GrGym: When GNU Radio goes to (AI) Gym

Anatolij Zubow, Sascha Rösler, Piotr Gawłowicz, Falko Dressler
TU Berlin, Germany
{zubow,gawlowicz,dressler}@tkn.tu-berlin.de

ABSTRACT

Trends like softwarization through the usage of flexible Software-defined Radio (SDR) platforms together with the usage of Machine Learning (ML) techniques are key enablers for building and running future high-performance communication networks in a cost-efficient way. In particular, Reinforcement Learning (RL) becomes very popular as an agent can explore the environment and adapt its behavior based on collected observations and reward values. However, for early deployments there is a pressing need for well-defined environments so that ML/RL-based algorithms can learn the best policies. In this paper, we present GrGym, a framework enabling the design of RL-driven solutions for communication networks based on the OpenAI Gym toolkit and the GNU Radio SDR platform. The GrGym framework allows integrating any GNU Radio program as an environment in the Gym framework by exposing its state and control knobs for the agent's learning purposes. Our framework is generic and can be easily extended to cover various communication problems. We present an illustrative example, where an IEEE 802.11 transmitter learns to adapt its modulation and coding rate based on the observed channel conditions.

CCS CONCEPTS

• **Networks** → **Wireless access networks**.

KEYWORDS

Communication Networks, Reinforcement Learning, Wireless, Software-defined Radio, GNU Radio, OpenAI Gym

ACM Reference Format:
Anatolij Zubow, Sascha Rösler, Piotr Gawłowicz, Falko Dressler. 2021. GrGym: When GNU Radio goes to (AI) Gym. In *The 22nd International Workshop on Mobile Computing Systems and Applications (HotMobile 2021), February 24–26, 2021, Virtual, United Kingdom.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3446382.3448358

1 INTRODUCTION

Modern wireless communication networks, e.g., WiFi or 3GPP LTE, have evolved into extremely complex and dynamic systems. However, traditional designs are based on (over-)simplified models of the environment; hence, they bring only limited performance gains. This is because they are mostly focused on a single component (e.g., protocol layer), neglecting the end-to-end and cross-layer network's nature. Hence, today we see two major trends in communication networks. First, we see the usage of machine learning (ML) techniques to improve the efficiency of such networks [9]. This is promising as networks generate a large amount of data that can be observed and analyzed by ML in order to optimize the network behavior. Here, techniques like Reinforcement Learning (RL) are becoming very popular [7, 17]. The second major trend is softwarization through the usage of flexible Software-defined Radio (SDR) architectures, which allows quick adaptation to dynamic environments and changing application requirements [1]. We strongly believe that such a flexible and adaptive ML-driven communication system is able to overtake traditionally designed ones in terms of performance and efficiency. However, in order to achieve that, ML algorithms require a well-defined environment so that they can learn the best policies.

Let us take WiFi as an example. The 802.11 standard continues to develop very quickly and a plethora of new features like HARQ, multi-link, null-steering, and queue management are expected in IEEE 802.11be amendment [10]. The following problem arises. Each corresponding PHY/MAC algorithm needs to be parameterized which is costly and error-prone due to large configuration space. Moreover, there are often interactions between PHY/MAC parameters and hence a joint optimization is needed. The goal is not just increasing the throughput, as there is a variety of KPIs ranging from efficiency to fairness to latency and to energy consumption, which need to be optimized depending on the situation.

Furthermore, we believe that the usage of ML techniques in communication networks is not yet widely used because of several reasons. First, there is the *existence of a knowledge gap*. Specifically, communication technology experts lack ML-related knowledge and experience and it would be helpful to have a sandbox-like playground to easily explore ML algorithms and their configurations. Second, there is a *lack of training environments*. RL requires a large amount of training, i.e., a high number of interactions with the environment. The best way is to use real-world network setups or at least testbeds. However, it might be sometimes too 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. Having a simulated environment would be helpful.

Contribution: In this paper, we present GrGym, a framework enabling the design of RL-driven solutions for communication networks based on the OpenAI Gym toolkit and the GNU Radio SDR platform. The GrGym framework allows integrating any GNU Radio program as an environment in the Gym framework by exposing its state and control knobs for the agent's learning purposes. Our framework is generic and can be easily extended to cover various communication problems. We present an illustrative example, where an IEEE 802.11 transmitter learns to adapt its modulation

HotMobile 2021, February 24–26, 2021, Virtual, United Kingdom

ACM ISBN 978-1-4503-8323-3/21/02…$15.00
https://doi.org/10.1145/3446382.3448358

and coding rate based on the observed channel conditions. Our toolkit is provided to the community as Open Source.[1]

2 BACKGROUND

2.1 Reinforcement Learning

Reinforcement learning (RL) is one of three basic machine learning paradigms, alongside supervised and unsupervised learning, and is being already used in robotics for years as it allows the design of sophisticated control algorithms [15]. With RL it is no longer necessary to generate large sets of labeled training data before training can start. Instead, an RL agent interacts with the environment and reinforces (or inhibits) particular patterns of behavior depending on the resulting reward (or penalty). In particular, the learning process is as follows: (1) an agent observes the current state of the environment, (2) based on the observation and its policy, the agent selects and executes an action in the environment, (3) the environment, in turn, returns a reward associated with this specific action in the particular state, and (4) the agent updates its policy based on the obtained reward. The way an environment transforms the agent's action taken in the current state into the next state and the reward is unknown. Hence, the agent's main goal is to learn a policy to select always the best action and to maximize its cumulated reward.

2.2 OpenAI Gym

OpenAI Gym [6] is a framework for benchmarking RL-based algorithms. It provides a simple and unified API that structures the interactions between an RL agent and the environment. Specifically, to integrate an environment to the Gym, its observations, actions, and rewards have to be represented as structured (e.g., vector or map) numerical data. This way, the framework was already interfaced with a large set of diverse environments in areas ranging from video games to robotics [6].

An RL agent learns its policy (i.e., the mapping between the observed states to the best action) by interacting with the environment through the unified interface. The Gym framework makes no assumptions about agent implementation. However, due to the complexity of the learning process and a large observation space, often RL agents are based on very complex neural networks (NN). During the learning process, the NN parameters are tuned. In recent years, the task of developing NNs was simplified by the emergence of powerful libraries like Keras and Tensorflow. Both can make use of ML accelerators like GPUs and FPGAs.

2.3 GNU Radio

GNU Radio [5] is an open source software toolkit, which provides a rich library of signal-processing blocks that can be glued together for building complex software-defined radios (SDR). With GNU Radio, a radio system can be built by designing a flow graph (XML or Python) where the vertices are signal processing blocks (implemented in C++) and the edges represent the data flow between them. Each signal processing block processes in real-time an infinite stream of data flowing from its input ports to its output ports. Each block is described by the number of input and output ports as well as the type of data that flows (e.g., complex numbers). There are special blocks called data sources and sinks. The first has only output ports whereas the latter only input ports. GNU Radio contains a myriad of sources and sinks allowing read/write from files or real SDR hardware like USRPs. More than one hundred blocks are currently provided. Moreover, there are partial/full implementations of radio technologies like IEEE 802.11[abgp] [3, 4], IEEE 802.15.4 [2], and 3GPP LTE [12]. It is possible to run GNU Radio programs on either real hardware (e.g., USRP SDR) or loopback in a fully simulated environment allowing application of channel propagation models to synthetically generated signals [18].

[1] https://github.com/tkn-tub/gr-gym

3 GRGYM – DESIGN PRINCIPLES

The main goal of our work is to facilitate and shorten the time required for developing novel RL-based communication networking solutions. We believe that developing RL-driven control algorithms and training them with the data generated in a simulated environment is very often more practical (i.e., easier, faster, less expensive, and safer) in comparison to directly running experiments in the testbed or real world. From the simulated environment, it should be easy to switch to testbeds and real deployments without requiring many changes. Here we believe in the power of SDR platforms like GNU Radio. Furthermore, thanks to *transfer learning*, i.e., the ability to reuse previously acquired learned knowledge in a new (more complex) system or a real environment, the ML/RL-agent trained in a simulation can directly interact or be retrained in the real world much faster than when starting from the scratch [8].

How well the agent copes with the real-world environment depends on the accuracy of the simulation channel models that were used during training. In GNU Radio, there are lots of simulated channel models available ranging from very simple ones (e.g., Additive White Gaussian Noise, AWGN) to more realistic ones which take multi-path, mobility, and fast fading into account. Moreover, the impact of interference can be easily simulated in GNU Radio as well. Finally, note that our framework is not constrained 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).

Designing such a framework is challenging. First, GNU Radio programs need to run in real-time and cannot get paused as it is possible with pure simulations running in simulation time [11]. So, no computational expensive tasks can be executed within the agent's main control loop where the `step()` function is executed. However, offloading CPU intensive ML tasks is still feasible, e.g., running an ML agent on different machines is possible as there is a loose coupling between Gym and GNU Radio using inter-process communication.

4 GRGYM – DETAILED DESCRIPTION

The `GrGym` architecture is depicted in Figure 1. It consists of the following major components: GNU Radio and OpenAI Gym. The main contribution of this work is the design and implementation of a generic interface between OpenAI Gym and GNU Radio that allows for seamless integration of those two frameworks. In the following, we describe our `GrGym` framework in detail.

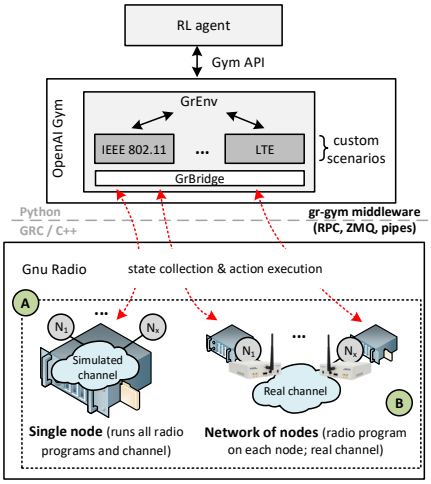


Figure 1: Architecture of the GrGym framework.

4.1 GNU Radio Side

Any radio transmission system implemented using the Open Source GNU Radio platform can be used with GrGym. However, the simplest integration is to use a radio program in form of a GNU Radio Companion (GRC) flow graph. Here, the following modifications need to be made to expose the data needed to capture the desired RL observation space and information needed to compute the reward value. Moreover, the possibility to change values of variables is needed as a mean to execute RL actions. This is achieved by modifying the GRC file of the radio program to be integrated into GrGym by adding additional GNU Radio blocks and wiring them accordingly. GrGym uses inter-process communication (IPC) for communication with GNU Radio processes, thus, we need the following blocks:

(1) `XMLRPC Server`: the server is needed for life-cycle management of the GNU Radio program by GrGym, i.e., start/stop, and (remote) execution of RL actions.

(2) `* Sink`: sink nodes are needed to capture both observations and reward. GrGym is able to use different IPC mechanisms: named pipes (files), ZeroMQ, UDP/TCP. The usage depends on whether GrGym and GNU Radio are co-located or not, i.e., ZeroMQ can be used if both processes are located on different machines.

4.2 OpenAI Gym Side

The main purpose of the Gym framework is to provide a standardized interface allowing for accessing the state and executing actions in an environment implemented using GNU Radio. This makes the RL-agent's code (Python) environment-independent, which allows for exchanging the agent's implementation while keeping the reproducibility of the environment's conditions.

4.3 GrGym Middleware

GrGym takes care of transferring state (i.e., observations, reward) and control (i.e., actions) between the Gym agent and the network of nodes running GNU Radio. The middleware consists of two parts: a generic part and a scenario-specific implementation. The generic part couples OpenAI Gym with GNU Radio. The scenario-specific part needs to be provided when adding new GNU Radio environments to GrGym framework. This is achieved by providing a custom Python class which derives from `GrScenario` and implements the required functions like `get_observation_space()`. Here, the user can access all the data exposed by the GNU Radio program using the `GR_Bridge` class. Note, that the GrGym middleware transfers the state and actions as numerical values and it is up to the user to define their semantics.

4.4 GrGym: Hello World!

In Listing 1, we present an example RL-agent, written in Python, showing the usage of the GrGym framework. First, the GNU Radio environment and agent are initialized — lines 4–7. Note that the creation of the `grgym:grenv-v0` environment is achieved using the standard Gym API. Behind the scenes, the GrGym engine instantiates the scenario class (derived from `GrScenario`) and starts the GNU Radio program(s) configured in `config.yaml`, establishes several IPC connections (XML-RPC, ZMQ, pipes), and waits for the environment initialization. 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 the agent class is not provided in the framework and the developers are free to define them as required. In the example, the simple agent performs random actions. The execution of the episode terminates (lines 13–14) when the environment returns `done=true`, that can be caused by the end of the GNU Radio program or meeting the predefined game-over condition.

```
1  import gym
2  import MyAgent
3
4  env = gym.make('grgym:grenv-v0')
5  env.seed(47)
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
15
16 env.close()
```

Listing 1: Example Python program showing the interaction between an agent and the GNU Radio environment

4.5 Implementation Details

Next, we describe the interactions between the different components of GrGym with GNU Radio. Specifically, we show how the different Gym functions, namely `Gym::make()`, `Gym::reset()`, and `Gym::step(action)` are implemented. By calling the `Gym::make()` function the framework compiles the GRC flow graph and start the Gnu radio process in case both GrGym and the GRC are located on the same machine, i.e., executed locally. Next, is the `Gym::reset()` function. First, it starts the GNU Radio process in case the GNU Radio autostart parameter is deactivated. Next, the custom scenario is reset and both the observation and the action space descriptions are collected for which internal data structures are created. Finally, we have the `Gym::step(action)` function (Figure 2). First, it executes the given action on the GNU Radio program by passing through the custom scenario class. Second, after waiting `step_time`, the data for reward, done, and info is collected. Note that the wait time is optional and no waiting happens in case GrGym is

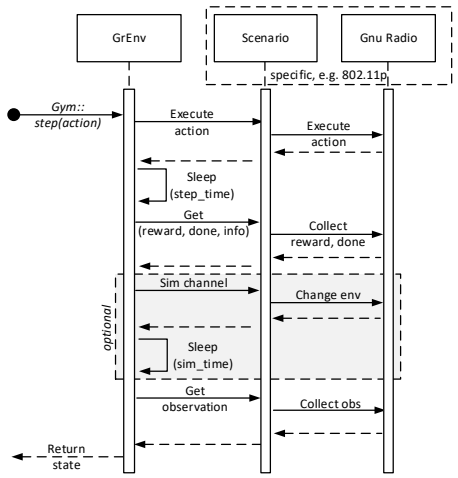


Figure 2: Implementation of `Gym::step(action)`.

running in `eventbased` mode where the appearance of a new observation completes a step. The third operation is also optional and is needed in case there is no real communication channel between the GNU Radio processes. Here, the channel is simulated in GNU Radio and this operation is responsible for changing its parameters, e.g., changing the SNR value. The last operation is the collection of the new observation state.

4.6 Typical Workflow

To summarize, the typical workflow consists of six steps:

(1) Setup a network of GNU Radio nodes. It can be simple as having a single node running a single GNU Radio process or a distributed system composed of multiple nodes each running a GNU Radio process.

(2) The GNU Radio programs should be described as a GNU Radio Companion (GRC) flow graph. Modify the GRC files by adding additional GNU Radio blocks (e.g., from `GrGym`) to get the data needed to compute observation and reward. Add also blocks needed for IPC communication with `GrGym` Python framework, i.e., XML RPC for both control of life cycle and execution of actions and ZMQ/file sink blocks for collecting observation and reward data.

(3) Write a specific `GrGym` scenario class (Python) which derives from framework's `GrScenario` base class and implements all needed functions, e.g., `execute_action()`. This class maps the generic framework functions on a specific scenario, e.g., action number is mapped on the MCS index.

(4) Wiring as defined in `config.yaml`, i.e., select the scenario and configuration (e.g., `eventbased` mode) – cf. Listing 2.

(5) Develop your RL-based agent using available numerical Python libraries (e.g., Keras), which interacts with the environment using the standard `Gym::step(action)` function.

(6) Train the agent and analyze the results.

5 GRGYM EXAMPLE: IEEE 802.11P

As a proof-of-concept, we implemented an IEEE 802.11p communication scenario in `GrGym`. We used the GNU Radio implementation provided by Bloessl et al. [3, 4]. The first step was to analyze the implementation and to expose as many parameters as possible to

```
grgym_environment:
  run_local: True # GNU Radio is local or remote
  timebased: # a step is progress in time
    step_time: 0.5 # step duration (in s)
  eventbased: True # if false use time based
  max_steps_zero_reward: 30 # max steps with no reward
grgym_local: # used if grgym_environment.run_local == True
  compile_and_start_gr: True # disable if remote
  host: localhost # local GNU Radio process
  rpc_port: 8080 # GNU Radio RPC port
  gr_ipc: ZMQ # IPC between grgym and gnuradio
  gr_grc: benchmark_ieee80211_wifi_loopback_zmq # used GRC flow graph
grgym_remote: # if grgym_environment.run_local == False
  num_nodes: 1
  node0:
    name: TX_RX_channel
    host: 10.0.0.2 # remote GnuRadio process
    rpc_port: 8080 # RPC port of remote GnuRadio
grgym_scenario:
  scenario_class: benchmark.BenchmarkScenario # used GrGym scenario
  mcs: 3 # scenario specific arguments
```

Listing 2: Example `GrGym` configuration file

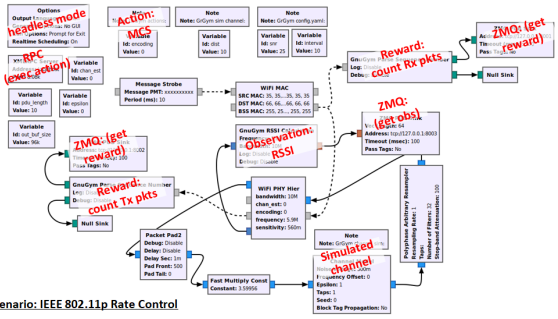


Figure 3: IEEE 802.11p flowgraph with additional blocks needed to expose it as specific scenario in `GrGym`.

the `GrGym` framework so that they can be controlled or observed by RL agents. For ease of use, a standalone configuration was selected. Here both the IEEE 802.11p transmitter and receiver are implemented in the same GNU-Radio flowchart and are connected by a simulated radio channel (here AWGN). As there is no test on a real channel no SDR hardware is needed to run this scenario. However, to introduce changes in the environment, here due to mobility, the distance between sender and receiver was varied during execution resulting in changing receive signal strength. The resulting GNU Radio flowchart is shown in Figure 3. The modifications and the additional blocks (marked in red) are needed to compute the reward as well as to capture the observation provided to the agent.

5.1 Building the Model

To use ML, at first a model is required. For the selected IEEE 802.11p scenario reward, action, and observation are defined as follows:

- **Action**: There are many parameters, which can be controlled in the IEEE 802.11p scenario. Examples include packet size, packet interval, and transmit power. As a proof-of-concept, we selected the configuration of the Modulation and Coding Scheme (MCS) of a packet as action. Note: IEEE 802.11p supports 8 different MCS values which corresponds to the action space.
- **Reward**: The reward is set as effective data rate during the last step. It is computed from the packet success rate during the last step multiplied by the data rate of the selected MCS.
- **Observation**: The observation is a vector of either RSSI per OFDM subcarrier or Signal to Noise Ratio (SNR) per subcarrier

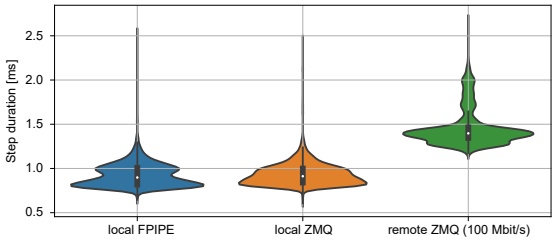


Figure 4: Framework performance.

of the last received packet – four subcarriers are pilot tones and at the border of the spectrum there are null tones. As the observation is measured during the synchronization phase of packet reception, it is independent of the selected MCS, i.e., action.

5.2 Additional GNU Radio Blocks

To calculate both the reward and observation, the GNU Radio 802.11p flow graph has to expose some state to be monitored by GrGym. First, knowledge about the packet success rate during the last step is needed. This is achieved by having a block for parsing the sequence numbers from the MAC header on both the transmitter and the receiver side. Second, the measurement of the RSSI or the SNR values is required. Therefore, an additional block extracts the data at the first or the first and the second sample of the synchronization word of an IEEE 802.11p packet. The RSSI block returns the absolute value of the Fast Fourier Transformation (FFT) output. The SNR block calculates the difference between the FFT value of the first two samples as noise and the sum of the first two values as signal. The resulting SNR is the division of signal and noise.

6 EVALUATION

First, as GNU Radio programs need to run in real-time, e.g., when using SDR systems like USRP, the latency of communication between the gym agent and the GNU Radio process becomes an important aspect to be considered when designing RL-based solutions. Second, we want to understand whether a distributed setup is meaningful, i.e., RL agent runs on one machine whereas the GNU Radio processes run on other machines. Such a configuration is sometimes needed in order to minimize the impact of the RL agent on the radio program running in real-time. Moreover, the agent's machine can be equipped with special hardware needed to speedup ML computations, e.g., graphic cards, which is not required for the machines running GNU Radio. For the experiments, we use the IEEE 802.11p example scenario described in Section 5. We measured the duration of a single step for three different configurations:

(1) Both the Gym agent and the GNU Radio process are co-located on the same machine. For the collection of observation data a named pipe is used.

(2) Similar to (1), except that ZMQ is used instead of the pipe.

(3) The Gym agent and the GNU radio process are running on two different machines using ZMQ for IPC. Moreover, both machines were connected by a rather slow network connection (100 Mbit/s).

As hardware we used embedded Intel NUC (i5, 2.3 GHz). `GrGym` was used in `eventbased` mode: a step consists of executing the action, i.e., sending a single 802.11p packet, and collecting the observation, i.e., the RSSI vector of the received packet. Figure 4 shows the step duration for the three configurations. We see that there is no significant disadvantage of using the ZMQ IPC protocol for collecting RSSI observations as compared to local named pipes. This is because the communication delay is mostly determined by the expensive XML RPC call, which is needed to execute an action. Moreover, we can see that there is no need to co-locate both the agent and the radio program on the same machine. In a remote setup the median step duration only slightly increases, i.e., from 0.9 ms to 1.4 ms. This confirms that a distributed setup is feasible.

7 CASE STUDY: RL-BASED RATE CONTROL

The goal of the agent is to decide on the MCS to be used for the next packet transmissions (step) based on the observation of the current channel condition, which is the absolute signal strength (RSSI) per OFDM subcarrier. This is a challenging task as the RSSI is uncalibrated, i.e., level of the noise floor is unknown. Therefore, the agent needs to learn which MCS to use given the absolute RSSI values reported. We use RL, in particular the Actor-Critic (AC) method [16], to learn to select the best MCS given observed RSSI. As a reward, we use the effective throughput, which is defined as the packet success rate $\times$ bitrate. In this case study, the agent had to choose from three different MCS values, i.e., QPSK 3/4, 16QAM 3/4, and 64QAM 3/4. The agent pre-processes the raw observation (per OFDM subcarrier RSSI) reported by the environment. As the simulated channel is an AWGN channel on top of distance-dependent pathloss, we compressed the observation as follows: First, we removed the DC and null subcarriers. Second, we computed the RSSI mean value, which is fine as the channel is frequency flat. Third, the mean RSSI value is normalized into an $[0, 1]$ interval. The corresponding neural network is shown in Listing 3. In summary, our RL mapping is:

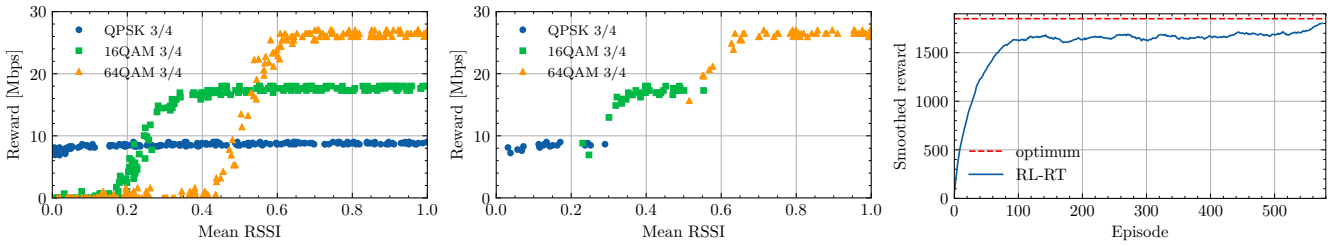
- **Observation** — mean RSSI normalized into $[0, 1]$ interval,
- **Action** — set the MCS to be used for the next time slot,
- **Reward** — effective throughput computed over last step,
- **Gameover** — if effective throughput was zero during the last ten time-slots.

```
inputs = layers.Input(shape=(1,))
common = layers.Dense(128, activation="relu")(inputs)
action = layers.Dense(env.action_space.n, activation="softmax")(common)
critic = layers.Dense(1)(common)
model  = keras.Model(inputs=inputs, outputs=[action, critic])
```

Listing 3: Neural Network used by Actor Critic method

`GrGym` was run in standalone mode and the 802.11p GNU Radio stack was operated in loopback mode, i.e., the channel was simulated in GNU Radio. To simulate mobility, the distance between the transmitter and receiver was changed randomly every 100 ms.

In the beginning, our RL-agent randomly tests different MCS regardless of the observed RSSI (Figure 5a). Even at very low RSSI, a high MCS was often selected resulting in a zero reward. Similarly, at high RSSI low-order MCS are selected resulting in a low reward. After learning the environment for 570 episodes, the agent perfectly selects the correct MCS rates (Figure 5b). We see that the agent learned three different RSSI regions, i.e., one for each MCS. Finally, we see that the learning is quite fast: after 570 episodes the agent learns to select the proper MCS (Figure 5c).



(a) Random probing at $t = 0$. **(b) After learning ≈ 570 episodes.** **(c) Smoothed reward vs. episodes.**

Figure 5: RL-based MCS (rate) control.

Considerations when using a Real Channel

So far our RL agent was trained in an environment with a simulated radio channel, i.e., AWGN on top of a distance-dependent pathloss model. However, the agent can be also trained in a real testbed using SDR hardware with a real wireless channel. In such a configuration the latency introduced by our GrGym framework becomes an important aspect to be considered as it may lead to situations where the agent decides on an action based on outdated observation. In the following, we want to analyze this using our example of RL-based rate control. The efficiency of such a closed-loop rate control depends on two variables: i/ the channel coherence time T_c, i.e., the time the channel is not varying and hence having the same SNR and ii/ the time duration τ_{obs} needed for collecting the observation (channel SNR) and duration τ_{act} for executing the action (setting MCS on packet). Note, that T_c depends on the carrier frequency and the speed of the mobile v.

For our analysis we consider 802.11a PHY with values for T_c provided by Jung *et al.* [14] while assuming a block fading channel model, i.e., the channel is changing i.i.d. (randomly) every T_c, and the set of available MCS is large so that after each T_c a different MCS has to be selected which represents a worst case analysis. Using our results from Section 6 we set $\tau = \tau_{\text{obs}} + \tau_{\text{act}}$ to 0.9 ms and 1.4 ms for the local and remote GrGym configuration respectively. Now we can define the miss ratio M which represents the percentage of time the incorrect MCS is selected as: $M = \tau / T_c$. Table 1 shows the results for M for different v and GrGym configuration.

From the results we can see that for static and low-speed scenarios the average miss ratio is small, i.e., only 3.5-5.5% of the time the incorrect MCS is selected due to outdated observation and delayed execution of actions. However, in environments with moderate speed, e.g., 5 m/s = 18 km/h, and remote GrGym configuration the M increases significantly. Here 1 out of 4 packets will be transmitted on possibly wrong MCS. We can conclude that the latency introduced by the GrGym framework needs to be taken into account when designing wireless RL solutions.

v [m/s]	T_c [ms]	$M(\%, local)$	$M(\%, remote)$
1	25.4	3.54	5.51
2	12.7	7.09	11.03
3	8.5	10.63	16.54
4	6.3	14.18	22.06
5	5.1	17.72	27.57

Table 1: Coherence time vs. miss ratio.

8 RELATED WORK

In the RL community, the concept of a gym has gained much attention after being proposed by Brockman et al. [6] within the OpenAI project. Since then, gyms have been used well beyond robotics [20]. For example, Vinitsky et al. [19] used road traffic simulations as a gym environment. It enables research of road traffic control and aims to provide benchmarks for relevant traffic situations that can be handled using RL agents. Moreover, Hein et al. [13] published an industrial benchmark allowing studying the performance of RL algorithms applied to different real-world industry control problems. Recently, we published ns3-gym [11], which allows using a simulated communication network like WiFi or LTE as a gym environment so that RL agents could control the behavior of network protocols. This was achieved by providing an extension to the ns3 network simulator. Now, GNU Radio has to process sample streams in real-time, whereas the ns3 simulator is event-based and much easier to integrate with the gym concept. O'Shea and West [18] also covers applications of ML to the radio signal processing domain. They proposed to use GNU Radio as an enabler to create good datasets for model learning by simply using the already existing processing blocks. However, the interaction with the ML framework was done only in postprocessing mode.

9 CONCLUSIONS

We presented the GrGym toolkit that simplifies the usage of Reinforcement Learning (RL) for solving problems in the area of communication networks. In particular, we interconnected the OpenAI Gym with the GNU Radio framework. Our Open Source framework is generic: It can be used by the research community to tackle a variety of communication and networking problems using an RL approach. We plan to provide custom scenario implementations for technologies like IEEE 802.15.4 and LTE. Moreover, we envision to set up a global leaderboard allowing researchers to share and compare their results for various environments.

As future work, we plan to address the limitations of GrGym like the framework latency which adversely impacts both the agent's learning as well as the inference phase. The former can be partially solved by using simulated or emulated channels because GrGym has the possibility to control the evolution of the channel. Here the framework latency is not an issue and only increases the learning time. The inference phase can be accelerated by directly using the agent with its trained neural network in GNU Radio using the C++ interface of ML libraries like Tensorflow. We plan to provide software support to make this transition easy. Finally, the framework

latency itself can be improved by using shared memory for IPC between the GNU Radio processes and our Python framework.

Finally, we want to evolve the framework beyond just parameter learning for a pre-selected radio flowgraph. Instead, the RL/ML agent learns itself to build the best flowgraph from a repository of available radio components.

REFERENCES

[1] Rami Akeela and Behnam Dezfouli. 2018. Software-defined Radios: Architecture, state-of-the-art, and challenges. *Elsevier Computer Communications* (2018), 106–125. https://doi.org/10.1016/j.comcom.2018.07.012

[2] Bastian Bloessl, Christoph Leitner, Falko Dressler, and Christoph Sommer. 2013. A GNU Radio-based IEEE 802.15.4 Testbed. In *12. GI/ITG KuVS Fachgespräch Drahtlose Sensornetze (FGSN 2013)*. Cottbus, Germany, 37–40.

[3] Bastian Bloessl, Michele Segata, Christoph Sommer, and Falko Dressler. 2013. An IEEE 802.11a/g/p OFDM Receiver for GNU Radio. In *ACM SIGCOMM 2013, 2nd ACM Software Radio Implementation Forum (SRIF 2013)*. ACM, Hong Kong, China, 9–16. https://doi.org/10.1145/2491246.2491248

[4] Bastian Bloessl, Michele Segata, Christoph Sommer, and Falko Dressler. 2018. Performance Assessment of IEEE 802.11p with an Open Source SDR-based Prototype. *IEEE Transactions on Mobile Computing (TMC)* 17, 5 (May 2018), 1162–1175. https://doi.org/10.1109/TMC.2017.2751474

[5] Eric Blossom. 2004. GNU Radio: Tools for Exploring the Radio Frequency Spectrum. *Linux Journal* 2004, 122 (June 2004).

[6] Greg Brockman, Vicki Cheung, Ludwig Petterson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. 2016. *OpenAI Gym*. cs.LG 1606.01540. arXiv. https://arxiv.org/abs/1606.01540

[7] 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 *32nd AAAI Conference on Artificial Intelligence (AAAI-18)*. New Orleans, LA.

[8] 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*. cs.RO 1610.03518. arXiv.

[9] Zubair Md. Fadlullah, Fengxiao Tang, Bomin Mao, Nei Kato, Osamu Akashi, Takeru Inoue, and Kimihiro Mizutani. 2017. State-of-the-Art Deep Learning: Evolving Machine Intelligence Toward Tomorrow's Intelligent Network Traffic Control Systems. *IEEE Communications Surveys & Tutorials* 19, 4 (Oct. 2017), 2432–2455. https://doi.org/10.1109/comst.2017.2707140

[10] Adrian Garcia-Rodriguez, David Lopez-Perez, Lorenzo Galati-Giordano, and Giovanni Geraci. 2020. *IEEE 802.11be: Wi-Fi 7 Strikes Back*. cs.NI 2008.02815. arXiv.

[11] Piotr Gawłowicz and Anatolij Zubow. 2019. ns-3 meets OpenAI Gym: The Playground for Machine Learning in Networking Research. In *22nd ACM International Conference on Modelling, Analysis and Simulation of Wireless and Mobile Systems (MSWiM 2019)*. ACM, Miami Beach, FL. https://doi.org/10.1145/3345768.3355908

[12] Ismael Gomez-Miguelez, Andres Garcia-Saavedra, Paul D. Sutton, Pablo Serrano, Cristina Cano, and Doug J. Leith. 2016. srsLTE: An Open-Source Platform for LTE Evolution and Experimentation. In *22nd ACM International Conference on Mobile Computing and Networking (MobiCom 2016), 10th ACM International Workshop on Wireless Network Testbeds, Experimental evaluation and Characterization (WiNTECH 2016)*. ACM, New York City, NY, 25–32. https://doi.org/10.1145/2980159.2980163

[13] Daniel Hein, Stefan Depeweg, Michel Tokic, Steffen Udluft, Alexander Hentschel, Thomas A. Runkler, and Volkmar Sterzing. 2017. A benchmark environment motivated by industrial control problems. In *IEEE Symposium Series on Computational Intelligence (SSCI 2017)*. IEEE, Honolulu, HI. https://doi.org/10.1109/ssci.2017.8280935

[14] Hakyung Jung, Ted "Taekyoung" Kwon, Kideok Cho, and Yanghee Choi. 2011. REACT: Rate Adaptation using Coherence Time in 802.11 WLANs. *Elsevier Computer Communications* 34, 11 (July 2011), 1316–1327. https://doi.org/10.1016/j.comcom.2011.01.011

[15] Jens Kober, J. Andrew Bagnell, and Jan Peters. 2013. Reinforcement learning in robotics: A survey. *The International Journal of Robotics Research* 32, 11 (Aug. 2013), 1238–1274. https://doi.org/10.1177/0278364913495721

[16] Vijay R. Konda and John N. Tsitsiklis. 2003. On Actor-Critic Algorithms. *SIAM Journal on Control and Optimization* 42, 4 (Jan. 2003), 1143–1166. https://doi.org/10.1137/s0363012901385691

[17] Hongzi Mao, Mohammad Alizadeh, Ishai Menache, and Srikanth Kandula. 2016. Resource Management with Deep Reinforcement Learning. In *15th ACM Workshop on Hot Topics in Networks (HotNets 2016)*. ACM, Atlanta, GA. https://doi.org/10.1145/3005745.3005750

[18] Timothy O'Shea and Nathan West. 2016. Radio Machine Learning Dataset Generation with GNU Radio. In *GNU Radio Conference*. Boulder, CO.

[19] Eugene Vinitsky, Aboudy Kreidieh, Luc Le Flem, Nishant Kheterpal, Kathy Jang, Cathy Wu, Fangyu Wu, Richard Liaw, Eric Liang, and Alexandre M. Bayen. 2018. Benchmarks for reinforcement learning in mixed-autonomy traffic. In *2nd Conference on Robot Learning (CoRL 2018) (Conference on Robot Learning, Vol. 87)*, Aude Billard, Anca Dragan, Jan Peters, and Jun Morimoto (Eds.). PMLR, Zürich, Switzerland, 399–409.

[20] 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*. cs.RO 1608.05742. arXiv.