ns-3 meets OpenAI Gym: The Playground for Machine Learning in Networking Research

Piotr Gawłowicz
Technische Universität Berlin
Germany
gawlowicz@tu-berlin.de

Anatolij Zubow
Technische Universität Berlin
Germany
zubow@tu-berlin.de

ABSTRACT

Recently, we have seen a boom attempts to improve the operation of networking protocols using machine learning (ML) techniques. The proposed reinforcement learning (RL) based control solutions very often overtake traditionally designed ones in terms of performance and efficiency. However, in order to reach such a superb level, an RL control agent requires a lot of interactions with an environment to learn the best policies. Similarly, the recent advancements in image recognition area were enabled by the rise of large labeled datasets (e.g. ImageNet [8]). This paper presents the ns3-gym — the first framework for RL research in networking. It is based on OpenAI Gym, a toolkit for RL research and ns-3 network simulator. Specifically, it allows representing an ns-3 simulation as an environment in Gym framework and exposing state and control knobs of entities from the simulation for the agent's learning purposes. Our framework is generic and can be used in various networking problems. Here, we present an illustrative example from the cognitive radio area, where a wireless node learns the channel access pattern of a periodic interferer in order to avoid collisions with it. The toolkit is provided to the community as open source under a GPL license.

CCS CONCEPTS

• **Networks → Network simulations**; • **Computing methodologies → Reinforcement learning**; **Simulation tools**.

KEYWORDS

reinforcement learning, networking research, OpenAI Gym, network simulator, ns-3

ACM Reference Format:
Piotr Gawłowicz and Anatolij Zubow. 2019. ns-3 meets OpenAI Gym: The Playground for Machine Learning in Networking Research. In *MSWiM '19: ACM International Conference on Modeling, Analysis and Simulation of Wireless and Mobile Systems, Nov 25–29, 2019, Miami Beach, USA.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/1122445.1122456

1 INTRODUCTION

Modern communication networks have evolved into extremely complex and dynamic systems. Although a network makes use of rather simple to understand protocols, their composition makes network's behavior non-trivial with very often hidden (i.e. not directly explainable) dependencies between components' parameters and network performance metrics. For this reason, traditional approaches for the design of new solutions or the optimization of existing ones provide only limited gains as they are based on (over-)simplified models created according to people's understanding. Moreover, the approaches are mostly focused on a single component (e.g. protocol layer) neglecting the end-to-end network's nature.

Furthermore, today's networks generate a large amount of monitoring data, that can help to improve the design and management of them. This, however, requires the processing of the raw data in order to find hidden dependencies. All this together makes machine learning (ML) techniques the perfect fit for modern networking [25]. ML can provide estimated models with tunable accuracy, that will help researchers to tackle intractable old problems, as well as encourage new applications in the networking domain potentially leading to breakthroughs [28]. We have already seen a boom in the usage of machine learning in general and reinforcement learning (RL) in particular for the optimization of communication and networking systems ranging from scheduling [2, 6], resource management [18], congestion control [14, 15, 29], routing [1] and adaptive video streaming [19]. Each proposed approach shows significant improvements compared to traditionally designed algorithms.

However, we believe that RL in networking (RLN) research is slowed down by the following factors:

- *The existence of a knowledge gap* — networking researchers lack ML related knowledge and experience while ML researchers lack knowledge in networking [28].
- *Lack of training environments* — RL requires a large number of interactions with an environment to properly train an agent. The best way is to use a real-world environment. However, it is time-consuming and researchers usually lack skills and/or hardware to setup a testbed, while an exploration (required in RL to learn) in real network deployments can be unsafe for their operation.
- *The need for reliable benchmarking* [4] — currently, researchers build and use their environments on a case-by-case basis. Some of them use network simulators, others real-world hardware, i.e. testbed. This issue makes the direct comparison of the performance of published algorithms difficult while tracking the RLN progress almost impossible.

Contribution: In this paper, we propose ns3-gym — a first attempt to fix all the above problems. It is a benchmarking system for networking based on two well-known and acknowledged by research community frameworks, namely, ns-3 and OpenAI Gym. It combines the advantages of these two, namely, the verified and

MSWiM '19, Nov 25–29, 2019, Miami Beach, USA

ACM ISBN 978-1-4503-9999-9/18/06…$15.00
https://doi.org/10.1145/1122445.1122456

tested models of ns-3 and the simplicity of prototyping RL-based algorithms in Python using numerical computation libraries (e.g. Tensorflow[1] and Keras[2]). Specifically, ns3-gym simplifies feeding the RL models with the data generated in the simulator.

The framework is generic and it can be easily extended and used in a wide range of networking problems. We provide the first set of problems along with baseline solutions that can be used by the community to directly compare the performance of different RL-based algorithms (i.e. agents) using the same virtual conditions of well-defined simulation scripts (i.e. environments). We believe that our work will help to motivate researchers from both networking and ML areas to collaborate in order to develop and share innovative algorithms and challenging environments, and hence speed up research and development in RLN area.

2 BACKGROUND

In this section, we provide an overview of reinforcement learning technique together with tools that simplify the development and training of RL models. Then, we briefly introduce the ns-3 simulator.

2.1 Reinforcement Learning

Reinforcement learning is being successfully used in robotics for years as it allows the design of sophisticated and hard to engineer behaviors [13]. The main advantage of RL is its ability to learn to interact with the surrounding environment based on its own experience. An RL agent interacts with an environment in the following way: i) it observes the current state of the environment, ii) based on the observation it selects an action and executes it in the environment, iii) which in turn returns a reward associated with this specific action in the particular state – Fig. 1. The way an environment transforms the agent's action taken in the current state into the next state and a reward is unknown. Hence, the agent's main goal is to approximate the environment's function and learn the best policy allowing it to select always the best action and maximize its cumulated reward.

2.2 RL Tools

OpenAI Gym [4] is a toolkit for developing and comparing reinforcement learning algorithms. It provides a simple API that unifies interactions between an RL-based agent and an environment. Specifically, any environment can be integrated into the Gym as long as all the observations, actions, and rewards can be represented as numerical values. Note, that Gym makes no assumptions about the structure of an agent (just provides input data and action knobs) and it is compatible with any numerical computation library. Based on the unified interface Gym provides access to a collection of standardized environments. The framework was already integrated with the variety of environments in areas ranging from video games (e.g. Ping-Pong) to robotics [4, 23, 24].

As mentioned in the previous section (§2.1), an RL agent attempts to approximate the environment's function. Usually, such functions are very complex and cannot be represented in closed-form. Fortunately, neural networks cope well in such cases. They use coefficients to approximate the function transforming inputs

[1]https://www.tensorflow.org/
[2]https://keras.io/

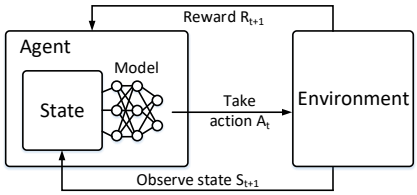

Figure 1: Reinforcement Learning.

to outputs and during the learning process, they try to find the coefficients' values yielding the best approximation. The learning is an iterative process during which a solver follows gradients that lead to smaller errors. In recent years, the task of developing and solving the neural networks was simplified by the emergence of numerical computation libraries. For instance, Keras provides a high-level API allowing to create and optimize even a very complex neural network in just a few lines of code. Keras runs on top of TensorFlow that allows representing numerical computation as a data flow graph, i.e. nodes represent mathematical operations, while edges represent the multidimensional data arrays that flow between them. Finally, with only minor changes, TensorFlow allows performing the computations on a single CPU and GPU as well as distributed clusters of them using the same code.

The replication of the published RL algorithms is a challenging task, especially for people entering the field of machine learning, as usually they are very complex and even a small difference (e.g. bug) in the implementation may affect their performance. Releasing a code repository along with the published paper is a good practice, however, still not the case for most of the publications. Fortunately, recently the high-quality reference implementations of RL algorithms become available [9, 10, 16]. Researchers can use them as a base and apply to the problems in their respective areas.

2.3 ns-3 Network Simulator

ns-3 is a discrete-event network simulator for networking systems, targeted primarily for research and educational use. It is an open-source project developed in C++ using object-oriented programming model [21, 22]. It became a standard in networking research as the results obtained are accepted by the community.

ns-3 tries to reflect the reality as close as possible, therefore it uses several core concepts and abstractions that well map to how computers and networks are built, i.e. a `Node` is a fundamental entity connected to the network. It is a container for `Applications`, `Protocols` and `Network Devices`. An application is a user program that generates packet flows. A protocol represents a logic of network and transport level protocols, e.g. TCP, OSRL routing. A `Network Device` is an entity connected to a `Channel` that is the basic communication sub-network abstraction. As in real-world, in order to build a network system, one has to perform set of task including installing network devices in nodes and connecting them to channels, allocating proper MAC addresses, configuring the protocol stacks of all nodes, etc. ns-3 provides a set of helpers that simplifies the tedious work behind easy to use API.

The introduced abstractions came with the unified interfaces between entities, what allowed the research community to work in parallel on different parts of the protocol stack without any problems during integration, e.g. any application can send packets

over Ethernet or WiFi network devices. Based on the core concepts, the ns-3 community developed a vast set of networking protocols (e.g. IP, TCP, UDP) and communication technologies (e.g. Ethernet, WiFi, LTE, WiMAX) with detailed modeling of physical layer operations. Furthermore, ns-3 offers a variety of statistical models for wireless channels, mobility, and traffic generation. In addition, ns-3 can interact with external systems (e.g. real-time LTE testbed [11]), applications (e.g. using direct code execution technique [27]) and libraries (i.e. Click [26]).

Finally, ns-3 provides also generic *tracing* and *attribute configuration* subsystems, that signal state changes in a network model and allow monitoring the internal state and parameters of any entity (e.g. node, protocol, device) in a simulation; and control its parameters and attributes at run-time, respectively. Both subsystems serve as a basis for the ns3-gym framework.

3 MOTIVATION

The main goal of our work is to facilitate and shorten the time required for prototyping of novel RL-based networking solutions. We believe that developing control algorithms and training them with the data generated in a simulation is very often more practical (i.e. easier, faster and safer) in comparison to running experiments in the real world. Moreover, it gives an opportunity for everybody to test his/her ideas, without a need to buy and set up costly testbeds.

Furthermore, thanks to *transfer learning*, i.e. the ability to reuse previously acquired knowledge in a new (more complex) system or an environment, the agent trained in a simulation can directly interact or be retrained in the real world much faster than when starting from the scratch [7]. How well the agent copes with the real-world environment, depends on the accuracy of the simulations models that were used during training. Since ns-3 community strives to make its models reflect reality as close as possible, we believe the knowledge acquired in a simulation should remain reasonable also for the real world.

Finally, note that our framework is not constrained only to RL as one can use it to obtain observations from the simulation in order to generate data-sets and use them for the offline learning using a variety of ML algorithms (e.g. supervised learning).

4 SYSTEM ARCHITECTURE

The architecture of ns3-gym as depicted in Fig. 2 consists of the following major components, namely: ns-3 network simulator and OpenAI Gym framework. The former one is used to implement environments, while the latter one unifies their interface. The main contribution of this work is the design and implementation of a generic interface between OpenAI Gym and ns-3 that allows for seamless integration of those two frameworks. In the following, we describe our ns3-gym framework in detail.

4.1 Network Simulator

ns-3 is a core part of our framework since it is used to implement a *simulation scenario* serving as an environment for an RL agent. A simulation scenario contains a network model together with scheduled changes in simulation conditions. One can create even very complex network models and study them under various traffic and

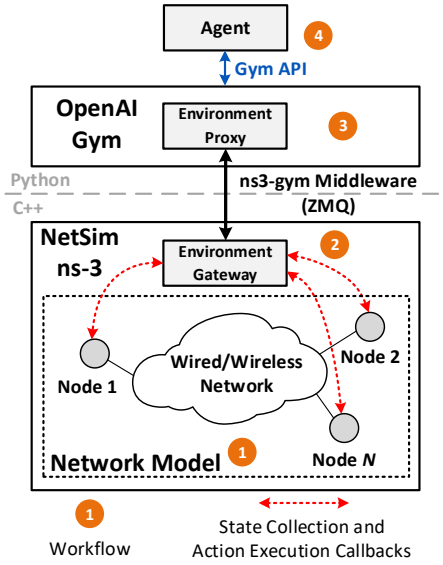

Figure 2: Architecture of ns3-gym framework.

mobility patterns by assembling the detailed models of communication components and channels provided in ns-3. An experimenter triggers changes in some conditions during the course of a simulation by scheduling proper events, e.g. start/stop traffic sources.

The state of the entire network model is a composition of states of its elements. The state representation of each entity depends on its implementation. For example, the state of a packet queue is a numerical value indicating the number of enqueued packets, while most of the protocols (e.g. TCP) are implemented as finite-state machines (FSM) jumping between predefined states. Note that in the latter case the state can be also encoded into numerical values. ns-3 provides proper interfaces allowing reading the internal state of each entity.

It is up to the designer to decide which part of the simulation state is going to be shared with the agent for the learning purpose. In most cases, it will be sub-set of the network model state together with some statistics collected during the last step execution (e.g. number of TX/RX packets in a network device and mean inter-packet arrival interval). Usually, the observed state will be limited to a state of a single instance of the protocol that the RL-based agent is going to control. The rest entities of the network model implement and evolve the complex state of the environment the agent is going to interact with. Similarly, the possible actions are limited to changes of parameters of the observed entity. In other words, the agent is able to only partially observe the network model by interacting with it though taken actions and experiencing its response in changes of the local observations.

4.2 OpenAI Gym

The main purpose of the Gym framework is to provide a standardized interface allowing to access the state and execute actions in an environment. Note that the environment is defined entirely inside the simulation scenario making the Python code environment-independent, which allows to easily to exchange the agents' implementation while keeping the reproducibility of the environment's conditions.

4.3 ns3-gym Middleware

ns3-gym middleware interconnects ns-3 network simulator and OpenAI Gym framework. Specifically, it takes care of transferring state (i.e. observations) and control (i.e. actions) between the Gym agent and the simulation environment.

The middleware consists of two parts, namely Environment Gateway and Environment Proxy. The gateway resides inside the simulator and is responsible for gathering environment state into structured numerical data and translating the received actions, again encoded as numerical values, into corresponding function calls with proper arguments. The proxy receives environment state and expose it towards an agent through the *pythonic* Gym API. Note, that ns3-gym middleware transfers the state and actions as numerical values and it is up to the researcher to define their semantics.

5 IMPLEMENTATION

`ns3-gym` is a toolkit that consists of two software components (i.e. Environment Gateway written in C++ and the Environment Proxy in Python) being add-ons to the existing ns-3 and OpenAI Gym frameworks. The toolkit simplifies the tasks of development of the networking environments and training RL-based agents by taking care of the common tasks and hiding them behind easy to use API. Specifically, `ns3-gym` provides a way for the collection and exchange of information between frameworks (including connection initialization and data (de)serialization), takes care of the management of ns-3 simulation process life-cycle as well as freezing the execution of simulation during the interaction with an agent. The communication between components is realized with ZMQ[3] sockets using the Protocol Buffers[4] library for serialization of messages.

The software package together with clarifying examples is provided to the community as open source under a GPL in our online repository: `https://github.com/tkn-tub/ns3-gym`. It is also available as a so-called ns-3 App, that can be integrated with any version of the simulator: `https://apps.nsnam.org/app/ns3-gym`

In the following subsections, we describe the `ns3-gym` components in details and explain how to use them with code examples.

5.1 Environment Gateway

In order to turn a ns-3 simulation scenario into a Gym environment, one need to *i)* instantiate `OpenGymGateway` and *ii)* implement its callbacks functions listed in Listing 1. Note, that the functions have to be registered in gateway object.

```
Ptr<OpenGymSpace> GetObservationSpace();
Ptr<OpenGymSpace> GetActionSpace();
Ptr<OpenGymDataContainer> GetObservation();
float GetReward();
bool GetGameOver();
std::string GetExtraInfo();
bool ExecuteActions(Ptr<OpenGymDataContainer> action);
```

Listing 1: ns3-gym C++ interface

The functions `GetObservationSpace` and `GetActionSpace` are used to define observation and action spaces (i.e. data structures storing observations and actions encoded as numerical values), respectively. Both spaces descriptions are created during the initialization of the environment and send to the environment proxy object in the initialization message – Fig.3, where they are used to create corresponding spaces in Python domain. The following spaces defined in the OpenAI Gym framework are supported, namely:

(1) **Discrete** — a discrete number between 0 and N.
(2) **Box** — a vector or matrix of numbers of single type with values bounded between *low* and *high* limits.
(3) **Tuple** — a tuple of simpler spaces.
(4) **Dict** — a dictionary of simpler spaces.

Listing 2 shows an example definition of the observation space as C++ function. The space is going to be used to observe queue lengths of all the nodes available in the network. The maximal queue size was set to 100 packets, hence the values are integers and bounded between 0 and 100.

```
Ptr<OpenGymSpace> GetObservationSpace() {
  uint32_t nodeNum = NodeList::GetNNodes ();
  float low = 0.0;
  float high = 100.0;
  std::vector<uint32_t> shape = {nodeNum,};
  std::string type = TypeNameGet<uint32_t> ();
  Ptr<OpenGymBoxSpace> space =
      ↪CreateObject<OpenGymBoxSpace>(low,high,shape,type);
  return space;}
```

Listing 2: An example definition of the `GetObservationSpace` function

The step in ns3-gym framework can be executed *synchronously*, i.e. scheduled in predefined time-intervals (time-based step), e.g. every 100 ms, or *asynchronously*, i.e. fired by an occurrence of specific event (event-based step), e.g. packet loss. In both cases, one has to define a proper callback that triggers the `Notify` function of the `OpenGymGateway` object.

After being notified about the end of a step – Fig.4 – the gateway collects the current state of the environment by calling the following callback functions:

(1) `GetObservation` – collect values of observed variables and/or parameters in simulation;
(2) `GetReward` – get the reward achieved during last step;
(3) `GetGameOver` – check a predefined gameover condition;
(4) `GetExtraInfo` – (optional) get an extra information associated with current environment state.

The listing 3 shows example implementation of the `GetObservation` observation function. First, the box data container is created according to the observation space definition. Then the box is filled with the current size of the queue of WiFi interface of each node.

```
Ptr<OpenGymDataContainer> GetObservation() {
  uint32_t nodeNum = NodeList::GetNNodes ();
  std::vector<uint32_t> shape = {nodeNum,};
  Ptr<OpenGymBoxContainer<uint32_t>> box =
      ↪CreateObject<OpenGymBoxContainer<uint32_t>>(shape);

  uint32_t nodeNum = NodeList::GetNNodes ();
  for (uint32_t i=0; i<nodeNum; i++) {
    Ptr<Node> node = NodeList::GetNode (i);
    Ptr<WifiMacQueue> queue = GetQueue (node);
    uint32_t value = queue->GetNPackets();
    box->AddValue(value);
  }
  return box; }
```

Listing 3: An example definition of the `GetObservation` function

[3] http://zeromq.org/
[4] https://developers.google.com/protocol-buffers/

The ns3-gym middleware delivers the collected environment's state to an agent that in return sends the action to be executed. Note, that the execution of a simulation is stopped during this interaction. Similarly to the observation, the action is also encoded as numerical values in a container. The user is responsible to implement the `ExecuteActions` callback, that maps the numerical values to proper actions, e.g. setting minimum MAC contention window size for the 802.11 WiFi interface of each node – Listing 4.

```
1  bool ExecuteActions(Ptr<OpenGymDataContainer> action) {
2    Ptr<OpenGymBoxContainer<uint32_t> > box =
         ↪DynamicCast<OpenGymBoxContainer<uint32_t> >(action);
3    std::vector<uint32_t> actionVector = box->GetData();
4
5    uint32_t nodeNum = NodeList::GetNNodes ();
6    for (uint32_t i=0; i<nodeNum; i++) {
7      Ptr<Node> node = NodeList::GetNode(i);
8      uint32_t cwSize = actionVector.at(i);
9      SetCwMin(node, cwSize);
10   }
11   return true;}
```

Listing 4: An example definition of the `ExecuteActions` function

5.2 Environment Proxy

The environment proxy is the northbound part of the middleware. It is wrapped by the `Ns3GymEnv` class that inherits from the generic Gym environment, which makes it accessible through OpenAI Gym API. Specifically, the proxy translates the Gym function calls into messages and sends them towards an environment gateway over ZMQ socket.

In the code listing 5, we present example Python script showing the usage of ns3-gym framework. First, the ns-3 environment and agent are initialized — lines 5–7. Note, that the creation of `ns3-v0` environment is achieved using the standard Gym API. Behind the scene, the ns3-gym engine starts a ns-3 simulation script located in the current working directory, establishes a ZMQ connection and waits for the environment initialization message — Fig.3. Optionally, the ns-3 environment can be adjusted by passing command line arguments during the start of the script (e.g. seed, simulation time, number of nodes, etc.). This, however, requires to use `Ns3Env(args={arg=value,...})` constructor instead of standard `Gym::make('ns3-v0')`.

```
1  import gym
2  import ns3gym
3  import MyAgent
4
5  env = gym.make('ns3-v0')
6  obs = env.reset()
7  agent = MyAgent.Agent()
8
9  while True:
10   action = agent.get_action(obs)
11   obs, reward, done, info = env.step(action)
12
13   if done:
14     break
```

Listing 5: Example Python script showing interaction between an Agent and ns-3 environment

At each step, the agent takes the observation and returns, based on the implemented logic, the next action to be executed in the environment — lines 9–11. Note, that agent class is not provided in the framework and the developers are free to define them as they want. For example, the simplest agent performs random actions.

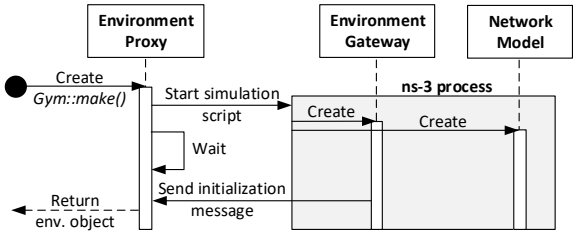

Figure 3: Implementation of the `Gym::make()` function.

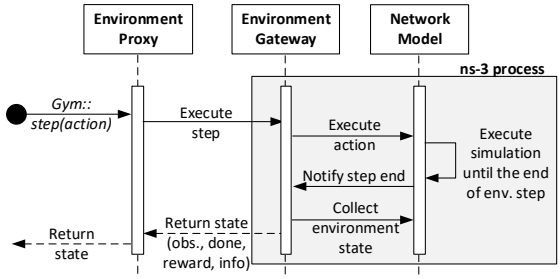

Figure 4: Implementation of the `Gym::step(action)` function.

The execution of the episode terminates (lines 13–14) when the environment returns `done=true`, that can be caused by the end of the simulation or meeting the predefined game-over condition.

In addition, a Gym environment exposes also a `Gym::reset()` function, that allows reverting the environment into the initial state. The ns3-gym implements the reset function by simply terminating the simulation process and starting a new one reusing mechanisms of the `make` function. Note, that the mapping of all the described functions between corresponding C++ and Python functions is done by the ns3-gym framework automatically hiding the entire complexity behind easy to use API.

5.3 Discussion

Although the ns-3 project supports Python bindings that would allow us to integrate it with OpenAI Gym directly in a single Python process, we have not taken this approach and decided to split ns3-gym into two communicating processes, ie. ns-3 (C++) and OpenAI Gym (Python). We believe that the split is essential due to the following reasons. First, during the learning process, an OpenAI Gym agent has to keep its state (i.e. gained knowledge) across multiple episodes (i.e. simulation runs). Having two separate processes makes this requirement easier to fulfill. Moreover, it allows running multiple ns-3 instances in parallel even in a distributed environment. Hence, the agent learning process can be executed on powerful machines with the support of GPUs, while ns-3 instances on ones equipped only with decent CPUs. This feature is especially important for techniques like A3C [20], that uses multiple agents interacting with their own copies of the environment for more efficient learning. Then, the independent, hence more diverse, experience of all agents is periodically fused to the global learning network. Second, the ns-3 project is developed in C++, while Python bindings are generated automatically as an add-on that allows only writing simulations scripts. Apparently, the development of scripts in C++ is easier for newcomers as there are much more

code examples (while only a few in Python) and documentation as well as more support from the community. Finally, having the C++ implementation allows updating existing C++ simulation scripts and examples to be used as OpenAI Gym environment.

6 ENVIRONMENTS

In the following subsections, we present a typical workflow when using the ns3-gym framework. Then, we briefly describe the environments provided as examples. Finally, we discuss the feasibility of implementation of multi-agent environments and the direct usage of agents trained in a simulated environment in a real-world experiment using emulation technique.

6.1 Typical Workflow

A typical workflow of developing and training an RL-based agent is shown as numbers in Fig. 2: (1) Create a model of the network and configure scenario conditions (i.e. traffic, mobility, etc.) using standard functions of ns-3; (2) Instantiate ns3-gym environment gateway in the simulation, i.e. create `OpenGymGateway` object and implement callbacks functions that collect a state of the environment to be shared with the agent and execute actions received from it; (3) Create the ns3-gym environment proxy, i.e. create `ns3-gym` using the standard `Gym::make('ns3-gym')` function; (4) Develop an RL-based agent using available numerical Python libraries, that interacts with the environment using the standard `Gym::step(action)` function; (5*) Train the agent.

6.2 Example Environments

In addition to the generic ns3-gym interface when one can observe any variable in a simulation, we provide also custom environments for specific use-cases, e.g. in `TcpNs3Env` where for the problem of flow & congestion control (TCP) the observation state, action and reward function are predefined using the RL mapping proposed by [15]. This simplifies dramatically the development of own RL-based TCP solutions and can be further used as a benchmarking suite allowing to compare the performance of different RL approaches in the context of TCP.

`DashNs3Env` is another predefined environment for testing adaptive video streaming solutions using our framework. Again the RL mapping for observation state, action and reward is predefined, i.e. as proposed by [19].

6.3 Multi-Agent Environments

In multi-agent environments, a number of agents must collaborate or compete to achieve a predefined goal, e.g. maximize utilization of wireless resources. They belong to the most complex branch of RL research, as traditional RL approaches fail to learn in such environments, i.e. directly applying single-agent RL algorithms and treating other agents as part of the environment is problematic as the environment is non-stationary from the view of any agent, what eventually violates Markov assumptions required for convergence and prevents learning [17].

As a communication network by nature is a multi-agent environment (e.g. wireless network), we believe that our ns3-gym framework may be a useful tool helping to advance research and knowledge in the multi-agent RLN area. Note, that implementation of multi-agent environment can be achieved using multiple (i.e. one for each agent) or a single instance of the `OpenGymGateway` in ns-3 simulation script. In the former case, the gateways communicate with corresponding agents in separate Python processes, while in the latter case all agents are being trained in a single Python process. This, however, requires to implement a switching mechanism steering observations and actions to the proper agents (e.g. based on node ID added to the observation vector).

6.4 Emulation

Since the ns-3 allows for usage of real Linux protocol stacks inside simulation [27] as well as can be run in the emulation mode for evaluating network protocols in real testbeds [5] (possibly interacting with real-world implementations), it can act as a bridge between an agent implemented in Gym and a real-world environment. Those capabilities give the researchers the possibility to train their RL agents in a simulated network (possibly very fast using parallel environments) and test them afterward in real testbed without having to change any single line of code. We believe that this intermediate step is of the great importance for the developing of ML-based network control algorithms reducing the possible discontinuities when moving from simulation to real-world.

7 CASE-STUDY EXAMPLES

In this section, we present two examples of a Cognitive Radio (CR) transmitter, that learns the pattern of a periodic interferer in order to avoid collisions with it. In the first example, the transmitter is able to sense the entire bandwidth (i.e. $M = 4$ wireless channels) whereas in the second example it can only monitor its own channel.

Specifically, we consider the problem of radio channel selection in a wireless multi-channel environment, e.g. 802.11 networks with external interference. The objective of the agent is to select for the next time slot a channel free of interference. We consider a simple illustrative example where the external interference, e.g. a microwave oven, follows a periodic pattern, i.e. sweeping over all four channels in the same order as shown in Fig. 5.

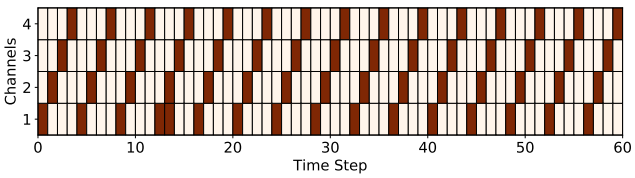

Figure 5: Channel access pattern of periodic interferer.

We created a simple simulation scenario using existing functionality from ns-3, i.e. interference created using `WaveformGenerator` class and sensing performed using `SpectrumAnalyzer` class. Our proposed RL mapping is:

- observation — occupation on each channel in the last time slot, i.e. a vector indicating whether a received signal power on each channel is below (-1) or above (+1) a predefined threshold, e.g. -82 dBm; in addition, we can combine observations from last N time-steps;
- action — set the channel to be used for the next time slot,
- reward — +1 if no collision with interferer; otherwise -1,

- gameover — if more than three collisions happened during the last ten time-slots

Our simple RL-based agent is based on deep Q-learning technique. It uses a small neural network with two fully connected layers, i.e. input and output. The $M \cdot N$ neurons in input layer use *ReLU* activation function, while the output layer uses *softmax* activation whose output $\mathbf{p} \in (0,1)^M$ is a probability vector over the four possible channels. We use Adam solver [12] during training to tune the neural network parameters.

The source code of both examples (§7.1 and §7.2) is available in the online ns3-gym repository.

7.1 CR – Wideband Sensing

Fig. 6 shows the learning performance. We see that after around 30 episodes, using only its local (but wideband) observations, the agent has perfectly learned the behavior of the periodic interferer and was able to properly select the channel for the next time-slot avoiding any collision. Although it is not shown here, we observed that combining observations of last N time-steps before passing them to the agent, speeds up the learning process, i.e. with longer observation, the agent can learn more dependencies in a single step. In Fig. 6 we show the case when the agent was feed with the data containing the observations from the last four time-steps (i.e. history size $N = 4$).

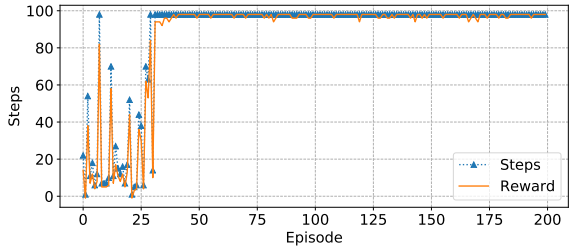

Figure 6: Learning performance of RL-based Cognitive Radio transmitter in case of wideband sensing.

7.2 CR – Narrowband Sensing

In contrast to the previous example, the CR transmitter has to learn to adapt to the interferer by performing narrowband sensing. Hence at a given point in time, the agent can only monitor the state of the channel it is operating on. The narrow-band observations of the CR transmitter are illustrated in Fig. 7. Note that in a single time-slot the TX can determine only whether the currently used channel is occupied by interfered (red, +1) or is free (blue, -1) and has no information about the other channels (white, 0). As shown in Fig. 8, we can observe that even narrowband sensing is sufficient to learn the behavior of the periodic interferer and to select an interference-free channel for the following time-slot. However, it requires a longer training process (i.e. around 100 episodes) comparing to the wideband sensing. Similar as in case of wideband sensing, we observe that longer observation history shortens the learning process. Again in Fig. 8 we show the agent's learning performance with the observation history size of $N = 4$.

8 RELATED WORK

Related work falls into three categories:

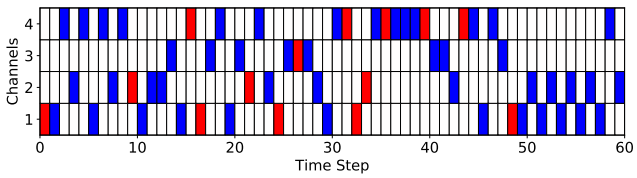

Figure 7: Narrow-band observations of cognitive radio transmitter: collision with interferer (red), no collision (blue), no information (white).

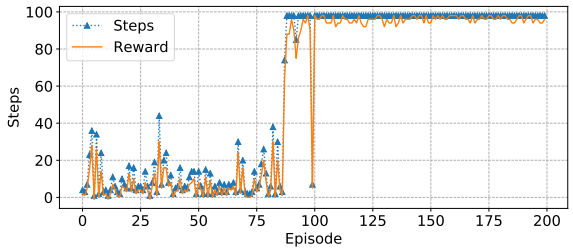

Figure 8: Learning performance of RL-based Cognitive Radio transmitter in case of narrowband sensing.

RL for networking applications: In the literature, a variety of works can be found proposing to use RL to solve networking related problems. We present two of those in more detail with emphasis on the proposed RL mapping.

Li et al. [15] proposed RL-based Transmission Control Protocol (TCP) where the objective is to learn to adjust the TCP's congestion window (CWND) to increase an utility function, which is computed based on the measurement of flow throughput and latency. The identified state space consists of EWMA of the ACK inter-arrival, EWMA of packet inter-sending time, RTT ratio, slow start threshold and current CWND is available in the provided environment. Moreover, the action space consists of increasing and decreasing the CWND respectively. Finally, the reward is specified by the value of a utility function, reflecting the desirability of the action picked.

Mao et al. proposed an RL-based adaptive video streaming [19] called Pensieve which learns the Adaptive Bitrate (ABR) algorithm automatically through experience. The observation state consists among other things of past chunk throughput, download time and current buffer size. The action space consists of the different bitrates which can be selected for the next video chunk. Finally, the reward signal is derived directly from the QoE metric, which considers the three QoE factors: bitrate, rebuffering, smoothness.

Extension of OpenAI Gym: Zamora et al. [30] provided an extension of the OpenAI Gym for robotics using the Robot Operating System (ROS) and the Gazebo simulator with a focus on creating a benchmarking system for robotics allowing direct comparison of different techniques and algorithms using the same virtual conditions. Our work aims similar goals but targets the networking community. Chinchali et al. [6] build a custom network simulator for IoT using OpenAI's Gym environment in order to study the scheduling of cellular network traffic. With ns3-gym, it would be easier to perform such an analysis as the ns-3 contains lots of MAC schedulers which can serve as the baseline for comparison.

Custom RL solutions for networkings: Winstein et al. [29] implemented a RL-based TCP congestion control algorithm on the

basis of the outdated ns-2 network simulator. Newer work on Q-learning for TCP can be found in [19]. In contrast to our work both proposed approaches are not generic as only an API meant for reading and controlling TCP parameters was presented. Moreover, custom RL libraries were used. Komondor[3] is a low-complexity wireless network simulator for the next-generation high-density WLANs including support for novel WLAN mechanisms like dynamic channel bonding (DCB) or spatial reuse. In addition, Komondor permits including intelligent ML-based agents in the wireless nodes to optimize their operation based on an implemented learning algorithm. In contrast to ns-3, Komondor is focused only on simulating WLAN operation and does not provide detailed models of other layers of the protocol stack nor different wireless technologies. This limits the potential of ML for cross-layer control/optimization or cross-technology cooperation.

9 CONCLUSIONS

In this paper, we presented the ns3-gym toolkit that simplifies the usage of reinforcement learning for solving problems in the area of networking. This is achieved by connecting the OpenAI Gym with the ns-3 simulator. As the framework is generic, it can be used by the community in a variety of networking problems.

For the future, we plan to extend the set of available environments, which can be used to benchmark different RL techniques. Moreover, we are going to provide examples showing how it can be used with more advanced RL techniques, e.g. A3C [20]. We believe that ns3-gym will foster machine learning research in the networking area and research community will grow around it. Finally, we plan to set up a website – so-called leaderboard – allowing researchers to share their results and compare the performance of algorithms for various environments using the same virtual conditions.

ACKNOWLEDGMENTS

We are grateful to Georg Hoelger for helping us with the implementation of the CR examples.

REFERENCES

[1] Dafna Shahaf Aviv Tamar Asaf Valadarsky, Michael Schapira. 2017. Learning To Route with Deep RL. In *NIPS*.
[2] R. Atallah, C. Assi, and M. Khabbaz. 2017. Deep reinforcement learning-based scheduling for roadside communication networks. In *WiOpt*.
[3] Sergio Barrachina-Muñoz, Francesc Wilhelmi, Ioannis Selinis, and Boris Bellalta. 2018. Komondor: a Wireless Network Simulator for Next-Generation High-Density WLANs. *CoRR* abs/1811.12397 (2018). arXiv:1811.12397 http://arxiv.org/abs/1811.12397
[4] Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. OpenAI Gym. *CoRR* (2016). http://arxiv.org/abs/1606.01540
[5] Gustavo Carneiro, Helder Fontes, and Manuel Ricardo. 2011. Fast prototyping of network protocols through ns-3 simulation model reuse. *Simulation modelling practice and theory, Elsevier* (2011).
[6] Sandeep Chinchali, Pan Hu, Tianshu Chu, Manu Sharma, Manu Bansal, Rakesh Misra, Marco Pavone, and Sachin Katti. 2018. Cellular Network Traffic Scheduling With Deep Reinforcement Learning. In *AAAI*.
[7] Paul F. Christiano, Zain Shah, Igor Mordatch, Jonas Schneider, Trevor Blackwell, Joshua Tobin, Pieter Abbeel, and Wojciech Zaremba. 2016. Transfer from Simulation to Real World through Learning Deep Inverse Dynamics Model. *CoRR* (2016). http://arxiv.org/abs/1610.03518
[8] J. Deng, W. Dong, R. Socher, L. Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *IEEE CVPR*.
[9] Prafulla Dhariwal, Christopher Hesse, Oleg Klimov, Alex Nichol, Matthias Plappert, Alec Radford, John Schulman, Szymon Sidor, Yuhuai Wu, and Peter Zhokhov. 2017. OpenAI Baselines. https://github.com/openai/baselines.
[10] Yan Duan, Xi Chen, Rein Houthooft, and John Schulman and@inproceedingsliang2018rllib, Author = Eric Liang and Richard Liaw and Robert Nishihara and Philipp Moritz and Roy Fox and Ken Goldberg and Joseph E. Gonzalez and Michael I. Jordan and Ion Stoica, Title = RLlib: Abstractions for Distributed Reinforcement Learning, Booktitle = International Conference on Machine Learning (ICML), Year = 2018 Pieter Abbeel. 2016. Benchmarking Deep Reinforcement Learning for Continuous Control. *CoRR* abs/1604.06778 (2016). arXiv:1604.06778 http://arxiv.org/abs/1604.06778
[11] Rohit Gupta, Bjoern Bachmann, Russell Ford, Sundeep Rangan, Nikhil Kundargi, Amal Ekbal, Karamvir Rathi, Maria Isabel Sanchez, Antonio de la Oliva, and Arianna Morelli. 2015. Ns-3-based Real-time Emulation of LTE Testbed Using LabVIEW Platform for Software Defined Networking (SDN) in CROWD Project. In *Proceedings of the 2015 Workshop on Ns-3 (WNS3 '15)*.
[12] Diederik P. Kingma and Jimmy Ba. 2015. Adam: A Method for Stochastic Optimization. In *ICLR*. http://arxiv.org/abs/1412.6980
[13] Jens Kober, J Andrew Bagnell, and Jan Peters. 2013. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research* (2013).
[14] Yiming Kong, Hui Zang, and Xiaoli Ma. 2018. Improving TCP Congestion Control with Machine Intelligence. In *ACM NetAI*.
[15] Wei Li, Fan Zhou, Kaushik Roy Chowdhury, and Waleed M Meleis. 2018. QTCP: Adaptive Congestion Control with Reinforcement Learning. *IEEE Transactions on Network Science and Engineering* (2018).
[16] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph E. Gonzalez, Michael I. Jordan, and Ion Stoica. 2018. RLlib: Abstractions for Distributed Reinforcement Learning. In *International Conference on Machine Learning (ICML)*.
[17] Ryan Lowe, Yi Wu, Aviv Tamar, Jean Harb, OpenAI Pieter Abbeel, and Igor Mordatch. 2017. Multi-agent actor-critic for mixed cooperative-competitive environments. In *Advances in Neural Information Processing Systems*.
[18] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. 2016. Resource Management with Deep Reinforcement Learning. In *ACM HotNets*.
[19] Hongzi Mao, Ravi Netravali, and Mohammad Alizadeh. 2017. Neural adaptive video streaming with pensieve. In *ACM SIGCOMM*.
[20] Volodymyr Mnih, Adrià Puigdomènech Badia, Mehdi Mirza, Alex Graves, Timothy P. Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. 2016. Asynchronous Methods for Deep Reinforcement Learning. *CoRR* (2016). http://arxiv.org/abs/1602.01783
[21] NS-3 Consortium. [n. d.]. ns-3 documentation. *https:www.nsnam.org*. Accessed: 2018-09-20.
[22] NS-3 Consortium. [n. d.]. ns-3 source code. *http:code.nsnam.org*. Accessed: 2018-09-20.
[23] OpenAI. [n. d.]. OpenAI Gym documentation. *https:gym.openai.com*. Accessed: 2018-09-20.
[24] OpenAI. [n. d.]. OpenAI Gym source code. *https:github.comopenaigym*. Accessed: 2018-09-20.
[25] Wojciech Samek, Slawomir Stanczak, and Thomas Wiegand. 2017. The Convergence of Machine Learning and Communications. *CoRR* (2017). http://arxiv.org/abs/1708.08299
[26] P. Lalith Suresh and Ruben Merz. 2011. Ns-3-click: Click Modular Router Integration for Ns-3. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques (SIMUTools '11)*. 8.
[27] Hajime Tazaki, Frédéric Uarbani, Emilio Mancini, Mathieu Lacage, Daniel Camara, Thierry Turletti, and Walid Dabbous. 2013. Direct Code Execution: Revisiting Library OS Architecture for Reproducible Network Experiments. In *ACM CoNEXT*.
[28] M. Wang, Y. Cui, X. Wang, S. Xiao, and J. Jiang. 2018. Machine Learning for Networking: Workflow, Advances and Opportunities. *IEEE Network* (2018).
[29] Keith Winstein and Hari Balakrishnan. 2013. TCP Ex Machina: Computer-generated Congestion Control. In *ACM SIGCOMM*.
[30] Iker Zamora, Nestor Gonzalez Lopez, Victor Mayoral Vilches, and Alejandro Hernandez Cordero. 2016. Extending the OpenAI Gym for robotics: a toolkit for reinforcement learning using ROS and Gazebo. *arXiv preprint arXiv:1608.05742* (2016).