectively. In our implementation, actor network contains two fully-connected hidden layers with 64 and 32 neurons respectively, and the activation function is Leaky Rectifier function [28]. The input layer (output layer) has a number of units matching the length of the state vector (action vector). The output layer takes Softmax function [28] as the activation function so as to follow the action constraints as shown in Eq. (1) and Eq. (4). Critic network contains three hidden layers with 64, 32, and 64 neurons respectively, and the activation function is Leaky Rectifier function. The input includes state vector as well as action vector output by actor network. The first two hidden layers are fully-connected and take state vector as the input. The third hidden layer takes the concatenation of action vector and the outputs of the second hidden layer as the input and is fed into a one-unit output layer.

At each iteration step t, online actor network takes an action based on the environment. Then, a four-tuple sample can be collected which consists of $\mathbf{s}_t$, $\mathbf{a}_t$, $\mathbf{r}_t$, and $\mathbf{s}_{t+1}$. $\mathbf{s}_{t+1}$ is the new state vector after action $\mathbf{a}_t$ is taken in state $\mathbf{s}_t$. The new sample will be stored in a buffer with fixed size and will overwrite the oldest sample in the buffer if the buffer is full. Then, we select a batch of samples (indexed by k) randomly from the buffer. The online critic network can be updated by $\boldsymbol{\theta}^{Q} := \boldsymbol{\theta}^{Q} + \eta^{Q}\nabla_{\boldsymbol{\theta}^{Q}} L$, where $L = E[\sum_k (y_k - Q(\mathbf{s}(k), \mathbf{a}(k)|\theta^{Q})^2]$, and $y_k = r(k) + \gamma Q'(\mathbf{s}(k+1), \pi'(\mathbf{s}(k+1)))$. y_k is the target Q-value for sample k and is computed with the help of two target networks. $\gamma \in [0, 1]$ is a discount factor. η^{Q} decides the learning rate of the online critic network. Next, actor network can be updated by $\boldsymbol{\theta}^{\pi} := \boldsymbol{\theta}^{\pi} + \eta^{\pi}\nabla_{\boldsymbol{\theta}^{\pi}} J$, where

$$\nabla_{\boldsymbol{\theta}^{\pi}} J = E[\nabla_{\mathbf{a}} Q(\mathbf{s}, \mathbf{a}|\theta^{Q})|_{\mathbf{s}=\mathbf{s}(k), \mathbf{a}=\pi(\mathbf{s}(k))} \cdot \nabla_{\theta^{\pi}} \pi(\mathbf{s}|\theta^{\pi})|_{\mathbf{s}=\mathbf{s}(k)}]$$

η^{π} decides the learning rate of the online actor network. Finally, target actor network and target critic network are slowly updated based on online networks respectively, i.e., $\boldsymbol{\theta}^{Q'} = \tau\boldsymbol{\theta}^{Q} + (1-\tau)\boldsymbol{\theta}^{Q'}$ and $\boldsymbol{\theta}^{\pi'} = \tau\boldsymbol{\theta}^{\pi} + (1-\tau)\boldsymbol{\theta}^{\pi'}$. τ is a small value for adjusting the learning rate of the networks.

To learn satisfactory DNN parameters, the online actor network will add a noise value to its output for action exploration during the training stage as described previously. Particularly, at each iteration step, the action is computed by

$$\mathbf{a}_t = \pi(\mathbf{s}_t|\boldsymbol{\theta}^{\pi}) + \varepsilon(t) \cdot \mathcal{N}(t), \tag{6}$$

where $\mathcal{N}(t)$ is a random noise for action exploration and $\varepsilon(t)$ is a parameter for balancing action exploration and action exploitation. We use $\varepsilon(t) = \varepsilon_0 \cdot \zeta^t$ where ε_0 is the initial value and the constant parameter ζ has $0 < \zeta < 1$. We can find $\varepsilon(t)$ decreases as t increases, and finally $\mathbf{a}(t) \approx \pi(\mathbf{s}(t)|\boldsymbol{\theta}^{\pi})$. Suppose there are totally T iterations and $t = 1, 2, ..., T$. $\varepsilon(T)$ should be small enough so that the action output becomes stable. Given $\varepsilon_0, \varepsilon(T)$, and T, we have $\zeta = \left(\frac{\varepsilon(T)}{\varepsilon_0}\right)^{1/T}$.

Using the above algorithm, the agents can learn good policies after enough iteration steps (i.e., a large enough T) for the simulated network with a given TM. While, it is desired that traffic dynamics, as well as link failures, can be accommodated well during the online stage. Thus, we leverage the method of incremental training to enhance the adaptiveness of the agents. Particularly, we use a set of TMs (called training dataset) to train the agents. The agents will be trained for T iteration steps in the environment set with every TM in the training dataset, and the parameters of agents will be updated incrementally. By creating a dynamic environment, the agents can learn policies that are able to adapt to various traffic patterns. Since the agents deal with link failures in a way similar to dealing with link congestion as described previously, incremental training can also facilitate the improvement of TE performance in link failure scenarios. In practice, the TMs for offline training can be extracted from captured data offline.

The offline training process can be executed either in a centralized manner or in a decentralized manner. For the former, a centralized server will be used in which the whole network environment and all the agents are maintained. That is to say, the centralized server is reliable for all the regions. After training, the centralized server will distribute well-trained agents (i.e., DNN parameters) to the corresponding regions. For the latter, agent training is conducted in individual regions simultaneously. In particular, each region uses a local server to maintain a partial environment only simulating its own region as well as the region's two agents (one T-agent and one O-agent). At each iteration step, adjacent regions will exchange some information such as traffic statistics and reward values. Thus regions can compute states and rewards of O-agents normally. The second training manner makes the regions avoid sharing some sensitive internal information such as region structure, edge capacities, and edge weights. During the online stage, the agents can extend their knowledge continuously with little communication among regions in a way similar to the second manner.

V. Simulations

We implement our proposed framework using TensorFlow [18] and conduct extensive simulations.

A. Simulation Setup

We do evaluation using both real-world and synthetic network topologies. First, we use a measured topology called Telstra (AS 1221) obtained from the Rocketfuel project [29]. The network nodes are scattered across Australia. We consider each state or territory of Australia as a region and ignore the regions with few nodes. Thus we obtain five regions. We also remove the nodes whose degree is no larger than one, which does not affect the evaluation of routings [30]. Particularly, the reduced Telstra topology contains 38 nodes and 152 edges. Second, we use a real topology obtained from

Google cloud [31]. Particularly, we consider three regions: Europe, Asia, and North America, and there are a total of 44 nodes and 160 edges. Third, we use a large-scale synthetic topology whose region-level topology is a 2D 4×4 grid. Thus there are 16 regions in total. We use BRITE [32] to generate each region's topology randomly. In particular, each region's topology contains 10 to 15 nodes, and the link density (the ratio of link number divided by node number) is set to 2 (i.e., 20 to 30 pairs of edges in one region) according to our analysis of many available topologies [33] [29]. For any two adjacent regions, we generate 2 to 4 pairs of edges by selecting border nodes in each region randomly. Particularly, we use a synthetic topology (named as BRITE) with 204 nodes and 964 edges.

The edge capacities of the above three topologies are determined based on the fact that a node with a big degree is likely to hold edges with a large capacity [34]. In particular, an edge's capacity is set to 10 Gbps if either end node of the edge has a degree no smaller than four. Otherwise, the edge capacity is set to 5 Gbps.

We use the Gravity model [19] to generate synthetic TM sequences. In the Gravity model, demand sizes in a TM are proportional to the source node's and the destination node's outgoing capacities and are inversely proportional to the square of the distance between the two nodes. To generate gravity TM sequences, we first compute a base TM using the Gravity model, and $d_{base}(s,t)$ is denoted as the demand size between source node s and destination node t in the base TM. Then we generate TMs one by one by choosing the value of each demand size $d(s,t)$ from the uniform distribution $[-0.5 \cdot d_{base}(s,t), 3.0 \cdot d_{base}(s,t)]$. Given a node pair (s,t), the demand size in some generated TMs can be at most 3.5 times larger than that in other TMs, which is used to simulate traffic dynamics. In our simulations, we use a sequence of 40 TMs for training agents and a sequence of 160 TMs for testing.

We set α=0 in the edge cost function $h(\cdot)$, i.e., the objective of TE is to minimize the maximum edge utilization of the whole network. By default, mice flow ratio ρ is set to 0.8, and mice flows are determined according to the TMs in the training dataset. β in the O-agent's reward function is set to 0.7. We set ε_0=1.0, $\varepsilon(T)$=0.05. T is set to 3,000 for the Telstra topology and the Google cloud topology for enough action explorations, and we set T=6,000 for the BRITE topology which is larger than the first two topologies. We set the discount factor γ=0.99 following [35]. To update DNN parameters smoothly, we use small learning rates: η^{π}=0.0001, η^{Q}=0.001, and τ=0.001.

For convenience, we use MRTE (Multi-Region Traffic Engineering) to represent our proposed framework in the following. We compare MRTE with both traditional and data-driven TE approaches. First, we compare with hot potato routing (HPR), which is the most popular method in multi-region networks and does not import communication overhead during routing computation. In our implementation, HPR delivers outgoing demands to the nearest border node and computes optimal routing for terminal demands by solving MCF problems [36] so as to minimize the maximum edge utilization of each region. Second, we compare with equal-cost multi-path (ECMP) routing which means splitting traffic among the pre-computed paths evenly. ECMP is used to demonstrate that multi-path alone does not lead to superior performance. Third, we implement the single-agent scheme (named as TRPO in our paper) proposed in [16], which is one of the state-of-the-art data-driven TE approaches. The scheme assumes the Deep RL agent has a global view and control of the whole network. The Deep RL agent takes a continuous of TMs as the input and outputs link weights for routing computation. We implement the system and set parameters following [16], and the Deep RL agent is trained sufficiently.

We define *normalized congestion* which can be computed by the maximum edge utilization under our scheme divided by that under the globally optimal routing. The globally optimal routing can be obtained by solving the MCF problem of the TE problem described in Section III-A. In addition, the results of running time are measured on a server with a 4-core Intel 3.6 GHz CPU and 32 GB memory.

B. Simulation Results

Normalized congestion. Fig. 3 shows the empirical CDF of normalized congestion in the Telstra topology. We can find that our scheme, i.e., MRTE, outperforms all the other compared schemes. Particularly, MRTE gets normalized congestion smaller than 1.2 in around 97% of tests. TRPO also performs well, and the normalized congestion of TRPO is smaller than 1.2 in around 91% of tests. However, the performance of TRPO is unstable. For example, TRPO gets result values larger than 1.6 sometimes. ECMP performs worse and HPR performs the worst. HPR always gets normalized congestion larger than 1.3. As described previously, delivering outgoing traffic to the closest border node will overuse some critical edges and may also lead to unexpected congestion in neighboring regions. In such a case, the global TE performance is limited even under locally optimal routing for minimizing each region's maximum edge utilization. Similar results can be found in the Google cloud topology as shown in Fig. 4.

Fig. 5 shows the simulation results of the BRITE topology with over 200 nodes. We can find that our system shows good scalability and performs the best. In contrast, TRPO performs the worst in Fig. 5. This is because the agent of TRPO can hardly converge well due to the high dimension of input and output. Overall, the results in Fig. 3 - Fig. 5 indicate that our proposed scheme limits congestion within 1.2 times of the global optimal solution in over 90% of tests for all the three topologies. Therefore, MRTE is able to achieve stable performance gains under different topologies and TMs.

Normalized congestion in single link failure scenarios. We consider the scenarios where one random link (one pair of edges between two adjacent nodes) is broken down. We manually set the utilization of broken edges to a relatively large value (i.e., 1.0) in the state input so that the corresponding agents can adjust the actions to accommodate the failures just like dealing with edge congestion. On the other hand, as the failure will break down some pre-computed paths, we take a heuristic method that the traffic on the broken paths will be

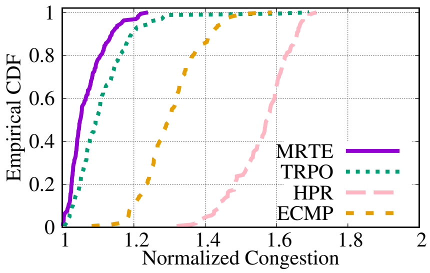

Fig. 3. Empirical CDF of normalized congestion in the Telstra topology.

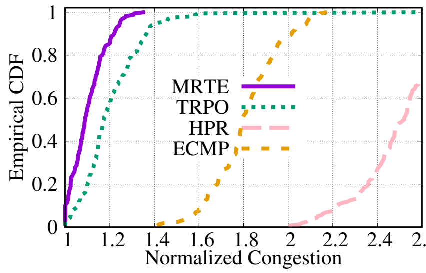

Fig. 4. Empirical CDF of normalized congestion in the Google cloud topology.

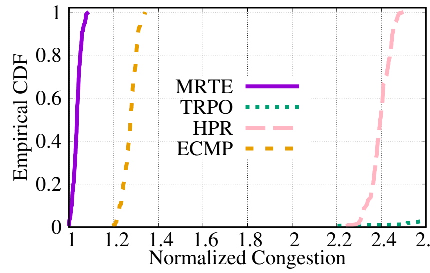

Fig. 5. Empirical CDF of normalized congestion in the BRITE topology.

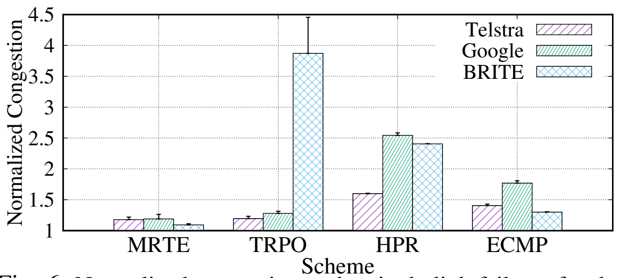

Fig. 6. Normalized congestion under single link failures for the three topologies.

rescaled to other normal paths in the same path set by taking their splitting ratios as rescaling weights. We do 100 tests by generating TMs and selecting broken edge pairs randomly.

Fig. 6 shows the results for all the three topologies. Each bar shows the average value and variance. We can see that MRTE performs the best all the time. TRPO achieves similar TE performance to MRTE for the first two topologies but gets very bad results for the largest topology, i.e., BRITE. For example, MRTE improves the average performance in the BRITE topology by 75.4%, 54.2%, and 15.5% compared with TRPO, HPR, and ECMP, respectively. The results in Fig. 6 illustrate that our scheme using multiple learning agents (as opposed to single-agent RL like TRPO) enables more robust TE performance with respect to random link failures.

Effect of the number of regions. Given a network with a fixed topology, dividing it into more regions tends to make smaller region sizes. We explore the effect of the number of regions (also region size) on TE performance to investigate how MRTE benefits from decentralized control. We do tests on the BRITE topology by re-dividing the topology into different numbers of regions. Obviously, there will be only one region if we consider the topology as a whole. Recalling that the region-level structure of BRITE is originally a 2D 4×4 grid, we can divide the whole topology into two parts evenly, and each part is a 2×4 grid. Then we get a topology with two regions by considering each 2×4 grid as one new region. Similarly, we can divide the network into four regions (four 2×2 grids), eight regions (eight 1×2 grids), and sixteen regions (same as the originally generated multi-region topology).

We train the agents for 6,000 iteration steps with respect to a random TM. Fig. 7 shows the average normalized congestion after convergence and the training time. We can find that more regions lead to better TE performance because of smaller state/action space, which, in some way, illustrates why multi-agent design has potentially better scalability than single-agent design like TRPO. What needs to be pointed out is that too many regions will instead limit the flexibility of traffic routing and harm TE performance. In an extreme case, every router node belongs to one individual region. Then, the demands can only be delivered through the static inter-region path like shortest path routing (see Section III-A). In our test, the normalized congestion in the extreme case is 1.8, which is much larger than all the congestion values shown in Fig. 7.

Table I: A comparison of training speed (per 1k iterations).

	Telstra	Google	BRITE
TRPO	2.1h	4.0h	7.7h
MRTE	1.3min	2.4min	21min

Fig. 7 also shows more regions lead to longer training time because of more agents for training. For example, having 16 regions reduces normalized congestion by 22.7% but increases training time by 29.9% compared with having only one region. Even so, we note that our system can be trained much more efficiently than existing single-agent approaches like TRPO. Table I compares the time of training MRTE and TRPO for 1,000 iteration steps. We can find that MRTE gets $20\times$-$100\times$ the training speed of TRPO.

Investigation of framework design. We evaluate the effectiveness of the framework design, i.e., maintaining two agents for each region instead of one agent. In our design, two agents in the same region use different reward functions as described previously. Particularly, T-agents have the reward function of Eq. (3), and O-agents use the reward function of Eq. (5). Now, we compare our proposed design with another two designs that maintain only one agent for each region. Using one agent for each region means both terminal and outgoing demands are controlled by only one agent. The agents in the first compared design use the reward function of Eq. (3). The agents in the second compared design use the reward function of Eq. (5). We use "one-agent design with Eq. (3)" and "one-agent design with Eq. (5)" to represent the two compared designs, respectively. The systems of the two compared designs are trained and tested in a way similar to our system.

To evaluate the effectiveness of these framework designs, we define five metrics. The first metric is the maximum edge utilization of the whole network, which is our TE objective. The second metric (*resp.* the third metric) is the average value of the top 5% (*resp.* 10%) largest edge utilization. The second and third metrics consider a set of top congested edges instead

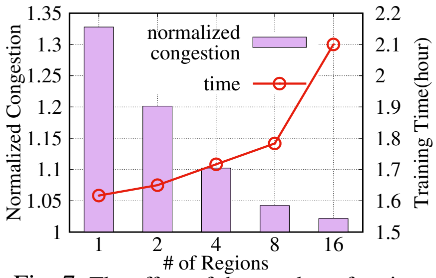

Fig. 7. The effect of the number of regions in the BRITE topology.

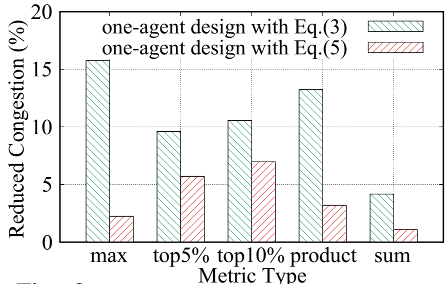

Fig. 8. Reduced congestion of different system designs in the Telstra topology.

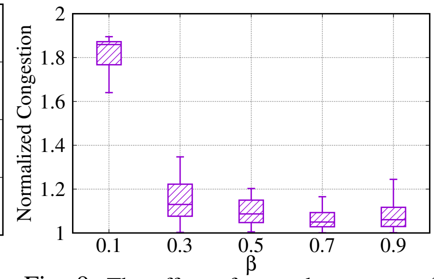

Fig. 9. The effect of reward parameter β on TE performance in the Telstra topology.

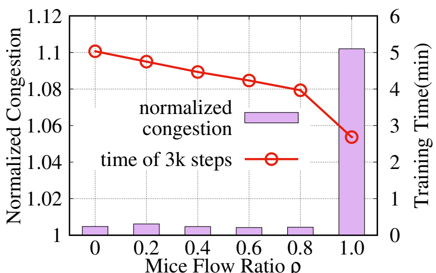

Fig. 10. The effect of mice flow ratio ρ in the Telstra topology.

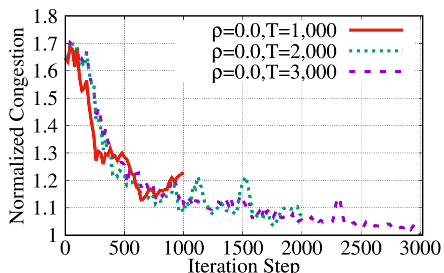

Fig. 11. The effect of T on system convergence when $\rho = 0.0$ in the Telstra topology.

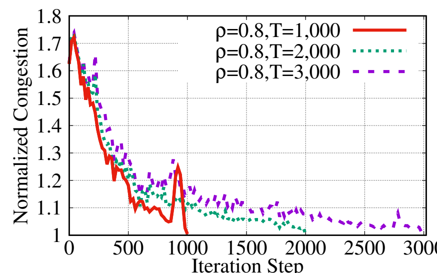

Fig. 12. The effect of T on system convergence when $\rho = 0.8$ in the Telstra topology.

of only the most congested edge. The fourth metric is defined as the product of the maximum edge utilization of each region. The fifth metric is the sum of the maximum edge utilization of each region. The last two metrics are used to validate whether each region's local objective can be optimized simultaneously.

We evaluate reduced congestion which is defined as the reduced metric value of the design compared with our proposed design. Fig. 8 shows the results of reduced congestion in the Telstra topology. We can find the five metric values of the two "one-agent" designs are reduced greatly. The results demonstrate the effectiveness of our system design.

Parameter setting investigation. First, we explore the effect of β in Eq. (5) on TE performance. Fig. 9 shows the normalized congestion of different β values in the Telstra topology. Each box contains the median value, 5%-quantile, 25%-quantile, 75%-quantile, and 95%-quantile. As the increment of β, the result values get smaller at the beginning and become larger when β is too large. When β=0.7, our scheme gets the smallest median value and the most stable results.

Second, we explore the effect of mice flow ratio ρ which is defined for terminal traffic in Section IV-A. A larger ρ means that more terminal demands will use static routing (i.e., ECMP) and that the action space becomes smaller. ρ=1.0 means all terminal demands are delivered through static routing. Fig. 10 shows the results in the Telstra topology. We can observe that the normalized congestion is close to 1.0 when $\rho \leq 0.8$ and that the normalized congestion is large (i.e., over 1.1) when ρ=1.0. This is because too large ρ limits the potential of balancing traffic loads. We can also find that the training time gets smaller as ρ increases. This is because a larger ρ leads to smaller action space, which introduces fewer DNN parameters for updating. Therefore, a proper ρ value makes efficient training while the TE performance is good.

Third, we investigate the influence of parameter T and ρ in the convergence of our system. We perform 20 runs of independent simulations for different TMs and find ρ=0.8 requires significantly less steps for good convergence compared with ρ=0.0 due to reduced action space. Fig. 11 and Fig. 12 show the convergence results for a random TM when ρ=0.0 and ρ=0.8, respectively. We can see ρ=0.8 makes at least 2 times improvement of convergence speed in the simulation.

VI. Conclusion

In this paper, we propose a novel framework for distributed TE in multi-region networks. Two types of agents, namely T- and O-agents, are designed to optimize the routing of terminal and outgoing traffics in each region, respectively. In particular, the agents collect local link statistics, make their own decisions in reduced action space, and measure congestion-related rewards. Simulations based on real-world network topologies show that over 90% of tests are able to limit congestion within 1.2 times the optimal. Note that we set the number of forwarding paths by making a tradeoff between potential TE performance and the overhead of Deep RL. More detailed explorations on this are left to our future work. Investigation of robust algorithms that can cope with topology changes and some other TE objectives (e.g., end-to-end QoE metrics) is also an important future direction.

VII. Acknowledgment

We thank the reviewers and the shepherd for their valuable comments. The co-authors, Geng, Yang, and Xu, from Tsinghua University are supported by the National Natural Science Foundation of China under Grant (61625203, 61832013, and 61872209), the National Key R&D Program of China under Grant (2017YFB0801701), and China Scholarship Council.

References

[1] G. Shrimali, A. Akella, and A. Mutapcic, "Cooperative interdomain traffic engineering using Nash bargaining and decomposition," *IEEE/ACM Transactions on Networking*, vol. 18, no. 2, pp. 341–352, 2009.

[2] F. Paraiso, N. Haderer, P. Merle, R. Rouvoy, and L. Seinturier, "A federated multi-cloud PaaS infrastructure," in *IEEE CLOUD*, 2012.

[3] S. Hassas Yeganeh and Y. Ganjali, "Kandoo: a framework for efficient and scalable offloading of control applications," in *ACM HotSDN*, 2012.

[4] M. Musolesi and C. Mascolo, "A framework for multi-region delay tolerant networking," in *ACM WiNS-DR*, 2008.

[5] P. Kumar, Y. Yuan, C. Yu, N. Foster, R. Kleinberg, P. Lapukhov, C. L. Lim, and R. Soulé, "Semi-oblivious traffic engineering: The road not taken," in *USENIX NSDI*, 2018.

[6] R. Hartert, S. Vissicchio, P. Schaus, O. Bonaventure, C. Filsfils, T. Telkamp, and P. Francois, "A declarative and expressive approach to control forwarding paths in carrier-grade networks," in *ACM SIGCOMM*, 2015.

[7] M. Al-Fares, S. Radhakrishnan, B. Raghavan, N. Huang, and A. Vahdat, "Hedera: Dynamic flow scheduling for data center networks." in *USENIX NSDI*, 2010.

[8] C.-Y. Hong, S. Mandal, M. Al-Fares, M. Zhu, R. Alimi, C. Bhagat, S. Jain, J. Kaimal, S. Liang, K. Mendelev *et al.*, "B4 and after: Managing hierarchy, partitioning, and asymmetry for availability and scale in google's software-defined WAN," in *ACM SIGCOMM*, 2018.

[9] M. Leconte, A. Destounis, and G. Paschos, "Traffic engineering with precomputed pathbooks," in *IEEE INFOCOM*, 2018.

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

[11] U. Feige and P. Raghavan, "Exact analysis of hot-potato routing," in *IEEE FOCS*, 1992.

[12] T. Yang, X. Yi, J. Wu, Y. Yuan, D. Wu, Z. Meng, Y. Hong, H. Wang, Z. Lin, and K. H. Johansson, "A survey of distributed optimization," *Elsevier Annual Reviews in Control*, 2019.

[13] K. Srivastava, A. Nedić, and D. Stipanović, "Distributed min-max optimization in networks," in *IEEE ICDSP*, 2011.

[14] T. Koponen, M. Casado, N. Gude, J. Stribling, L. Poutievski, M. Zhu, R. Ramanathan, Y. Iwata, H. Inoue, T. Hama *et al.*, "Onix: A distributed control platform for large-scale production networks." in *USENIX OSDI*, 2010.

[15] Q. Xu, Y. Zhang, K. Wu, J. Wang, and K. Lu, "Evaluating and boosting reinforcement learning for intra-domain routing," in *IEEE MASS*, 2019.

[16] A. Valadarsky, M. Schapira, D. Shahaf, and A. Tamar, "Learning to route with deep RL," in *NIPS*, 2017.

[17] 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.

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

[19] M. M. Rahman, S. Saha, U. Chengan, and A. S. Alfa, "IP traffic matrix estimation methods: Comparisons and improvements," in *IEEE ICC*, 2006.

[20] S.-C. Lin, I. F. Akyildiz, P. Wang, and M. Luo, "QoS-aware adaptive routing in multi-layer hierarchical software defined networks: A reinforcement learning approach," in *IEEE SCC*, 2016.

[21] B. Mao, Z. M. Fadlullah, F. Tang, N. Kato, O. Akashi, T. Inoue, and K. Mizutani, "Routing or computing? the paradigm shift towards intelligent computer network packet transmission based on deep learning," *IEEE Transactions on Computers*, vol. 66, no. 11, pp. 1946–1960, 2017.

[22] K. Rusek, J. Suárez-Varela, A. Mestres, P. Barlet-Ros, and A. Cabellos-Aparicio, "Unveiling the potential of Graph Neural Networks for network modeling and optimization in sdn," in *ACM SOSR*, 2019.

[23] M. Shafiee and J. Ghaderi, "A simple congestion-aware algorithm for load balancing in datacenter networks," *IEEE/ACM Transactions on Networking*, vol. 25, no. 6, pp. 3670–3682, 2017.

[24] M. Andrews, A. F. Anta, L. Zhang, and W. Zhao, "Routing for energy minimization in the speed scaling model," in *IEEE INFOCOM*, 2010.

[25] 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:1509.02971v6*, 2019.

[26] Y. Bao, Y. Peng, and C. Wu, "Deep learning-based job placement in distributed machine learning clusters," in *IEEE INFOCOM*, 2019.

[27] N. Geng, Y. Yang, and M. Xu, "Flow-level traffic engineering in conventional networks with hop-by-hop routing," in *IEEE/ACM IWQoS*, 2018.

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

[29] N. Spring, R. Mahajan, and D. Wetherall, "Measuring ISP topologies with rocketfuel," *ACM SIGCOMM CCR*, vol. 32, no. 4, pp. 133–145, 2002.

[30] D. Applegate and E. Cohen, "Making routing robust to changing traffic demands: algorithms and evaluation," *IEEE/ACM Transactions on Networking*, vol. 14, no. 6, pp. 1193–1206, 2006.

[31] "Google cloud topology," https://cloud.google.com/about/locations/.

[32] "BRITE," http://www.cs.bu.edu/brite/.

[33] "The Internet topology zoo," http://www.topology-zoo.org/dataset.html.

[34] T. Hirayama, S. Arakawa, S. Hosoki, and M. Murata, "Models of link capacity distribution in ISP's router-level topology," *International Journal of Computer Networks & Communications*, vol. 3, no. 5, p. 205, 2011.

[35] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *ICML*, 2015.

[36] S. Even, A. Itai, and A. Shamir, "On the complexity of time table and multi-commodity flow problems," in *IEEE SFCS*, 1975.