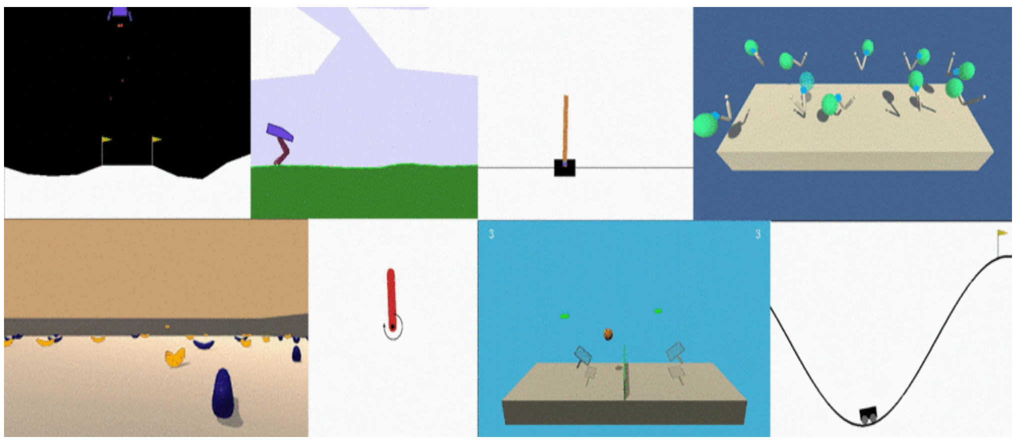
GrGym: When GNU Radio goes to (AI) Gym

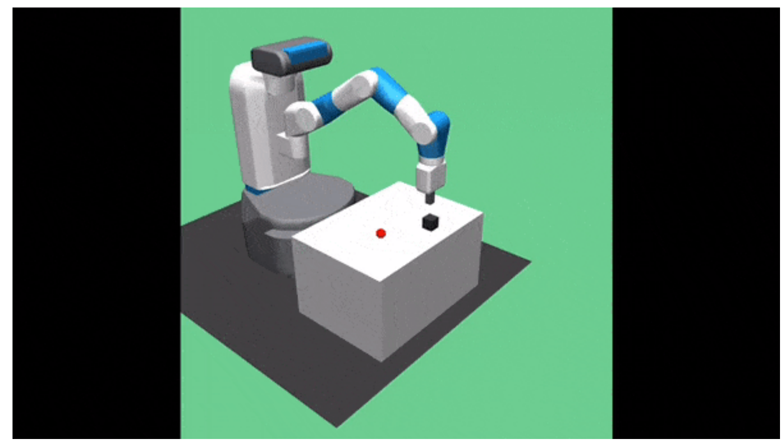
Anatolij Zubow, Sascha Rösler, Piotr Gawłowicz, Falko Dressler

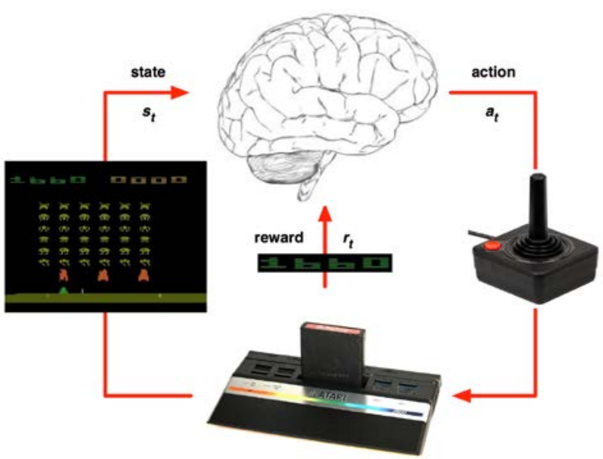
Talk@ACM HotMobile 2021

Motivation

- Boom of applications using **Reinforcement Learning**







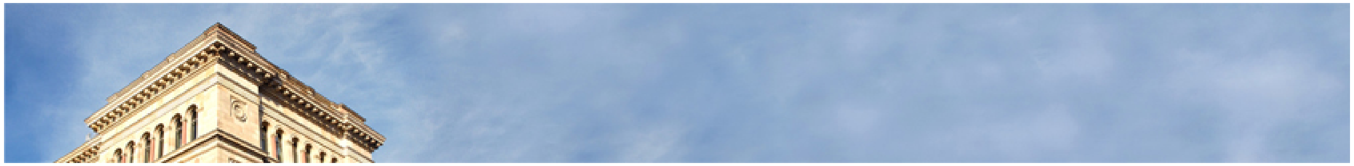
OpenAI Gym

- **Gym** is **open-source** framework with vast set of standardized environments including algorithmic examples, games and 3D robots
- **Gym** allows for developing and comparing **Reinforcement Learning** (RL) algorithms in the same virtual conditions
- **Gym** is a wrapper that provides an **unified environment API**:
  - reset()
  - next_state = step(action)
  - (optional) render()
- New environment can be integrated:
  - Need to represent state & actions as numerical values

```
import gym

env = gym.make('CartPole-v0')
obs = env.reset()
agent = MyGreatAgent()
done = False

while not done:
    action = agent.get_next_action(obs)
    obs, reward, done, info = env.step(action)
```

GNU Radio

- Toolkit with rich library of signal-processing blocks for building **software-defined radios**
- Design of flow graph (XML/Python): vertices are signal processing blocks (C++), edges represent data flow between them
- Each block processes in real-time an infinite stream of data flowing from its input ports to its output ports
- Partial/full implementations of 802.11, 802.15.4, LTE
- GNU Radio programs run on **real hardware** (USRP) or loopback in a fully **simulated environment**



GrGym framework: Design Principles

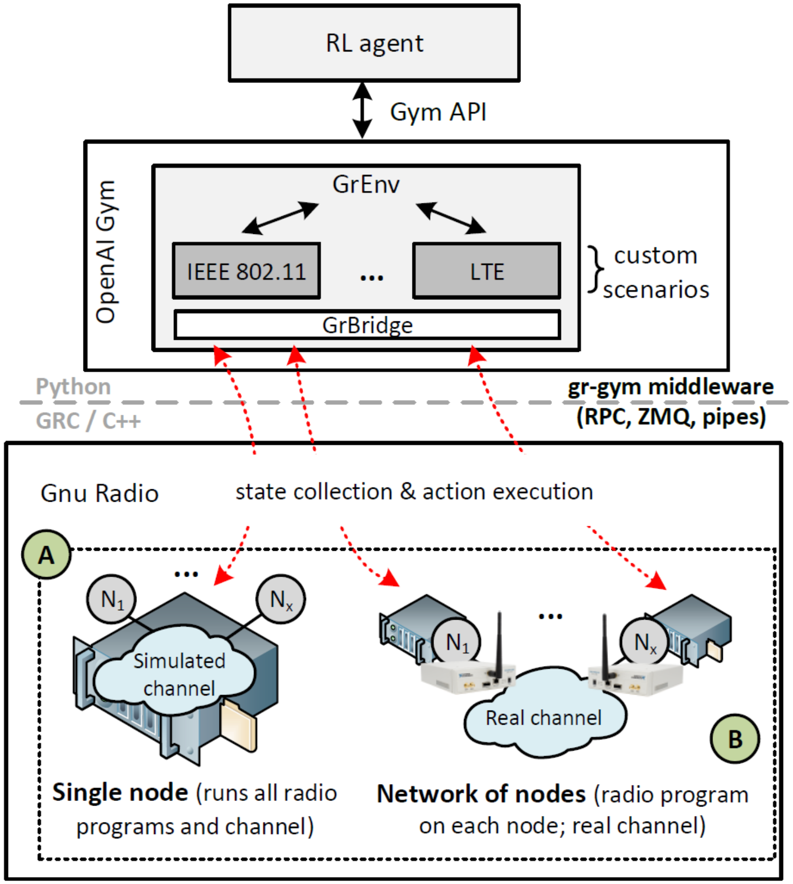
- Modern (wireless) communication networks have evolved into **complex & dynamic systems**, e.g. hundreds of knobs in 802.11ax/be
- **New approaches** needed for control & management of such networks, i.e. application of ML techniques like RL
- **Goal**: facilitate and shorten time required for developing novel **RL-based communication networking** solutions
  - RL-driven control algorithms should be trained in a simulated environment before running in real world
  - Flexibility of SDR platforms allows to quickly switch from simulated environment to real-world
  - **Transfer learning**

bit.ly/3upLFk3

TKN

GrGym framework

- A **generic interface** between OpenAI Gym & GNU Radio
- Only **small changes** to radio programs (GRC flow graph) needed to make them usable by framework:
  - Blocks added for IPC with framework
  - Life-cycle management
  - Collection of observation & reward
- GrGym **middleware** takes care of transferring state (observations, reward) & control (actions) between agent and network of GNU Radio nodes:
  - Two parts: i) generic and ii) scenario-specific implementation



Architecture of **GrGym** framework

GrGym: Basic Example

1. Configuration file (YAML)
2. RL agent (Python)

1

```yaml
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
     # scenario specific arguments
```

remote or local ?

GrGym scenario class

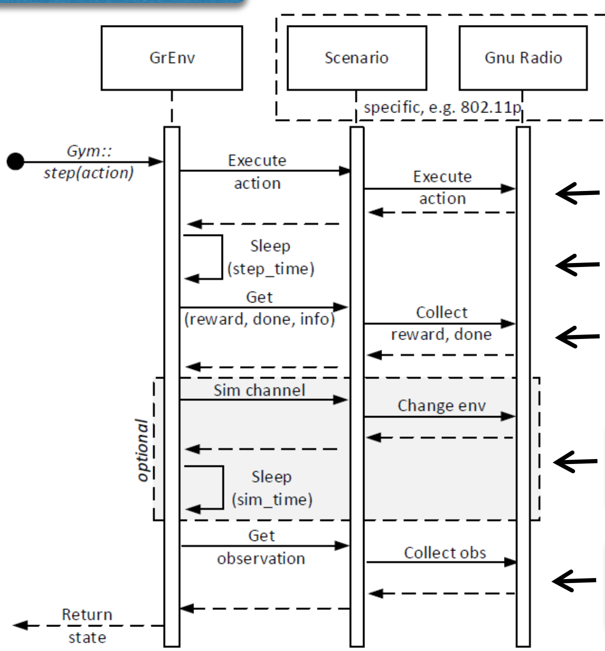
2

```python
ort gym
ort MyAgent

 = gym.make('grgym:grenv-v0')
.seed(47)
obs = env.reset()
ent = MyAgent.Agent()

le True:
ction = agent.get_action(obs)
obs, reward, done, info = env.step(action)

f done:
  break

env.close()
```

Start GrGym

Interact with env. via step()

Implementation of step()



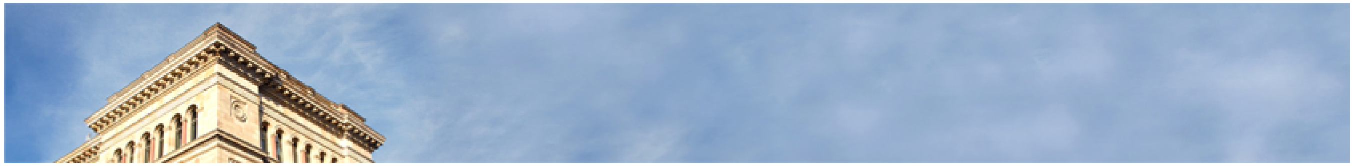
Exec action

Pause

Get reward

Sim channel

Get observation



GrGym: Workflow

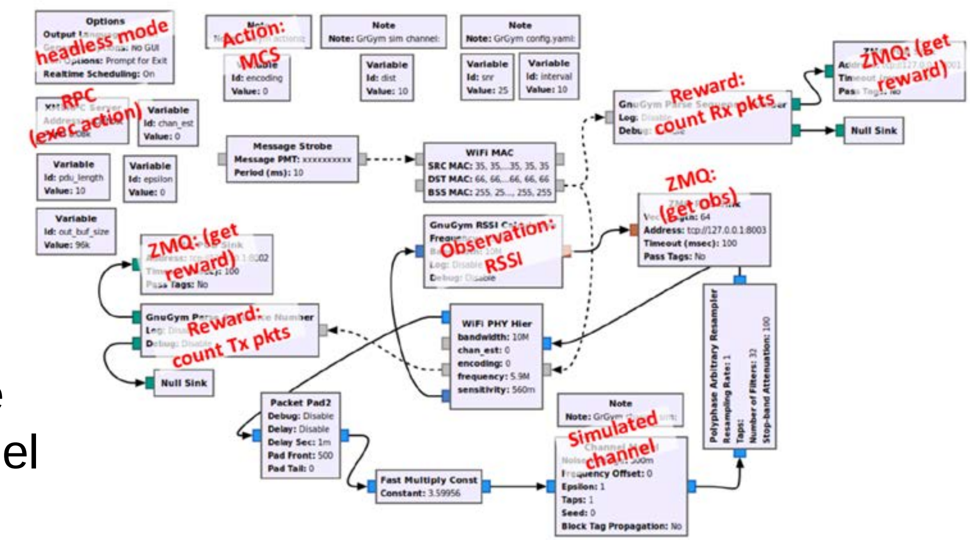
- **Workflow** consists of **6 steps**:

  1. **Setup** single or network of GNU Radio nodes
  2. **Modify** radio programs (described as GRC flow graph)
     - Add blocks to get data for observation/reward
     - Add blocks for IPC with GrGym (XML RPC, ZMQ/file)
  3. **Write** GrGym scenario (Python) which implements all functions,
     - Maps generic framework functions to scenario, e.g., action= MCS index
  4. **Wire** everything with *config.yaml*
  5. **Write** RL agent which interacts with environment via Gym API
  6. **Train** the agent and analyze results

Example Scenario: IEEE 802.11 Rate Control

- 802.11p based on [1] as proof-of-concept scenario
- **RL modeling for closed-loop rate control**:
  - Action (MCS)
  - Reward (effective data rate)
  - Observation (RSSI)
- **GrGym configuration**:
  - Single flowgraph: TX & RX are connected by simulated channel
  - Additional GNU Radio blocks added (counting sequence no., RSSI)

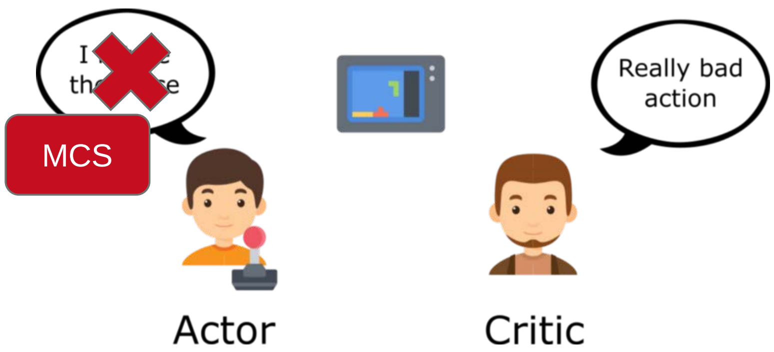
GRC flowgraph



[1] Bloessl et al., „An IEEE 802.11a/g/p OFDM Receiver for GNU Radio“, ACM SIGCOMM 2013

Case Study: RL-based Rate Control

- **Objective:** agent decides on MCS for next packet transmissions in 802.11p scenario
- Observation is current channel condition, i.e. absolute signal strength (RSSI) per OFDM subcarrier
- Challenging as RSSI is uncalibrated, i.e., unknown noise floor
- **Learn to map** absolute RSSI to MCS
- Agent uses **Actor-Critic** (AC) method
- Reward = effective throughput,
  i.e. PSR × bitrate



bit.ly/2Nul30i

TKN

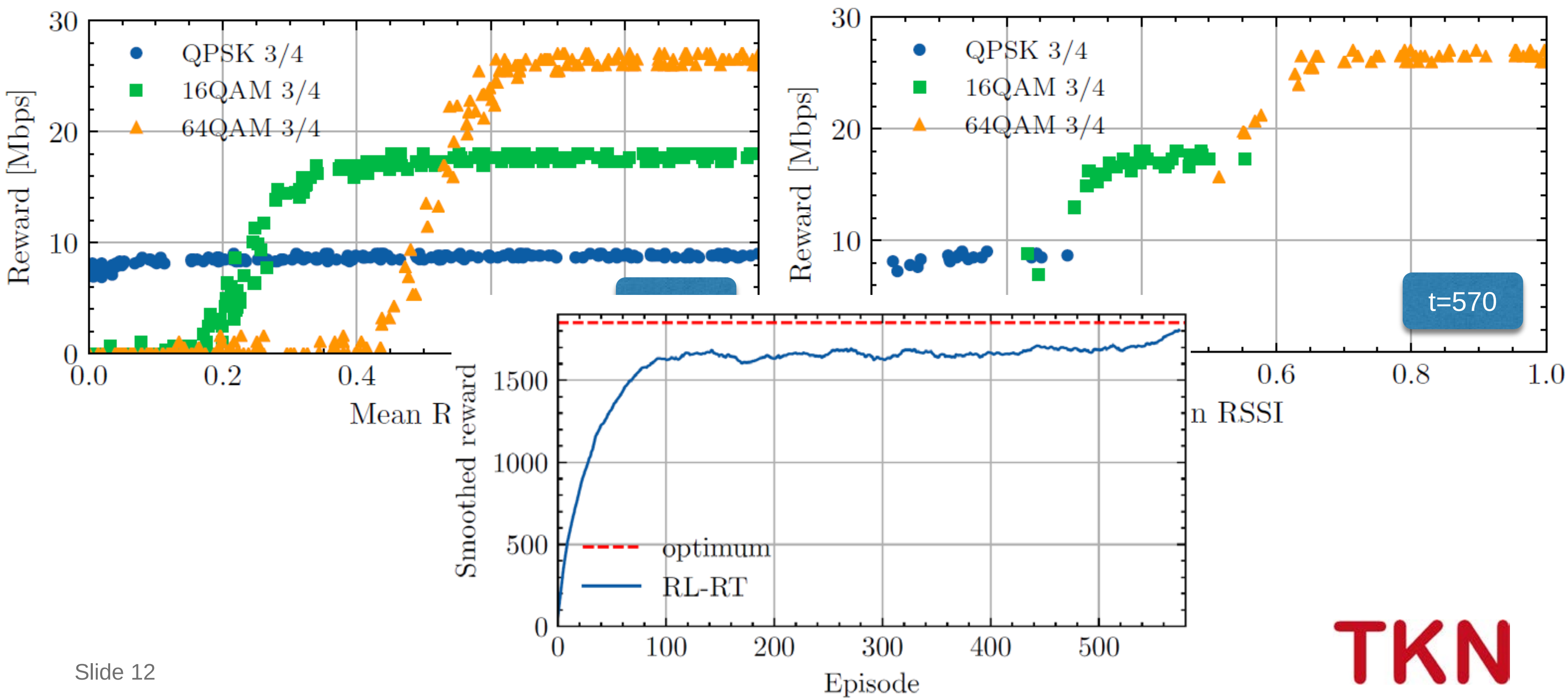
Case Study: RL-based Rate Control (II)

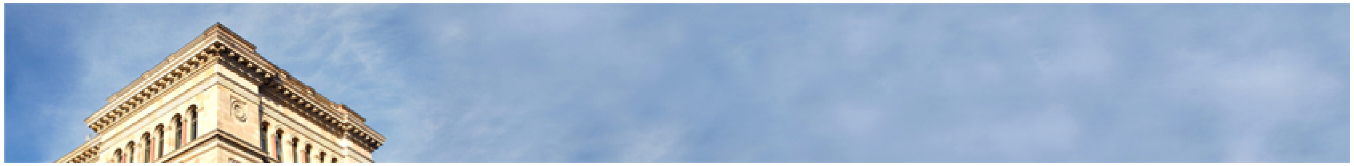
- GrGym setup:
  - Standalone mode with simulated channel
  - AWGN, mobility => distance changed randomly every 100 ms
- RL mapping due to further simplifications:
  - Observation — mean RSSI normalized into [0, 1],
  - Action — MCS for next time slot,
  - Reward — effective throughput computed over last step,
  - Gameover — if effective throughput was 0 during last 10 time slots
- Neural network used:

```
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])
```

Case Study: RL-based Rate Control (III)

- Results:
  - At t=0 RL-agent randomly tests different MCS regardless of RSSI
  - After t=570 episodes agent perfectly selects correct MCS

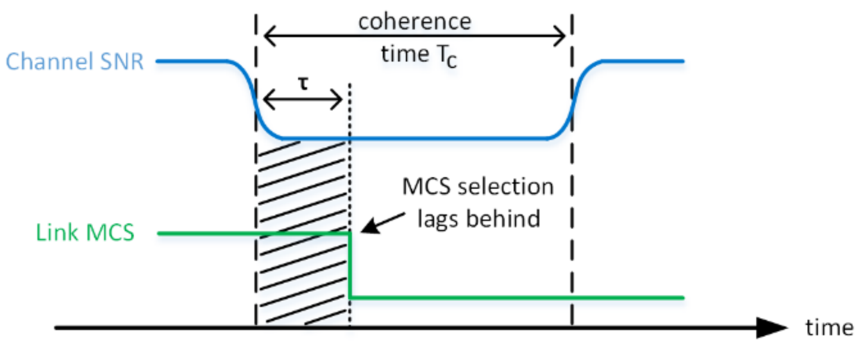




Case Study: RL-based Rate Control (IV)

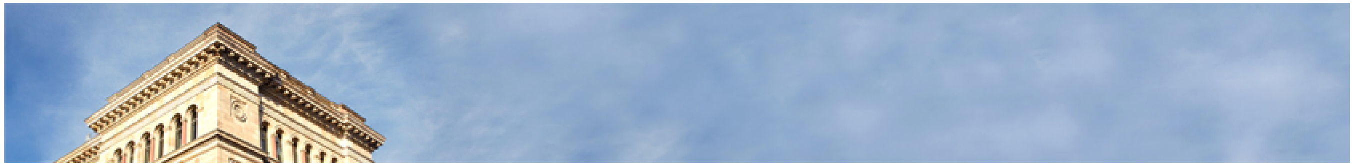
- Can we use a **real wireless channel**?
  - … so far RL agent trained in an environment with simulated channel
  - But agent can be trained in real testbed using SDR hardware with real **mobile (!)** wireless channel

  - Here framework **latency** becomes an issue!
    - agent should not decide on an action based on outdated observation
  - Let's analyze efficiency of RL-based rate control, i.e. miss ratio $M = \tau / T_c$



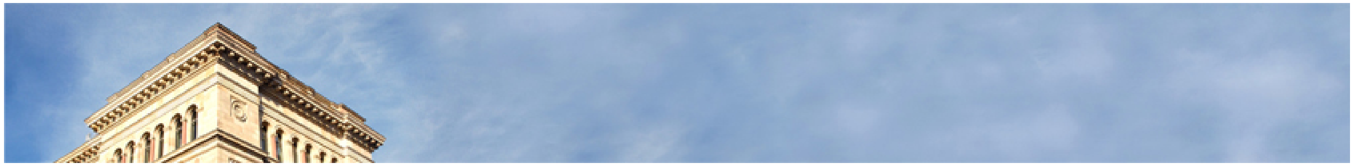
v [m/s]	T_c [ms]	$M(\%, local)$	$M(\%, remote)$
1	25.4	3.54	5.51
2	12.7	7.09	11.03
3	8.5	10.63	16.54
4	6.3	14.18	22.06
5	5.1	17.72	27.57

Coherence time vs. miss ratio



Conclusions

- GrGym – framework that simplifies usage of RL for solving problems in area of (wireless) communication networks
- It is based on OpenAI Gym and GNU Radio framework
- Plans for **future**:
  - Custom scenario implementations for ZigBee & LTE
  - Addressing framework limitations like latency
  - Going beyond simple parameter learning
- We hope for research community to grow around it



Thank you!

Q&A

Check GrGym on GitHub

https://github.com/tkn-tub/gr-gym