Graph Convolutional Reinforcement Learning for Load Balancing and Smart Queuing

Hassan Fawaz*, Omar Houidi*, Djamal Zeghlache*, Julien Lesca†, Pham Tran Anh Quang†, Jérémie Leguay†, and Paolo Medagliani †
*SAMOVAR, Télécom SudParis, Institut Polytechnique de Paris, 91120 Palaiseau, France
† Huawei Technologies Ltd., Paris Research Center, France

***Abstract*—In this paper, we propose a graph convolutional deep reinforcement learning framework for both smart load balancing and queuing agents in a collaborative environment. We aim to balance traffic loads on different paths, and then control how packets belonging to different flow classes are dequeued at network nodes. Our objective is twofold: first to improve general network performance in terms of throughput and end-to-end delay, and second, to ensure meeting stringent service level agreements for a set of classified network flows. Our proposals use attention mechanisms to extract relevant features from local observations and neighborhood policies to limit the overhead of inter-agent communications. We assess our algorithms in a Mininet testbed and show that they outperform classic approaches to load balancing and smart queuing in terms of throughput and end-to-end delay.**

***Index Terms*—Smart Queuing, Load Balancing, Deep Reinforcement Learning, Multi-Agent Systems.**

I. Introduction

With an ever-increasing demand for higher throughput and lower latency values, traditional traffic engineering and queue management approaches in networks are showing their weaknesses. Load balancing, as well as traffic scheduling, play a primary role in determining the efficient sharing of bandwidth, and in particular, the capability of networks to satisfy Service Level Agreements (SLAs) for flows across different metrics such as throughput, delay, jitter, and others.

In a Software-Defined Wide Area Network (SD-WAN), a central controller holds a set of network policies that are deployed at edge routers charged with interconnecting different sites and branches. The edge routers are configured to transmit traffic over different transport networks such as internet, MPLS, or private lines. These edge devices are tasked with making real-time decisions following how the traffic varies in order to sustain SLAs and optimize network performance.

The usage of load balancing and queuing mechanisms to improve network performance is not a new concept. Nonetheless, classic approaches of the former such as Equal-Cost Multi-Path (ECMP), which evenly splits traffic across different routes, and traditional mechanisms of the latter such as Weighted Fair Queuing (WFQ), which schedules traffic based on pre-selected weights per flow, have been shown to be extremely limited in efficiency. As a result, reliance started to grow on adaptive and machine learning-based approaches to these two tasks.

In the case of load balancing, deep learning mechanisms, such as ReInforcement Learning NETworking (RILNET) [1], grew in pertinence. RILNET, a model-free actor-critic algorithm, uses deep function approximators to develop continuous action space policies. This centralized framework uses Maximum Link Utilization (MLU) as basis for its reward, and assumes its actor and critic have complete knowledge of the network. Nonetheless, RILNET does not count for SLA requirements or any Quality-of-Service (QoS) constraints.

In the case of queue management, several adaptive queuing and active queue management techniques [2] have been shown to be capable of sustaining certain delay and throughput requirements. The automatic adaptation of queuing parameters, such as the weights in Adaptive Weighted Fair Queuing (AWFQ) [3]–[5], improves network performance. However, these mechanisms function at the level of individual routers, without any cooperation or communication towards globally improving the QoS.

In our work, we propose a joint smart load balancing and queue management approach. As in traditional routers, the two algorithms work in cascade. Nonetheless, in our approach, they are based on deep learning mechanisms. Specifically, in order to develop these traffic engineering agents, we utilize a subset of machine learning known as graph convolutional reinforcement learning. The latter models the network nodes as the vertices of the graph and its links as the edges. This modeling encapsulates inter-node relationships and suits the general network structure, enabling thus the reduction in inter-agent communications. We consider both a typical SD-WAN scenario and a generalized network topology and deal with a set of classified network flows divided, following their priority, into three groups: gold, silver, and bronze, representing traffic such as real-time, business, and bulk, respectively. Our objective is twofold: first, we aim to use deep learning agents to select the best paths for the flows across the network, and second, based on the priority of these routed flows, we use another set of agents to determine how they are queued and dequeued at network routers.

Furthermore, we discuss how the load balancing and smart queuing agents learn. Their states, actions, rewards, and the used methods for inter-agent cooperation are detailed next. We build our proposals in a Mininet testbed [6], and perform packet-level simulations in both SD-WAN and classic network topologies. We show that our proposals scale well and that they outperform classic approaches such as ECMP and traditional weighted fair queuing in terms of maximum throughput and minimum delay. Finally, we discuss the overhead incurred due to inter-agent communications and show that the reduction in

ISBN 978-3-903176-57-7

inter-agent communications, as a result of the neighborhood-based cooperation mechanisms of the used learning approach, leads to insignificant signaling loads.

Section II of this paper describes the related works in the state-of-the-art. Section III discusses the system architecture, including our SD-WAN use case. Section IV introduces the graph convolutional reinforcement learning-based algorithm that we used for both our smart queuing and smart load balancing agents. Section V details our approach to the smart load balancing problem including the considered actions, states, and rewards. Section VI details our smart queuing proposal and the queuing approach it is built upon. Finally, Section VII presents the simulation results and analysis, while Section VIII concludes this paper.

II. Related Works

In this section, we take a glimpse at the state-of-the-art in relation to load balancing and queue management in networks. Load balancing (LB) aims at better managing and distributing traffic in networks according to heterogeneous and dynamic requirements and has been the focus of much research and investigation in the academic and industrial worlds.

One of the most widely used strategies in LB is ECMP [7], a routing strategy where packet forwarding to a single destination can occur over multiple best paths with equal routing priority. ECMP splits flows evenly over candidate paths between origin (source) and destination pairs. It suffers when confronted with bursty traffic and in the presence of elephant flows. Uneven flow splitting across multiple paths, like in Unequal Cost Multi-Path routing (UCMP), improves the performance slightly but does not resolve issues with long-term optimization for load balancing when traffic patterns are unknown and hard to forecast or predict.

Even if network calculus or queuing models can estimate latency, packet loss, and jitter, the associated models remain quite complex and difficult to integrate in global load balancing solutions. Model-free reinforcement learning approaches have been proposed as a promising solution to optimize parameters such as Quality of Experience (QoE) [8] and MLU, as well as correctly predicting future conditions. For these reasons, some previous works such as RILNET [1] integrate Reinforcement Learning (RL) in their approach. The RL algorithm used in RILNET is Deep Deterministic Policy Gradient (DDPG) [9], which is a model-free, off-political actor-critic algorithm that uses deep function approximators that can learn policies in continuous action spaces. The reward used in the agent learning phase is the MLU.

The RL algorithms have also been extended from one to multiple agents (MARL) that work together to seek the best decision that maximizes the desired reward through interaction with the environment. MARL [10] algorithms can rely on centralized control, where a central unit periodically makes a decision for each agent; or decentralized control, where each agent makes its own local decisions. Agents can also work cooperatively to achieve a common goal, or they can compete with each other to maximize their own reward.

A relevant paper for smart load balancing is DGN [11], even if its focus is on the more general routing problem in networks. DGN proposes a multi-agent cooperation algorithm based on convolutional graph reinforcement learning suitable for capturing the dynamics of the multi-agent environment underlying graph. The proposed DGN routing solution aims at finding the next hop, hop-by-hop, for packets at each router from the origin to the destination while avoiding congestion. The considered protocol is based on a distributed decision at each router in order to optimize the average delay of data packets. The packet-based routing results obtained by DGN are found promising compared with traditional techniques, and thus motivate our interest in using DGN to realize load balancing on flows rather than packets for dynamic end-to-end path selection.

Existing traffic engineering approaches focus on the network performance and do not take into account the intents of users. In [12], the authors introduce an intent-based load balancing approach that can take users' intent, such as commercial cost, into consideration. Houidi et al. [13] use graph convolutional reinforcement learning to propose a QoE-based load balancing algorithm. Their approach models the network as a graph and derives, through a graph convolutional method, a policy that splits video traffic flows across end-to-end candidate paths while meeting application QoE requirements.

In this context, we propose in this paper to apply DGN philosophy to design a smart load balancing and queuing protocol. However, in our work, we will not adopt the same hop-by-hop prediction policy used in [11] (where data packets are the agents). Rather, our model will be based on the path between the origin and the destination routers. The paths are pre-calculated using the shortest path method, and DGN will be applied to load balance between the selected paths by estimating the percentage of the flow for each path.

The main objective is to find the load balancing weights that determine the path (among candidate paths) for every incoming flow. The ultimate goal is to optimize QoS by leveraging only network-level KPIs and deriving the probabilities for path selection. For this reason, we identify the combination of network metrics necessary to design an efficient reward function that maximizes the QoS. We show that by selecting flow delivery delay and throughput measurements, we can improve the QoS compared with ECMP.

For the state-of-the-art on Smart Queuing (SQ) and the utilization of Deep Reinforcement Learning (DRL) in network management, Kim et al. [14] propose a Deep Q-Network (DQN) based Active Queue Management (AQM) scheme to reduce queuing delay in fog/edge networks. Their proposal relies on DRL techniques for efficient queue management to handle latency and trade-off queuing delay with throughput, while maintaining QoS in terms of low jitter. It is compared to other RL-based solutions and shown to outperform the other methods in terms of delay and jitter while maintaining above-average throughput and being an efficient network congestion manager.

A reinforcement learning based approach, via an index policy for bursty load management, is proposed by Balasubramanian et al. [15] to improve average wait times and over-saturation in the queues. RL is shown here also to provide benefits and improvements compared with non-RL solutions. The work of Liao et al. [16] addresses the scheduling of flows in Multi-Path TCP (MPTCP), with a focus on short and long

MPTCP flows, and also uses RL to improve performance compared with traditional traffic scheduling methods. The proposed DDPG-based DRL framework determines how to distribute the packets over multiple paths while decreasing the out-of-order queue size under such paths. It reduces the gap between the faster and slower paths. The RL model is solved via an actor-critic framework and transformer encoders to process the states of dynamic sub-flows for the deep neural networks (in the actor and critic networks).

Furthermore, all these previous works conduct evaluations based on simulations, while this paper implements and evaluates MARL-based load balancing algorithms in a real-world testbed. To the best of our knowledge, we are the first to propose an RL-based system that considers two reference use cases: Smart Load Balancing (sLB) combined with an SQ mechanism to provide an intelligent routing architecture based on an intelligent network architecture design and collaboration between agents using an attention-based technique [17]. Our algorithm is used to guide online packet load balancing decision-making to achieve intelligent routing and smart queuing in the entire network.

Additionally, in contrast to the majority of the existing works, our proposal was also evaluated and validated using larger-scale networks alongside a testbed framework. Compared with the best state-of-the-art distributed and centralized software-defined networking solutions, our algorithm improves SLA satisfaction. Our proposed model is adaptable and supports different topologies, traffic distribution scenarios, and network scales.

III. System Architecture

We consider a semi-distributed architecture where edge devices are controlling traffic based on real-time measurements using local agents sharing certain information with their peers. The agents are centrally trained, but their execution is done in a distributed manner. In this section, we detail the architecture of the SD-WAN use case that we focus on, and the general node architecture in which our agents are built. In the simulation and results section, we also discuss the implementation of our proposal in a more generalized network topology.

Figure 1 presents a typical SD-WAN use case where an enterprise network headquarters (HQ) and three remote branches are interconnected by the numbered nodes and through Access Routers (ARs) located at the branches. Flows issued by user applications are grouped into *flow groups* that correspond to traffic classes with different SLA requirements. A typical traffic scenario includes gold, silver, and bronze groups for multimedia, business critical, and non-critical applications, respectively.

The system architecture is split into two control entities. In the first control entity, traffic engineering and load balancing policies are taken. In the second control entity, AR devices take tactical decisions to follow the evolution of traffic and network conditions. Figure 2 depicts the architecture of AR devices. The decisions are taken in cascade. The traffic of each flow group is first load balanced over available access networks using a *routing agent*. Afterwards, a scheduling engine at each port (each access network link), controlled by a *QoS agent*, applies a QoS policy, *i.e.*, the WFQ-based RL approach we

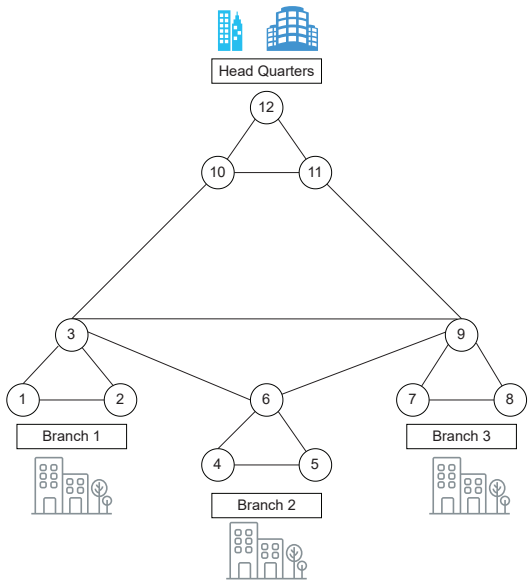

Figure 1: SD-WAN based network topology

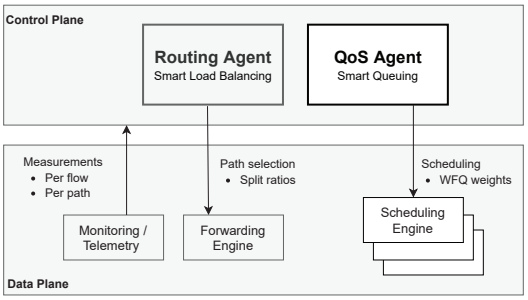

Figure 2: Access router architecture

describe later on. The monitoring block provides information on the network at path and flow group levels such as jitter, delay, and throughput metrics, some of which are factored into our deep learning decision-making.

Our objective is twofold. First, we want to use deep learning, specifically graph convolutional reinforcement learning, to load balance the flows on the different available paths in the network. Secondly, we aim to satisfy SLAs for the same set of classified network flows. In particular, we aim at meeting performance targets for each flow group in terms of minimum throughput and maximum end-to-end delay.

Finally, we note that in addition to our SD-WAN use case, we also test our proposal in a more generalized network topology, namely based on the ION-NY network topology. We show that our learning models are resilient to different topologies and can scale well.

IV. Graph Convolutional Reinforcement Learning For Multi-agent Systems

In the case of smart load balancing, the agents are placed at source nodes. In the case of smart queuing, the HQ nodes also have embedded agents. In our work, we use graph convolutional reinforcement learning, or DGN, to dictate the relationship between agents and how they learn. The latter

models the network as a graph $G = (V, E)$, where its nodes are represented by the agents, and its vertices by the links connecting them. Every agent i, $i \in \mathcal{N}$, possesses a set of neighboring agents with which it can communicate and exchange information. We use the existence of direct links between agents as the main factor in determining the adjacency. An adjacency matrix $\mathcal{C}$ defines which agents are neighbors, and as such, which agents will exchange information throughout training and execution. This reduces inter-agent communications and its costs in terms of signaling and complexity.

Multi-agent environment. Each agent i will run its own DRL algorithm. In the case of smart load balancing, it aims to learn how to distribute the loads across the different paths. In the case of smart queuing, it seeks to determine the best weights per flow group (w_i^G, w_i^S, w_i^B). During each time iteration t, the agents receive local observations o_i^t. In the case of load balancing, they consist of throughput metrics. In the case of smart queuing, the latency per flow class is also factored in. Observations from neighboring (communicating) agents, impact how the agents learn and their decision-making as well. As a result of these observations, an action a_i^t is taken, and a reward r_i^t is issued.

In Figure 3, we show the composition of the DRL agents. They are formed from three main modules. The first module is an encoder comprised of a multi-layer perceptron (MLP). As input, the encoder takes the set of local observations of the agents o_i^t and outputs the relevant features f_i^0. Each agent i will share its relevant features with its neighbors as input to the second module.

The second module is composed of convolutional layers. Each of these layers takes as input the output of the previous layer from its own agent, and from other communicating agents as well. The convolutional layers are used to further extract relevant features among neighbors. Multiple convolutional layers can be used. The number of convolutional layers in the module could be likened to distance-vector routing and determines the distance at which communicating agents could be placed in the network graph while still being able to exchange relevant information from the entire network. That is to say, even if an agent does not communicate with all agents in the network, the communications with its neighbors enable it to obtain a full view of the topology.

The third and final module in the agent composition is a Q-network. It takes as input a vector of features from the convolutional module and outputs the action to be taken. As in classic deep Q-learning, the objective is to take the decisions which maximize the expected reward.

Attention. DGN implements attention mechanisms through convolutional kernels. Initially used in Convolutional Neural Networks (CNNs) for image and pattern recognition and extraction, these kernels enable the integration of features in the receptive fields of the agents to extract latent features. They enable learning how to quantify the relationship between agents and the integration of their respective features. DGN uses a multi-head dot-product convolutional kernel to compute the different interactions among agents. A more illustrated discussion on how attention works in neural networks can be found in [17].

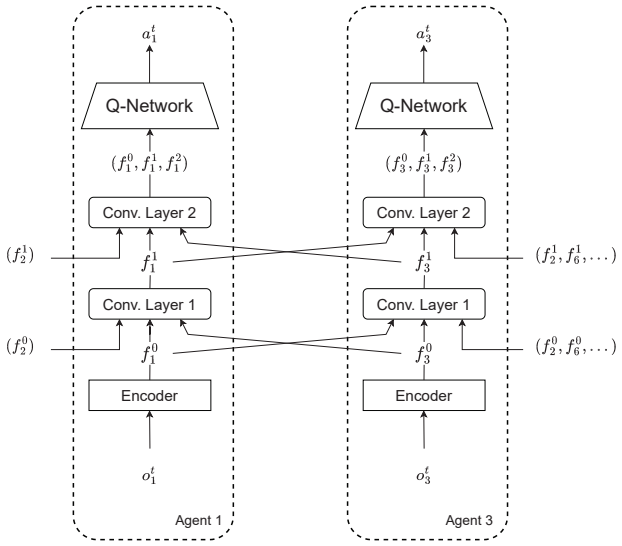

Figure 3: Structure of two communicating DGN agents

Replay buffer. We use an experience (replay) buffer in our work. Experiences are stored and afterwards randomly selected for training. This reduces the risk of correlated data. These samples are formed as ($\mathcal{O}$,$\mathcal{A}$,$\mathcal{O}'$,$\mathcal{R}$,$\mathcal{C}$), where $\mathcal{O}$ is the set of agent observations $\{o_1, ..., o_N\}$, $\mathcal{A}$ is the group of agent actions $\{a_1, ..., a_N\}$, and $\mathcal{O}'$ is the set of new observations $\{o'_1, ..., o'_N\}$ resulting from these actions. $\mathcal{R}$ represents the rewards issued to the agents $\{r_1, ..., r_N\}$, and $\mathcal{C} = \{C_1, ..., C_N\}$ is the collection of adjacency matrices for the agents. The latter define the neighborhoods for the agents, and indicate with which agents each agent is able to communicate. We dropped the time index t from these expressions for the sake of simplicity.

Agent training. Alongside graph convolutional reinforcement learning, we enlist the aid of a classic deep learning tool in the usage of a target network [18]. The latter stores a copy of the main Q-network of the agents. Nonetheless, the parameters in this case are not trained but slowly updated using the main network's parameters. A random minibatch of size S is used to train the agents with the objective of minimizing the loss:

$$\mathcal{L}(\theta) = \frac{1}{S} \sum_{S} \frac{1}{N} \sum_{i=1}^{N} (y_i - Q(O_{i,C}, a_i; \theta))^2. \tag{1}$$

Recall that N is the total number of agents and that

$$y_i = r_i + \gamma \max_{a'} Q(O'_{i,C}, a'_i; \theta'). \tag{2}$$

$O_{i,C} \subseteq \mathcal{O}$ represents the observations of i's neighbors. Q is the Q-function, θ' represents the parameters of the target network, and γ is the discount factor, which weighs the impact of future rewards. The gradients of the loss of all the agents are accumulated and used to update the main network parameters. The target network parameters are smoothly updated every training iteration as:

$$\theta' = \tau\theta + (1 - \tau)\theta', \tag{3}$$

where τ represents the smoothness of the update. When τ=1, the update is considered to be "hard" and the parameters of the main network are directly cloned onto the target network.

V. DGN BASED SMART LOAD BALANCING

In this section, we discuss the states, actions, and rewards associated with our deep reinforcement learning approach to load balancing in networks.

- The states are the amount of traffic (average throughput) and the number of active flows on each outgoing path (route). We use one agent for each Origin-Destination (OD) flow. For the policy decisions, each agent is positioned at the source node and decides about load balancing weights for the L paths towards the destination.
- The actions in our case are the load balancing weights for each OD flow. Each time a new flow arrives, the path is selected accordingly. Due to the discrete nature of RL algorithms, we choose to discretize the decision-making process for the agents: at each time step/period P a new policy will be selected by the agent and will be applied during the entire time step P: each new flow arriving between the current time t and $t+P$ will follow the same policy ($i.e.$, set of load balancing weights). We explore the use of QoS metrics collected in the network as an alternative to QoE-based DRL methods. The latter assume that full access to clients and applications on the endpoints or terminals is readily available and can be collected to monitor the evolution of the reward at every iteration.
- Equation 4 presents the reward expression considered and used for SLB comparisons in the performance evaluation.

$$\mathcal{R} = Throughput - \lambda \times c(s,a) \tag{4}$$

where $c(s,a) = |Reference_{Delay} - Current_{Delay}(s,a)|$ As shown in Equation 4, we introduce a constrained reward expressions that penalize decisions if the experienced flow delivery delay violates a tolerable delay bound, called the *Reference Delay* (arbitrarily set to 0.5s in our case). Recall that our goal is to use only network KPIs in the smart load balancing algorithms to intelligently distribute traffic to optimize QoS. The constrained rewards use parameter λ to penalize the algorithms if flow delivery delays violate target SLAs. While algorithms like RCPO [19] can be implemented to find optimal values of λ, in our case, we performed an iterative search for simpler implementation.

VI. DGN BASED SMART QUEUE MANAGEMENT

For the smart queuing part of our joint approach, DRL and specifically graph convolutional RL, is used to aid a WFQ scheduler placed at key nodes in the network. Weighted fair queuing is used to sustain fairness. The weights of the scheduler can be modified and flow groups would receive bandwidth allocation in proportion of their weights. These weights would also impact the end-to-end latency experienced by the flow groups. Let $\{1,\ldots,K\}$ represent the flows. In a WFQ scheduler, each of them achieves an average rate R_k equal to:

$$R_k = \frac{w_k}{\sum_{i=1}^{K} w_i} R, \tag{5}$$

where R represents the overall capacity of the link, and w_k is the weight of flow k. The larger the weight of a flow is, the more bandwidth it is allocated and the lower its latency is.

We assume that every flow in the network can be classified into one of three groups. These groups are: gold, silver, and bronze in descending order of importance. Each has a set of minimum throughput thresholds to be respected: T_g, T_s, and T_b for gold, silver, and bronze, respectively and an upper limit of maximum end-to-end delay values to be sustained: d_g, d_s, and d_b. The QoS agents aim to meet these constraints by continuously updating the weights for each group of flows served. Each agent is built on top of a WFQ scheduler. While we use the latter in our work, we believe that our learning model is generic enough to handle any other scheduling architecture. The agents require observations in terms of the throughput and end-to-end delays of the flow groups. Based on these observations, each agent will take a decision on whether to increase or decrease the weight of each group of flows it serves.

The set of observations, actions, and rewards are detailed as follows:

- The local observation is a tuple containing the end-to-end throughput and delay values of the flows served by the node: $\{\overline{T_g}, \overline{d_g}, \overline{T_s}, \overline{d_s}, \overline{T_b}, \overline{d_b}\}$. $\overline{T_g}$ is the throughput of the served gold flows, and $\overline{d_g}$ is their average end-to-end delay, and so on for the rest of the classes. The end-to-end delays are measured using in-band network telemetry.
- Each agent takes the action of either increasing or decreasing the weight of each flow group by a constant value δ. At each time step t, each QoS agent will change all the weights of the flow groups it is serving ($\pm$ δ). With three flow groups being considered, that is a total of eight different actions that could be taken.
- As a result of the action taken, each agent receives a reward that reflects its contribution to satisfying the throughput and delay requirements for the flow groups. We denote η_j be the reward for meeting the throughput thresholds of flow group j, and ϕ_j its end-to-end delay equivalent. The reward r_t issued for a certain action is as such calculated as:

$$\omega_g^{th}\cdot\eta_g + \omega_g^{d}\cdot\phi_g + \omega_s^{th}\cdot\eta_s + \omega_s^{d}\cdot\phi_s + \omega_b^{th}\cdot\eta_b + \omega_b^{d}\cdot\phi_b, \tag{6}$$

where ω_j^{th} is equal to -1 if the throughput threshold for j is violated and +1 otherwise. ω_j^{d} is its equivalent for the end-to-end delay constraints. As a result, the reward could be a negative value, $i.e.$, a penalty.

The rewards/penalties for meeting the thresholds for the gold flows are higher than those for the silver, and for the silver flows higher than those for the bronze. The agents are better rewarded, and in turn more harshly penalized, for meeting (or violating) the requirements for the gold flows, than they are for the silver and bronze flows, respectively. Additionally, the rewards for the delay thresholds with respect to the throughput can be weighted. For instance, we configure $\phi_g = \kappa_g \cdot \eta_g$. If $\kappa_g < 1$, the agents will aim to meet the throughput thresholds ahead of the delay ones.

Finally, in a continuous space such as weight selection, it is important to introduce discrete states. We start by defining the observation space size. We set the latter to 20 in our work, meaning that there are 20 different possible values for

each element of the observation. Afterwards, we calculate the window size as the maximum value for each element of the observation sextuplet minus the minimum value for said element. Finally, we calculate the discrete observation as the integer ratio of the actual observation value minus the minimum value, with the result being divided by the calculated window size.

We note that there is an inversely proportional relationship between the size of the state space *i.e.,* the total number of possible values the observation can take, and the time needed for convergence. Nonetheless, the former should be large enough to give the algorithm sufficient room to explore.

VII. Simulation and Results

In this section, we detail a set of simulations and results with the objective of highlighting the efficiency of our joint proposal. We start by introducing the emulator we used in our work, and afterwards discuss the training of the agents, the considered network topology, and the benchmarks used to assess our proposals. Finally, we demonstrate that our joint approach is scalable and discuss the overhead that our proposals impose.

A. Emulation and Traffic Generation Environment

We used Mininet to test our proposed algorithms. Mininet provides a virtual testbed and development environment for Software-Defined Networks (SDNs). Mininet enables SDN development on any processor, and SDN designs can move seamlessly between Mininet and the real hardware running at line rate in live deployments [6]. Mininet enables:

- Rapid prototyping of software-defined networks
- Complex topology testing without the need to wire up a physical network
- Multiple concurrent developers to work independently on the same topology

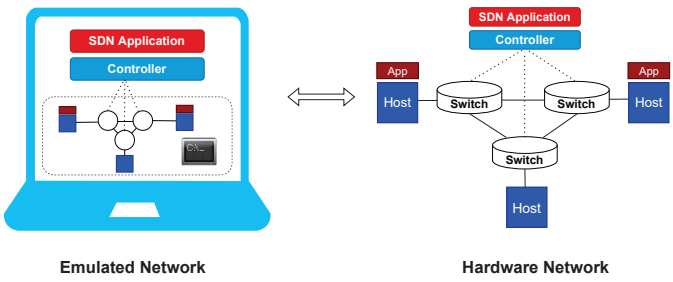

Figure 4: The Mininet stack

Figure. 4, illustrates how Mininet works. It creates a realistic virtual network, running real kernel, switch, and application code. Alongside Mininet, we use the Distributed Internet Traffic Generator (D-ITG) to generate traffic [20]. D-ITG is a platform capable of producing traffic at packet-level accurately, replicating appropriate stochastic processes for both inter departure time and packet size random variables (Exponential, Uniform, Cauchy, Normal, Pareto, and others).

B. SD-WAN Scenario

We start by considering the topology shown in Figure 1. Each node at the branches (1, 2, 4, 5, 7, 8) has an attached source of traffic. Only edge nodes have agents attached to them. For DGN, nodes with a link are considered neighbors and can communicate. For the SQ approach, the agents can communicate with the headquarters node as well (12).

Table I: Simulation parameters for the SD-WAN scenario

Parameter	Value
Number of O-D pairs	6 with 18 different flows
Transport protocol	Mixed UDP-TCP
SLA Throughput G/S/B flows	3000, 1200, 500 Kbps
SLA Delay G/S/B flows	0.2/0.5/1 seconds
Link BWs	variable 20-50 Mbps per link
Transmit rate per source	900-1200 packets/second
Packet size	512 Bytes

Table II: Parameters for SQ DGN agents

Parameter	Value
N^o of neighbors	3
N^o of convolutional layers	2
Reward relative to flows G/S/B	$3x/2x/x$
N^o attention heads	8
N^o of encoder MLP layers	2
N^o of encoder MLP units	(128,128)
Target network update rate τ	0.01
Discount factor γ	0.99
Training batch size	32

Table III: Parameters for sLB DGN agents

Parameter	Value
N^o of neighbors	3
N^o of convolutional layers	2
N^o attention heads	8
N^o of encoder MLP layers	2
N^o of encoder MLP units	(128,128)
Target network update rate τ	0.01
Discount factor γ	0.99
Training batch size	32
Policy refreshing period (time period)	10 s

Traffic scenario. Table I shows the simulation parameters for the SD-WAN scenario. The transmit rate of the sources follows diurnal and sinusoidal patterns between 900-1200 packets/second. The simulations are done as a series of snapshots, the duration of each being 10 seconds. The duration is enough to achieve a steady state for TCP in our topology, and it has no impact on the results. The agents are queried for new weights at the same frequency of 10 seconds. It is important that the policy refresh rate is not significantly lower than the rate at which the traffic varies. As we verify later on in the simulations, increasing the frequency of agent querying does not incur any great costs in communication. The delay metric considered is the average end-to-end delay for flow groups.

Benchmarks. We consider three different simulation cases. The first is smart load balancing alongside smart queuing (our two DGN based approaches). The second is smart load balancing alongside classic WFQ with fixed weights relative

to the reward values, and the third is smart load balancing alongside FIFO queue management.

Training of agents. Tables II and III detail the parameters for the smart queuing and smart load balancing DGN agents, respectively. When choosing these sets of hyper-parameters, the objective is to create the smallest neural network capable of addressing the problem. Each of these learning models needs its hyper-parameters tuned. For instance, the DQN algorithm's performance would degrade if it had one fully connected layer instead of two, but it wouldn't improve if it had three. These parameters were set intuitively following models in the state-of-the-art and through testing. The exploration rate ϵ dictates how often during training we take random actions, and how often we utilize the trained model.

1) Smart load balancing and queue management: We first consider the double learning approach. The parameters for the DGN agents are listed in Tables II and III. Figure 5 has the results in terms of throughput per flow class. The dashed lines show the throughput requirements. We note that the joint approach is able to meet all the requirements. For the gold flows, it shows a minimum value close to 3050 Kbps, for the silver flows a median close to 1300 Kbps and for the bronze flows at around 600 Kbps. All above the requirements.

We additionally look at the results in terms of the end-to-end delay. Figure 6 has the results. With the requirements set at 0.2/0.5/1 seconds for gold/silver/bronze flows, we note that we are able to satisfy all the requirements. With median gold flow delays well below 0.1 seconds, and bronze median delay values at around 0.85 seconds.

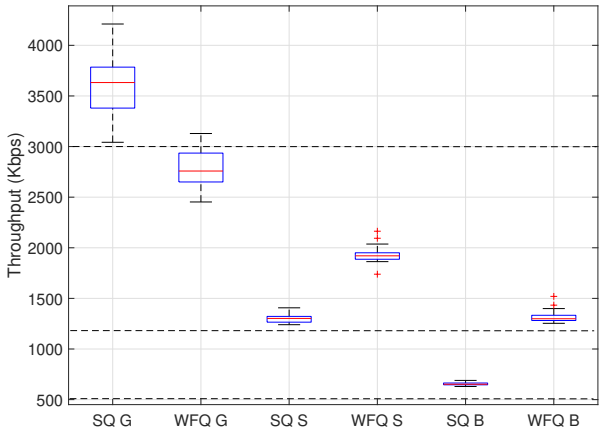

Figure 5: sLB and SQ vs sLB and WFQ: throughput results. The removal of SQ leads to violations in gold flows' throughput results.

2) Smart load balancing with WFQ: We substitute the smart queuing with a classic weighted fair queuing approach. The weights are chosen following the importance of the flows in proportion to the reward values used as input to the smart queuing learning approach before. Although aided with the help of the smart load balancing approach, removing the SQ approach in this case prevents us from meeting the set requirements. Figures 5 and 6 have the results for the throughput and

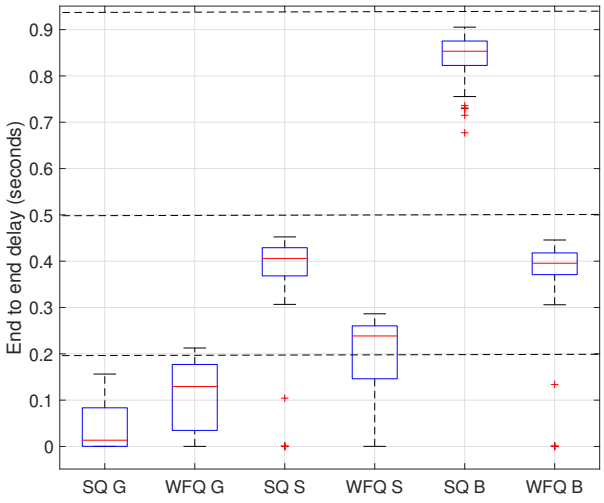

Figure 6: sLB and SQ vs sLB and WFQ: delay results. The removal of SQ leads to violations in gold flows' delay results.

delay, respectively. In both cases, the requirements for the gold flows are severely violated. For the throughput, the median value is about 2500 Kbps well below the requirements, with violations recorded in more than 75% of the cases. For the delay values, there are violations in about 20 % of the cases for the gold flows.

3) Smart load balancing with FIFO: Finally, we additionally replace the SQ part with classic first in, first out (FIFO). The objective of this experiment was to highlight the need for queue management when we have an SLA objective to meet, and show that the network we considered on its own is unable to sustain the requirements for all the flow types. Figure 7 has the throughput results. We note that all the gold flow results are in violation of their requirements. Similarly, for the delay results shown in Figure 8, we note that more than half the gold flows show delay violations.

C. Scalability

We aim to address the scalability of our joint proposal. As such, we consider a scenario based on the ION-NY topology from the topology zoo dataset (Figure 9). The network has 125 nodes. We consider 20 hosts connected to like numbered nodes. 17 of them are transmitting and three are acting as receiving ends. The hosts are spread out to count for different cases and possible impacts. Sending hosts are placed at nodes such as s0, s1, s9, s40, s41, s100, etc, and the receiving hosts at nodes s4, s44, and s114. The parameters for the DGN agents are the same as in the previous scenario. Only edge nodes have agents attached. Table IV has the simulation parameters for this scenario.

We compare our joint learning approach (smart load balancing alongside smart queuing) to the case where smart load balancing is replaced with ECMP, with SQ still present on the ports. We look at the results in terms of throughput and delay.

For the throughput results, seen in Figure 10, we note that the joint learning approach is able to meet the throughput requirements for all flow classes. It shows a minimum value

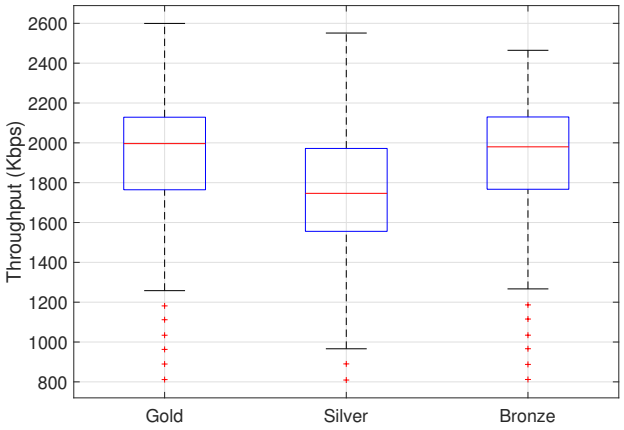

Figure 7: sLB and FIFO throughput results. Removal of SQ removes any aspect of intelligent queuing causing significant violations in gold flow throughput requirements.

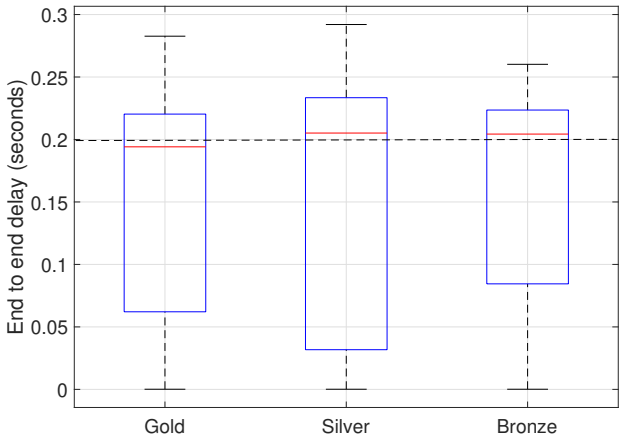

Figure 8: sLB and FIFO delay results. Complete removal of SQ causes significant violations in gold flow delay requirements.

Table IV: Simulation parameters for the large-scale scenario

Parameter	Value
Number of O-D pairs	17 with 51 different flows
Transport protocol	Mixed UDP-TCP
SLA Throughput G/S/B flows	3000, 1200, 600 Kbps
SLA Delay G/S/B flows	0.2/0.5/1 seconds
Link BWs	variable 20-80 Mbps per link
Transmit rate per source	1200-1400 packets/second
Packet size	512 Bytes

higher than 3000 Kbps for the gold flows, and a median value close to 1500 for the silver ones. Violations are recorded for all the ECMP-based flows, with almost all the bronze flows being in violations and more than 60 % of their gold counterparts as well.

For the delay results, in Figure 11, with ECMP alongside smart queuing, only the gold delay requirements are met.

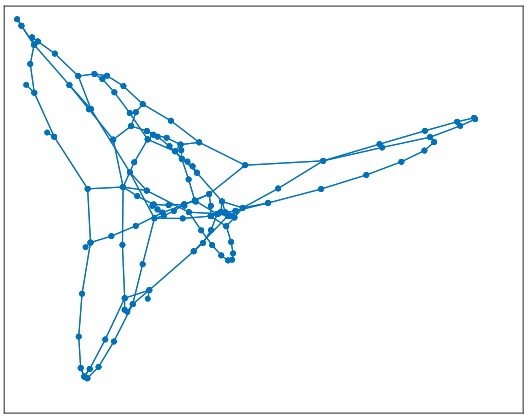

Figure 9: ION network topology

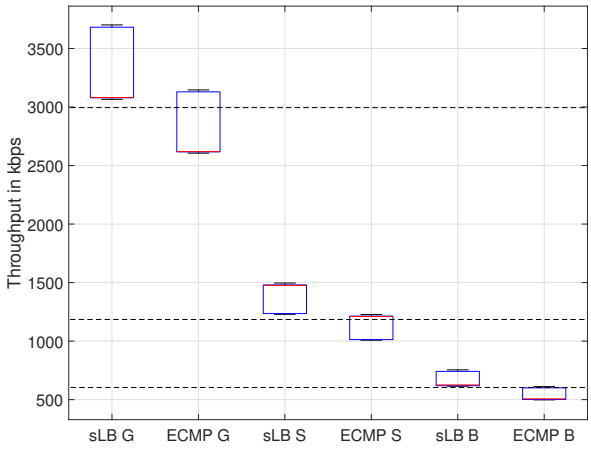

Figure 10: Smart load balancing vs ECMP in the presence of SQ: throughput results. Removal of sLB reduces throughput.

However, we record violations in about 30 % of the silver flows and similarly for the bronze flows. Our joint learning approach can meet all the set requirements.

D. Communication Overhead

During the execution phase, the agents need to communicate their feature vectors *i.e.,* the output of the convolutional layers. While the size of inter-agent communications is usually of concern in distributed learning approaches, we aim to prove that it is not an issue with our proposals. We determine the incurred overhead using two methods. First, in order to quantify the overhead involved, as proposed in [21], we compute it as a function of the total number of pairs of agents that communicate during a certain time instance $t \in T$, denoted g_t, and the total number of agent pairs R as:

$$\beta = \frac{\sum_{t=1}^{T} g_t}{RT} \tag{7}$$

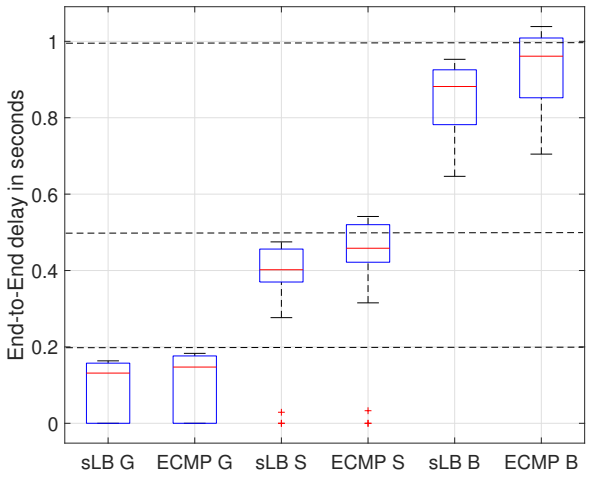

Figure 11: Smart load balancing vs ECMP in the presence of SQ: delay results. Removal of sLB increases latency.

If the value of this ratio is close to zero, it means the overhead is negligible. On the other hand, if the value is close to 1, it means all agents communicate with each other *i.e.,* equivalent to a full mesh. In the case of the SD-WAN scenario, the ratio is about 0.35 for our smart queuing approach and 0.09 for the load balancing algorithm. In the case of the large-scale network topology, it is less than 0.16 for smart queuing and about 0.07 for load balancing. This indicates a very small overhead and communications limited to where needed.

We also assessed the overhead in terms of bandwidth required. In DGN, the agents exchange latent features. Let m be the number of convolutional layers, that is also the number of messages to be exchanged every time a decision needs to be taken. The agents are queried every 10 seconds to send $s.m$ bits, where s is the size of the message. To estimate the load: consider a scenario where we have 2 convolutional layers, $m = 2$. We have 128 nodes in the convolutional layers, which output a message of dimensions (1x128). We need 4 bytes to encode a float, that is a total of 512 bytes per message. On each link, for the communications between two agents, we are thus exchanging messages at the rate of $((512 * 8\ bits) * 2\ layers)/10s$ = 0.8192 Kbps. Even as the number of convolutional layers or communicating agents increases, we can note that the bandwidth needed to exchange messages between different agents remains quite limited.

VIII. Conclusion

In this paper, we proposed a joint smart load balancing and smart queuing approach for networks. We utilized a subset of deep learning known as graph convolution deep reinforcement learning to model the different agents as nodes of a network-representing graph. We compared our joint algorithms to classical methods such as Equal-Cost Multi-Path for load balancing and traditional Weighted Fair Queuing for queue management, and showed that our proposals improve throughput and end-to-end delay and are better suited to meet service level agreements. Finally, we computed the overhead required for our algorithms to function during their execution phase and showed that the utilized attention mechanisms and neighborhood policies keep this overhead at a minimum.

References

[1] Q. Lin, Z. Gong, Q. Wang, and J. Li, "RILNET: A Reinforcement Learning Based Load Balancing Approach for Datacenter Networks," in *Machine Learning for Networking - First International Conference, MLN 2018, Paris, France, November 27-29, 2018*, ser. Lecture Notes in Computer Science, vol. 11407, 2018, pp. 44–55.

[2] R. Adams, "Active queue management: A survey," *IEEE Communications Surveys Tutorials*, vol. 15, no. 3, pp. 1425–1476, 2013.

[3] T. Frantti and M. Jutila, "Embedded fuzzy expert system for adaptive weighted fair queueing," *Expert Systems with Applications*, vol. 36, no. 8, pp. 11 390–11 397, 2009.

[4] A. Sayenko, T. Hämäläinen, J. Joutsensalo, and L. Kannisto, "Comparison and analysis of the revenue-based adaptive queuing models," *Computer Networks*, vol. 50, no. 8, pp. 1040–1058, 2006.

[5] M.-F. Homg, W.-T. Lee, K.-R. Lee, and Y.-H. Kuo, "An adaptive approach to weighted fair queue with QoS enhanced on IP network," in *Proc. IEEE TENCON*, vol. 1, 2001, pp. 181–186.

[6] K. Kaur, J. Singh, and N. S. Ghumman, "Mininet as software defined networking testing platform," in *International Conference on Communication, Computing & Systems (ICCCS)*, 2014, pp. 139–42.

[7] C. Hopps, "Analysis of an Equal-Cost Multi-Path Algorithm," RFC 2992, November, Tech. Rep., 2000.

[8] O. Houidi, D. Zeghlache, V. Perrier, P. T. A. Quang, N. Huin, J. Leguay, and P. Medagliani, "Constrained Deep Reinforcement Learning for Smart Load Balancing," in *2022 IEEE 19th Annual Consumer Communications & Networking Conference (CCNC)*, 2022, pp. 207–215.

[9] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016*, 2016.

[10] A. Oroojlooyjadid and D. Hajinezhad, "A Review of Cooperative Multi-Agent Deep Reinforcement Learning," *ArXiv*, vol. abs/1908.03963, 2019.

[11] J. Jiang, C. Dun, T. Huang, and Z. Lu, "Graph Convolutional Reinforcement Learning," in *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[12] P. T. Anh Quang, S. Martin, J. Leguay, X. Gong, and X. Huiying, "Intent-based routing policy optimization in sd-wan," in *ICC 2022 - IEEE International Conference on Communications*, 2022, pp. 4914–4919.

[13] O. Houidi, S. Bakri, and D. Zeghlache, "Multi-Agent Graph Convolutional Reinforcement Learning for Intelligent Load Balancing," in *NOMS 2022-2022 IEEE/IFIP Network Operations and Management Symposium*, 2022, pp. 1–6.

[14] M. Kim, M. Jaseemuddin, and A. Anpalagan, "Deep reinforcement learning based active queue management for iot networks," *Journal of Network and Systems Management*, vol. 29, no. 3, pp. 1–28, 2021.

[15] V. Balasubramanian, M. Aloqaily, O. Tunde-Onadele, Z. Yang, and M. Reisslein, "Reinforcing Cloud Environments via Index Policy for Bursty Workloads," in *NOMS 2020*, 2020, pp. 1–7.

[16] B. Liao, G. Zhang, Z. Diao, and G. Xie, "Precise and Adaptable: Leveraging Deep Reinforcement Learning for GAP-based Multipath Scheduler," in *Proc. IFIP Networking*, 2020.

[17] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proc. NeurIPS*, 2017.

[18] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[19] C. Tessler, D. J. Mankowitz, and S. Mannor, "Reward constrained policy optimization," 2018.

[20] S. Avallone, S. Guadagno, D. Emma, A. Pescapè, and G. Ventre, "D-itg distributed internet traffic generator," in *First International Conference on the Quantitative Evaluation of Systems, 2004. QEST 2004. Proceedings.* IEEE, 2004, pp. 316–317.

[21] S. Q. Zhang, Q. Zhang, and J. Lin, "Efficient communication in multi-agent reinforcement learning via variance based control," *Advances in Neural Information Processing Systems*, vol. 32, 2019.