PRISMA: A Packet Routing Simulator for Multi-Agent Reinforcement Learning

Redha A. Alliche⋆, Tiago Da Silva Barros⋆, Ramon Aparicio-Pardo⋆, Lucile Sassatelli†
⋆ Université Côte d'Azur, Inria, CNRS, I3S, France
† Université Côte d'Azur, CNRS, I3S, Institut Universitaire de France, France
Contact: alliche@i3s.unice.fr

Abstract—**In this paper, we present PRISMA: Packet Routing Simulator for Multi-Agent Reinforcement Learning. To the best of our knowledge, this is the first tool specifically conceived to develop and test Reinforcement Learning (RL) algorithms for the Distributed Packet Routing (DPR) problem. In this problem, where a communication node selects the outgoing port to forward a packet using local information, *distance-vector* routing protocol (e.g., RIP) are traditionally applied. However, when network status changes very dynamically, is uncertain, or is partially hidden (e.g., wireless ad hoc networks or wired multi-domain networks), RL is an alternate solution to discover routing policies better fitted to these cases. Unfortunately, no RL tools have been developed to tackle the DPR problem, forcing the researchers to implement their own simplified RL simulation environments, complicating reproducibility and reducing realism. To overcome these issues, we present PRISMA, which offers to the community a standardized framework where: (i) communication process is realistically modelled (thanks to ns3); (ii) distributed nature is explicitly considered (nodes are implemented as separated threads); (iii) and, RL proposals can be easily developed (thanks to a modular code design and real-time training visualization interfaces) and fairly compared them.**

Index Terms—**ns-3; Multi-Agent; Packet Routing; Reinforcement Learning; Network Simulation; ML tool.**

I. Introduction

In the last years, Reinforcement Learning (RL) [1] using Deep Neural Networks (DNNs) [2], also called Deep Reinforcement Learning (DRL), has obtained ground-breaking results in solving highly complex tasks, such as human-like performance results at Atari video games [3] or beating AlphaGo [4] world champion. In communication networks, DRL has also been widely used in many networking technologies and problems. One of them is the Distributed Packet Routing [5]–[8]. In this problem, no complete and centralized view of network topology and traffic demands is available, which poses a challenge. This is the case in multi-hop wireless networks [9] or in multi-domain optical networks [10]. More precisely, in the DPR problem, each packet can be potentially routed differently regardless of the flow they belong to. Besides, the per-packet decisions are made locally by distributed agents placed at the routing nodes. These agents exploit local information, such as packet headers and neighboring nodes and link states.

However, in all the above-mentioned works, the proposed DRL approaches were evaluated on ad hoc discrete-time packet-level simulation environments (typically implemented in python) tailored to the assumptions and simplifications made by each study. Namely, these works do not generally implement the RL agents as separated threads or processes (although they claim to be multi-agent studies); and, they usually assume that at most one packet per router can be transmitted at each time step, which introduces an artificial synchronization into the routers' dynamics. This has two main consequences: (i) a reduced realism of the simulated communication process; and, (ii) an obstacle to fairly comparing these DRL proposals to state-of-the-art approaches or even against each other. This lack of standardized Machine Learning (ML) tools in the networking community [11], [12] represents a major issue to guarantee reproducibility. We point out that ML reproducibility issues are becoming a serious concern [13].

As a consequence, in the last years, some tools [12], [14] devoted to the application of RL to networks have been developed. These tools allow python RL agents to interact with the popular ns-3 network simulator [15]. Nevertheless, these proposals are not natively compatible with the *collaborative and distributed multi-agent* setting of the DPR problem, where agents collaborate to take decisions in a distributed manner.

In this paper, we propose an open-source RL simulation environment designed to overcome the aforementioned drawbacks in the application of DRL to the DPR problem. We bring the following main contributions:

1) a RL framework designed specifically for considering the distinctive characteristics of the DPR problem, serving as a playground where the community can easily validate their RL approaches and compare them.
2) more realistic modelling of the communication process based on: (i) the ns-3 [15] network simulator; and, (ii) a multi-threaded implementation for each agent.
3) a modular code design, which allows researchers to test their own RL algorithm, without needing to work on the implementation of the environment.
4) visual tools based on tensorboard [16] allowing to track training and test phases.

The rest of this paper is organized as follows. Section II discusses the related works. In Section III, we present the pa-

The author acknowledges the support of the French Agence Nationale de la Recherche (ANR), under grant ANR-19-CE-25-0001-01 (ARTIC project). This work was performed using HPC resources from GENCI-IDRIS (Grant 2021-AD011012577).

ISBN 978-3-903176-48-5© 2022 IFIP

per's background. Section IV presents PRISMA. The simulator usage is detailed in Section V and illustrated in Section VI. Finally, we conclude the paper in Section VII.

II. Related works

Related works can be grouped into the next two categories.

DRL for packet routing. In recent years, the first works applying DRL to the DPR problem have been published. DPR has been studied in [5]–[8], [17]. The seminal paper [17] was the first to apply a RL method, the Q-learning [18], to DPR, giving rise to the *Q-routing* paradigm. More details about this paradigm will be given in the next section. The study in [5] applies the Deep Q-Network (DQN) framework (i.e., a neural network fits the Q-value function) to the *Q-routing* algorithm, yielding to the *DQN routing*. In all these works, the node state only codes the destination of the packet to forward. More recently, works [6]–[8] extended the *DQN routing* by adding to the destination additional inputs, particularly node *buffer occupancy*. *DQRC (DQR with Communication)* [6] also considers *action history* and *future destinations*. Authors in [8] focus on the *relational features* (independent of the network graph), such as the *distance to the destination*. Finally, in [7], the meta-learning framework is applied to the *DQN routing* to learn different tasks, where a task corresponds to a given traffic matrix.

RL environments for networking. Only two tools have been developed to allow a Python agent to interact with a network simulation: first, ns3-gym [12]; and, more recently, ns3-ai [14]. Both of them are based on the well-known ns-3 network simulator, but they follow different approaches to implement a RL environment. Fist, ns3-ai is not only restricted to Gym as Python environment (as ns3-gym does). Second, it makes use of a shared memory pool as mechanism to connect the ns-3 simulator with the Python framework (agent and environment), differently from the ns3-gym sockets-based approach. The ns-3, Gym and ns3-gym tools are detailed in the next section.

III. Background

In this section, we present the elements which PRISMA is built over: the RL framework used to solve the packet routing problem, the RL toolkits, and, the ns-3 network simulator.

A. Q-routing: Reinforcement Learning for Packet Routing

The DPR problem can be formalized as Multi-Agent Partially Observable Markov Decision Process (POMDP) [19]: a Markov Decision Process where each agent can only locally observe its environment. The first RL method proposed to tackle the DPR was the *Q-routing* [17], based on the classical *Q-learning* [18], which we will use to describe the POMDP and the RL approach used to solve it. Note that other RL methods could be also adapted to the DPR problem.

Let $\mathcal{N}$ be the set of routing nodes (*agents*), where each *agent* n has its own local observation space $\mathcal{O}_n$ and its own action space $\mathcal{A}_n$. When a new packet arrives at time t at the node n, the node n selects as action a_n the *next hop node* n' to forward the packet. This decision is taken depending on the local observation of the router o_n (e.g., *the current packet destination*, *the node buffers' occupancy,* $\ldots$). As a consequence, the *agent* n receives a reward r_n: the *next-hop packet delay* (i.e. the packet delay to travel from n to n'), and the whole network makes a transition to a new state. When this packet (or another one) arrives to a node, this procedure is repeated. The action a_n is selected to minimize an estimate of the expected *end-to-end packet delay* from node n to its final destination. Under the DRL scheme, this estimate, denoted as $Q_n(o_n, a_n; \theta_n)$ (the *Q-value*), is the output of a DNN with θ_n weights. This DNN is trained to fit the target value Y_n^Q below:

$$Y_n^Q = r_n + \gamma \cdot \tau \cdot (1 - f) \tag{1}$$

where f indicates if the next hop is the packet destination, $\gamma \in [0, 1]$ is a discount factor, r is the *next-hop packet delay* and τ is *the remaining end-to-end delay* from the next hop n' to the destination computed as follows:

$$\tau = \min_{a_{n'} \in \mathcal{A}_{n'}} Q_{n'}(o_{n'}, a_{n'}; \theta_{n'}) \tag{2}$$

where $Q_{n'}(\cdot; \theta_{n'})$ is the output of next hop agent. The θ_n weights are updated by stochastic gradient descent when minimizing the so-called *Temporal Difference (TD) error*: $\big(Q(s, a; \theta_n) - Y_n^Q\big)^2$.

B. The ns-3 Network Simulator

The ns-3 simulator [15] is a stable discrete-event network simulator, largely adopted by the research and educational community and accepted as a standard. It is free and open-source under *GPLv2* and developed in C++ using object-oriented programming. The success of this tool is based on two main features: (i) a realistic modelling of the network systems: the protocol stack is properly abstracted by the simulator classes; (ii) a wide range of networking protocols (e.g., IP, TCP, UDP), communication technologies (e.g., Ethernet, Wi-Fi, LTE) and statistical models for channels, mobility, and traffic generation are supported by built-in classes. As a consequence, ns-3 becomes a natural choice to use as a network simulator for PRISMA.

C. Reinforcement Learning Tools

RL agents are usually implemented using Python packages (e.g., TensorFlow [20]) and interfaced with an environment where agents take decisions. The OpenAI™Gym [21] is one of the most popular toolkits to define RL environments in Python. Gym is used to test and compare different RL algorithms for a variety of problems, like Atari games or robotics [22]. Gym also provides a simple interface to pass the interactions between the agent and the environment, without making assumptions about the agent structure. Moreover, the Gym toolkit allows the creation of customized environments, which is very convenient since no network environment is natively available in Gym.

The first solution to this lack of networking RL environments was ns3-gym [12], which interfaces OpenAI ™Gym with the ns-3 network simulator. In other words, ns3-gym "transforms" a ns-3 network simulation into an RL environment by serving as gateway connecting ns-3 with Gym. Then, it provides an easy-to-use platform for developing and testing RL algorithms for networking problems. The Python process (the agent and the Gym environment) communicates with the C++ process (the ns-3 network simulation) via ZMQ sockets [23]. ns3-gym is also open source under a GPL licence and can be easily extended. In PRISMA, we opt for this tool (instead ns3-ai) since ns3-gym is better documented.

IV. PRISMA DESCRIPTION

In this section, we present the technical details of PRISMA. First, a description of the design of the framework is given. Then, each feature is explained, ranging from the network definition to simulation visualization using tensorboard [16]. The PRISMA source code is publicly available at [24].

A. Framework design principles

We consider the following principles:

1) *Training-Forwarding separation*: the agent training does not disturb the packet forwarding.
2) *Modularity*: code easy to reuse and modify.
3) *Realistic simulation*: implement the agent as close as possible to the real world.
4) *Fast prototyping*: easy and rapid modifications of the decision model.
5) *Online simulation tracking*: visualize in real-time the simulation evolution.

For point 1), we choose a multi-threading approach: each node will be run on two separate threads. One is dedicated to the training process, and; the other one to the decision process. Hence, the node agents can be trained at a timestamp without disturbing the action computation at each packet arrival.

For point 2), we built node agents in an object-oriented manner, with separated functions for each task, so that a user can easily modify the behaviour of the agent without affecting the rest of the code. We document each part of the code and provide scripts for running the simulation.

Point 3) is a very important aspect of the PRISMA design, since we need reliable results as close as possible to the deployment of the agents in the real world. For that, we keep the link propagation delay and use ns-3 to simulate the network. We have also integrated the action handling to the environment and added a Poisson random traffic generator.

For point 4), PRISMA is separated into independent modules, so that a user may easily modify the code. For example, changing the neural network or the inputs of the model will not affect the other parts of the framework.

Finally, for point 5), we give to the user the ability to monitor the simulation progress in real-time. For example, the user can track some network or ML metric (e.g., average delay, buffers occupancy, error progression) to identify training issues in real-time. To do so, tensorboard has been integrated into the framework offering network monitoring. In addition, it adds the possibility to customize plots.

B. Feature description

Figure 1 depicts the PRISMA code structure, showing in *green* our contribution over the existing ns-3 and ns3-gym modules. We describe next these *green* modules.

Main code. The core of the tool enables to:

- Gather the user arguments from the Arguments parser and manage the simulation parameters.
- Instantiate the agents from the Agent class and create the threads for each node.
- Run ns-3 simulator and tensorboard server in separated processes.
- Log the simulation statistics using the Stats writer.

Agent class. This class represents a node agent, containing two main methods: run_forwarder and run_trainer. The first one interacts with the ns3-gym environment as follows: (i) retrieve the observations, (ii) take action, (iii) compute the reward, (iv) store the states' transitions in the experience replay buffers, and, (v) update the information about the environment. The second method trains the model performing a gradient descent step. The run_forwarder also tracks the packets using their IDs to build the target value Y_n^Q at the node n, since the reward r, the next hop agent observation o'_n and the next hop agent *Q-value* $Q_{n'}(\cdot;\theta_{n'})$ are associated to the packet ID.

Multi-threaded agent instances. These are Agent class instances. Each one corresponds to a network node. They share different information among them and with the Main code using static variables. As aforementioned, we decide to allocate two threads for each instance: one calling run_forwarder method and the other one calling run_trainer method to guarantee that training and forwarding can take place separately.

Stats writer. This module is called by the Main code and Agent class instances to log information about the environment or the training progress. The following variables are tracked to be visualized using tensorboard: average packet delay, loss ratio, Temporal Difference error at each agent, replay buffers' occupancy at each node, exploration ratio for each node, the number of new arriving packets, the number of delivered packets, and, the number of buffered packets. We also add the ability to define custom plots and to compare between execution's hyperparameters.

Argument parser. This module is used to retrieve and parse the arguments given by the user in the script call.

Poisson traffic generator. This is a ns-3 application class that generates random packets following a *Poisson* process at each node. The average rate is retrieved from a given traffic matrix and the packet length is fixed to a given value. The application is not installed in the node itself, but in a virtual one directly connected to the real node.

Action and observation handler. As indicated in Section III-A, the action a_n and the observation o_n are defined as the output network interface index and the packet destination, respectively. For the local observation o_n, other data such as the buffer occupancy of the outgoing interfaces

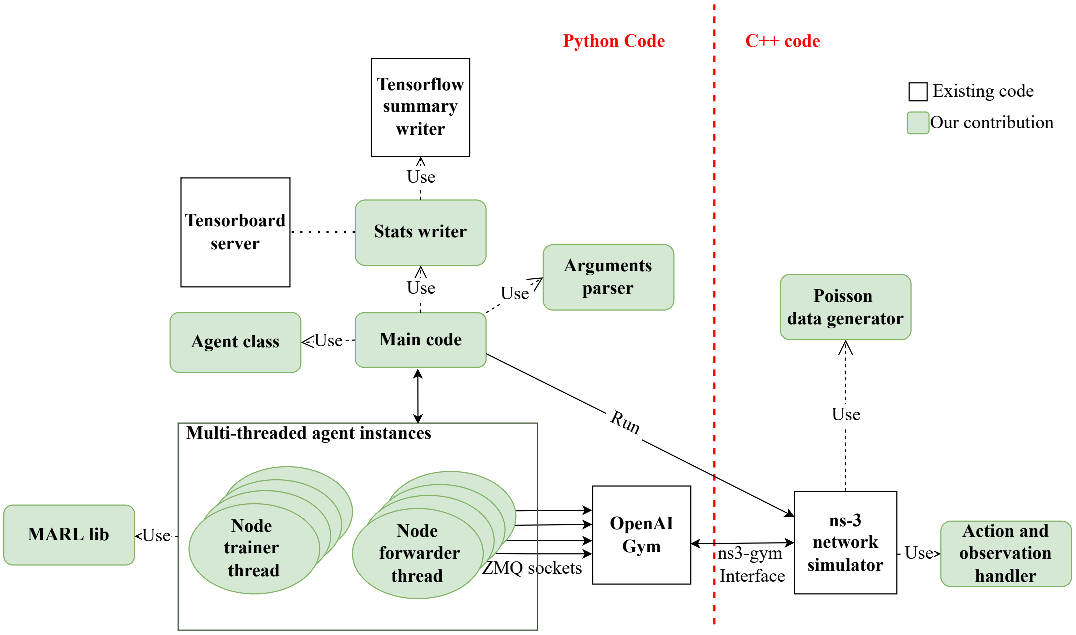


Fig. 1: PRISMA code structure, separated in two parts : Python code and C++ code, highlighting our contribution in green. The figure shows the multi-threaded approach, where each node agent is composed of two threads: one forwarding the packets and another training the model.

could be considered. This module, implemented as a ns-3 handler, retrieves o_n from the network simulation when a new packet arrives at the node n, and forwards the packet to the output interface a_n. If the buffer interface is full, then the packet is discarded. We also provide the packet ID in the info field, so we can track each packet.

MARL lib. With this module, we extend the OpenAI™ Baselines library [25] to the Multi-Agent Deep Reinforcement Learning (MA-DRL) approach. The MARL lib provides the tools for defining, training, and testing an agent. It contains the following modules:

- models: a module containing the DNN models.
- Replay buffer: a class handling the experience replay buffer. It contains methods like: add, sample or save.
- agent: a class defining an DQN agent in the context of DPR. It contains methods like step, train and sync neighbor target neural network
- utils: a set of utility functions like save_model and load_model.

V. Usage

In this section, we describe the parameters of PRISMA, and we present the guidelines about how to use the framework.

A. Framework parameters

PRISMA parameters are separated into the following:

- *Global simulation arguments*: it contains the simulation time, the base port, the seed, and whether to run training or not.
- *Network parameters*: it contains network attributes like the path to the adjacency matrix, the maximum output buffer size, or the load factor.
- *DRL agent argument*: it concerns the training of the agent and contains parameters like the batch size, the learning rate, or the training time step.
- *Session logging arguments*: it contains parameters about the session, like the name of the session and the path to store the result of the simulation.
- *Other parameters*: some misc arguments like whether to run a tensorboard server and its port number.

B. Usage guide

First, a user must install the ns3-gym, operational system and Python dependencies. To do that, we provide installation files.

The main prisma folder is composed of four folders: source, ns-3, examples, and scripts. The source folder contains the MARL lib, the agent class and utils modules. The ns3 folder contains the ns-3 files: (i) the Poisson data generator, and; (ii) the ns-3 simulation itself (sim.cc). The latter file creates and runs the simulation scenario defined by the examples folder files, that define the network topologies and traffic matrices. Scripts for launching the simulations of Section VI are provided in the scripts folder.

The DNN model can be changed in models.py. The reward, observation, and action definitions can be modified in the methods _get_reward (in agent_class), Get_Observation (in packet-routing-gym.cc) and Get_action (in packet-routing-gym.cc), respectively. Finally, a user may test the framework by calling main.py with the corresponding arguments. Calling the latter file with ”–help” argument will show all the possible parameters.

VI. Illustrative Example

This section aims to show how PRISMA can be used to assist the training and testing of a DRL algorithm to solve the Distributed Packet Routing problem. Namely, we make a Deep Q-Network routing model [5] learn the *Shortest Path (SP) routing* in backbone network scenarios. The tool can be applied to more challenging ad-hoc wireless networks by simply modifying the sim.cc file and properly adapting the DQN routing model. Here, we use a more simple case for illustrative purposes.

A. Simulation settings

We ran PRISMA in two different machines: (i) a Dell Precision 7920 workstation equipped with an Intel Xeon Gold 6230R Dual CPU (26 Cores, 2.1-4.0GHz Turbo, 128 GB RAM) with 2 NVIDIA RTX A5000 GPUs; and, (ii) an Intel Inspiron 14 laptop equipped with an Intel Core i7-8565U CPU (Quad Core, 1.80 GHz, 8 GB RAM) with a 1 GPU NVIDIA GeForce MX130 GPU.

The DRL agent is implemented in TensorFlow [20] as a three-layer neural network in models.py with a first dense layer of 32 nodes, then two layers of 128 nodes each having a Leaky ReLU activation function. This model considers only the packet destination as input and outputs the estimated end-to-end packet delay based on the selected network interface. The action is retrieved by applying an argmin function to the output layer. In the examples folder, we define the two topologies considered in this experiment: *Abilene* (11 nodes) and *Geant* (23 nodes), as well as the traffic matrices (randomly generated using a uniform distribution). We fix the packet size to 512 KB, the link propagation delay to 2 ms, and the maximum output buffer length to 30 packets. We chose the reward r_n (the *next hop packet delay*) to be 1, which means that the agent policy will aim to minimize the total number of hops from source to destination These parameters along with the model hyperparameters are set as arguments when calling main.py. We expect that *DQN-routing* will have a similar performance to *Shortest Path routing*, since both algorithms minimize the number of hops taken by a packet to reach its destination.

B. Model training

In this section, we show how the training process is performed successfully. The model is trained during 1 $minute$ at each 7 ms (network simulation time) with a learning rate of 0.001, batch size of 512, load factor of 50% and γ of 1. The duration of the simulation time for training was 30 seconds. Moreover, we use an ϵ-greedy approach (ϵ decays from 1 to 0.01) to move from exploration to exploitation.

Thanks to tensorboard visualization tools (see Figure 2), we can watch the training progress. Figures 3 show in detail the most relevant metrics to watch the learning progress for *Abilene* and *Geant*. Figure 3a and 3b show the average cost over the simulation time. The cost is computed by dividing the sum of the rewards over time by the total number of packets. Since the reward is 1 at each hop, the cost represents the average hop count per packet. We can observe an early cost increase due to the initial exploration: the network is flooded with packets needing many hops to arrive at their destinations, since the forwarding decisions are mainly random. Afterward, we move progressively to the exploitation phase, where the decision comes from the Deep Neural Network (DNN) model. As we see, the packets reduce the hop count to reach their destination, since model decisions improve. Figs 3c and 3dshow the Temporal Difference (TD) error for each node over the simulation time. We can observe that this error is decreasing, which means that all the agents are converging (learning) to a feasible routing policy (packets are not forwarded indefinitely).

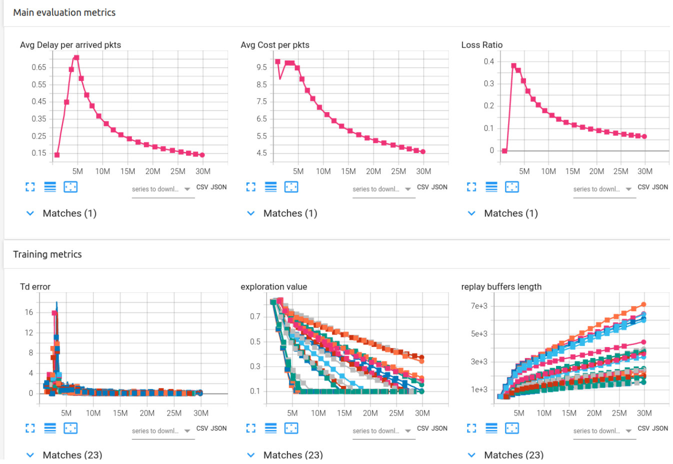
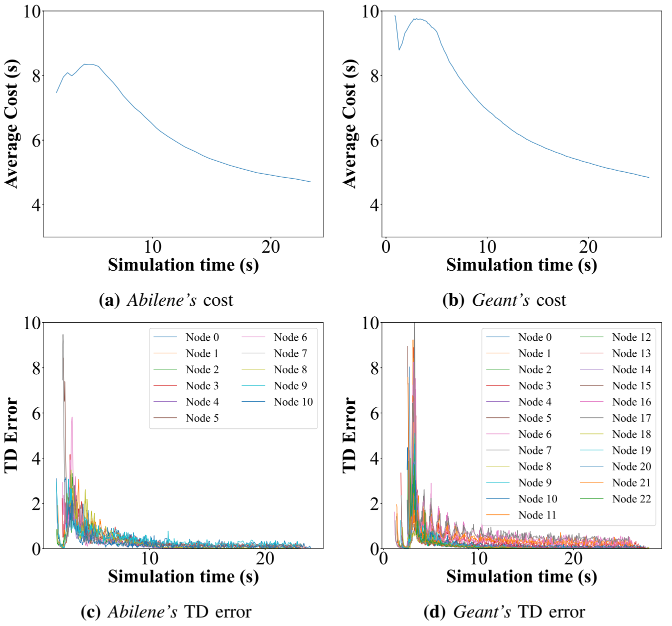


Fig. 2: Screenshot of tensorboard interface. The x axis represents the time in ns, and the y axis represents the variables tracked during the simulation. For the training metrics (bottom subfigures), each curve corresponds to an agent.



(a) *Abilene's* cost

(b) *Geant's* cost

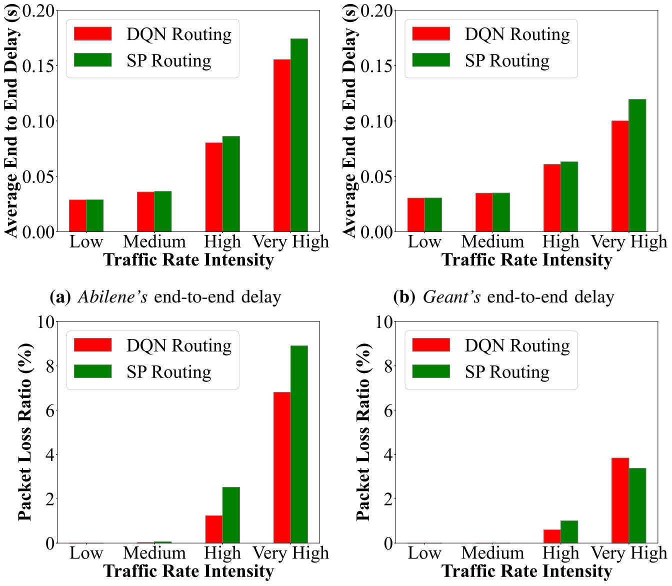
(c) *Abilene's* TD error

(d) *Geant's* TD error

Fig. 3: The average cost per packet and TD errors at each node during training for Abilene (rightmost subfigures) and Geant (leftmost subfigures) topology.

C. Model testing

In this part, we evaluate the performances of the trained *DQN-routing* agents at different traffic intensities (low, medium, high, and very high traffic, in which the load factor was 50%, 100%, 150% and 200%, respectively). In Figure 4, we compare the results with SP routing in terms of the *average end-to-end delay* and the *packet loss ratio*. For low and medium traffic rates, *DQN-routing* presents the same performance as *SP* since both methods are using the shortest path decision policy, which is sufficient to handle all the packets at this rate. For high traffic rate, *DQN-routing* outperforms *SP routing* in terms of the *end-to-end delay* for both topologies; and, in terms of *loss ratio*, for Abilene. On the contrary, *SP routing* is slightly better in *loss ratio* in Geant. Anyway, these differences are not significant since we intend to learn a routing close to the SP, which is the case.



(a) *Abilene's* end-to-end delay

(b) *Geant's* end-to-end delay

(c) *Abilene's* packet loss ratio

(d) *Geant's* packet loss ratio

Fig. 4: The average end-to-end delay per arrived packet and packet loss ratio for different traffic rate intensities for Abilene (right row) and Geant (left row) topology

VII. Conclusion

In this paper, we have presented the PRISMA tool, which is, to the best of our knowledge, the first Deep Reinforcement Learning (DRL) framework specifically devoted to the Distributed Packet Routing (DPR) problem. In this problem, routing agents take packet forwarding decisions based only on local information. When the network state is uncertain or partially hidden (e.g., a global view is not available) or changes very dynamically, DRL appears as a solution to discover an efficient routing policy. In this vein, PRISMA aims to provide a playground for researchers interested in the application of this machine learning paradigm to this challenging problem, allowing fast prototyping and bench-marking. We illustrated its main functionalities by applying the tool to learn in a distributed manner a Shortest Path routing policy in two backbone networks.

References

[1] R. S. Sutton *et al.*, *Reinforcement learning: An introduction*. MIT press, 2018.

[2] Y. Bengio, *Learning deep architectures for AI*. Now Publishers, 2009.

[3] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015.

[4] D. Silver *et al.*, "Mastering the game of Go without human knowledge," *Nature*, vol. 550, no. 7676, pp. 354–359, Oct. 2017.

[5] D. Mukhutdinov *et al.*, "Multi-agent deep learning for simultaneous optimization for time and energy in distributed routing system," *Future Generation Computer Systems*, vol. 94, pp. 587–600, 2019.

[6] X. You *et al.*, "Toward packet routing with fully distributed multiagent deep reinforcement learning," *IEEE Transactions on Systems, Man, and Cybernetics: Systems*, pp. 1–14, 2020.

[7] L. Chen *et al.*, "Multiagent meta-reinforcement learning for adaptive multipath routing optimization," *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–13, 2021.

[8] V. Manfredi *et al.*, "Relational deep reinforcement learning for routing in wireless networks," in *2021 IEEE 22nd International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM)*, 2021-06, pp. 159–168.

[9] L. Tassiulas *et al.*, "Stability properties of constrained queueing systems and scheduling policies for maximum throughput in multihop radio networks," *IEEE transactions on automatic control*, vol. 37, no. 12, pp. 1936–1948, 1992.

[10] X. Chen *et al.*, "Demonstration of distributed collaborative learning with end-to-end qot estimation in multi-domain elastic optical networks," *Optics express*, vol. 27, no. 24, pp. 35 700–35 709, 2019.

[11] A. Mestres *et al.*, "Knowledge-defined networking," *SIGCOMM Comput. Commun. Rev.*, vol. 47, no. 3, p. 2–10, sep 2017.

[12] P. Gawłowicz *et al.*, "ns-3 meets OpenAI Gym: The Playground for Machine Learning in Networking Research," in *Proc. ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM)*, November 2019.

[13] J. Pineau *et al.*, "Improving reproducibility in machine learning research(a report from the NeurIPS 2019 reproducibility program)," *Journal of Machine Learning Research*, vol. 22, no. 164, pp. 1–20, 2021.

[14] H. Yin *et al.*, "Ns3-ai: Fostering artificial intelligence algorithms for networking research," in *Proc. ACM 2020 Workshop on Ns-3 (WNS3)*, New York, NY, USA, 2020, p. 57–64.

[15] nsnam, "Ns-3 documentation website," 11/05/22. [Online]. Available: https://www.nsnam.org/documentation/

[16] "Tensorboard: Tensorflow's visualization toolkit," 11/05/22. [Online]. Available: https://www.tensorflow.org/tensorboard

[17] J. A. Boyan *et al.*, "Packet routing in dynamically changing networks: A reinforcement learning approach," in *Proc. Neural Information Processing Systems (NeurIPS)*, 1993, pp. 671–678.

[18] C. J. C. H. Watkins, "Learning from delayed rewards." Ph.D. dissertation, University of Cambridge, 1989.

[19] M. L. Littman, "Markov games as a framework for multi-agent reinforcement learning," in *Proc. Intl. Conf. on Machine Learning (ICML)*, 1994, pp. 157–163.

[20] "Tensorflow: An end-to-end open source machine learning platform," 11/05/22. [Online]. Available: https://www.tensorflow.org/

[21] "Gym: Gym toolkit for creating reinforcement learning environments," 11/05/22. [Online]. Available: https://gym.openai.com

[22] I. Zamora *et al.*, "Extending the openai gym for robotics: a toolkit for reinforcement learning using ros and gazebo," *arXiv preprint arXiv:1608.05742*, 2016.

[23] "Zero mq: An open-source universal messaging library," 11/05/22. [Online]. Available: https://zeromq.org/

[24] "Prisma tool: An open marl framework for packet routing." 11/05/22. [Online]. Available: https://github.com/raparicio pardo/PRISMA

[25] "Openai baselines: high-quality implementations of reinforcement learning algorithms," 11/05/22. [Online]. Available: https://github.com/openai/baselines