5G Down Link Power Control to Maximize Sum Throughput Using Reinforcement Learning

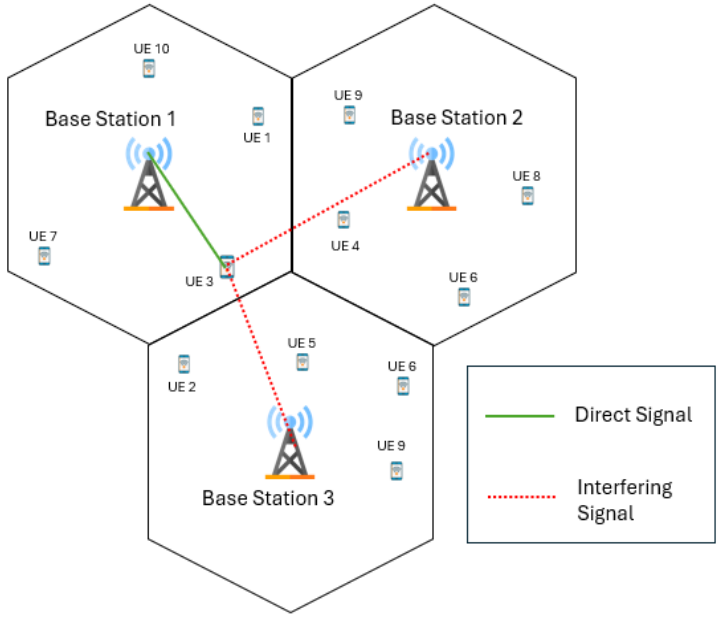
Implementation in NetSim

Applicable Release: NetSim v14.1 or higher
Applicable Version(s): NetSim Standard
Project download link: https://github.com/NetSim-TETCOS/RL_Based_Tx_Power_Control_v14.1/archive/refs/heads/main.zip
The URL has the exported NetSim scenario for the example used in this document and the python script to run the reinforcement learning simulation

Please refer to slide 21 for the steps to run the reinforcement learning (RL) algorithm.

Power Control: Introduction

- Due to the broadcast nature of wireless communication, signals interfere with each other.
- Interference can degrade the performance of the 5G RAN
- DL power control can be used to mitigate interference, ensure spectral reuse and improve user experience
- However, direct application to practical systems is impaired by the dependency on various simplifying assumptions
  - Knowledge of channel gains from all BSs to all UEs,
  - Full buffer traffic,
  - Knowledge of user positions, number of users, resources allocated to each user etc.
- In this project, we use reinforcement learning (RL) for downlink (DL) power control to mitigate interference, boost SINR and maximize sum throughput



Ghadimi et al, "A Reinforcement Learning Approach to Power Control and Rate Adaptation in Cellular Networks", arXiv:1611.06497v1 [math.OC] 20 Nov 2016

The Optimization Problem

- Optimization Problem: Received Signal to Interference plus Noise Ratio (SINR) of link i in slot t is a function of power allocation $\boldsymbol{p} = [p_1, \dots, p_n]^T$

$$\gamma_i^{(t)}(\boldsymbol{p}) = \frac{g_{ii}^t P_i}{\Sigma_{j \neq i} g_{ji}^{(t)} P_j + \sigma^2}$$

where g_{ii}^t is time varying due to mobility and fading.

- The objective is to maximize a weighted sum-rate utility function. The dynamic power allocation problem in time slot (t) is formulated as:

$$maximize \sum_{i=1}^{M} R_i\,(t)$$

where $R_i(t)$ is a function of $\gamma_i^{(t)}(\boldsymbol{p})$, since the instantaneous rate (MCS) can be obtained from 3GPP SE-MCS tables, and spectral efficiency (SE) is dependent on SINR.

Reinforcement Learning

- Environment:
  - The 5G cellular network (see next slide for details)
  - Nodes are stationary; Fading channel
- Agent:
  - Centralized oracle. Controls power in each gNB
- State:
  - Vector of received SINRs at the UEs
  - Sate changes every coherence time
- Action: Power Control i.e., power-up, power-down, power-hold
  - A control of ΔP_i applied to gNB_i
  - $\Delta P_i \in \{0\ dB, \pm 1, dB, \pm 3\ dB\}$
  - Agent applies power control every 3 frames (30 ms)
- Reward function:
  - Sum throughput
  - Throughput is obtained every 3 frames i.e., time between two consecutive actions

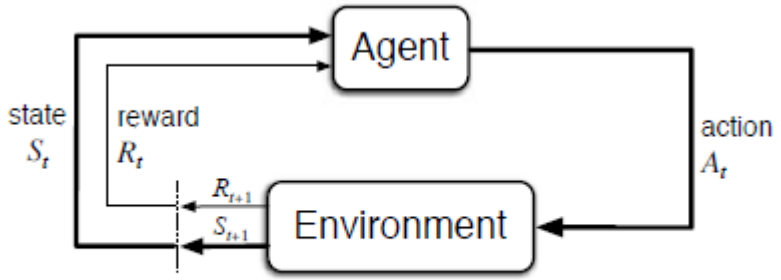
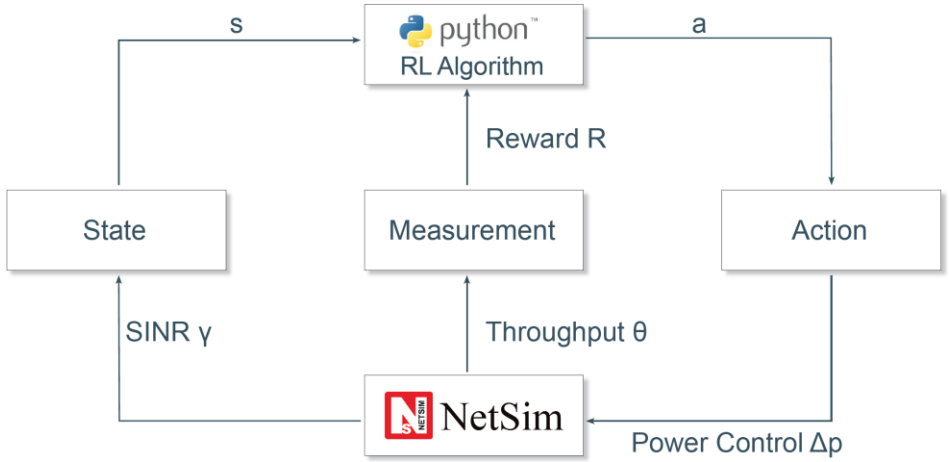


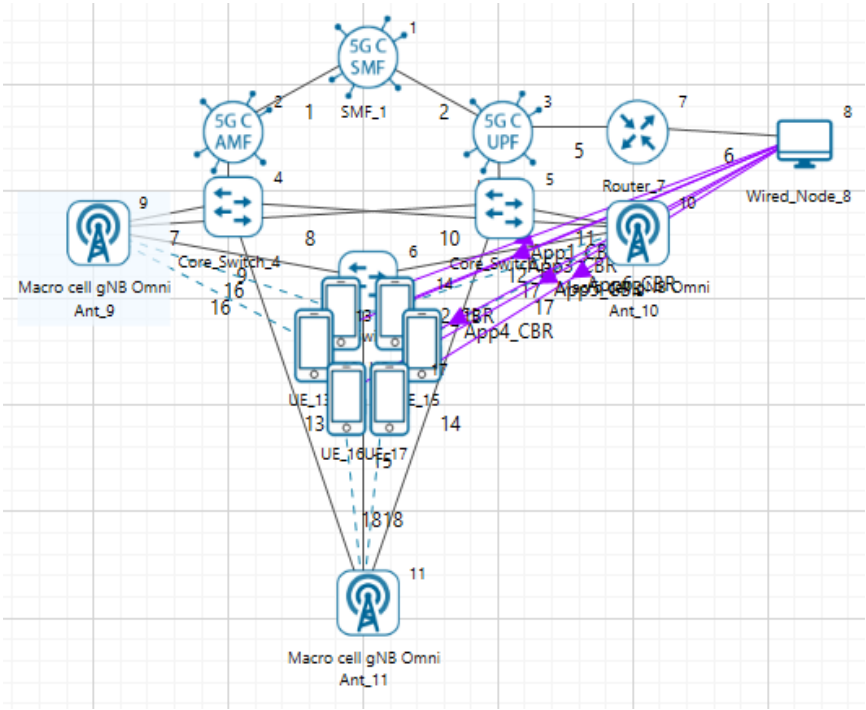
Figure 3.1: The agent–environment interaction in a Markov decision process.



The agent can be a “basic” user developed RL algorithm or an “advanced” Open AI Gym linked RL algorithm

NetSim Model

- 3gNBs and 6 UEs
  - Stationary UEs download data from a remote server
  - Each gNB transmits at a fixed power to all associated UEs
- Channel model
  - Pathloss: Log distance with $\eta = 2$
  - Rayleigh fading. Wideband fading
- “Geometric” interference in DL
- BS (gNB) determines the rate based on 3GPP standards
  - Maps received SINR to an MCS, per 3GPP table 2 (256QAM)
  - Rates seen will be different due to fading
- Antenna counts at gNBs and UEs: 1Tx, 1 Rx
- No error
- Full buffer traffic in all UEs.
  - Packet size 1500 B.



Agent Environment Interactions

NetSim (environment) provides a TCP socket with which a socket program/application written in any programming language can establish a connection.

In this example, a client program (agent) written in Python connects to the NetSim process at the specified port using a TCP connection for exchanging data.

The following is executed when the user runs a simulation:

- Initialize a listening socket in NetSim, that binds to any vacant port, say 12345
- For each episode
  - Initiate a new NetSim simulation and then create a client socket at Python that binds to the same port
  - For each iteration
    - Form an array of gNB powers in python, serialize the data into bytes and send the serialized data to Netsim
    - Receive the serialized data in Netsim, deserialize the data and convert the bytes into an array of gNB powers
    - Update the gNB powers in NetSim, and send back an array of SINRs and the sum throughput over to python after serializing the data into bytes
    - Receive the SINRs and the reward (sum throughput) from Netsim and take necessary action after updating the Q table (or model in case of A2C, PPO)
  - End
  - Terminate the client connection on the Python side
- End

NetSim Python Interfacing

Python C Socket Interfacing:

- Python facilitates seamless integration with C for socket programming tasks.
- Python uses the socket module to interact with C side functions.

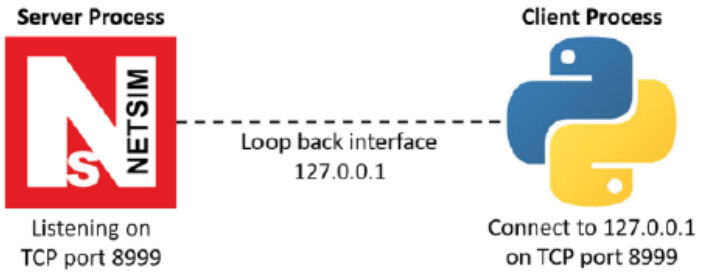
Python Code:

- Python creates a client socket in the host machine and connects with the C side server

Key function:

def NETSIM_interface(gNB_powers):

- Sends the gNB powers to NetSim with a header
- Receives acknowledgement from NetSim
- Sends a request value to NetSim to receive the updated SINR values and the reward
- Receives the next state and the reward from NetSim



NetSim Python Interfacing

NetSim C code

- Key functions include:
  - void init_waiting_struct_socket1() :
    - Initializes the Winsock library and calls the listenForPython() function
  - bool listenForPython() :
    - Resolves the server address and port
    - Creates a socket for connecting to the server
    - Sets up the TCP listening socket
    - Waits for client socket connection
  - void handle_Send_Receive(struct SINR_Values_Reward* Param1, struct gNB_Powers* Param2)
    - Receives the message type from Python
    - If message type is to “receive gNB powers”, receives the powers from Python and sends back an acknowledgement message
    - If message type is to “send SINRS and rewards”, sends back the state and reward back to Python using the send_SINRS_Rewards_at_Time_Step() function
  - void send_SINRS_Rewards_at_Time_Step() :
    - sends the list of updated SINRs and the sum throughput received from the LTENR project in NetSim back to python

Q- Learning Algorithm Overview

Q- Learning Algorithm:

- Aims to compute an optimal policy π that maximizes the expected reward.
- It does not require knowledge of the reward function form or state transitions

Future Cumulative Discounted Reward:

$$R^{(t)} = \sum_{\tau=0}^{\infty} \gamma^{\tau}\, r^{(t+\tau+1)}$$

where $\gamma \in (0,1]$: Discount factor for future use

Q- Function:

- The Q-function associated with policy π is defined as:

$$Q^{\pi}(s,a) = \mathbb{E}^{\pi}\left[R^{(t)} \middle|\, s^{(t)} = s, a^{(t)} = a\right]$$

- Represents the expected rewards when action a is taken under state s.

Bellman Equation for Q-Function:

$$Q^{\pi}(s,a) = R(s,a) + \gamma \sum_{s^1 \epsilon S} p^{a}_{ss^1} \left(\sum_{a^1 \epsilon A} \pi\,(s',a') Q^{\pi}(s',a') \right)$$

$R(s,a) = \mathbb{E}^{\pi}[r^{t+1} |\, s^{(t)} = s, a^{(t)} = a]$, expected rewards for action a at state s.

$P^{a}_{ss'} = \Pr\left(s^{(t+1)} = s' \middle| s^{(t)} = s, a^{(t)} = a\right)$, Probability of transitioning from state s to state s' at given action a

$\pi(s^1, a^1) Q^{\pi}(s^1, a^1)$

Q-Learning Algorithm

Q-Learning Algorithm:

- Constructs a lookup table $q(s, a)$ as an option for the optimal Q-function.
- Lookup table is randomly initialized.

ε – Greedy policy:

- With probability $1 - \epsilon$, the agent takes the action a that maximizes the lookup table value for the current state.
- With probability ϵ, the agent picks a random action to avoid non-optimal policies.

Q-Value Update Rule:

$$q(s^t, a^t) \leftarrow (1 - \alpha) q(s^t, a^t) + \alpha (r^{t+1} + \gamma \max_{a^1} q(s^{t+1}, a^1))$$

where $\alpha \epsilon (0,1]$: Learning rate.

γ: Discount factor for future rewards.

Tabular Q Learning: Algorithm

Algorithm 1 Tabular Q-learning Algorithm

```
Initialize all necessary variables and parameters as specified in the input.
Initialize the Q-table with random values representing the state-action values.
for episode = 1 to num_episodes do
    Reset the environment and receive initial SINR values.
    Initialize the state based on the initial SINR values from NETSIM.
    Set current gNB powers to the initial powers.
    for each time step within the episode do
        Choose an action using ϵ-greedy strategy
        Update gNB powers based on the chosen action.
        Receive new SINR values and reward from NETSIM.
        Calculate the next state based on the new SINR values.
        Update the Q-value for the current state-action pair
        Update the current state to the next state.
        Update the total reward for the episode.
        Decay ϵ if it's above the minimum threshold.
    end for
end for
```

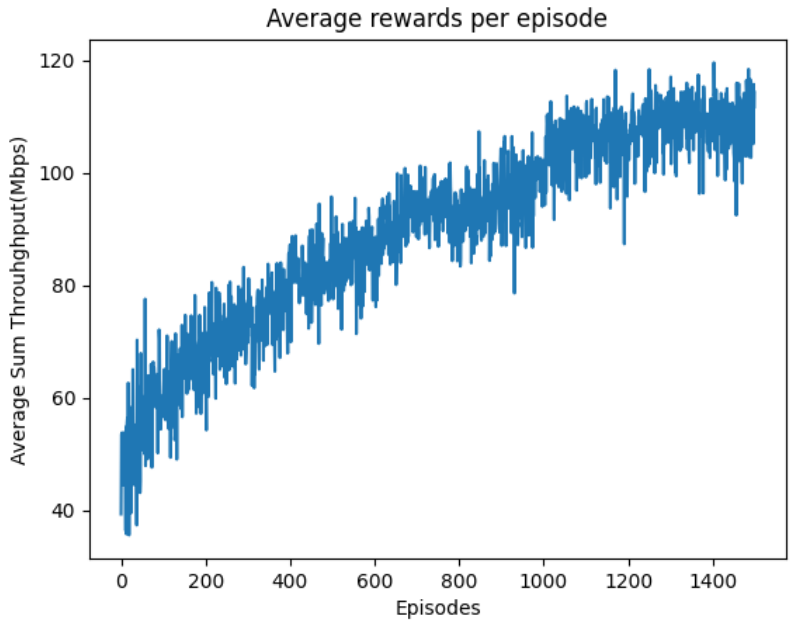
Hyper parameter	Value
Discount factor, γ	0.9
Learning rate, α	0.3
Epsilon greedy, ϵ	0.25

- To avoid the state and action space explosion, both state and action spaces are discretized.
- The MDP is therefore a finite-state, finite-action, discrete-time formulation.
- The state space is discretized based on SINR
  - Bucket 1: < -5 dB
  - Bucket 2: 5 dB to +3 dB
  - Bucket 3: +3 dB to +11 dB
  - Bucket 4: $\geq$ 11 dB
- Action space is discretized based on power up/down
  - $\Delta P = 0\ dB, \pm 1, dB, \pm 3\ dB$
  - Transmit power limits: [27, 46] dBm
- Number of entries in Q-table
  - States: 4^6 = 4096
  - Actions: 5^3 = 125
  - Total: 600k

Q- Learning tabular results

Scenario – 3 gNBs and 6 UEs

1500 Episodes with 500 iterations of each episode



Average rewards vs episode count

Average Sum Throughput (Mbps)	Without RL	With RL
	54.6	105.5

RL gives an **93.22 %** improvement

Interfacing with OpenAI Gymnasium for advanced RL algorithms

- OpenAI Gymnasium is a toolkit developed by OpenAI for developing and comparing reinforcement learning algorithms
- Allows to make a custom environment that will be used by the agent during learning.
- Custom environment framework includes:
  - reset(): to reset environment to the initial starting position, in our case [40,40,40] gNB
  - step(a): to take action a and return next state, reward, termination to agent.
- Allows use of different agents as DQN, PPO, A2C automatically vectorized for speed and making the environment a blackbox-like object for these to learn.
- The agents and environment are completely independent; we do not need to change anything in env to facilitate the agent.

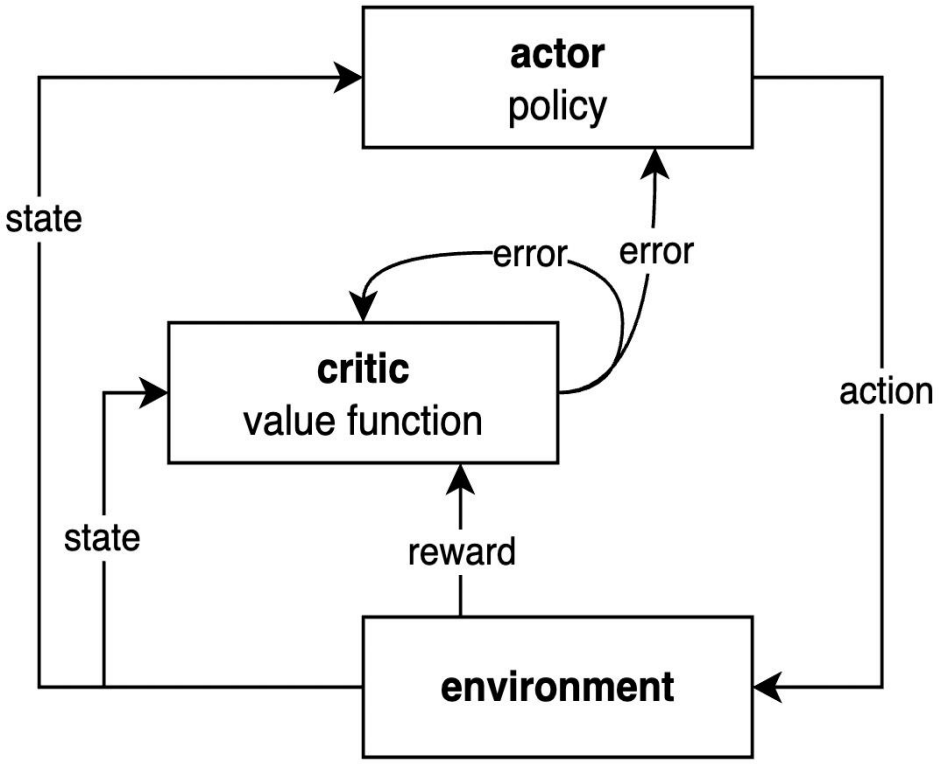
Deep Q Learning: Algorithm

Algorithm 4 Deep Q-Learning with Experience Replay

Initialize replay memory D with capacity N
Initialize Q-values with randoms weights
for $t = 1 \dots T$ **do**
  With probability ϵ select a random action a_t,
  otherwise select $a_t = \max_a Q^*(\phi(s_t), a, w_{t-1})$
  Execute action a_t
  Receive reward r_t and new state s_{t+1}
  Store transition $(\phi(s_t), a_t, r_t, \phi(s_{t+1}))$ into D
  Sample experience $(\phi(s_j), a_j, r_j, \phi(s_{j+1}))$ from D
  Set $y_j = \begin{cases} r_j & \text{if } s_{j+1} \text{ is terminal state} \\ r_j + \gamma max_{a_{j+1}} Q(\phi(s_{j+1}), a_{j+1}, w_{t-1}) & \text{otherwise} \end{cases}$
  Perform gradient descent update using $[y_j - Q(\phi(s_j, a_j, w_t)]^2$ as loss
end for

- Neural network Specifications:
  - Input: Array of SINR values for each UE, here (6)
  - Output : Index of action to be performed, here 125.
  - Architecture: Two 64 neuron fully connected layers with no/Linear activation.
- Action space is discretized based on power up/down
  - $\Delta P = 0\ dB, \pm 1, dB, \pm 3\ dB$
  - Transmit power limits: [27, 46] dBm
- Using the action index, we find the new gNB powers and new SINR.
- (Input SINR,action,reward,Output SINR) is saved in buffer.
- Double DQN is a technique to use 2 different neural nets and updating the second one less frequently than first.

Actor Critic Model: Introduction



- Components of Actor critic:
  - Actor: The policy model, which decides the actions to take based on the current state. It parameterizes the policy $\pi(a \mid s; \theta)$, where θ are the parameters of the actor network.
  - Critic: The value model, which evaluates the action taken by the actor by estimating the value function $V(s; w)$, where w are the parameters of the critic network. The critic provides feedback on how good the action taken by the actor is.
  - Advantage Function: A measure of how much better or worse an action is compared to the average action, calculated as:
    - $A(s, a) = Q(s, a) - V(s)$
    - In A2C, the advantage function is used to reduce variance in the policy gradient updates.(Input SINR,action,reward,Output SINR) is saved in buffer.
- DQN only focuses on value model and keep the policy constant (argmax) while A2C changes the policy mapping as well.

Actor Critic Model: Algorithm

Algorithm 3 N-step Advantage Actor Critic

Initialize actor network π_θ
Initialize critic network V_w
for iteration $= 1, 2, \ldots,$ num_episodes **do**
  Generate an episode $S_0, A_0, R_1, S_1, \ldots, S_{T-1}, A_{T-1}, R_t$ following π_θ
  for $t = 0, 1, 2, \ldots, T-1$ **do**
    $V_{\text{end}} = \begin{cases} V_w(S_{t+N}) & \text{if } t+N < T \\ 0 & \text{otherwise} \end{cases}$
    $G_t = (\sum_{k=t+1}^{\min(t+N,T)} \gamma^{k-t-1} R_k) + \gamma^N V_{end}$
  end for
  $L(\theta) = -\frac{1}{T} \sum_{t=0}^{T-1} (G_t - V_w(S_t)) \ln \pi_\theta(A_t|S_t)$
  $L(w) = -\frac{1}{T} \sum_{t=0}^{T-1} (G_t - V_w(S_t))^2$
  Update π_θ using Adam$(\nabla_\theta L(\theta))$
  Update V_w using Adam$(\nabla_w L(w))$
end for

- Neural network Specifications:
  - Input: Array of SINR values for each UE, here (6)
  - Output : Index of action to be performed, here 125.
  - Architecture: Two 64 neuron fully connected layers with no/Linear/Tanh activation.
- Action space is discretized based on power up/down
  - $\Delta P = 0\ dB, \pm 1, dB, \pm 3\ dB$
  - Transmit power limits: [27, 46] dBm
- Contains 2 independent Neural nets for the actor and the critic.
- More variants of actor critic exist like Soft Actor critic A3C

Proximal Policy Optimization: Algorithm

Algorithm 1 PPO-Clip

Input: initial policy parameters θ_0, initial value function parameters ϕ_0
for $k = 0, 1, 2, \ldots$ **do**
  Collect set of trajectories $\mathcal{D}_k = \{\tau_i\}$ by running policy $\pi_k = \pi(\theta_k)$ in the environment.
  Compute rewards-to-go $\hat{R}_t$.
  Compute advantage estimates, $\hat{A}_t$ (using any method of advantage estimation) based on the current value function V_{ϕ_k}.
  Update the policy by maximizing the PPO-Clip objective:

$$\theta_{k+1} = \arg\max_{\theta} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \min\left(\frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_k}(a_t|s_t)} A^{\pi_{\theta_k}}(s_t, a_t),\ \ g(\epsilon, A^{\pi_{\theta_k}}(s_t, a_t)) \right),$$

  typically via stochastic gradient ascent with Adam.
  Fit value function by regression on mean-squared error:

$$\phi_{k+1} = \arg\min_{\phi} \frac{1}{|\mathcal{D}_k|T} \sum_{\tau \in \mathcal{D}_k} \sum_{t=0}^{T} \left(V_\phi(s_t) - \hat{R}_t \right)^2,$$

  typically via some gradient descent algorithm.
end for

- Neural network Specifications:
  - Input: Array of SINR values for each UE, here (6)
  - Output : Index of action to be performed, here 125.
  - Architecture: Two 64 neuron fully connected layers with no/Linear/Tanh activation.
- Action space is discretized based on power up/down
  - $\Delta P = 0\ dB, \pm 1, dB, \pm 3\ dB$
  - Transmit power limits: [27, 46] dBm
- PPO is similar to A2C, but the main difference is the policy optimization:
  - It uses a clipped objective function

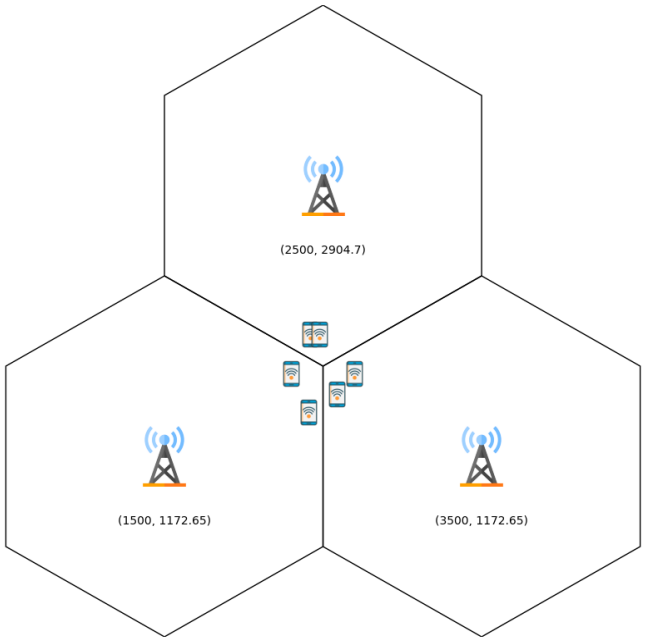
$$L^{CLIP}(\theta) = E_t\left[\min\left(r_t(\theta)\widehat{A_t}, clip(r_t(\theta), 1-\epsilon, 1+\epsilon)\widehat{A_t}\right)\right]$$

  - $r_t(\theta)$ is the probability ratio of old policy and new policy

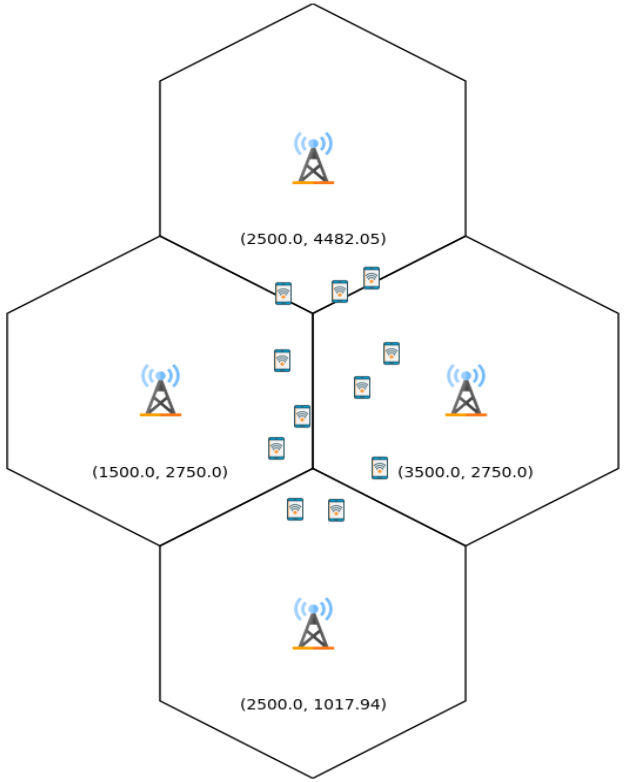
Example Scenarios

User Equipment (UE)

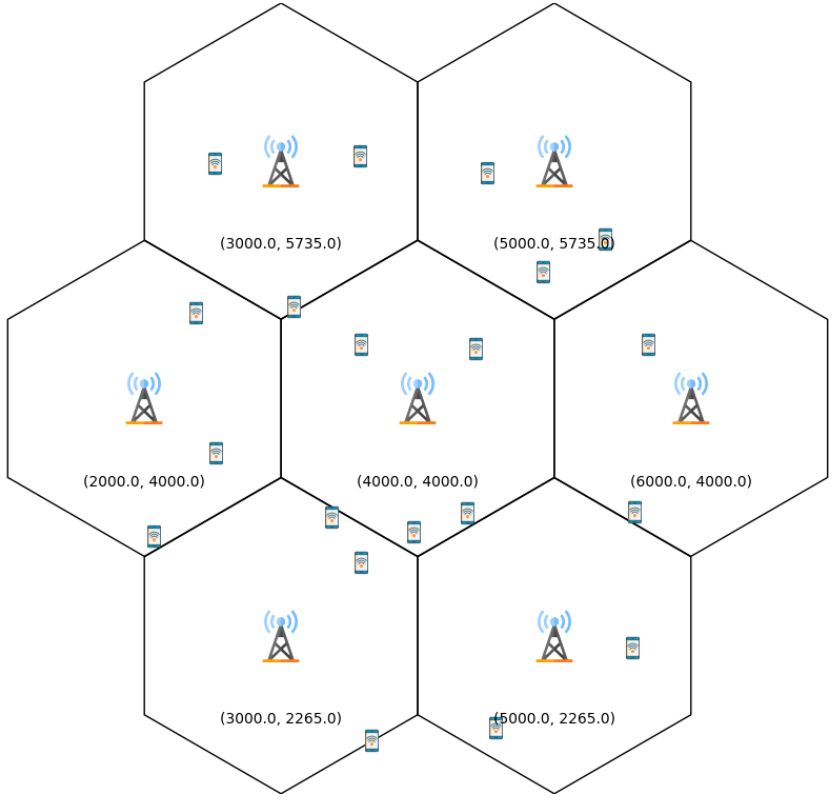
Base station (gNB)



3 gNBs 6 UEs
Tabular Q learning & PPO



4 gNBs 11 UEs
Tabular Q learning and PPO



7 gNBs 20 UEs
PPO only

State space is too large for tabular Q learning

System params. and Q-learning hyper params.

Scenario 1: 3gNBs and 6 UEs

System Parameter	Value
No of gNBs	3
No of UEs	6
Pathloss model	Log distance
Pathloss Exponent (η)	3
Fading model	Rayleigh
Scheduling	Round Robin
Error model	No error
gNB Bandwidth	50 MHz
Transmit Power Limit	[27, 46] dBm
Noise Power	-96.83 dB
Tx-Rx Antenna counts	1×1
Traffic Model	DL Full buffer

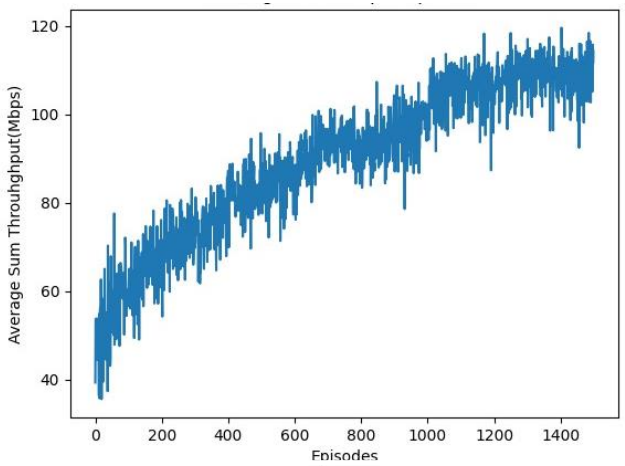
Scenario 2: 4 gNBs and 11 UEs

System Parameter	Value
No of gNBs	4
No of UEs	11
Pathloss model	Log distance
Pathloss Exponent (η)	2
Fading model	Rayleigh
Scheduling	Round Robin
Error model	No error
gNB Bandwidth	50 MHz
Transmit Power Limit	[27, 46] dBm
Noise Power	-96.83 dB
Tx-Rx Antenna counts	1×1
Traffic Model	DL Full buffer

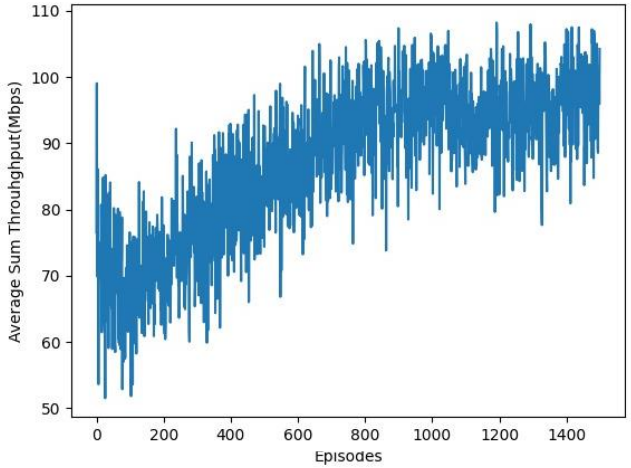
Scenario 3: 7 gNBs and 20 UEs

System Parameter	Value
No of gNBs	7
No of UEs	20
Pathloss model	Log distance
Pathloss Exponent (η)	2
Fading model	Rayleigh
Scheduling	Round Robin
Error model	No error
gNB Bandwidth	50 MHz
Transmit Power Limit	[27, 46] dBm
Noise Power	-96.83 dB
Tx-Rx Antenna counts	1×1
Traffic Model	DL Full buffer

RL leads to > 100% performance improvement

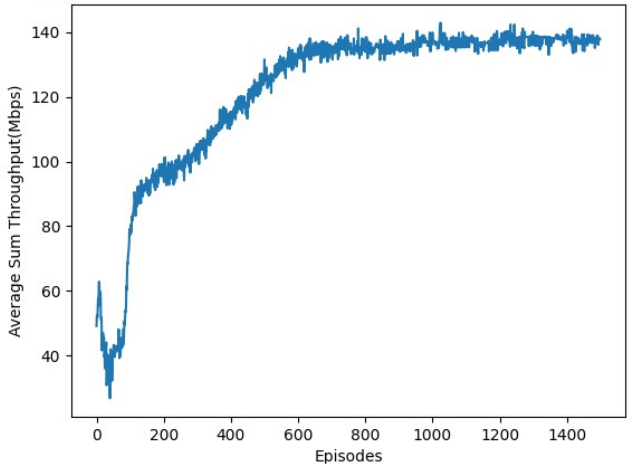


3 gNBs 6 UEs; RL algorithm: Q learning

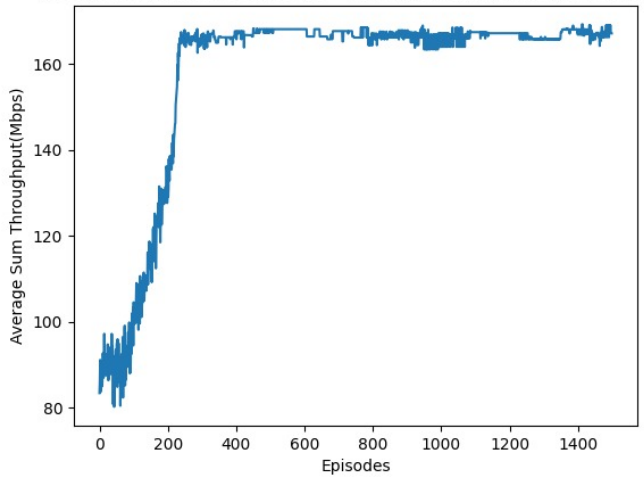


4 gNBs 11 UEs; RL algorithm: Q learning

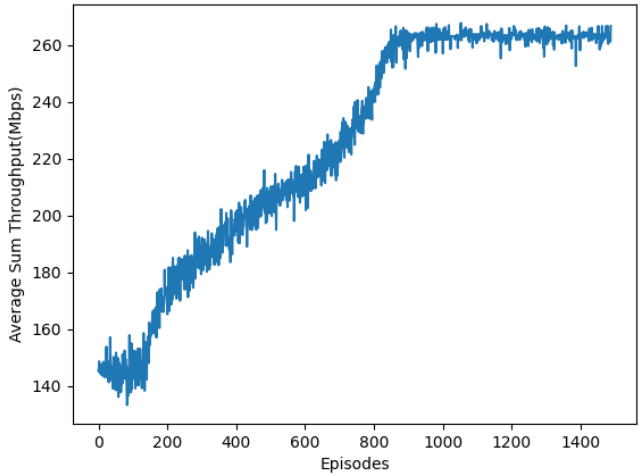
Scenario	Avg. Sum Thput. (Mbps)	
	Without RL	With RL (PPO)
3 gNB 6UEs	55	140
4 gNB 11 UEs	86	165
7 gNBs 20 UEs	181	265



3 gNBs 6 UEs; RL algorithm: PPO



4 gNBs 11 UEs; RL algorithm: PPO

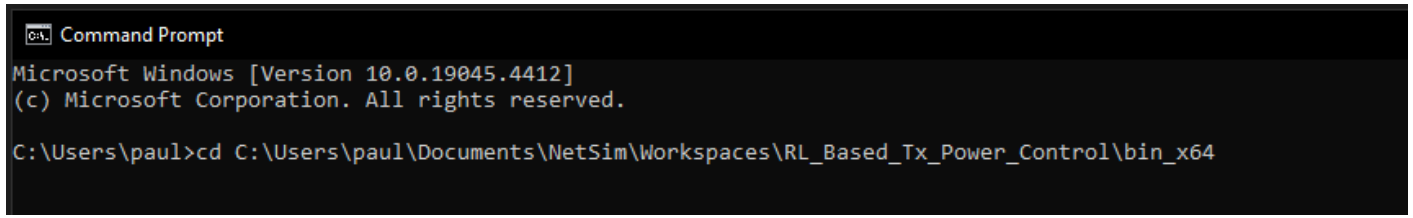


7 gNBs 20 UEs; RL algorithm: PPO

Appendix

How to run the RL simulation?

- Download the project from Github link provided in slide 1.
- Follow the instructions provided in the following link to setup the project in NetSim
  - https://support.tetcos.com/support/solutions/articles/14000128666-downloading-and-setting-up-netsim-file-exchange-projects
- For the RL simulation, we first need to run NetSim using the command line interface(CLI)
- Open the Run menu with Windows Key + R, then type "cmd." Press "Enter" to open Command Prompt
- Note the application path. Application path is the current workspace location of the NetSim that you want to run. The default application path will be something like "***C:\Users\PC\Documents\NetSim\Workspaces\\<Your default workspace>\bin_x64***" for 64-bit.
- Change the directory to the application path using the following command, below is an example command
  >cd \<app path>



How to run the RL simulation ?

- Type the following in the command prompt

FOR /L %i IN (1,1,<NUM_EPISODES>) DO NetSimCore.exe<space>-iopath<space><io path><space>-license<space>5053@<Server IP Address>

or

FOR /L %i IN (1,1,<NUM_EPISODES>) DO NetSimCore.exe<space>-iopath<space><io path><space>-license<space><license path>

  - **<iopath>** is the path to the current scenario which needs to be run, which in this case is “RL_based_Power_Control_Sample”
  - **<Server IP Address>** is the IP address of the system where NetSim license server is running.
  - **<NUM_EPISODES>** is the number of episodes the simulation will run for; default value is 1500, below is an example command
  - **<license file path>** path where the license file is present. This is generally the <NetSim_Installation_Directory>/bin folder

```
C:\Users\paul\Documents\NetSim\Workspaces\RL_Power_model_Nitin\bin_x64>for /L %i (1,1,1500) do NetSimCore.exe -iopath
"C:\Users\paul\Documents\NetSim\Workspaces\RL_Power_model_Nitin\RL_based_Power_Control_3x6" -license 5053@192.168.0.9
```

- Open another command prompt window and change the directory to where the python file and requirements.txt is saved
- To run Tabular Q Learning:
  - Type python <filename>, in this case, filename is “RL_Power_Control” and hit enter to run the python file
  - The python script will ask for the number of episodes as input, enter the default value, 1500 and hit enter to start the simulation

How to run the RL simulation ?

- For PPO and A2C Algorithms
  - Using a Virtual environment:
    - Open the folder containing requirements.txt in command prompt
    - Create Environment: **>python -m venv netsim_power_control**
    - Activate Environment:
      - Windows: **>netsim_power_control\Scripts\activate**
      - Mac/Linux : **>source netsim_power_control/bin/activate**
    - Install dependencies**.** : **>pip install -r requirements.txt**
  - Type **>python <filename>**, in this case, filename is “agents_netsim” and hit enter to run the python file.
  - The default Simulation will run for 1500 episodes with 1000 steps.
  - To deactivate the environment after use: **>deactivate**
  - To **use GPU** for training:
    - Make sure CUDA 11.8 and cuDNN 8.7.0.x are installed to support Pytorch 2.3
    - use **>pip install -r requirements-gpu.txt**

How to run the RL simulation ?

- To change the number of episodes, we need to make two changes. First, we need to change the input we give to the python script. Second, we need to edit the looping command which we use to run NetSim. Edit the **<NUM_EPISODES>** variable in the command
- For Tabular Q Learning:
  - Upon running the simulation, the python script will create two folders, named “plots” and “logs” where it will save the result plots and log files, respectively. These folders will be created in the same working directory as the python script
  - Close the plots after viewing them, to allow them to be saved.
  - In the “logs” folder, all the log files will be saved which can be used for debugging
- For Proximal Policy Optimization (PPO) and Advantage actor-critic (A2C) Algorithms:
  - The agents (A2C or PPO) and activation functions can be changed by changing the “model” variable in the agents_netsim.py file.
  - Upon Running the simulation, a <name>.zip file will be saved which has the trained model and weights saved for future use and a plot for the average rewards per episode.

System requirements and execution guidelines

System Specs

- 16GB RAM, Intel-i5/AMD-Ryzen 5 or above
- Win 10/ Win 11 OS

Guidelines

- To turn on GPU usage, PyTorch can be utilized.
  - CUDA, CuDNN required in PPO/A2C cases
- If the NetSim terminal works very slow or stops in simulation
  - Decrease number of steps per episode
  - Then simulation time in NetSim should be set equal to $T = \left(Steps \times \frac{3}{100} + 0.21\right)\ s$
- A2C: n_steps should be small and be a factor of total number of steps.
- PPO: n_steps should be large and be a factor of total number of steps (generally half).
- Keep the NetSim terminal to be the main screen. This will ensure it gets a higher priority process.
  - If the simulation runs in background and it would slow down. Uncomment the psutil code in agents file to decrease process priority.
- Using a complex network architecture can help in better learning in cases and prevent overfitting.
- Keyboard-Interrupt can be used to stop learning prematurely if required
  - First NetSimCore terminal then the main simulation terminal