QCell: Self-optimization of Softwarized 5G Networks through Deep Q-learning

Bernardo Casasole,* Leonardo Bonati,† Salvatore D'Oro,†
Stefano Basagni,† Antonio Capone,* Tommaso Melodia†
*Politecnico di Milano, Milan, Italy
Email: bernardo.casasole@mail.polimi.it, antonio.capone@polimi.it
†Institute for the Wireless Internet of Things, Northeastern University, Boston, MA, U.S.A.
Email: {bonati.l, s.doro, s.basagni, t.melodia}@northeastern.edu

***Abstract*—With the unprecedented rise in traffic demand and mobile subscribers, real-time fine-grained optimization frameworks are crucial for the future of cellular networks. Indeed, rigid and inflexible infrastructures are incapable of adapting to the massive amounts of data forecast for 5G networks. Network softwarization, i.e., the approach of controlling "everything" via software, endows the network with unprecedented flexibility, allowing it to run optimization and machine learning-based frameworks for flexible adaptation to current network conditions and traffic demand. This work presents QCell, a Deep Q-Network-based optimization framework for softwarized cellular networks. QCell dynamically allocates slicing and scheduling resources to the network base stations adapting to varying interference conditions and traffic patterns. QCell is prototyped on Colosseum, the world's largest network emulator, and tested in a variety of network conditions and scenarios. Our experimental results show that using QCell significantly improves user's throughput (up to 37.6%) and the size of transmission queues (up to 11.9%), decreasing service latency.**

***Index Terms*—Deep Q-Network, Network Slicing, Open RAN.**

I. Introduction

The 5th generation (5G) of cellular networks will leverage softwarization and virtualization principles to provide unprecedented levels of service to mobile subscribers. Telecom operators will be allowed to instantiate many heterogeneous cellular networks on a common physical infrastructure (*network slicing*) and to optimize the use and allocation of network resources via software to improve the performance of the network [1]. Custom algorithms and frameworks can now be programmed as *virtual network functions* to optimize one or multiple network functionalities. Software-based cellular optimization is however no trivial feat, as the network is required to dynamically adapt in real time to multiple unpredictable conditions. For instance, traffic demands might suddenly spike in certain areas due to nearby events, and saturate network resources. Clearly, poorly designed strategies—possibly relying on historic operator data—would be unable to swiftly adapt to the new network conditions and might result in poor service.

To address these issues, we introduce QCell, a Deep Q-Network (DQN)-based framework for scalable and real-time self-optimization of cellular networks. In a nutshell, QCell is a multi-agent framework that dynamically adapts network slicing policies and allocates radio resources in real time, adapting to the current network conditions and traffic demand. Our work makes the following contributions:

- We define QCell, a DQN-based model-free framework for cellular network self-optimization that dynamically: (i) Allocates resources (e.g., bandwidth) to multiple slices of the network, and (ii) determines the optimal scheduling policy for each network slice. QCell is able to adapt in real time to varying network conditions and traffic demand by querying the performance at the Base Stations (BSs), quickly adapting to the heterogeneous and time-varying context of 5G cellular networks. The QCell modular design allows it to either execute as a standalone distributed framework on each network BS, or as part of larger frameworks such as O-RAN [1]. In the latter case, QCell can run as an xApp in the O-RAN near real-time Radio Access Network (RAN) Intelligent Controller, and make control decisions on one or multiple BSs interfaced through the standardized E2 interface [1].
- We build a prototype of QCell on srsRAN (formerly srsLTE [2]), an open-source protocol stack for cellular BSs, User Equipments (UEs) and core network that is compliant with 3rd Generation Partnership Project (3GPP) directives.
- We train and test QCell on Colosseum, arguably the world's largest wireless network emulator with hardware in the loop [3]. Colosseum is an experimental testbed that allows to design, prototype and test wireless solutions at scale in a variety of emulated network scenarios and conditions. Specifically, it allows to recreate virtually any wireless environment (e.g., indoor, outdoor, etc.) and channel effects (e.g., path loss, fading, mobility) through FPGA-based Finite Impulse Response (FIR) filters. Experimental results involving 25 Software-defined Radios (SDRs) over a variety of different network scenarios show that QCell significantly improves the throughput of cellular users up to 37.6%, and decreases the size of downlink transmission queues by up to 11.9%, thus decreasing the service latency.

We believe that our work advances future cellular networking with respect to previous attempts at building network optimization frameworks. Previous solutions in this realm are either tied to traditional optimization techniques, and therefore less adaptable to unpredictable network dynamics, or use machine learning techniques for optimizing network

This work was partially supported by the U.S. National Science Foundation under grants CNS-1923789 and CNS-1925601 and by the U.S. Office of Naval Research under grant N00014-20-1-2132.

performance. Works belonging to the first category include CellOS, an automated framework for cellular network optimization using Lagrangian multipliers and duality theory [4]. The very nature of these techniques makes CellOS slower to react to network changes. Machine learning-based solutions for optimized cellular performance concern specific aspects of the network architecture [5, 6]. For instance, Li et al. devise a Deep Reinforcement Learning (DRL) framework for resource management in network slicing in which actions are defined as radio resource allocations [7]. Although simplifying system design, this choice significantly increases the policy space, making it harder to find optimal solutions. The works by Chinchali et al. [8] and by Bonati et al. [9] present learning-based schedulers that adapt to traffic demand and different reward functions set by the telecom operators. Differently from QCell, these works focus mainly on scheduling optimization, disregarding optimal allocation of network slicing policies.

Overall, QCell takes a DQN approach to self-optimization of softwarized cellular networks by making decisions on slicing and scheduling policies of the BSs. As demonstrated through an extensive experimental campaign on Colosseum, QCell self-adapts to varying network conditions and traffic, significantly improving key performance metrics such as throughput and size of the transmission buffers.

The remainder of this work is organized as follows. Section II provides some background knowledge on DQNs. The QCell architecture is detailed in Section III. Section IV presents our QCell prototype and experimental results. Conclusions are drawn in Section V.

II. Deep Q-Networks: A Primer

A Deep Q-Network (DQN) is a DRL technique used to determine policies π from high-dimensional inputs by leveraging reinforcement learning techniques [10, 11]. In a DQN model, an *agent* interacts with an *environment* through a sequence of *observations* of the environment (formed by a series of *states*), *actions*, and *rewards*. A sequence of states, actions, and rewards that lead to a terminal state of the system is called an *episode*. The agent's purpose is to choose the action that maximizes the cumulative future rewards.

The optimal state-action value function $Q^*(s,a)$, *aka* the *Q-function,* of a policy π, namely, $Q^\pi(s,a)$, measures the expected return—or sum of discounted rewards—that is obtained by taking action a from state s, and then following the policy π thereafter. $Q^*(a,s)$ is defined as the maximum return that is obtainable by taking action a in state s, and then following the optimal policy. This is approximated with a Deep Neural Network (DNN), which produces the cumulative future reward r_t at time t, discounted by a factor $\gamma \in [0,1]$.

$$Q^*(s,a) = \left[r + \gamma \max_{a'} Q^*(s',a') \right], \qquad (1)$$

where s' is the next state, and a' is the action taken from it.

To cope with instability due to the correlation between consecutive episodes, the *experience replay* D_t at time step t needs to be considered [10]. This is a set of agent's experiences $\{e_1, \ldots, e_t\}$ that randomizes over data, where $e_i = (s_i, a_i, r_i, s_{i+1})$ is the set of state, action, reward, and next state. At each time step t of the learning phase, the agent performs an action a_t, which causes the system to transition to state s_t. The experience vector $e_t = (s_t, a_t, r_t, s_{t+1})$ is, then, stored in the experience replay D_t. To reduce the correlation between Q-function value Q and the optimal Q^*, a second Q-Network—namely target Q-Network—is introduced. The structure of the target Q-Network is the same as that of the original Q-Network, while its weights are updated periodically to those of the original Q-Network. Finally, the Q-learning update at iteration t leverages the following loss function:

$$L_t(\theta_t) = \mathbb{E}_{s,a,r,s'} \left[\left(r + \gamma \max_{a'} Q(s',a';\theta_t^-) - Q(s,a;\theta_t) \right)^2 \right], \qquad (2)$$

where θ_t (θ_t^-) are the weights of the original (target) Q-Network.

III. QCell Architecture

The QCell architecture consists of a distributed multi-agent optimization framework. A DQN agent runs on each cellular BS and makes control decisions on its configuration, e.g., scheduling and slicing policies. These decisions are based on the real-time performance of the network, i.e., the network state, signaled through messages exchanged among the agents of the interfering BSs. In this way, agents are able to learn the best action to optimize the performance of their own BS, while keeping the interference to the other BSs as low as possible.

Fig. 1 shows the architecture of the QCell DQN framework. This is formed by three components: (i) The agent; (ii) the BS protocol stack, and (iii) the BS connector.

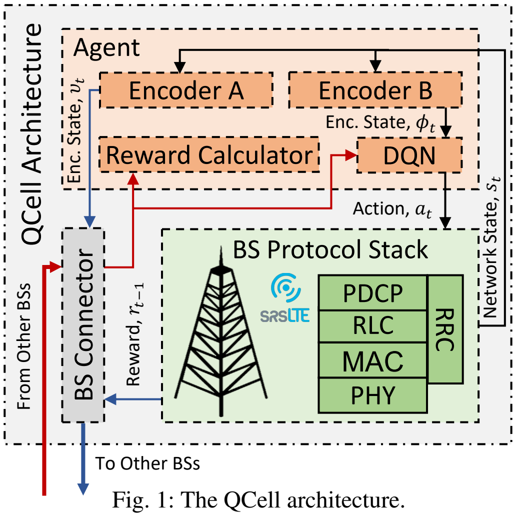

Fig. 1: The QCell architecture.

Each *agent* is equipped with two different encoders (A and B in the figure) used to reduce the dimensionality of the input data, a reward calculator, and a DQN. At each time instant t, the agent reads the network state (s_t), which reflects the performance of the BS on which QCell is deployed, and of the UEs served by it. The feed-forward multi-layer neural network encoders are used to normalize the dimensionality of the input, which depends on the number of users served by the BS. Both encoders A and B are fed with the current network state and retain as output the information relevant to the DQN. The output of encoder A (ν_t) represents the state of the QCell BS (regardless of the state of each user). This is sent to the interfering BSs by the *BS connector*, along with the reward of the action taken at time step $t - 1$. The output of encoder B (ϕ_t) is used to set the input of the DQN to a given dimension, while retaining more than 85% of the original dataset variance through the Principal Component Analysis process [12].

The *reward calculator* receives the metrics sent from the interfering BSs from the BS connector and computes the reward for the current time step t. This is the linear combination of the variation of the metrics of the users served by the BS, and the rewards received in input by the interfering QCell BSs.

The DQN is formed by a 4-layer deep convolutional neural network. It takes as input the output of encoder B and the state signaled by the other QCell BSs (received by the BS connector), and returns as output a set of Q-values, each of which is associated with a different action and network configuration. Among the possible returned actions, the one associated with the highest Q-value (a_t) is implemented at the BS. For the sake of improving the efficiency of the training, the DQN has been decoupled from the encoders, and the three components have been trained separately [11]. The action space of each agent is defined by a finite set of possible network configurations. For each network slice, actions are the combination of a scheduling policy (we considered the round-robin, waterfilling and proportionally fair scheduling policies) and resource allocation of the slice, i.e., the number of Resource Block Groups (RBGs) allotted to the slice [6].

The reward function of each agent is shown in Algorithm 1. This is a function of the average variation of the metrics of the users the BS is serving, and of those sent by the interfering QCell BSs. Specifically, the goal of the QCell agents is to simultaneously optimize the downlink throughput of the users and the size of their downlink transmission buffer, i.e., the buffer containing the data the BS needs to transmit to each user in downlink, which directly reflects on the service latency.

The *BS connector* allows the communication between the BSs of the network. It receives the state encoded by encoder A (ν_t) and the previous-step reward (r_{t-1}), and sends them to the interfering BSs of the network. Similarly, it also receives analogous information from the interfering BSs and forwards it to the *reward calculator* and to the DQN.

Finally, the *BS protocol stack* implements the softwarized cellular BS, including layers such as PHY, MAC, RLC, PDCP, and RRC [1]. This enables the communication with the UEs of the network and can be implemented through open source software solutions, e.g., srsRAN (see Section IV-A). Regardless of the specific implementation, this element provides the network state to encoders A and B of the QCell agent, and the reward obtained from the last time step to the BS connector.

Algorithm 1: QCell agent reward function at time t.

```
Result: reward $r_t$
Initialize: my reward $r_t^m = 0$
for all users $u \in \mathcal{U}$ do
    for all metrics $m$ do
        sign $\leftarrow \frac{m_t - m_{t-1}}{|m_t - m_{t-1}|}$
        if sign $= -1$ then
            sign $\leftarrow -1.2$
        variation $z \leftarrow \frac{|m_t - m_{t-1}|}{m_{t-1}}$
        if $z \leqslant 0.01$ then
            $r_t^m \leftarrow r_t^m +$ sign
        else if $z \leqslant 0.05$ then
            $r_t^m \leftarrow r_t^m + 5 \cdot$ sign
        else if $z \leqslant 0.1$ then
            $r_t^m \leftarrow r_t^m + 10 \cdot$ sign
        else
            $r_t^m \leftarrow r_t^m + 20 \cdot$ sign
$r_t^m \leftarrow \frac{r_t^m}{|\mathcal{U}|}$
send($r_t^m$)
other BSs' rewards $r_{t-1}^{oth} \leftarrow$ receive_rewards()
$r_t \leftarrow 0.8 \cdot r_t^m + 0.1 \cdot$ sum($r_{t-1}^{oth}$)
```

A. QCell Example

We will now give a high-level example of the QCell algorithm workflow. We consider a scenario with 3 interfering BSs, namely a, b, and c (see Fig. 2). This process is also detailed in Algorithm 2.

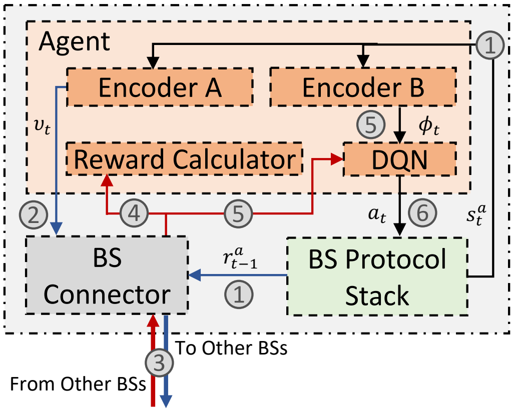

Fig. 2: Example of the QCell algorithm workflow.

As the first step, the two encoders (A and B in the figure) read the network state s_t^a, and the BS connector reads the previous reward r_{t-1}^a (step 1 in Fig. 2). Encoder A encodes the state information of the BS into ν_t (step 2). This is sent to the agents of the interfering QCell BSs from the BS connector, together with the information about the reward of the agent at time $t - 1$, i.e., r_{t-1}^a (step 3, outgoing blue arrow). At the same time, this component receives the signaling messages from the interfering BSs (step 3, incoming red arrow). These messages contain the network state and previous-step reward of the QCell agents of the nearby BSs. In step 4, the received rewards (r^b_{t-1} and r^c_{t-1}) are used to compute the final agent reward according to Algorithm 1. The received states (s^b_t and s^c_t), instead, are fed to the DQN together with the output ϕ_t produced by encoder B (step 5). Finally, in step 6, the agent action at time t, a_t, is computed by the DQN.

Algorithm 2: QCell DQN algorithm.

```
Initialize: replay memory $D = N$;
  $Q \leftarrow$ random weights $\theta$; $\hat{Q} \leftarrow$ weights $\theta^- = \theta$;
  network state $\tilde{s}_t$
action $a_t \leftarrow argmax_a Q(\tilde{s}_t, a; \theta)$
while qcell is running do
    sleep 35 s
    $\tilde{s}_{t-1} \leftarrow \tilde{s}_t$
    get new network state $s_t$
    compute output encoder A $\nu_t \leftarrow \nu(s_t)$
    send $\nu_t$ to other BSs
    receive $\bar{\nu}_t^{oth}$ from other BSs
    compute output encoder B $\phi_t \leftarrow \phi(s_t)$
    $\tilde{s}_t \leftarrow \texttt{concat}\left(\phi_t, \bar{\nu}_t^{oth}\right)$
    $r_t \leftarrow r(\phi_t, \bar{\nu}_t^{oth})$
    $x \leftarrow \texttt{rand}(0, 1)$
    if $x \leq \epsilon$ then
        select random $a_t$
    else
        $a_t \leftarrow argmax_a Q(\tilde{s}_t, a; \theta)$
    store $(\tilde{s}_{t-1}, a_{t-1}, r_{t-1}, \tilde{s}_t)$ in $D$
    sample mini-batch of $(\tilde{s}_t, a_t, r_t, \tilde{s}_{t+1})$ from $D$
    $y \leftarrow r_t + \gamma \cdot max_a \hat{Q}\left(\tilde{s}_{t+1}, a; \theta^-\right)$
    do gradient descent on $\left(y - Q\left(\tilde{s}_t, a_t, \theta\right)\right)^2$ wrt $\theta$
    execute $a_t$
    every $C$ steps: $\hat{Q} \leftarrow Q$
```

IV. Experimental Results

In this section, we describe the network emulator we leveraged to develop our QCell prototype, the DQN training process, and the obtained experimental results.

A. QCell Prototype

We prototyped QCell on Colosseum, the world's largest wireless network emulator [3]. Colosseum was developed by DARPA for the Spectrum Collaboration Challenge as the ideal environment for machine learning development and testing [13]. It is composed of 256 USRP X310 SDRs, half of which are controlled by high-performance, scalable servers—namely Standard Radio Nodes (SRNs)—and used as communications devices. The remaining 128 SDRs, instead, compose the Colosseum Massive Channel Emulator (MCHEM). This leverages FPGA-based FIR filters to emulate heterogeneous environments and conditions of the wireless channel (e.g., path loss, multi-path, node mobility, etc.) through so called radio frequency scenarios.

We implemented 3GPP-compliant BSs and UEs through the SCOPE framework [14], which extends srsRAN with slicing capabilities and additional scheduling policies. Specifically, we leveraged this software to deploy 3 cellular BSs and up to 22 UEs on Colosseum SRNs. For each network BS, we considered 2 network slices and 3 different scheduling policies: Round-robin, waterfilling, and proportionally fair scheduling policies. The downlink center frequencies of the BSs were set to 1000.0 MHz, 1002.5 MHz, and 1005.0 MHz (i.e., partially interfering but not completely overlapped), while the channel bandwidth to 3 MHz, corresponding to 15 physical resource blocks (8 RBGs).

Traffic among BSs and UEs was generated through iPerf3, which allows to measure the network performance through TCP/UDP packet flows. To account for realistic—yet unpredictable—traffic conditions, we varied randomly the traffic rate and the length of the packets between BSs and UEs in $[3, 10]$ Mbps and $[3, 10]$ kB, respectively.

As for the DQN, we selected the behavioral distribution through an ϵ-greedy policy. At every time step, the environment state is defined by the state of every agent, which is determined by the metrics of the UEs served by the BSs. We considered the following metrics collected at each BS: (i) Slice resource ratio, as the fraction of RBGs assigned to the slice the UE belongs to; (ii) granted resource ratio, as the ratio between the resources granted and required by the UEs; (iii) channel quality information, which gives an indication of the quality of the channel, as perceived by the UEs; (iv) downlink throughput; (v) downlink buffer size, as the amount of data the BSs need to transmit to the UEs at any given moment, and (vi) modulation and coding scheme.

The agent actions are composed of the fraction of RBGs to allocate to each network slice, and the scheduling policy to use at the BS. A sample of the action space adopted in our QCell prototype is shown in Table I. We recall that the total number of RBGs used in our prototype is 8.

Finally, the agent reward at time *t* is computed as the linear combination of the variation of the metrics of the UEs served by the current BS, and the metrics sent by the agents of the interfering QCell BSs (see Algorithm 1).

A diagram of the QCell prototype we implemented on Colosseum is shown in Fig. 3. We deployed 3 softwarized BSs and 22 UEs on Colosseum SRNs, and emulated a urban

TABLE I: Sample action space of QCell prototype.

Action	Fraction of RBGs Slice 1	Fraction of RBGs Slice 2	Scheduling Policy
0	1/2	1/2	Round-robin
1	1/2	1/2	Waterfilling
2	1/2	1/2	Proportionally Fair
3	1/4	3/4	Round-robin
4	1/4	3/4	Waterfilling
5	1/4	3/4	Proportionally Fair
6	3/4	1/4	Round-robin
7	3/4	1/4	Waterfilling
8	3/4	1/4	Proportionally Fair

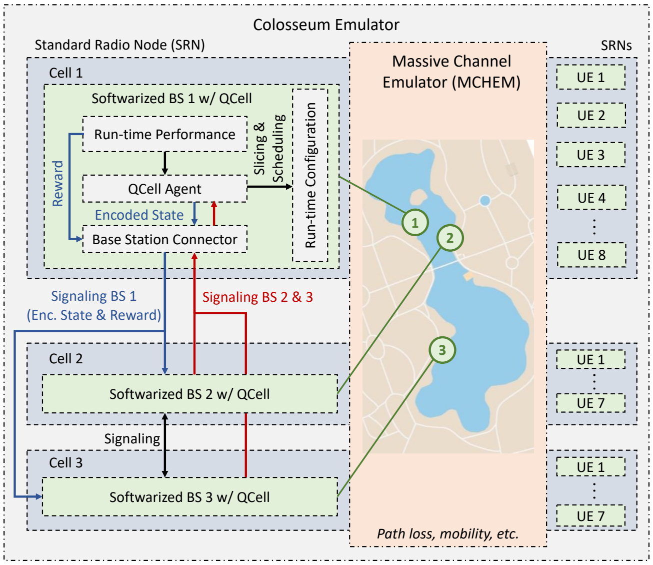

Fig. 3: The QCell prototype implementation on Colosseum.

scenario through MCHEM. Each BS runs a *QCell agent*. The agent reads the *run-time performance* of the BS (e.g., network conditions, traffic demand, and level of service provided to the UEs), which forms the network state. It then encodes this state and signals it, together with the previous reward, to the interfering BSs (which are also running QCell) through the *BS connector*. At the same time, this component receives the signaling messages from the interfering BSs and forwards them to the *QCell agent*. Finally, the agent leverages this information, together with the network state, to compute the optimal action (i.e., slicing and scheduling policies) through the DQN, and re-configures the BS at run time.

B. QCell Training

To let QCell interact dynamically with a diversified environment, in our training we periodically reinitialize the experiments on Colosseum, randomly assigning UEs to the network BSs. QCell periodically reads the up-to-date network performance collected at the BS by srsRAN and modifies slicing and scheduling policies used to serve the UEs, if necessary.

Relevant QCell hyperparameters are shown in Table II. We set the *minibatch size* to 32, which determines the number of training cases over which the Stochastic Gradient Descent (SGD) algorithm is computed. The *replay memory size* (set to 10000) regulates the number of training cases considered by the SGD algorithm, while the *target network update frequency*, measured as the number of network parameter updates, is set to 35. The *discount factor* γ of the Q-learning update, and the *learning rate* of the SGD algorithm are set to 0.9 and 0.01, respectively. Finally, the *initial* and *final exploration* ϵ values, which enable QCell to self-adapt to unseen network scenarios, are set to 1 and 0.05, respectively, and the *final exploration iteration* to 10000.

TABLE II: List of relevant QCell hyperparameters.

Hyperparameter	**Value**
Minibatch size	32
Replay memory size	10000
Target network update frequency	35
Discount factor, γ	0.9
Learning rate	0.01
Initial exploration value for ϵ	1
Final exploration value for ϵ	0.05
Final exploration iteration	10000

Our training consisted of almost 16000 episodes, for a total of 150 hours of training. Fig. 4 shows the loss value evolution over the training episodes. As the training goes on, we notice a decrease in the average loss value after around 1900 episodes. This metric, then, continues to decrease until the end of the training, indicating convergence of QCell learning process.

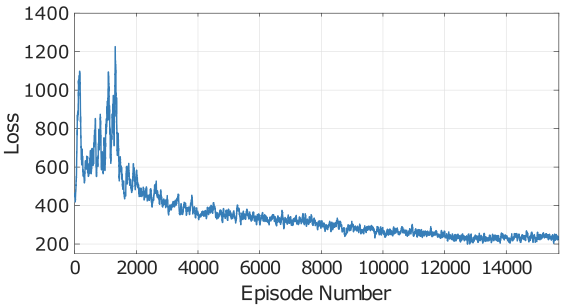

Fig. 4: Loss value vs. number of episodes.

Fig. 5 shows the reward of QCell agents, both instantaneous and average, over the various episodes of the training. We can see that the agent reward keeps increasing, on average, for the whole duration of the training, except for the final 3000 episodes, in which the reward oscillates around similar value intervals. This suggests convergence of the QCell algorithm, which is able to get the average optimal reward for each episode. Moreover, at the end of the training, the reward assumes positive values, demonstrating that QCell can efficiently optimize the performance of the network.

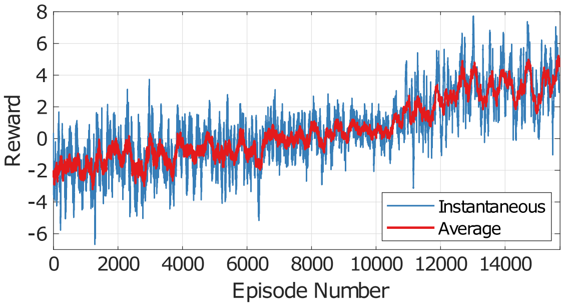

Fig. 5: Reward value vs. number of episodes.

C. QCell Testing

To test QCell, we ran over 17 hours of experiments on the Colosseum testbed (see Section IV-A). We compared the network performance in terms of downlink throughput and buffer queue length with and without QCell. In the case without QCell, we considered both the average of all the static configurations (see Table I), and the best performing static configuration (action 7 in Table I).

Fig. 6 depicts the downlink throughput with and without QCell in the above-mentioned cases. We notice that QCell is able to significantly improve the user throughput compared to both average and best static configurations. This is due to the tight interactions among the BS agents, which allow to dynamically reconfigure the network at run time based on the current network conditions and performance.

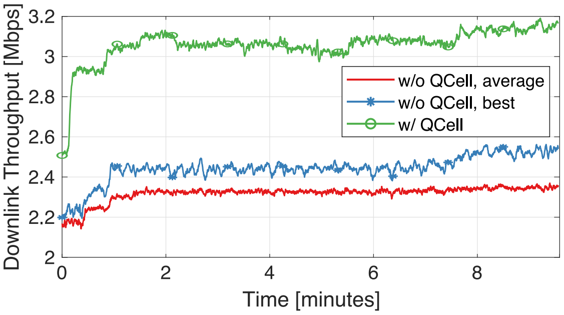

Fig. 6: Downlink throughput w/ and w/o QCell.

Fig. 7 shows the average downlink buffer occupancy with and without QCell. In general, the occupancy of the downlink buffer is reduced when QCell is active, implying a faster service to the UEs. This is due to the superior behavior of QCell, which leverages the UE feedback collected at the BSs to efficiently adapt to the varying network dynamics.

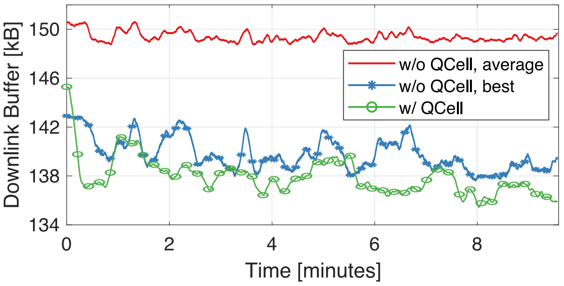

Fig. 7: Downlink buffer occupancy w/ and w/o QCell.

The action probability distribution of QCell agents is depicted in Fig. 8. This metric conveys how often a certain action is selected by the agents after the training phase has been completed. We observe that the actions (reported in Table I) are chosen with similar probabilities by QCell, which enacts the best-performing configuration based on the specific network scenario and traffic demand.

Finally, we analyzed the frequency at which QCell modifies the BS configuration after the training phase has been completed. Results confirmed the overall stability of QCell optimization framework, which varied the configuration of the BSs every 101.45 s, on average (corresponding to a 0.44 coefficient of variation).

V. Conclusions

We present QCell, a DQN-based framework for the real-time self-optimization of softwarized cellular networks. QCell follows a multi-agent approach for dynamic slicing and scheduling allocation to BSs at run time, adapting to current network and traffic conditions. We prototype QCell on Colosseum and test it in a host of different network scenarios and configurations. Our experimental results show that QCell is able to achieve up to 37.6% increase in user's throughput and shortens the transmission queue size by 11.9%, thus decreasing the service latency.

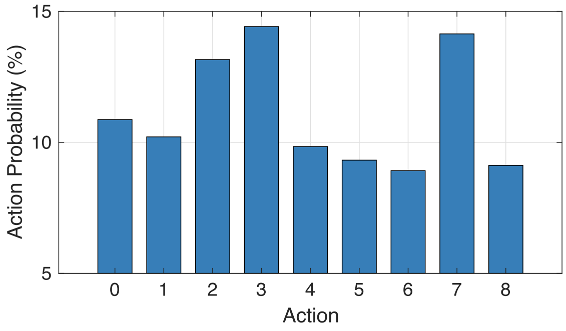

Fig. 8: Agent action probability distribution.

References

[1] L. Bonati, M. Polese, S. D'Oro, S. Basagni, and T. Melodia, "Open, Programmable, and Virtualized 5G Networks: State-of-the-Art and the Road Ahead," *Computer Networks*, vol. 182, pp. 1–28, December 2020.

[2] I. Gomez-Miguelez, A. Garcia-Saavedra, P. Sutton, P. Serrano, C. Cano, and D. Leith, "srsLTE: An Open-source Platform for LTE Evolution and Experimentation," in *Proceedings of ACM WiNTECH*, New York City, NY, USA, October 2016.

[3] Colosseum. https://www.colosseum.net. Accessed April 2021.

[4] L. Bonati, S. D'Oro, L. Bertizzolo, E. Demirors, Z. Guan, S. Basagni, and T. Melodia, "CellOS: Zero-touch Softwarized Open Cellular Networks," *Computer Networks*, vol. 180, pp. 1–13, October 2020.

[5] Z. Xiong, Y. Zhang, D. Niyato, R. Deng, P. Wang, and L.-C. Wang, "Deep Reinforcement Learning for Mobile 5G and Beyond: Fundamentals, Applications, and Challenges," *IEEE Vehicular Technology Magazine*, vol. 14, no. 2, pp. 44–52, June 2019.

[6] S. D'Oro, L. Bonati, F. Restuccia, and T. Melodia, "Coordinated 5G Network Slicing: How Constructive Interference Can Boost Network Throughput," *IEEE/ACM Transactions on Networking*, vol. 29, no. 4, pp. 1881–1894, August 2021.

[7] R. Li, Z. Zhao, Q. Sun, I. Chih-Lin, C. Yang, X. Chen, M. Zhao, and H. Zhang, "Deep Reinforcement Learning for Resource Management in Network Slicing," *IEEE Access*, vol. 6, pp. 1–4, November 2018.

[8] S. Chinchali, P. Hu, T. Chu, M. Sharma, M. Bansal, R. Misra, M. Pavone, and S. Katti, "Cellular Network Traffic Scheduling with Deep Reinforcement Learning," in *Proceedings of the AAAI Conference on Artificial Intelligence*, April 2018.

[9] L. Bonati, S. D'Oro, M. Polese, S. Basagni, and T. Melodia, "Intelligence and Learning in O-RAN for Data-driven NextG Cellular Networks," *IEEE Communications Magazine*, 2021.

[10] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level Control through Deep Reinforcement Learning," *Nature*, vol. 518, no. 7540, pp. 529–533, February 2015.

[11] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep Reinforcement Learning: A Brief Survey," *IEEE Signal Processing Magazine*, vol. 34, no. 6, pp. 26–38, November 2017.

[12] H. Abdi and L. J. Williams, "Principal Component Analysis," *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 2, no. 4, pp. 433–459, 2010.

[13] DARPA Spectrum Collaboration Challenge. https://www.darpa.mil/program/spectrum-collaboration-challenge. Accessed August 2021.

[14] L. Bonati, S. D'Oro, S. Basagni, and T. Melodia, "SCOPE: An Open and Softwarized Prototyping Platform for NextG Systems," in *Proceedings of ACM MobiSys*, Virtual Conference, June 2021.