Graph Convolutional Reinforcement Learning for Collaborative Queuing Agents

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

***Abstract*—This paper explores the use of multi-agent deep learning as well as learning to cooperate principles to meet strict service level agreements, in terms of throughput and end-to-end delay, for a set of classified network flows. We consider agents built on top of a weighted fair queuing algorithm that continuously set weights for three flow groups: gold, silver, and bronze. We rely on a novel graph-convolution based, multi-agent reinforcement learning approach known as DGN. As benchmarks, we propose centralized and distributed deep Q-network algorithms and evaluate their performances in different network, traffic, and routing scenarios, highlighting both the effectiveness of our proposals and the importance of agent cooperation. We show that our DGN-based approach meets stringent throughput and delay requirements across different scenarios, decreasing silver and bronze flow median waiting delays by more than 50 % and reducing the SLA violations of the latter by nearly 60 %, with respect to a classic priority queuing approach.**

***Index Terms*—Smart Queuing, Adaptive WFQ, Deep Reinforcement Learning, MADQN, DGN, Multi-Agent Systems.**

I. Introduction

Traffic scheduling is key to control how bandwidth is shared among different applications and in particular, to satisfy Service Level Agreements (SLA) of applications in terms of throughput, delay, loss and jitter. In typical Software-Defined Wide Area Networks (SD-WAN) architectures [1], a centralized controller maintains a set of policies deployed at edge routers that interconnect multiple sites (enterprise branches, data centers). Each edge router is configured to send traffic to its peers over several transport networks (*e.g.*, private lines based on MPLS or cheaper broadband internet connections). Typically, these routers are responsible for applying routing and queuing policies to meet SLA requirements in terms of end-to-end Quality of Service (QoS), security, etc. At a slow pace, the controller optimizes policies, while edge devices make real-time decisions.

Several solutions [1] have been proposed for the dynamic selection of paths in WAN networks to satisfy SLA requirements. The general idea is to compare the quality of paths with application requirements and update the path selection strategy inside routers when needed. Beyond path selection, a number of adaptive queuing and Active Queue Management (AQM) techniques [2] have been proposed to help sustain delay and throughput requirements. In particular, the dynamic adaptation of scheduling parameters, such as the weights in Adaptive Weighted Fair Queuing (AWFQ) [3]–[5], has been shown to significantly improve performance. Nonetheless, existing mechanisms are local and work at the level of individual routers in the network, without trying to explicitly cooperate to globally improve the QoS. In [6], for instance, an agent at the destination informs the source node of delay limit violations, so the upstream agent adjusts its queuing weight, but there is no cooperation or sharing of information across agents. In our work, we design a multi-agent system based on deep reinforcement learning with the objective of improving queue management in networks.

To this end, we propose a set of Deep Reinforcement Learning (DRL) algorithms that optimize queuing parameters to meet SLA requirements. We consider a typical SD-WAN scenario in which routers deal with an array of classified flow groups with different requirements in terms of throughput and latency. We consider that all these flows can be divided, according to their priority, into three classes: gold, silver, and bronze [7]. The gold group contains network control information and voice and video traffic. The silver group contains real-time service packets, and the bronze flows consist of batch and non-essential services. A WFQ approach is set up to control how each flow group is served at ingress nodes. Traffic belonging to each of these three classes is sorted into its own queue. Our DRL algorithms are embedded into agents controlling WFQ weights for each flow group depending on the traffic and network status at hand. The delay on each path, as well as the eventual achieved throughput by the flows, depends on the interfering traffic present in other queues and on other paths. This necessitates dynamically tuning the weights and motivates cooperation between agents managing the different nodes, *i.e.*, routers. While closed-form expressions for WFQ [5] can be used to tune weights locally, a machine learning approach can better adapt to realistic traffic patterns and generalize to the case where multiple agents are interfering (*e.g.*, sending traffic over the same links) and end-to-end QoS requirements must be met.

We utilize a multi-agent approach to tackle the problem. In Multi-Agent Reinforcement Learning (MARL), multiple methods exist to govern agent communication and cooperation. The MARL system could be completely centralized, fully distributed, or semi-distributed. In a centralized MARL system, all the agents act as one, sharing the same environment, states, actions and rewards. In a distributed one, the agents are completely independent, and in a semi-distributed MARL architecture, these distributed agents are able to communicate and cooperate.

In this paper, our main proposal consists of the application of a graph convolutional reinforcement learning (DGN) approach to the multi-agent smart queuing problem. DGN is a semi-distributed approach to MARL in which the collaboration between agents can be parameterized and learned. In addition, as benchmark solutions covering other MARL architectures, we propose two Multi-Agent Deep Q-Learning (MADQN) solutions based on DQN [8]. One is completely centralized and the other is fully distributed.

In our work, we discuss how both the DGN and MADQN agents learn. We detail their observations, actions, rewards, and the extent of their cooperation across the different considered approaches. We perform packet-level simulations in ns-3 [9] and compare our proposals against traditional Priority Queuing (PQ), in both SD-WAN and classic network topologies. We show that our proposals are better suited to deal with classified traffic than PQ. While DGN is always capable of meeting the required throughput and delay demands, we illustrate how the lack of agent cooperation in the distributed MADQN approach can cause the latter to falter in convoluted scenarios. And while the centralized approach to MADQN can meet the set objectives, we show that DGN can do it without the need for a centralized setting and with negligible overhead during execution.

Our main contributions in this work can be summarized as follows:

- We propose a graph convolutional reinforcement learning based approach to smart queuing.
- The approach uses attention mechanisms and neighborhood communication to set the weights per flow class served at routing nodes.
- We propose two deep learning mechanisms, one centralized and one distributed, to serve as reinforcement learning benchmarks to better assess our main proposal.
- We consider multiple network scenarios including an SD-WAN topology and a classic network in the Abilene topology. Our approach is studied for both UDP and TCP traffic.
- We show the gains that our approach brings in terms of decreasing SLA violations and reducing waiting delays with respect to the considered benchmarks.
- We study both the scalability of our DGN approach and the overhead incurred due to inter-agent communications, showing that our proposal fairs well on both fronts.

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 and the WFQ approach we build our agents upon. Section IV introduces our graph convolutional reinforcement learning based proposal for smart queue management. Section V details both our centralized and distributed deep Q-learning approaches to the smart queuing problem. Section VI presents the simulation results and analysis, while Section VII concludes this paper.

II. Related Works

In this section, we discuss the related works on smart queuing and the utilization of deep reinforcement learning (DRL) in network management. In this context, we first focus on active queue management and traffic engineering. Afterwards, we discuss graph convolutional reinforcement learning, which our proposal in this paper is based upon, and present multiple approaches in the state-of-the-art which utilized it. Finally, we highlight some algorithms which revolve around the SD-WAN environment, the system model we considered in our work.

In terms of what we aimed to accomplish in this work, the paper of Kim, Jaseemuddin and Anpalagan [10] is the closest. The authors propose a DQN based AQM algorithm in a single-agent environment, wherein the agent decides which packets to serve from the queue and which ones to drop. Other queue-based DRL usages can be seen in the paper of Balasubramanian *et al.* [11], where the agents decide which request traffic instances are to be served first, and in the work of Bachl, Fabini and Zseby [12], where they are tasked with finding the optimal buffer sizes.

DRL approaches in the domain of traffic engineering in general and Multi-Path TCP (MPTCP) specifically are also popular. Rosello [13] proposed a DQN agent with the purpose of selecting the optimal paths for MPTCP, while Liao *et al.* [14] used an actor-critic framework to the same end. Houidi *et al.* [15] proposed a multi-agent actor-critic framework to perform path selection and optimize Quality of Experience (QoE).

Kattepur *et al.* [16] used multi-agent deep reinforcement learning to sustain differentiated service guarantees in fat tree networks. While the agents at the spine and super-spine level may make use of Equal-Cost Multi-Path (ECMP) routing or intelligent load balancing, the leaf agents have other configurations such as decreasing flow rate, changing priority of flows, and increasing packet drop rates. The authors demonstrate the utility of their proposed agents and show that their approach achieves 6 % latency and 34 % throughput improvement with respect to vanilla ECMP.

Yao *et al.* [17] also use deep reinforcement learning to tackle the load balancing task. They formulate the latter as a decentralized partially observable Markov decision problem, which induces the MARL approach. They utilize experiments on a realistic test bed to show that their approach outperforms classic ones such as Weighted-Cost Multi-Path and Local Shortest Queue.

In this paper, we propose a graph convolutional reinforcement learning multi-agent approach for optimal weight selection in a network using WFQ schedulers. The objective is to meet delay and throughput requirements for a set of classified network flows. Originally proposed by Jiang *et al.* [18], DGN aims at learning how agents cooperate in a MARL environment. It uses attention [19] and adjacency matrices to extract relevant features and relay important information where needed. With respect to the state-of-the-art on cooperation in multi-agent deep reinforcement learning, DGN utilizes attention mechanisms similar to those proposed by Jiang & Lu [20], whilst avoiding its full-scale communication. It uses parameter sharing as done in the proposal by Zhang *et al.* [21], but without assuming a fully observable environment. And finally, while DGN was not the first proposal to utilize a graph convolutional network, it does so in a partially observable environment whilst allowing for a dynamic adjacency of agents.

Multiple papers in the state-of-the-art propose graph based deep reinforcement learning approaches. Houidi *et al.* [22] use graph convolutional reinforcement learning to propose a smart load balancing algorithm. Their approach models the network as a graph and derives, through a graph convolutional method, the policy that splits traffic flows across end-to-end candidate paths while meeting application QoE requirements. They use throughput and delay as performance metrics highlighting how graph convolutional deep reinforcement learning is ideal for decision making in networks. Jiang *et al.* [18] use graph convolutional reinforcement learning to propose a hop-by-hop routing prediction policy wherein each packet is an agent. Their approach attests to the scalability of these deep learning approaches and yet again shows how graph-based deep reinforcement learning is well suited for network problems. Another graph reinforcement learning approach to packet routing can be seen in the paper by Mai *et al.* [23]. In their proposal, the routers interact with the network and learn from experience the best routing configurations. They compare their approach to classic routing methods such as static shortest path and Q-learning, as well as deep learning approaches based on deep Q-networks, and show that their proposal outperforms these benchmark algorithms in terms of transmission delay and affordable load.

In the context of SD-WAN networks, Quang *et al.* [24] aimed at optimizing routing policies from a centralized network controller. As such, they integrated data-driven SLA predictions into a local search algorithm to optimize routing policies. Their proposed approach supports multiple intents such as the minimization of the congestion or the maximization of the network quality. Additionally, a dynamic control load balancing approach for SDN environments, dubbed MARVEL, is proposed by Sun *et al.* [25] utilizing multi-agent deep reinforcement learning. They use the latter to determine switch migration actions (controller-switch mapping). The training is done offline and the decision making is done online. The authors attest that their proposal improves the control plane's general processing ability by 27 % while reducing its processing taking time by 25 %.

Manel *et al.* [26] propose a multi-agent reinforcement learning approach to optimize routing decisions in SD-WAN networks. Mainly, they aim to ensure load balancing among each network as well as optimized resource utilization of inter-domain links. Their simulations show that their proposal outperforms solutions such as Border Gateway Protocol (BGP)-based routing. Finally, Troia *et al.* [27] develop an SD-WAN specific traffic engineering algorithm. The objective is to improve their performance in terms of service availability. They use three different deep learning approaches: policy gradient, TD-λ and deep Q-learning with a reward function representing the overall network availability.

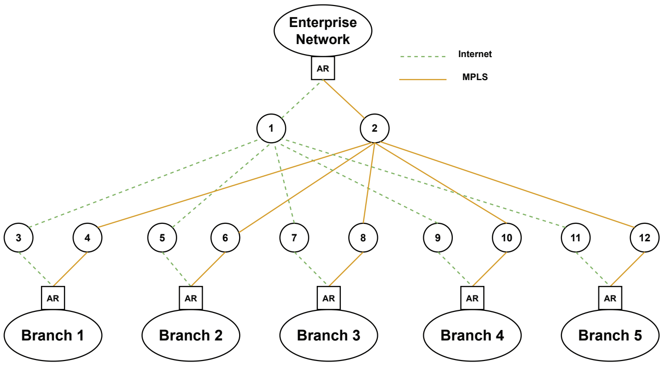


Figure 1: SD-WAN network with 5 branches

Our paper goes beyond the related works by considering a multi-agent architecture based on reinforcement learning algorithms for the adaptive tuning of queuing parameters. Unlike the majority of the works in the state-of-the-art, we don't look for path selection, but for adapting weights to guarantee QoS requirements. In our work, the agents are embedded in the routing nodes themselves, giving them real-time direct access to network decisions.

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 afterwards we discuss the scheduling approach on which we built our reinforcement learning proposals.

A. SD-WAN Use Case

Figure 1 presents a typical SD-WAN use case where an enterprise network headquarters (HQ) and five remote branches are interconnected by MPLS and broadband internet connections controlled by third-party operators. A controller is placed at the headquarter site and Access Routers (ARs) are responsible for the interconnection. 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 operating at different time scales. In the first control loop, the global controller (at the headquarter site) updates policies and communicates them to edge devices ($i.e.$, AR devices). In a second control loop, devices take tactical decisions to follow the evolution of traffic and network conditions. Figure 2 depicts the architecture of an AR device. The traffic of each flow group is first load balanced over available access networks

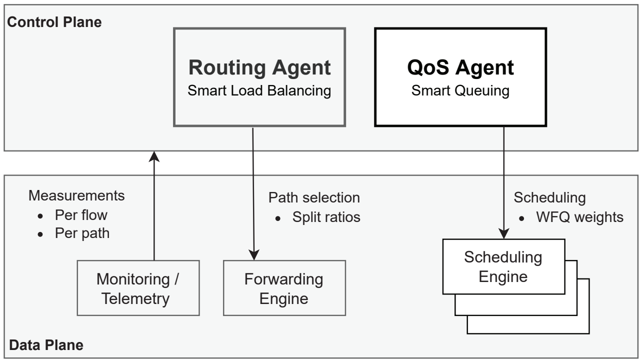


Figure 2: Access router architecture

($e.g.$, internet, MPLS) using a *routing agent* (as in [24], [25], and others) and then 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 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. The focus of this paper is on the smart queuing part of the aforementioned architecture. In what follows, we consider that the routing policy is already decided, and we discuss only traffic scheduling, with the integration of the two being the subject of our future works.

Our objective is to satisfy SLAs for 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. To do so, we enlist the aid of DRL to continuously optimize queuing parameters. In what follows, we discuss our WFQ approach and the QoS agent's role. Because of the graph-like structure of SD-WAN topologies, a graph convolutional approach to the problem was ideal. The different agents represent the different nodes of the graph, and the number of convolutional layers directly relate to the number of hop communications required to ensure coordination between all the agents, as we detail later on.

Finally, we note that in addition to our SD-WAN use case, we also test our proposal in a more generalized network topology, namely the Abilene topology, and in a large scale network topology as well. We show that our learning model is resilient to different topologies and can adapt to the objectives regardless of the scenario at hand.

B. Adaptive Weighted Fair Queuing and DRL Agents

While strict priority queuing is generally used to prioritize traffic, WFQ can be used to maintain fairness, and its weights can be adjusted so that each flow group of traffic receives a bandwidth proportional to its weight. The latter also impacts the resulting end-to-end delay experienced by the flows. Let $\{1, \ldots, K\}$ denote the set of flows. In a WFQ scheduler, each flow achieves an average data rate R_k equal to:

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

where R is the total link capacity, and w_k is the weight associated with flow k. As such, the greater the weight of the flow is, the higher its service rate and the lower its local queuing delay are (see latency-rate server model [28]).

We assume that every flow in the SD-WAN network can, based on packet priority, be classified into three groups. These flow groups are in order of importance: gold, silver, and bronze. Each group has its set of minimum throughput thresholds to be attained: T_g, T_s, and T_b for gold, silver, and bronze, respectively and an equivalent set of maximum end-to-end delay thresholds to be respected: d_g, d_s, and d_b. The objective of the DRL agents for QoS is to assist in meeting these thresholds by learning how to continuously update the weights (increase or decrease) for each flow group served by the WFQ algorithms. Each agent in this MARL deployment is built on top of a WFQ scheduler. While WFQ is used in our SD-WAN use case, the proposed solution is generic enough to handle any other scheduling architecture. The agents observe the throughput and end-to-end delay values attained by the flow groups, and then make individual decisions on whether to increase or decrease the weights for the flow groups that are served at their corresponding nodes. With the delay and throughput values being influenced by how the packets traverse the entire network, inter-agent communications are expected to be a key feature.

IV. Graph Convolutional Reinforcement Learning For Multi-agent Systems

The objective of the deep learning agents is to continuously adjust the weights, either by increasing or decreasing them, for a weighted fair queuing algorithm managing a set of classified network flows. These agents are situated at ingress nodes across the network, such as the numbered ones in our illustrated scenario in Figure 1. In this paper, our main proposal utilizes multi-agent graph convolutional reinforcement learning, or DGN, to manage both how the agents learn and communicate. DGN combines the ideas of graph neural networks and deep reinforcement learning. The agents are embedded in a graph $G = (V, E)$, whose topology is related to the computer network in our scenario. The existence of an edge between two agents in this graph means that they can exchange information. Each node (agent) $i \in \mathcal{N}$, where $\mathcal{N}$ denotes the set of agents, has a set of neighbors $\mathcal{B}_i$ with which it can communicate. This collaboration between agents can be parameterized and is dependent on an adjacency matrix $\mathcal{C}$ that defines which agents are neighbors. Limiting agent communication to neighbors reduces what could be costly interactions, in terms of bandwidth and complexity, while keeping the neighborhood present between agents that are likely to impact each other the most.

A. Multi-agent System, Replay Buffer and Target Network

In DGN, the learning problem is formulated as a partially observable Markov decision process. During every time iteration t, each agent i receives a local observation from the environment denoted o_i^t. The latter consists of a set of values

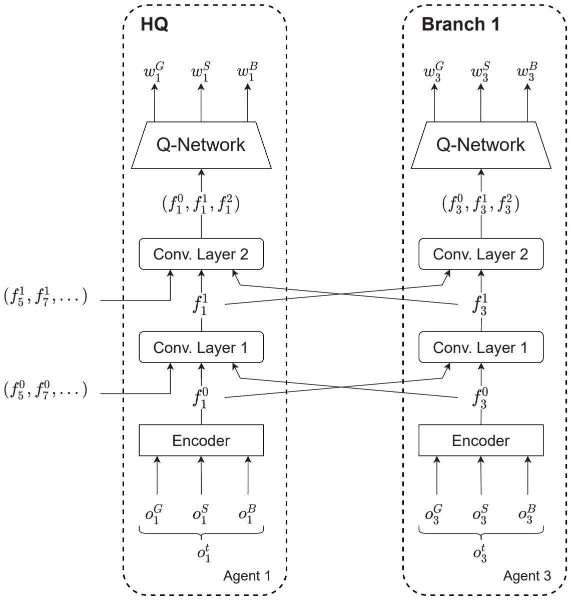


Figure 3: Structure of two DGN agents at HQ and Branch 1

detailing the end-to-end delay and throughput values of the flow groups it is serving. The agent then takes an action a_i^t, increasing or decreasing the WFQ weight of each flow group, and as a result is issued a reward r_i^t determined by whether the SLA requirements for the flows groups are met or not. The aim is to maximize the sum of the expected rewards of all the agents.

Multi-agent collaboration. Each agent i will run its own reinforcement learning algorithm, whose purpose is to learn how the weights (w_i^G, w_i^S, w_i^B), for the gold / silver / bronze flow groups, should change with respect to the local observations and information received from its neighbors. As displayed in Figure 3, this reinforcement learning agent is composed of multiple modules. The first module, a multi-layer perceptron (MLP) referred to as an encoder, takes as input the local observations of the agents and extracts the relevant features, referred to as f_i^0, of these observations. Once each agent i has its computed features f_i^0, it will send them to its neighbors and receive their features, which reflect their own observations.

In Figure 3, agent 1 at the HQ sends f_1^0 to all the branches it connects to and receives f_3^0, f_5^0, etc..., from the respective branches. Agent 1 at the HQ shares information with all the internet branch agents as justified by its adjacency matrix, which details its neighborhood. Recall that in our implementation, agents that share links at the network layer are considered to be neighbors. These features will be the input of the second module, which is a convolutional neural network. Similar to the encoder, the role of the convolutional network is to extract the relevant features of the combination of the local observations and the features received from neighbors. As suggested by the figure, a multiple convolutional layer module can be used. Each layer takes as input the features computed by the preceding convolutional layer, as well as a new set of features received from the neighbors. In our work, we consider two convolutional layers. Similarly, as it is performed by distance vector routing to learn the shortest path by exchanging routing tables with neighbors, the exchange of features between agents will permit the agents to obtain local knowledge from agents that are at a distance h from them, where h denotes the number of layers in the second module. For example, the second convolutional layer of node 3 at branch 1 will receive the feature f_1^1 from HQ node 1, which contains information received from the rest of the branches. Even if these branches cannot communicate directly, the exchange of features with the HQ nodes will permit them to have a full view of the network information. After several stages of convolutional layers, all the information computed will be gathered into a vector of features. The last module is a Q-learning algorithm. It takes as input the features produced by each layer of the convolutional layer. The reinforcement learning algorithm will run on this third module and the decisions, which maximize the expected reward, on the weights will be made by it.

Attention mechanisms. The convolutional layers of DGN implement attention mechanisms. Convolutional kernels, widely present in Convolutional Neural Networks (CNNs) and image recognition, enable extracting features from images. In DGN, these kernels integrate the features in the receptive field in order to extract the latent features. They should be able to learn how to abstract the relationship between agents as to integrate their input features. DGN uses a multi-head dot-product convolutional kernel to calculate the interactions between different agents. A more in depth illustration of how attention works in neural networks can be found in [19].

Replay buffer. DGN implements a replay (experience) buffer, $i.e.$, samples are stored in a memory and afterwards randomly sampled for training. This removes any correlation that might exist among consecutive samples. The experiences are of the type ($\mathcal{O}$,$\mathcal{A}$,$\mathcal{O}'$,$\mathcal{R}$,$\mathcal{C}$), where $\mathcal{O}$ is the set of agent observations $\{o_1, ..., o_N\}$, $\mathcal{A}$ is the set of agent actions $\{a_1, ..., a_N\}$, and as such $\mathcal{O}'$ is the set of new observations $\{o'_1, ..., o'_N\}$ as a result of the taken actions. $\mathcal{R}$ is the set of rewards issued to the agents $\{r_1, ..., r_N\}$, and finally $\mathcal{C}$ = $\{C_1, ..., C_N\}$ is the set of adjacency matrices for the agents. The adjacency matrices essentially define the neighborhoods for the agents. C_i, $\forall$ i, is constructed with dimensions ($|\mathcal{B}_i|$+1) $\times$ N, wherein the upper row is a one-hot representation of the index of the agent i, and the kth row, k = 2,...,$|\mathcal{B}_i|$+ 1, is a one-hot rendition of the index of the (k-1)th neighbor. Note that the time notation t is dropped from these expressions for the sake of simplicity.

Target network. With enough samples in the replay buffer, we are able to train the agents. The training is done with the aid of target networks [8]. A target network is a copy of the agent's main Q-network. Its parameters however are not trained every iteration, but rather updated slowly or every while. This helps root out any instability in training the main Q-network that could arise from consecutive states being very similar. The replay buffer is randomly sampled for a minibatch of size S on which each agent is trained with the purpose of minimizing the loss:

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

where we recall that N is the total number of agents and that

$$y_i = r_i + \gamma \max_{a'} Q(O'_{i,C}, a'_i; \theta'). \tag{3}$$

$O_{i,C} \subseteq \mathcal{O}$ represents the observations of i's neighbors. Q represents the Q-function, θ' the target network parameters, and γ is the discount factor. The latter 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 updated smoothly ($i.e.$, softly) every iteration following:

$$\theta' = \tau\theta + (1 - \tau)\theta', \tag{4}$$

where τ denotes the smoothness of the update. If τ=1, then the update is classified as "hard" and the parameters of the main network are simply copied onto the target network.

Finally, we note that during the training phase, as well as during the execution period, the agents are well aware of their neighborhoods, $i.e.$, their own adjacency matrices. That is to say that they know with which agents they would need to communicate. During this communication, agents share copies of their feature vectors, as illustrated in Figure 3. The significance of this overhead is discussed in the results section.

Spatial and Time Complexity. Following the discussion and details presented in [29], we can assert that the complexity of our approach, both spatially and time wise, is of the order of $O(m^2L)$, where L is the number of hidden layers and m the number of neurons each layer has. That same work shows that without the attention model and inter-agent communications provided by DGN, the complexity would become of the order $O(m^2L \cdot N)$, wherein N is the number of agents. This further attests to the scalability of the proposed approach.

B. DGN based Smart Queue Management

In our work, we enlist DGN to help with our problematic: to meet stringent SLA requirements for classified network flows. The different components presented in DGN are redefined as follows for our problem:

- The local observation, in our case, is a tuple representing the end-to-end throughput and delay values attained the flows served by the agent and denoted $\{\overline{T_g}, \overline{d_g}, \overline{T_s}, \overline{d_s}, \overline{T_b}, \overline{d_b}\}$, where $\overline{T_g}$ represents the throughput of the gold flows, $\overline{d_g}$ the average end-to-end delay of the gold flows, and so on. The end-to-end delays are typically measured using in-band network telemetry.
- The actions taken by every agent throughout the learning problem consist of either increasing or decreasing the weight of every flow group it is serving (gold, silver, bronze) by a preset constant value δ. Each agent will act on the weights of all three groups simultaneously ($\pm\delta$). This means that, in total, each agent has eight possible actions to take at every iteration.
- The reward issued for each agent after it takes an action is relative to whether it has helped meet the requirements for each flow group. Let η_j be the reward for meeting required throughput values of flow group j, and ϕ_j the reward for meeting the delay requirement of the group j. For the reward we are aiming to meet the an average end-to-end delay maximum for the flows of the groups. The total reward r_t issued for an action is then computed as follows:

$$\omega_g^{th}\cdot\eta_g + \omega_g^{d}\cdot\phi_g + \omega_s^{th}\cdot\eta_s + \omega_s^{d}\cdot\phi_s + \omega_b^{th}\cdot\eta_b + \omega_b^{d}\cdot\phi_b, \tag{5}$$

  where ω_j^{th} is set to -1 if the required throughput for flow j is not met and +1 otherwise. ω_j^{d} is its delay equivalent in regard to meeting the target delay values. Consequently, the agent reward can be negative, $i.e.$, a penalty.

The rewards/penalties for meeting the gold flow requirements are set higher than that for the silver, and for this latter higher than the bronze. That is to say, the agent is better rewarded, alternatively penalized more, for meeting or violating the gold flow requirements than they are for those of the silver and bronze flows, respectively. Note that we can weight the rewards/penalties for the delay with respect to those of the throughput. With gold group flows, for example, ϕ_g = $\kappa_g \cdot \eta_g$. If $\kappa_g < 1$, the agents are incentivized to meet the throughput requirements ahead of the delay ones, for the gold flows.

Note that the throughput and delay are continuous values. Since we cannot learn over a space of infinite states, it is important to discretize it. We first need to define the size of the observation space. In our work, we set it to 20. This means that for every observed element, we have 20 possible values. Since our observation is made up of six different inputs, the discrete observation space size is a sextuplet, with each element belonging to a set of 20 different values. The window size is computed as the maximum attainable value minus the corresponding minimum for each element of the state tuple. The discrete state is the integer value resulting from subtracting the minimum observation space from the state and dividing the result by the discrete window size.

Algorithm 1: Discretization of the States

```
Define the discrete observation space size:
    $DISCRETE\_OS\_SIZE = [20, 20, 20, 20, 20, 20]$
Compute the window size: $discrete\_os\_win\_size =$
    $(observation\_space.high -$
    $observation\_space.low)/DISCRETE\_OS\_SIZE$
Function $get\_discrete\_state\,(state)$:
    int $discrete\_state = (state -$
        $observation\_space.low)/discrete\_os\_win\_size$
    return $discrete\_state$
```

An increased state space would mean the algorithm has more room to explore for better solutions, but would incur a time penalty for convergence. That is to say if we set the discrete observation size to 100 for example, we would be encompassing a lot of additional states/observations. Nonetheless, the space would be too large. Striking the correct balance is mostly a matter of approximation and experimenting. Finally, in the annex of this manuscript we detail how the DGN agents are trained.

V. Deep Q-Learning Approach

We present two benchmark multi-agent solutions, based on DQN, to the problem. While DGN is semi-distributed, one of these DQN approaches is completely centralized, and the other fully distributed without any inter-agent communications. Deep Q-learning revolves around the idea of attaching a deep neural network to the traditional Q-learning problem. It aims at solving its memory problem by removing the Q-table and using neural networks to determine the best actions.

For the distributed approach, the agents are considered to be fully independent. They each have their own set of states, actions, and rewards, and they each view the environment from their local perspective. No inter-agent communications exist. Figure 4 shows the structure of the distributed DQN agents in our work. Unlike DGN, wherein we have an encoder and convolutional layers managing inter-agent relations, here we have two fully-connected layers in between the input and output layers. There are no built-in cooperation mechanisms. The actions taken by the agents and the rewards they are issued remain unchanged from before. In DQN, we again utilize two principle deep learning mechanisms: target networks and replay buffers. The replay buffer is filled with experiences of the type $(\mathcal{O}, \mathcal{A}, R, \mathcal{O}', done)$, $i.e.$, current observation, action taken, reward received, and the new observation. As in the case of DGN, the variable $done$ indicates if the learning reached its objective or not. The target network, with parameters θ', is a copy of the main Q-network and is used to stabilize the training. The predicted Q-values of the target Q-network are utilized to backpropagate through and train the main Q-network. However, they themselves are not trained but regularly updated with the values of the main Q-network. In this case, we use a hard update with $\theta' = \theta$.

We train on positive rewards, $i.e.$, the experiences in which the agent does not reach a terminal state are not used to minimize the loss. Once enough experiences are stored in the buffer, the training process starts on randomly selected S minibatches. For every experience, each agent acts as follows:

$$\begin{cases} y_i = r_i, & \text{If } done \text{ = True} \\ y_i = r_i + \gamma \cdot \underset{a' \in \mathcal{A}}{max}\ Q(o_i', a_i'; \theta'), & \text{Otherwise} \end{cases}$$

The loss, which is minimized using stochastic gradient descent, is then computed as follows:

$$\mathcal{L} = \frac{1}{S} \sum_{S} (Q(o_i, a_i; \theta) - y_i)^2. \qquad (6)$$

As for the approach to the weight selection problem itself, things remain virtually unchanged from DGN. We have the same states of states and observations, the same possible agent actions and the reward is calculated in the same manner.

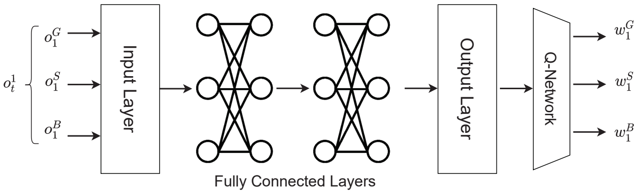


Figure 4: Structure of the DQN agents

For the centralized approach, the agents are trained as if they are one central unit, they interact jointly with the environment, $i.e.$, they share the same state and observations. They also take their actions jointly, as if it is one superimposed action, and they receive a single reward. The agents are practically sharing a complete vision of the environment and their individual interactions with it.

VI. Simulation and Results

We now evaluate the proposed multi-agent architecture for smart queuing using ns-3 [9] with the deep learning agents being built using Python and TensorFlow. We simulate SD-WAN network illustrated in Figure 1. As we considered that routing is controlled by a slower control loop, and it is in steady state (see Section III), the adjustment of queuing parameters inside transport networks ($i.e.$, internet, MPLS) can be considered independently. For this reason, we only simulated one type of transport network at a time, with a scenario of UDP traffic over the internet, and another of TCP traffic over Multiprotocol Label Switching (MPLS), being considered. We simulate HQ-branch links with propagation delays of 10 ms with capacities of 10 Mbps each. Small rates are chosen to speed up simulation duration, however we verified the results are not impacted when the bandwidths are of higher magnitude. We used on-off applications for traffic generation and build the WFQ approach over active queue management techniques, namely random early detection (RED). At each branch, origin-destination (OD) flows for two flow groups are generated towards HQ. For instance, at branch 1 we have gold and silver, at branch 2 we have silver and bronze, and at branch 3 we have gold and bronze, etc. The purpose of this variation is to avoid having homogeneity across the traffic treated by the different agents, and thus create a need for agent collaboration.

Agent communications. In the case of DGN, we recall that the adjacency of the agents in our scenario, $i.e.$, the neighbors with which each agent can communicate, is defined through the presence of links. In Figure 1, HQ agent 2 communicates with nodes 4, 6, etc..., but not 3 or 5, and so on. In the case of distributed MADQN, no inter-agent communications exist whatsoever. In the case of centralized MADQN, the agents act as if they are one unit, sharing the same environment, states, actions and rewards.

Traffic scenario. Table I details the simulation parameters. The transmit rate of the sources follows diurnal and sinusoidal patterns between 0 and 20 Mbps. The HQ-branch links (ex 1-3, 2-4, etc.) have limited bandwidths and are the links where congestion is likely to occur and impact the general performance of the network. The simulations are done as a series of 300 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 eventual results. The weights for the agents are randomly assigned at the start, with that of the gold flow being higher than the silver and the bronze, respectively. 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.

Table I: Parameters for the simulations

Parameter	**Value**
Number of O-D pairs	10, 4 gold, 3 silver, 3 bronze
Snapshot duration / # of snapshots	10 sec / 300
$T_g/T_s/T_b$	30 / 10 / 5 Mbps
$d_g/d_s/d_b$ for UDP	0.15 / 0.3 / 0.4 seconds
$d_g/d_s/d_b$ for TCP	0.1 / 0.15 / 0.2 seconds
Delay to throughput relevance $\kappa_g, \kappa_s, \kappa_b$	0.8
Reward relative to flows G/S/B	$3x/2x/x$
WFQ weight update δ	0.03

Benchmarks. In terms of deep reinforcement learning, we compare our proposal to two multi-agent DQN approaches. One is fully distributed with no agent communications, and the other completely centralized with shared states, actions, and rewards. In comparison to the DQN approaches, the DGN algorithm incorporates attention mechanisms and selective inter-agent communications. In terms of traditional approaches to QoS management in queuing, we simulate a classic priority queuing (PQ) algorithm. The latter serves the packets in descending order of priority. This means that all gold packets are dequeued first, the silver second and the bronze last.

Training of agents. Tables II and III detail the parameters for the MADQN and DGN agents, respectively. When choosing this set of deep learning parameters, our objective was to create the smallest neural network capable of converging towards an efficient solution. This was the case for all of our proposed learning algorithms. 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.

Table II: Parameters for MADQN agents

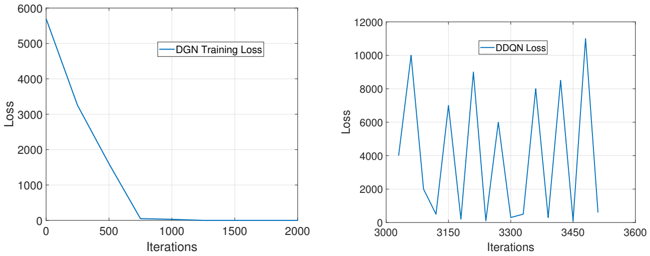
Parameter	**Value**
Activation function	ReLu
N^o of fully connected layers	2 each with 128 neurons
Exploration rate ϵ	starts with 1 and decays to 0.001
ϵ - decay ϵ	multiplied by 0.99955 per episode
Discount factor γ	0.99
Training batch size	32

Table III: Parameters for DGN agents

Parameter	**Value**
N^o of convolutional layers	2
N^o of encoder MLP layers	2
N^o of encoder MLP units	(128,128)
Scaling factor τ	0.01
Discount factor γ	0.99
Training batch size	32

A. Agent Convergence

We first assess if the agents converge or not. In this work, we assess the convergence analytically, a more theoretical discussion on the convergence of DGN is provided in [30][31]. For DGN, the agents are trained in a semi-centralized manner. As such, we track the global loss averaged across all the agents (Equation 2). The DGN approach was able to converge. This is illustrated in Figure 5a, where at around 800 training episodes the loss tends towards zero.



(a) DGN

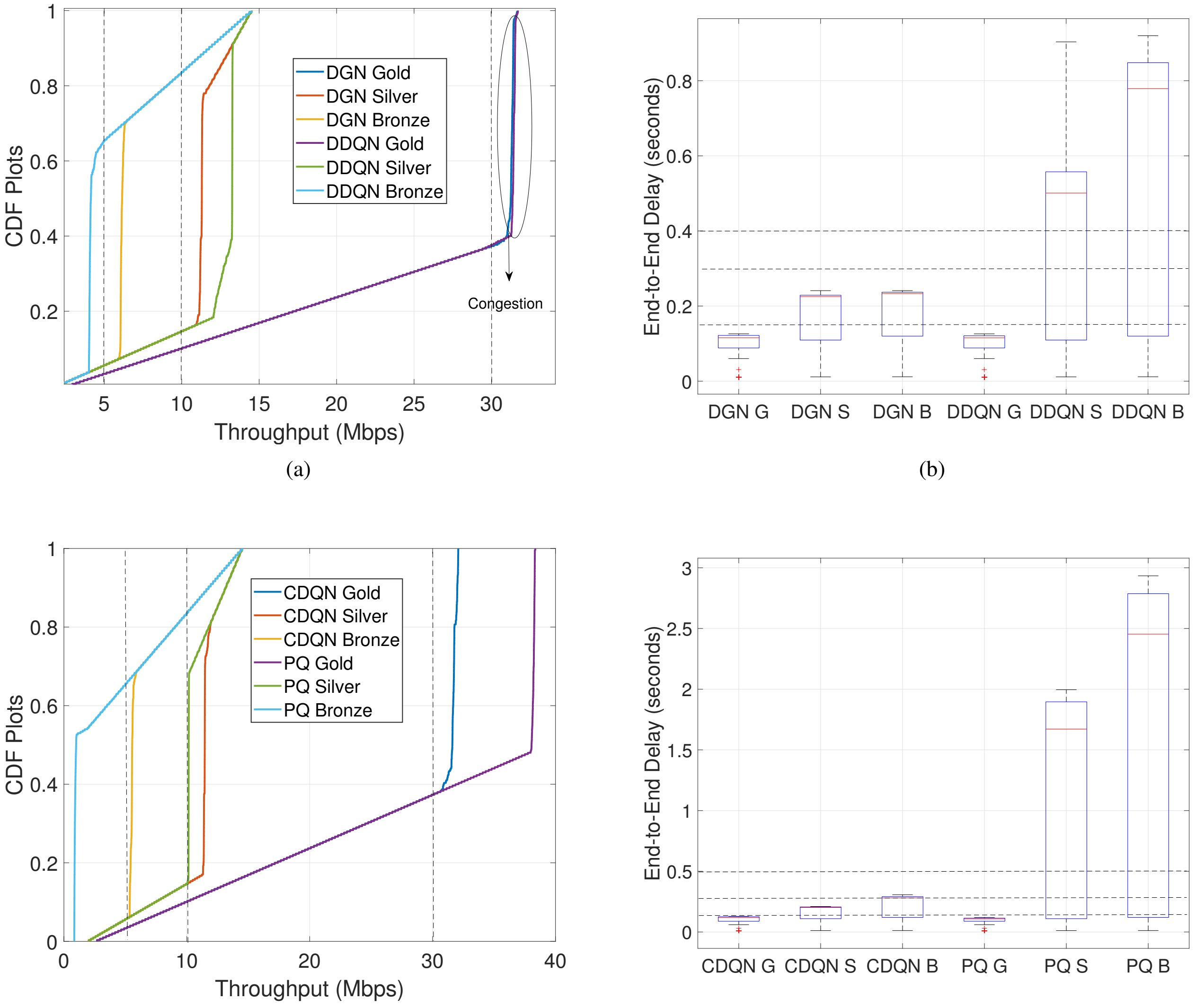
(b) Distributed DQN

Figure 5: Convergence of the learning approaches

For the multi-agent distributed DQN approach, Figure 5b tracks the loss function for a DQN agent after 3000 training iterations. The algorithm is not converging. Even if more time is given for the training, this oscillation remains present. This is mainly due to the completely distributed nature of this distribution. There is a lack of communication between agents in a scenario that requires it, and this is confirmed by the fact that the centralized approach converges without any issues.

B. UDP Scenario

First, we consider simulations with UDP traffic. Starting with DGN, we show a cumulative distribution function (CDF) plot with the throughput values attained by the different flow groups throughout all the snapshots. Figure 6a has the results. The vertical dashed lines show the SLA requirements in terms of throughput per class. The semi-vertical lines, for each flow group and each DRL solution, in the plots show how the network behaves when the traffic sources are transmitting at a congestion causing rate on the links. It can be seen as a sort of steady state. It is during this period that we are mainly concerned in verifying that the SLAs are met. When there is no congestion, in the lower parts of the diurnal traffic patterns for instance, the traffic sources are transmitting at a rate lower than that needed to maintain the required throughput thresholds. As such, we do not take throughput SLA violations in this region into consideration. We note that the DGN approach



(a)

(b)

(c)

(d)

Figure 6: UDP traffic: DGN vs. Distributed MADQN (DDQN) (a) and (b), Centralized MADQN (CDQN) vs. PQ in (c) and (d). G: Gold, S: Silver, B: Bronze. SD-WAN scenario.

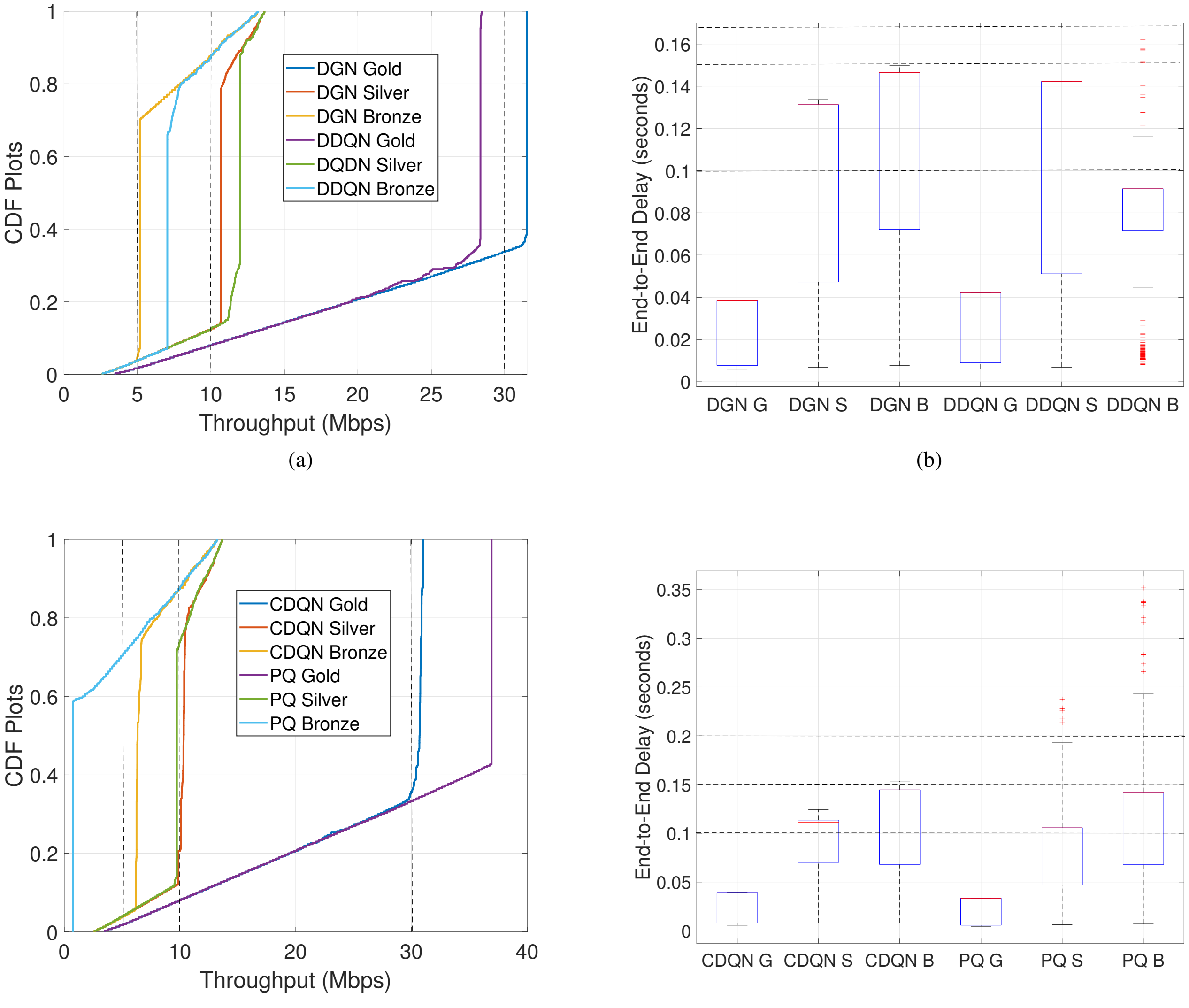
can always meet the required throughput. When the network is congested, the throughput for the bronze flow is just above 6 Mbps, for the silver flow about 11 Mbps, and for the gold flow is around 31 Mbps. All above the required throughput values of 30, 10, and 5 Mbps for gold, silver, and bronze flow groups, respectively.

We additionally look at how DGN performed in terms of the delay attained by the different flows. Figure 6b has box plots with the results. The requirements are 0.15, 0.3, and 0.4 seconds for the gold, silver and bronze flows, respectively. We note that DGN was able to meet all of these thresholds.

We observe on the other hand the results for distributed MADQN agents. Figure 6a has CDF plots with the results. The latter validate what is seen in the training convergence trend. The demands for the bronze flows are not met, and are below the 5 Mbps mark. While the algorithm does not converge, we did show that the average reward does go into the positives for multiple aggregated iterations as time goes on. As such, the algorithm still manages to meet certain demands.

In terms of the delay, the box plots of Figure 6b show a similar trend. The delay requirements for the silver flows are violated in more than half the instances. Out of six total constraints, distributed MADQN violates three.

Nonetheless, the centralized version of MADQN delivers the required thresholds. In Figure 6c, we note that the throughput values lie around 5.5, 11.4, and at above 31 Mbps for the bronze, silver, and gold flows, respectively. All above the required mark. Figure 6d shows that median delay values for the centralized MADQN flows at 0.128, 0.209, and 0.298 seconds meet all requirements. This however is not the case for PQ.



(a)

(b)

(c)

(d)

Figure 7: TCP traffic: DGN vs. Distributed MADQN (DDQN) (a) and (b), Centralized MADQN (CDQN) vs. PQ in (c) and (d). G: Gold, S: Silver, B: Bronze. SD-WAN scenario.

We see in the same figures that its bronze flows' throughput is less than 1 Mbps, while the maximum delay values, for both the silver and bronze flows, are around 2 and 3 seconds, respectively. Both are in violation of the required thresholds. Out of the six constraints, PQ meets only three.

In conclusion, the communication between agents was key to addressing the problem. The centralized MADQN approach was able to meet the demands unlike its distributed counterpart, highlighting that it is not an issue of the deep learning mechanism being used. DGN provides a solution to the problem without relying on the unrealistic centralized training and execution of the centralized MADQN approach.

C. TCP Scenario

Similarly, we now look at the results in the case of TCP traffic. Figure 7a has the throughput results for DGN. Again, DGN meets all the required demands. When the links are congested, the plots show throughput values of about 5.1, 10.6, and 31.5 Mbps for the bronze, silver, and gold flows. All above the set requirements of 5, 10, and 30 Mbps, respectively. The same cannot be said for distributed MADQN. Figure 7a shows that the throughput requirement for the gold group flows, sitting at around 28 Mbps, was violated.

We additionally observe the delay results as reported in the box plots in Figure 7b for DGN. The required delay thresholds are set at 0.1, 0.15 and 0.2 seconds for the gold, silver, and bronze flows. The DGN agents meet all these requirements. Distributed MADQN meets these demands, but in general with higher mean delay values compared with DGN. Furthermore, the centralized version of MADQN was able to meet all the required throughput and delay thresholds. Figure 7c shows throughput values at around 6.2, 10.1, and 31 Mbps for the bronze, silver, and gold flows, respectively. The same cannot be said regarding priority queuing. The silver flows throughput is about 9.7 Mbps and the bronze about 0.73 Mbps, both in violation of the requirements.

In terms of delay, Figure 7d shows that centralized MADQN group flows have maximum delays at around 0.04, 0.12, and 0.15 seconds, respectively. All within the required margins. As for PQ, it fails to meet the delay requirements for both the silver and bronze flows, with maximum values recorded at 0.23 and 0.35 seconds, respectively.

The results with TCP traffic validate the conclusions of their UDP counterparts. The lack of agent communications in the decentralized MADQN approach caused the algorithm to be inefficient. The results also show that our proposals are much more equipped to deal with the problem than priority queuing.

D. On Agent size, Complexity, and Communication Overhead

When setting neural network parameters for the deep learning agents, we always seek the smallest working configuration. That is independent of the MARL agent setting, whether the system is centralized, distributed or semi-distributed. In terms of size of agents, in bytes on disk, DGN agents are considerably larger. DGN has more layers, more structure, and even its own Q-network. A trained DQN agent takes an average 0.267 MBs of space, while DGN ones consume 1.9 MBs.

For DGN, during the agents' execution phase, the agents would need to communicate their feature vectors, *i.e.*, the output of the convolutional layers. The size of the latter can be a significant concern when implementing distributed learning approaches. We assess the incurred overhead using two methods. First, in order to quantify the amount of communications involved, as discussed in [32], the overhead is defined as a function 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:

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

A ratio closer to one, would mean all the agents are talking with each other. One closer to zero, means agents are barely communicating. In the case of the SD-WAN scenario, the ratio is 0.18 for our DGN approach. This indicates a very small overhead and communications limited to where needed. In comparison, the CDQN approach would yield a ratio of 1.

Additionally, we compute the bandwidth required for such inter-agent communications. In our DGN scenario, the agents refresh their policies every 10 seconds. At that time they need to communicate messages equal to the number of convolutional layers they have. The size of each message is equal to the size of the feature vector. For our implementation, we have 2 convolutional layers. The output of each is 1x128. Assuming a 64 bit machine, we would need 4 bytes to store each of these values. That means on each link between two communicating agents, we would only need around 0.8192 kbps of reserved bandwidth for inter-agent communications.

We also note that in this work, we considered a synchronous execution of agents so that they exchange information at the time is it needed by their neighbors. An interesting development would be to consider the asynchronous setting.

E. Impact of Varying the Number of Convolutional Layers

As discussed in Section IV-A, the number of convolution layers controls the communication between agents. Indeed, we verified experimentally the significance of the number of convolutional layers in the DGN agent modules. To do so, we ran the same experiment but this time with the convolutional layer module containing only one layer. The results, showing that the DGN approach no longer converges, can be seen in Figure 8. The main reason being that agents controlling the branches are not able to efficiently collaborate via the HQ. The presence of only one convolutional layer, means that the 2-hop communications, needed for non-linked nodes (as 3 and 5) to communicate through the HQ nodes, are not available.

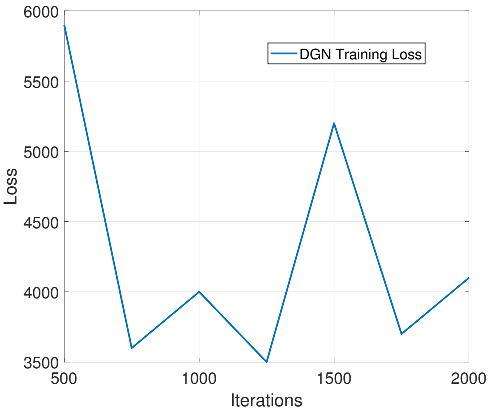


Figure 8: DGN convergence with one convolutional layer

F. Generalized Network Topology

We are additionally interested as well in testing our proposal in a more generalized network topology. For that, we choose the Abilene network topology illustrated in Figure 9. Traffic is generated from hosts connected to nodes 1, 2, and 3, and collected at a destination connected to node 7. Each of these

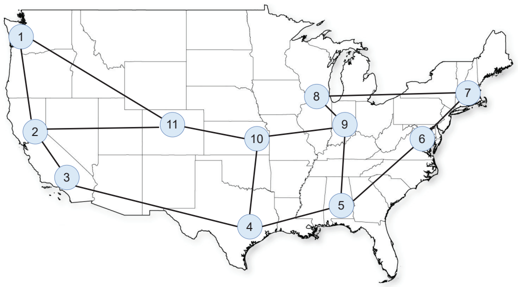


Figure 9: Abilene network topology

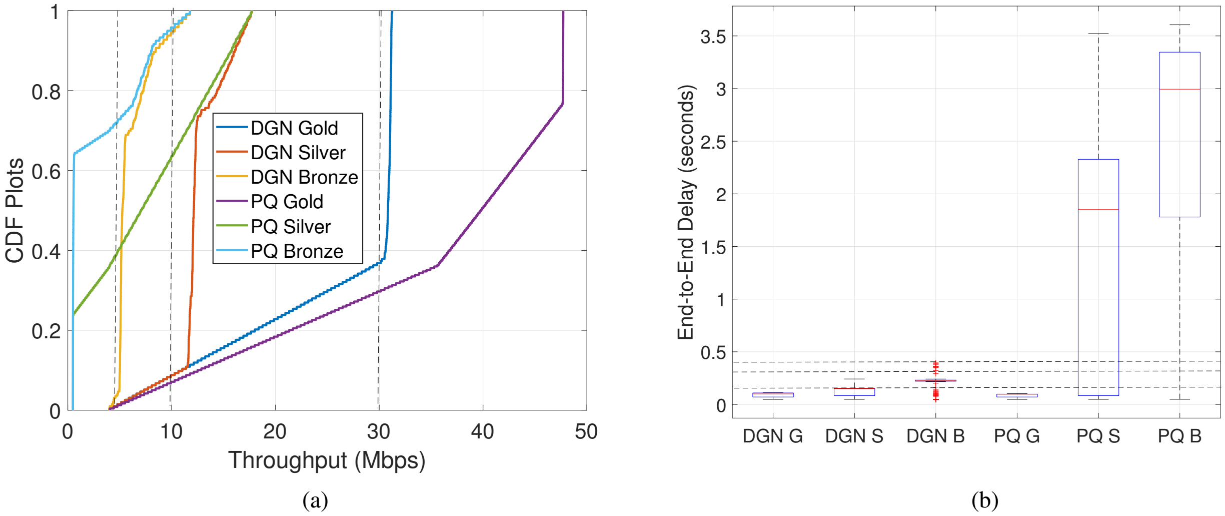


Figure 10: DGN vs. PQ in terms of throughput (a), and delay (b). G: Gold, S: Silver, B: Bronze. Abilene scenario.

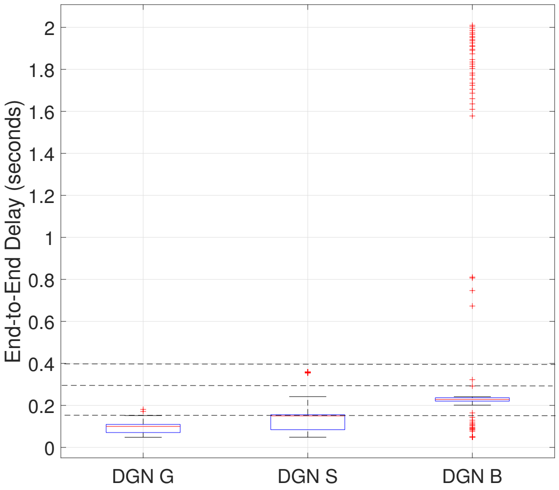


Figure 11: End-to-end delay values. One convolutional layer. Abilene scenario.

sources generates three flows, one of each type: gold, silver, and bronze. Similar to our previous scenario, we consider that the links interconnecting the main nodes shown in the figure are the ones with bandwidth constraints. Our DGN agents are placed on all the nodes (one through eleven). Two convolutional layers are considered for each agent. At each of the source nodes, we have a gold flow, a silver, and a bronze flow. The sources are of type UDP. Reminder that the latter are sinusoidal and initially the transmit rates are not enough to meet the throughput requirements. We are concerned with SLA violations only in the period where they can.

Figure 10a compares between our approach and priority queuing (PQ) in terms of throughput. The required thresholds are maintained as before. During congestion on the links, we note that our DGN approach is able to meet all the requirements for all flows with gold flows throughput being between 30.8 and 31.2 Mbps, the silver flows throughput at around 12 Mbps, and bronze flow throughput values just above 5 Mbps. For PQ, the algorithm is capable of meeting the gold flows' requirements, but records violations in both silver and bronze flows' requirements.

In Figure 10b, we compare between the two approaches in terms of the end-to-end delay. The requirements as before are set at 0.15, 0.3, and 0.4 seconds for gold, silver, and bronze, respectively. With DGN, all the flows meet their requirements with maximum values recorded at 0.112, 0.242, and 0.389 for the gold, silver, and bronze flows, respectively. The same cannot be said for PQ, which shows excessive violations for both the silver and bronze flows.

We are also interested in measuring the impact of the number of convolutional layers the DGN agents have on their performance. We reduce the number of these layers per agent from two to one, and afterwards repeat the training and the simulations under the same settings. In Figure 11, we show the resulting delay values achieved by the DGN agents with the aforementioned structure.

We notice that the approach no longer uniformly meets all the delay requirements, with several infringes recorded for gold and silver flows, and violations in more than 18 % of the cases for the bronze flows. As with before, the reduced number of convolutional layers causes the DGN to fail in extracting key relations between the different agents, that would have otherwise enabled it to succeed.

G. Scalability

Finally, we consider a scenario based on the ION-NY topology from the topology zoo dataset (see Figure 12). 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 scenarios and effects. 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. Only receiving and sending edge nodes have agents embedded in them. The topology is built in a Mininet testbed. Table IV has the simulation parameters for this scenario. Figure 13 depicts a boxplot with the average throughput results per flow. The required thresholds were set at 600, 1200, and 3000 Kbps for the gold, silver, and bronze flows, respectively. The figure shows gold flows with a median realized throughput of 3280 Kbps, the silver flows at 1316 Kbps, and the bronze at around 660 Kbps. All flow thresholds are met by our proposal.

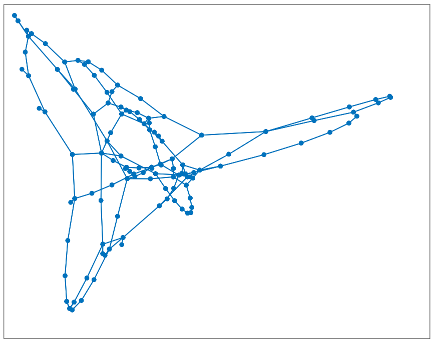


Figure 12: ION network topology

Similarly, we plot the values for the end-to-end delay achieved by the flows in Figure 14. Once again, our proposal is able to meet the set requirements, with the maximum delay values recorded being 0.15, 0.43, and 0.87 for the gold, silver, and bronze flows, respectively. In conclusion, we show that our proposal scales and functions well with larger network topologies.

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 60-80 Mbps per link
Transmit rate per source	1200-1400 packets/second
Packet size	512 Bytes

VII. Conclusion

This paper presented a multi-agent graph convolutional reinforcement learning approach built on top of a weighted fair queuing algorithm with the purpose of meeting stringent demands, in terms of throughput and delay, for a set of classified network flows. The deep learning agents continuously determine the weights with which the flow packets are dequeued. In addition, we implemented two classic multi-agent DQN solutions: one is completely centralized and the other fully distributed. We compare our approaches across different network topologies, scenarios, traffic types, and transport mechanisms, highlighting both their efficiency and the importance of inter-agent communication.

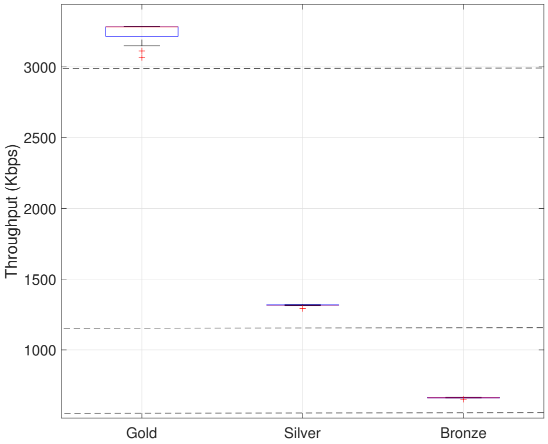


Figure 13: DGN throughput results in a large scale scenario

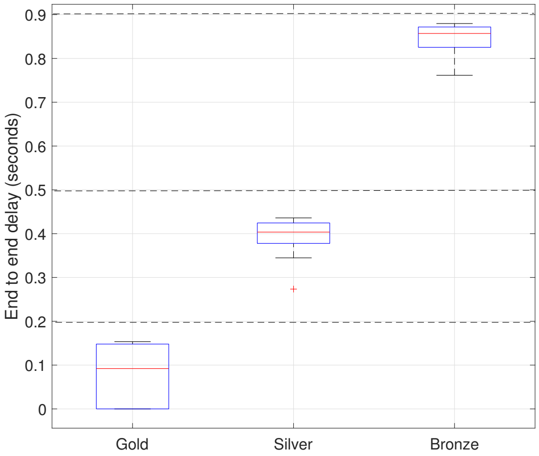


Figure 14: DGN delay results in a large scale scenario

These types of solutions are still in their infancy, but as we showed in this work, they can provide promising results. In future works, we will consider a larger scale scenario with multi-layer branches and non-direct connections to the HQ network. We will consider variable neighborhoods dependent on the links between routing nodes. Finally, we will assess our smart queuing proposals alongside a deep reinforcement learning assisted approach to load balancing in networks.

References

[1] Z. Yang, Y. Cui, B. Li, Y. Liu, and Y. Xu, “Software-defined wide area network (SD-WAN): Architecture, advances and opportunities,” in *Proc. IEEE ICCCN*, 2019.

[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] S. Hussain and A. Marshall, “An agent-based control mechanism for WFQ in IP networks,” *Control Engineering Practice*, vol. 11, no. 10, pp. 1143–1151, 2003.

[7] J. Follows and D. Straeten, *Application-Driven Networking: Class of Service in IP, Ethernet and ATM Networks*. IBM Corporation, 1999.

[8] 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.

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

[10] 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.

[11] V. Balasubramanian, M. Aloqaily, O. Tunde-Onadele, Z. Yang, and M. Reisslein, “Reinforcing cloud environments via index policy for bursty workloads,” in *Proc. IEEE NOMS*, 2020.

[12] M. Bachl, J. Fabini, and T. Zseby, “LFQ: Online Learning of Per-flow Queuing Policies using Deep Reinforcement Learning,” in *Proc. IEEE LCN*, 2020, pp. 417–420.

[13] M. M. Roselló, “Multi-path scheduling with deep reinforcement learning,” in *Proc. EuCNC*. IEEE, 2019.

[14] 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.

[15] O. Houidi, D. Zeghlache, V. Perrier, T. A. Quang Pham, N. Huin, J. Leguay, and P. Medagliani, “Constrained deep reinforcement learning for smart load balancing,” in *Proc. IEEE CCNC*, 2022.

[16] A. Kattepur and S. David, “Malta: Multi-agent reinforcement learning for differentiated services in fat tree networks,” in *2021 IEEE Conference on Network Function Virtualization and Software Defined Networks (NFV-SDN)*, 2021, pp. 129–134.

[17] Z. Yao, Z. Ding, and T. H. Clausen, “Multi-agent reinforcement learning for network load balancing in data center,” *arXiv preprint arXiv:2201.11727*, 2022.

[18] 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*, 2020.

[19] 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.

[20] J. Jiang and Z. Lu, “Learning Attentional Communication for Multi-Agent Cooperation,” in *NeurIPS*, 2018.

[21] K. Zhang, Z. Yang, H. Liu, T. Zhang, and T. Basar, “Fully decentralized multi-agent reinforcement learning with networked agents,” in *Proc. ICML*. PMLR, 2018.

[22] 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.

[23] X. Mai, Q. Fu, and Y. Chen, “Packet routing with graph attention multi-agent reinforcement learning,” in *2021 IEEE Global Communications Conference (GLOBECOM)*, 2021, pp. 1–6.

[24] 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.

[25] P. Sun, Z. Guo, G. Wang, J. Lan, and Y. Hu, “MARVEL: Enabling controller load balancing in software-defined networks with multi-agent reinforcement learning,” *Computer Networks*, vol. 177, p. 107230, 2020.

[26] M. Manel, A. El Kamel, and H. Youssef, “DQR: An Efficient Deep Q-Based Routing Approach in Multi-Controller Software Defined WAN (SD-WAN),” *Journal of Interconnection Networks*, vol. 20, p. 2150002, 12 2020.

[27] S. Troia, F. Sapienza, L. Varé, and G. Maier, “On deep reinforcement learning for traffic engineering in SD-WAN,” *IEEE Journal on Selected Areas in Communications*, vol. 39, no. 7, pp. 2198–2212, 2021.

[28] D. Stiliadis and A. Varma, “Latency-rate servers: a general model for analysis of traffic scheduling algorithms,” *IEEE/ACM Transactions on networking*, vol. 6, no. 5, pp. 611–624, 1998.

[29] H. Wei, N. Xu, H. Zhang, G. Zheng, X. Zang, C. Chen, W. Zhang, Y. Zhu, K. Xu, and Z. Li, “Colight: Learning network-level cooperation for traffic signal control,” in *Proceedings of the 28th ACM International Conference on Information and Knowledge Management*, 2019, pp. 1913–1922.

[30] J. Jiang, C. Dun, and Z. Lu, “Graph convolutional reinforcement learning for multi-agent cooperation,” *CoRR*, vol. abs/1810.09202, 2018. [Online]. Available: http://arxiv.org/abs/1810.09202

[31] K. Zhang, Z. Yang, H. Liu, T. Zhang, and T. Basar, “Fully decentralized multi-agent reinforcement learning with networked agents,” *CoRR*, vol. abs/1802.08757, 2018. [Online]. Available: http://arxiv.org/abs/1802.08757

[32] 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.

VIII. Annex

A. Training the DGN Agents

Algorithm 2: Training the DGN Agents

```
Initialize randomly the main Q network and its target
Initialize the agents and the environment at random states
while not converged do
    /* Sample phase in the replay buffer */
    for every agent $i \in \mathcal{N}$ do
        Generate a random number $e$
        if $e < \epsilon$ then
            Choose a random action $a_i$
        else
            Query the Q-network for the best action based on observation $o_i$
        Agent $i$ gets reward $r_i$ and next observation $o'_i$
    Store tuple ($\mathcal{O}$,$\mathcal{A}$,$\mathcal{O}'$,$\mathcal{R}$,$\mathcal{C}$,$done$) in replay memory $D$
    /* $done_i$ indicates if agent $i$ reached its set target or not */
    if enough experiences in D then
        /* Training phase */
        Sample a random $minibatch$ of transitions from $D$
        for every ($\mathcal{O}$,$\mathcal{A}$,$\mathcal{O}'$,$\mathcal{R}$,$\mathcal{C}$,done) do
            for every agent $i \in \mathcal{N}$ do
                if $done_i$ then
                    $y_i = r_i$
                else
                    $y_i = r_i + \gamma \max_{a'} Q(O'_{i,C}, a'_i; \theta')$.
            Calculate the Loss $\mathcal{L}(\theta) = \frac{1}{S}\sum_{S} \frac{1}{N}\sum_{i=1}^{N}(y_i - Q(O_{i,C}, a_i; \theta))^2$,
            Update Q by minimizing the loss $\mathcal{L}$
            Update the target network softly using Q's weights: $\theta' = \tau\theta + (1-\tau)\theta'$
```

In Algorithm 2 we summarize how DGN training is done. We detail the process starting from the filling of the experience replay buffer to the training of each agent, up until the update of the target network. The exploration rate ϵ determines how often the agents explore during learning, and it is kept constant during training in our approach. Convergence can be inferred from the training loss. When the experience buffer has enough samples, the training phase can begin. We train on positive rewards and the terminal state is when maximum reward is achieved. The variable $done$ indicates that the terminal state has been reached.

B. Acronyms

Table V lists the majority of this paper's acronyms.

Table V: Acronyms

Acronym	Definition
SLA	Service Level Agreement
SD-WAN	Software-Defined Wide Area Network
MPLS	Multi-protocol Label Switching
QoS	Quality of Service
WAN	Wide Area Network
AQM	Active Queue Management
WFQ	Weighted Fair Queuing
AWFQ	Adaptive WFQ
DRL	Deep Reinforcement Learning
MPTCP	Multi-Path TCP
MARL	Multi-Agent Reinforcement Learning
MA/DQN	Multi-Agent / Deep Q-Network
PQ	Priority Queuing
QoE	Quality of Experience
ECMP	Equal-Cost Multi-Path
AR	Access Routers
MLP	Multi-Layer Perceptron
CDQN /DDQN	Centralized / Distributed Multi-Agent DQN

IX. Author Biographies

Hassan Fawaz received his masters degree in Telecom Networks and Security and his Ph.D. in Wireless Communications from Saint Joseph University of Beirut in 2016 and 2019, respectively. In 2020, he was a PostDoc researcher at the University of Versailles, Paris-Saclay, with his work revolving around resource allocation in IoT LoRaWAN. He is currently a researcher engineer at Télécom SudParis, France, with his work focusing on deep learning based algorithms for network management.

Julien Lesca is a senior research engineer at Huawei Technologies Research and Development, France. Previously, he was an assistant professor at Paris Dauphine University. He works on algorithmic game theory and more precisely on algorithmic mechanism design and algorithmic cooperative game theory. During his PhD at the University Pierre et Marie Curie in Paris, France, he worked on linear programming formulation for combinatorial problems where the aggregating function is non-affine.

Pham Tran Anh Quang received the Ph.D. degree in computer sciences from University of Rennes 1, Rennes, France in 2017. He then joined b<>com and INRIA, Rennes, as a research engineer. He is currently a Senior Researcher with Huawei, Paris, France. His research interests include network functions virtualization, software-defined networking, artificial intelligence, and real-time applications.

Djamal Zeghlache (Member, IEEE) received the Ph.D. degree in electrical engineering from Southern Methodist University, Dallas, TX, USA, in 1987. He was an Assistant Professor with Cleveland State University from 1987 to 1991. In 1992, he joined Telecom SudParis of Institut Mines-Telecom, where he acts as a Professor and the Head of the Wireless Networks and Multimedia Services Department. His current research concerns architectures, protocols, and interfaces for future networks addressing cloud, SDN, and NFV optimization, control, and management reinforced by data driven approaches and in network intelligence.

Paolo Medagliani received the Master's degree and Ph.D. degree in information technologies from the University of Parma, Parma, Italy, in 2006 and 2010, respectively. He is currently a Principal Engineer & Project manager at Huawei Technologies Research and Development, France. His work revolves around advanced research on mathematical tools and breakthrough algorithms for 5G and SDN networks.

Jérémie Leguay is Chief Expert for routing algorithms and Director of the Datacom Dijkstra Lab at Huawei Technologies. He received a Ph.D. degree in computer science from Pierre & Marie Curie University (Paris, France). From 2007 to 2014, he conducted research and led the Networking Lab at Thales Communications & Security where he developed activities on sensor networks, mobile networks and software-defined networks for mission-critical networked systems. In 2014, he joined at Huawei Technologies as leader of the network and traffic optimization team. His current research activities cover the planning and control of IP networks using optimization and machine learning tools. He is a Senior Member of IEEE.