Real-Time Network Slicing with Uncertain Demand: A Deep Learning Approach

Nguyen Van Huynh, Dinh Thai Hoang, Diep N. Nguyen, and Eryk Dutkiewicz

***Abstract*—Practical and efficient network slicing often faces real-time dynamics of network resources and uncertain customer demands. This work provides an optimal and fast resource slicing solution under such dynamics by leveraging the latest advances in deep learning. Specifically, we first introduce a novel system model which allows the network provider to effectively allocate its combinatorial resources, i.e., spectrum, computing, and storage, to various classes of users. To allocate resources to users while taking into account dynamic demands of users and resources constraints of the network provider, we employ a semi-Markov decision process framework. To obtain the optimal resource allocation policy for the network provider without requiring environment parameters, e.g., uncertain service time and resource demands, a Q-learning algorithm is adopted. Although this algorithm can maximize the revenue of the network provider, its convergence to the optimal policy is particularly slow, especially for problems with large state/action spaces. To overcome this challenge, we propose a novel approach using an advanced deep Q-learning technique, called deep dueling that can achieve the optimal policy at few thousand times faster than that of the conventional Q-learning algorithm. This approach is especially significant for large-scale implementation to deal with dynamic demands of users and real-time resource constraints. Simulation results show that our proposed framework can improve the long-term average return of the network provider up to 40% compared with other current approaches.**

***Index Terms*—Network slicing, resource allocation, uncertain demand, real-time requests, MDP, Q-Learning, deep reinforcement learning, and deep dueling networks.**

I. Introduction

Network slicing is a novel virtualization mechanism that enables multiple logical networks, i.e., slices, with specific service and resource requirements to be created and simultaneously run on top of a single physical infrastructure by using software-defined networking (SDN) and network functions virtualization (NFV). By allowing network providers to provide network infrastructure with flexible demands and requirements as a service, network slicing thereby boosts the operational efficiency and reduces time-to-market for new services. However, there are two main challenges when allocating network resources to slices. First, various emerging services, e.g., IoT services require not only radio resources for communications but also other resources, e.g., computing and storage resources to meet their users' quality-of-service (QoS) requirements. Thus, how to effectively and simultaneously manage these combinatorial resources is a crucial challenge for the network provider. Second, due to the uncertainty of service demand/requirement, e.g., frequency of requests and occupation time, how to optimally and dynamically allocate resources in an online manner to maximize the long-term revenue is another challenge.

To address these problems, several solutions have been introduced recently. In particular, the authors in [2] introduce a two-tier model to obtain the optimal admission policy and to effectively allocate radio resources to the accepted slices. To do so, an extensive searching method is developed to find the globally optimal solution for the network provider. Nevertheless, this searching method is impractical for complex systems which have a large number of resources. To address this problem, the authors in [3] propose a three-step heuristic method to effectively allocate resources to slices. However, this heuristic scheme cannot guarantee the optimality for the network provider. Moreover, these solutions may not be feasible to apply for dynamic network slicing systems with a diversity of resource demands and service time.

To deal with the dynamic of the environment, the authors in [4] introduce a model that can predict future demands of slices, thereby maximizing the system resource utilization for the network provider. To that end, the authors adopt the Holt-Winters theory to predict network slice usage demands through analyzing the traffic usage of slices in the past. However, the heavy-tailed distribution functions and control parameters such as scale factor, least-action trip planning, and potential gain strongly affect the accuracy of this prediction. Moreover, the proposed solution only considers the short-term reward for the network provider, and thus the long-term profit may not be able to obtain. Thus, the authors in [5] and [6] propose reinforcement learning algorithms to address these problems. In particular, reinforcement learning allows the network control to learn from its decisions in a real-time manner to obtain the optimal policy through the trial-and-error learning process [7]. However, this approach converges slowly to the optimal solution, especially for large-scale systems.

All aforementioned works only consider optimizing radio resources, while, as stated in [8], a typical network slice is composed of three main components, i.e., radio, computing, and storage. Thus, considering only radio resources is impractical and may not be able to obtain the optimal slicing solution.

Given the above, in this paper, we develop a dynamic network resource management model based on Semi-Markov decision process (SMDP) framework [9]. This model allows the network provider "slice" various combinatorial resources, e.g., radio, computing, and storage resources, in an online manner. However, jointly considering combinatorial resources together with the uncertainty of demands may lead to an intractable problem. This is because we need to simultaneously deal with a very large multi-dimension state space and real-time dynamic decisions. To tackle it, we propose a novel deep learning approach using the dueling neural network architec-

ture combined with the Q-learning algorithm [10]. Extensive simulations show that the proposed solution can't only deal effectively with the system's dynamic, but also significantly improve the system performance compared with all state of the art network slicing resource allocation approaches.

II. System Model

We consider a general network slicing model with three major parties: (i) the network provider, (ii) tenants, and (ii) end users [4], [5], [8]. The network provider is the owner of the network infrastructure who provides resource slices including radio, computing, and storage, to the tenants. The tenants request and lease resource slices to meet service demands of their subscribers. The end users run their applications on the slices of the subscribed tenants.

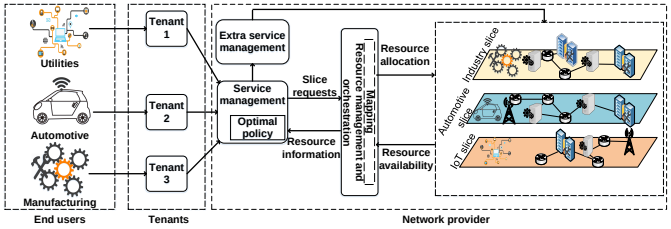


Fig. 1: System model.

Fig. 1 illustrates three tenants corresponding to three common classes of services, i.e., utilities, automotive, and manufacturing. Each class of service has some special features such as its functional, behavioral perspective, and requirements. For automotive class, a vehicle may need an ultra reliable slice for telemetry assisted driving, while security, resilience, and reliability services are of higher priority for slice requests from industry [11]. Hence, when a tenant sends a network slice request to the network provider, the tenant will specify requested resources and additional service requirements, e.g., security and reliability. As a result, depending on service demands, tenants may need to pay different prices for their requests. When a slice request arrives at the system, the service management component will analyze the requirements and make a decision, i.e., accept or reject the request, based on its optimal policy. The resource management and orchestration (RMO) block then allocate resources to the accepted slice request. If the tenant has additional service requirements, the extra service management (ESM) block will deploy extra services on the requested slice at the same time. Once a slice request is accepted, the network provider will receive an immediate reward (the amount of money) paid by the tenant for its granted resources and services.

Assume that there are C classes of slices, denoted by $\mathcal{C} = \{1, \ldots, c, \ldots, C\}$. Each slice from class c requires r_c^{re}, ω_c^{re}, and δ_c^{re} units of radio, computing, and storage resources, respectively. If a slice request from class c is accepted, the provider will receive an immediate reward r_c. The maximum radio, computing, and storage resources of the network provider are denoted by Θ, Ω, and Δ units, respectively. Let n_c denote the number of slices from class c being simultaneously run/served in the system. To ensure that the allocated resources do not exceed the available resources at any time, we have the following resource constraints:

$$\Theta \geq \sum_{c=1}^{C} r_c^{re} n_c, \quad \Omega \geq \sum_{c=1}^{C} \omega_c^{re} n_c, \quad \text{and} \quad \Delta \geq \sum_{c=1}^{C} \delta_c^{re} n_c. \tag{1}$$

III. Problem Formulation

To maximize the long-term reward of the network provider while considering the real-time arrivals/departures of slice requests, we adopt the semi-Markov decision process (SMDP) [9]. An SMDP is represented by a tuple $\langle t_i, \mathcal{S}, \mathcal{A}, \mathcal{L}, r \rangle$ where t_i is an decision epoch, $\mathcal{S}$ is the state space, $\mathcal{A}$ is the action space, $\mathcal{L}$ captures the state transition probabilities and the state sojourn time distribution, and r is the reward function. Unlike discrete Markov decision processes where decisions are made in every time slot, in an SMDP, one only makes decisions when an event occurs. This makes SMDP more effective to capture real-time systems.

A. Decision Epoch

A decision epoch is defined as the interval between two successive decisions of the system. Under our system model, the network provider needs to make decisions upon receiving requests from tenants. Thus, we define a decision epoch as the inter-arrival time between two successive slice requests.

B. State Space

The system state $\mathbf{s}$ of the SMDP at the current decision epoch is defined as the number of slices n_c from a given class $c \in \mathcal{C}$ simultaneously running in the system. Formally, we define $\mathbf{s}$ by $1 \times C$ vector:

$$\mathbf{s} \triangleq [n_1, \ldots, n_c, \ldots n_C], \tag{2}$$

Given the constraints in (1), the state space $\mathcal{S}$ of all possible states $\mathbf{s}$ is expressed as:

$$\mathcal{S} \triangleq \Bigg\{ \mathbf{s} = [n_1, \ldots, n_c, \ldots n_C] : R \geq \sum_{c=1}^{C} r_c^{re} n_c; \; \Omega \geq \sum_{c=1}^{C} \omega_c^{re} n_c; \Delta \geq \sum_{c=1}^{C} \delta_c^{re} n_c \Bigg\}. \tag{3}$$

We then define the *event vector* $\mathbf{e} \triangleq [e_1, \ldots, e_c, \ldots, e_C]$ with $e_c \in \{1, -1, 0\}, \forall c \in \mathcal{C}$ at the current system state $\mathbf{s}$. e_c equals to "1" if a new slice request from class c arrives, e_c equals to "-1" if a slice from class c departs, and e_c equals "0" otherwise (i.e., no slice request arrives nor completes/departs from the system). Then, all the possible events are defined as in (4).

$$\mathcal{E} \triangleq \Big\{ \mathbf{e} : e_c \in \{-1, 0, 1\}; \sum_{c=1}^{C} |e_c| \leq 1 \Big\}. \tag{4}$$

For the case in which there is no event occurring in the system, we define a *trivial event* $\mathbf{e}^* \triangleq [0, \ldots, 0] \in \mathcal{E}$.

C. Action Space

At state $\mathbf{s}$, if a slice request arrives, the network provider can choose either to reject or accept this request to maximize its long-term reward (defined below). Let $a_{\mathbf{s}}$ denote the action to be taken at state $\mathbf{s}$ where $a_{\mathbf{s}} = 1$ if an arrival slice is accepted and $a_{\mathbf{s}} = 0$ otherwise. The state-dependent action space $\mathcal{A}_{\mathbf{s}}$ is defined by:

$$\mathcal{A}_{\mathbf{s}} \triangleq \{a_{\mathbf{s}}\} = \{0, 1\}. \tag{5}$$

D. State Transition Probability

As aforementioned, in this work, we propose reinforcement learning approaches which can obtain the optimal policy for the network provider without requiring information from the environment. However, to lay a theoretical foundation and to evaluate the performance of our proposed solutions, we first assume that slice requests from class c arrive in the system in a Poisson process with mean rate λ_c. The network resources occupation time of slices from class c follows an exponential distribution with mean $1/\mu_c$. Through the uniformization technique [12], we can compute the occurrence rate $z_{\mathbf{s}}$ of the next event expressed as follows:

$$z_{\mathbf{s}} = \sum_{c=1}^{C} (\lambda_c + n_c \mu_c). \tag{6}$$

Given $z = \max_{\mathbf{s} \in \mathcal{S}} z_{\mathbf{s}}$, based on (6), we can calculate the probabilities of events occurring in the next event $\mathbf{e}$ as follows. A slice request from class c will arrive with probability λ_c/z. An accepted slice from class c will depart from the system with probability $n_c \mu_c / z$. Finally, the trivial event occurring in the system with probability $1 - z_{\mathbf{s}}/z$. Based on these probabilities, the state transition probabilities can be derived accordingly.

E. Reward Function

The immediate reward after action $a_{\mathbf{s}}$ is executed at state $\mathbf{s} \in \mathcal{S}$ is defined as follows:

$$r(\mathbf{s}, a_{\mathbf{s}}) = \begin{cases} r_c, & \text{if } e_c = 1,\ a_{\mathbf{s}} = 1, \text{ and } \mathbf{s}' \in \mathcal{S}, \\ 0, & \text{otherwise.} \end{cases} \tag{7}$$

At state $\mathbf{s}$, if an arrival slice is accepted, i.e., $a_{\mathbf{s}} = 1$, the system will move to next state $\mathbf{s}'$ and the network provider receives an immediate reward r_c. In contrast, the immediate reward is equal to 0 if an arrival slice is rejected or there is no slice request arriving at the system. The value of r_c represents the amount of money paid by the tenant based on resources and additional services required.

As the transition and reward function do not change with time, our system's statistical properties are time-invariant, i.e., stationary. Hence, the decision policy π, which is a pure strategy, i.e., accept or reject an arrival request, of the SMDP model can be defined as a time-invariant mapping from the state space to the action space: $\mathcal{S} \to \mathcal{A}_{\mathbf{s}}$. Thus, the long-term average reward starting from a state $\mathbf{s}$ can be formulated as:

$$\mathcal{R}_\pi(\mathbf{s}) = \lim_{K \to \infty} \frac{\mathbb{E}\{\sum_{k=0}^{K} r(\mathbf{s}_k, \pi(\mathbf{s}_k)) | \mathbf{s}_0 = \mathbf{s}\}}{\mathbb{E}\{\sum_{k=0}^{K} \tau_k | \mathbf{s}_0 = \mathbf{s}\}}, \forall \mathbf{s} \in \mathcal{S}, \tag{8}$$

where τ_k is the time interval between the k-th and $(k+1)$-th decision epoch, r is the immediate reward of the system, and $\pi(\mathbf{s})$ is the action corresponding to the policy π at state $\mathbf{s}$. In our SMDP model, the embedded Markov chain is unichain including a single recurrent class and a set of transient states for all pure policies π. Hence, the average reward $\mathcal{R}_\pi(\mathbf{s})$ is well defined and does not depend on the initial state, i.e., $\mathcal{R}_\pi(\mathbf{s}) = \mathcal{R}_\pi, \forall \mathbf{s} \in \mathcal{S}$ [9]. The average reward maximization problem is then written as:

$$\begin{aligned} \max_{\pi} \quad & \mathcal{R}_\pi = \frac{\overline{\mathcal{L}}_\pi r(\mathbf{s}, \pi(\mathbf{s}))}{\overline{\mathcal{L}}_\pi y(\mathbf{s}, \pi(\mathbf{s}))} \\ \text{s.t.} \quad & \sum_{\mathbf{s}' \in \mathcal{S}} \overline{\mathcal{L}}_\pi(\mathbf{s}'|\mathbf{s}) = 1, \forall \mathbf{s} \in \mathcal{S}, \end{aligned} \tag{9}$$

where $\overline{\mathcal{L}}$ is the limiting matrix of the transition probability matrix $\mathcal{L}$. Our objective is to find the optimal admission policy that maximizes the long-term average reward of the network provider, i.e., $\pi^* = \operatorname{argmax}_{\pi} \mathcal{R}_\pi$.

It is worth noting that the problem (9) requires environment information, i.e., arrival and completion rates of slice requests, to construct the transition probability matrix $\mathcal{L}$ [9]. Nevertheless, due to the uncertain demands and the dynamics of slice requests from tenants, these environment parameters might not be available and could be time-varying. To deal with the demand uncertainty and curse-of-dimensionality problems, in the following, we recruit Q-learning and deep dueling algorithms to find the optimal admission policy at the RMO to maximize the long-term average reward.

IV. Q-Learning Algorithm for Dynamic Resource Allocation Under Uncertainty

Q-learning [13] is a reinforcement learning technique which enables the controller to find the optimal policy through interaction processes with the environment without requiring environment information in advance. In particular, the Q-learning algorithm implements a Q-table to store the value for each pair of state and action. Given the current state, the network provider will make an action based on its current policy. After that, the algorithm observes the results, i.e., reward and the next state, and updates the value of the Q-table accordingly. In this way, the Q-learning algorithm can learn from its decisions to converge to the optimal policy after a finite number of iterations [13].

In this paper, we aim to find the optimal policy, i.e., a mapping from the state space to actions, $\pi^* : \mathcal{S} \to \mathcal{A}_{\mathbf{s}}$ for the network provider to maximize its long-term average reward. We first denote $\mathcal{V}^\pi(\mathbf{s}) : \mathcal{S} \to \mathbb{R}$ as the expected value function obtained by policy π from each state $\mathbf{s} \in \mathcal{S}$.

$$\begin{aligned} \mathcal{V}^\pi(\mathbf{s}) &= \mathbb{E}_\pi \Big[\sum_{t=0}^{\infty} \gamma^t r_t(\mathbf{s}_t, a_{\mathbf{s}_t}) | \mathbf{s}_0 = \mathbf{s} \Big] \\ &= \mathbb{E}_\pi \Big[r_t(\mathbf{s}_t, a_{\mathbf{s}_t}) + \gamma \mathcal{V}^\pi(\mathbf{s}_{t+1}) | \mathbf{s}_0 = \mathbf{s} \Big], \end{aligned} \tag{10}$$

where $0 \leq \gamma < 1$ is the discount factor that determines the importance of long-term reward [13]. In particular, if γ is close to 0, the RMO will prefer to select actions to maximize its

short-term reward. In contrast, if γ approaches 1, the RMO will make actions to maximize its long-term reward. $r_t(\mathbf{s}_t, a_{\mathbf{s}_t})$ is the immediate reward achieved by taking action $a_{\mathbf{s}_t}$ at state $\mathbf{s}_t$. Given a state $\mathbf{s}$, policy $\pi(\mathbf{s})$ is obtained by taking action $a_{\mathbf{s}}$ such that the value function is maximized [13]. At each state $\mathbf{s}$, an optimal action is determined based on the optimal value function as in (11):

$$\mathcal{V}^*(\mathbf{s}) = \max_{a_{\mathbf{s}}} \Big\{ \mathbb{E}_\pi [r_t(\mathbf{s}_t, a_{\mathbf{s}_t}) + \gamma \mathcal{V}^\pi(\mathbf{s}_{t+1})] \Big\}, \quad \forall \mathbf{s} \in \mathcal{S}. \tag{11}$$

For all state-action pairs $(\mathbf{s}, a_{\mathbf{s}})$, the optimal Q-functions are denoted by:

$$\mathcal{Q}^*(\mathbf{s}, a_{\mathbf{s}}) \triangleq r_t(\mathbf{s}_t, a_{\mathbf{s}_t}) + \gamma \mathbb{E}_\pi [\mathcal{V}^\pi(\mathbf{s}_{t+1})], \quad \forall \mathbf{s} \in \mathcal{S}. \tag{12}$$

Then, $\mathcal{V}^*(\mathbf{s})$ is expressed as in (13):

$$\mathcal{V}^*(\mathbf{s}) = \max_{a_{\mathbf{s}}} \{ \mathcal{Q}^*(\mathbf{s}, a_{\mathbf{s}}) \}. \tag{13}$$

By making samples iteratively, the problem is reduced to determining $\mathcal{Q}^*(\mathbf{s}, a_{\mathbf{s}})$ for all $(\mathbf{s}, a_{\mathbf{s}})$. Intuitively, to find the difference between the current Q-value and the predicted Q-value, the Q-function is updated as follows:

$$\mathcal{Q}_t(\mathbf{s}_t, a_{\mathbf{s}_t}) = \mathcal{Q}_t(\mathbf{s}_t, a_{\mathbf{s}_t}) + \alpha_t \Big[r_t(\mathbf{s}_t, a_{\mathbf{s}_t}) + \gamma \max_{a_{\mathbf{s}_{t+1}}} \mathcal{Q}_t(\mathbf{s}_{t+1}, a_{\mathbf{s}_{t+1}}) - \mathcal{Q}_t(\mathbf{s}_t, a_{\mathbf{s}_t}) \Big], \tag{14}$$

where α_t is the learning rate that determines the impact of new information to the existing value. Moreover, the learning rate α_t is deterministic, nonnegative, and satisfies (15) [13] to ensure that the Q-learning algorithm converges to the optimal solution.

$$\alpha_t \in [0, 1), \sum_{t=1}^{\infty} \alpha_t = \infty, \text{ and } \sum_{t=1}^{\infty} (\alpha_t)^2 < \infty. \tag{15}$$

Based on (14), the RMO can employ the Q-learning to obtain the optimal policy. Specifically, the algorithm first initializes the table entry $\mathcal{Q}(\mathbf{s}, a_{\mathbf{s}})$ arbitrarily, e.g., to zero for each state-action pair $(\mathbf{s}, a_{\mathbf{s}})$. Given the current state $(\mathbf{s}_t$, the algorithm chooses action $a_{\mathbf{s}}$ and observes results after performing this action. In practice, to select action $a_{\mathbf{s}_{t+1}}$, one can use ϵ-greedy algorithm [7]. Specifically, this method introduces a parameter ϵ which guides the controller to choose a random action with probability ϵ or select an action that maximizes the $\mathcal{Q}(\mathbf{s}, a_{\mathbf{s}})$ with probability $1-\epsilon$. In this way, the algorithm can explore the whole state space. The algorithm then determines the next state and reward after performing the chosen action and update the table entry for $\mathcal{Q}(\mathbf{s}_t, a_{\mathbf{s}_t})$ based on (14). The algorithm will be terminated when the number of iterations is reached or all the Q-values converge. The output of the algorithm is the optimal policy determining an action to be performed at a given state such that $\mathcal{Q}^*(\mathbf{s}, a_{\mathbf{s}})$ is maximized, i.e., $\pi^*(\mathbf{s}) = \arg\max_{a_{\mathbf{s}}} \mathcal{Q}^*(\mathbf{s}, a_{\mathbf{s}})$. Under (15), it was proved in [13] that the Q-learning algorithm will converge to the optimal solution with probability one.

Several research works in the literature introduce the application of the Q-learning algorithm to address the network slicing problem, e.g., [5]. However, the Q-learning algorithm often takes a long time to converge to the optimal solution when the state space or action space is large. For the practical combinatorial resource allocation problem considered in this work, the state space can be as large as tens of thousands. This makes Q-learning practically inapplicable, especially for real-time resource slicing. Therefore, we develop the efficient algorithm to overcome this shortcoming by leveraging the deep Q-learning and novel dueling architecture.

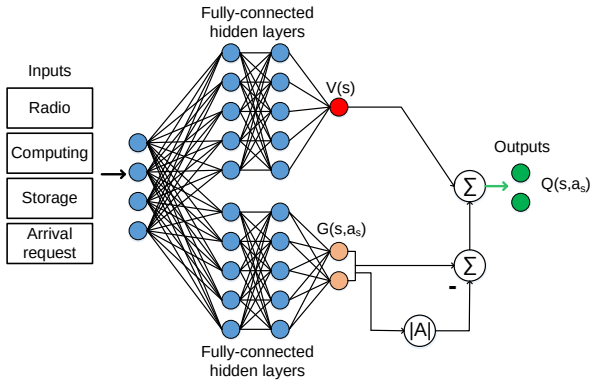


Fig. 2: Deep dueling model.

V. Fast and Optimal Resources Slicing with Deep Dueling Network

We propose deep dueling algorithm, which was originally developed by Google DeepMind in 2016 [10], to address the slow-convergence problem of the Q-learning algorithm. The key idea making the deep dueling superior to conventional approaches is its novel neural network architecture. In this algorithm, instead of estimating the action-value function (Q-function), as in deep Q-learning [15] and deep double Q-learning [16] algorithms, the values of states and advantages of actions are separately and simultaneously estimated by two sequences, i.e., two streams, of fully connected layers. The values and advantages are then combined at the output layer as shown in Fig. 2. This idea is from the fact that in many states it is unnecessary to estimate the value of corresponding actions as the choice of these actions has no repercussion on what happens [10]. In this way, the deep dueling algorithm can achieve more robust estimations of the state value, thereby significantly improving its convergence rate and stability.

The details of the deep dueling algorithm are provided in Algorithm 1. Specifically, the training phase consists of multiple episodes. In each episode, the RMO performs an action and learns from its observations corresponding to the taken action. Hence, the RMO needs to tradeoff between the exploration and exploitation processes over the state space. Given the current state, the algorithm will choose an action based on the ϵ-greedy algorithm. The algorithm will start with a fairly randomized policy and later slowly move to a deterministic policy. This means that, at the first episode, ϵ is set at a large value, e.g., 0.9, and gradually decayed to a small value, e.g., 0.1. After that, the RMO will perform the selected action and observe results, i.e., next state and reward. This transition is then stored in the replay memory for the training process at later episodes. The learning process is then

performed based on random samples from the memory pool, i.e., *experience replay mechanism*. By doing so, the previous experiences are exploited more efficiently as the algorithm can learn them many times. Additionally, by using the experience mechanism, the data is more like independent and identically distributed, thereby removing the correlations between observations. After that, the random samples of transitions from the replay memory will be fed into the neural network. In particular, for each state, we formulate four features, i.e., radio, computing, storage, and arrival event, as the input of the deep neural network. In this way, the training process is more efficient as all aspects of states are taken into account.

Recall that given a policy π, the values of the Q-function $\mathcal{Q}^{\pi}(\mathbf{s}, a_{\mathbf{s}})$ and state $\mathbf{s}$ are expressed as in (16).

$$\begin{aligned} \mathcal{Q}^{\pi}(\mathbf{s}, a_{\mathbf{s}}) &= \mathbb{E}\big[\mathcal{R}_t | \mathbf{s}_t = \mathbf{s}, a_{\mathbf{s}_t} = a_{\mathbf{s}}, \pi\big], \text{and} \\ \mathcal{V}_{\pi}(\mathbf{s}) &= \mathbb{E}_{a_{\mathbf{s}} \sim \pi(\mathbf{s})}\big[\mathcal{Q}^{\pi}(\mathbf{s}, a_{\mathbf{s}})\big]. \end{aligned} \tag{16}$$

The advantage function of actions can be computed as:

$$\mathcal{G}^{\pi}(\mathbf{s}, a_{\mathbf{s}}) = \mathcal{Q}^{\pi}(\mathbf{s}, a_{\mathbf{s}}) - \mathcal{V}^{\pi}(\mathbf{s}, a_{\mathbf{s}}). \tag{17}$$

Specifically, the value function $\mathcal{V}$ corresponds to how "good it is in a particular state $\mathbf{s}$" [10]. The state-action pair calculates the value of performing action $a_{\mathbf{s}}$ in state $\mathbf{s}$. The advantage function separates the state value from the Q-function to measure the importance of each action.

To estimate values of $\mathcal{V}$ and $\mathcal{G}$ functions, we use a dueling neural network in which one stream of fully-connected layers corresponds to $\mathcal{V}(\mathbf{s}; \beta)$ and the other stream is used to obtain an $|\mathcal{A}|$-dimensional vector $\mathcal{G}(\mathbf{s}, a_{\mathbf{s}}; \alpha)$ with α and β are the parameters of fully-connected layers. These two streams are then combined to obtain the Q-function by (18).

$$\mathcal{Q}(\mathbf{s}, a_{\mathbf{s}}; \alpha, \beta) = \mathcal{V}(\mathbf{s}; \beta) + \mathcal{G}(\mathbf{s}, a_{\mathbf{s}}; \alpha). \tag{18}$$

However, $\mathcal{Q}(\mathbf{s}, a_{\mathbf{s}}; \alpha, \beta)$ is only used to estimate the value of the Q-function. Moreover, given $\mathcal{Q}$, we cannot obtain $\mathcal{V}$ and $\mathcal{G}$ uniquely. Therefore, (18) is unidentifiable resulting in poor performance. To address this issue, we combine the outputs of the two streams as follows:

$$\mathcal{Q}(\mathbf{s}, a_{\mathbf{s}}; \alpha, \beta) = \mathcal{V}(\mathbf{s}; \beta) + \big(\mathcal{G}(\mathbf{s}, a_{\mathbf{s}}; \alpha) - \max_{a_{\mathbf{s}} \in \mathcal{A}} \mathcal{G}(\mathbf{s}, a_{\mathbf{s}}; \alpha)\big). \tag{19}$$

In this way, the advantage function estimator has zero advantage when choosing an action. Intuitively, given $a^*_{\mathbf{s}} = \arg\max_{a_{\mathbf{s}} \in \mathcal{A}} \mathcal{Q}(\mathbf{s}, a_{\mathbf{s}}; \alpha, \beta) = \arg\max_{a_{\mathbf{s}} \in \mathcal{A}} \mathcal{G}(\mathbf{s}, a_{\mathbf{s}}; \alpha)$, we have $\mathcal{Q}(\mathbf{s}, a^*_{\mathbf{s}}; \alpha, \beta) = \mathcal{V}(\mathbf{s}; \beta)$. (19) can be transformed into (20) by replacing the max operator with an average.

$$\mathcal{Q}(\mathbf{s}, a_{\mathbf{s}}; \alpha, \beta) = \mathcal{V}(\mathbf{s}; \beta) + \big(\mathcal{G}(\mathbf{s}, a_{\mathbf{s}}; \alpha) - \frac{1}{|\mathcal{A}|} \sum_{a_{\mathbf{s}}} \mathcal{G}(\mathbf{s}, a_{\mathbf{s}}; \alpha)\big). \tag{20}$$

Based on (20), the algorithm then updates the neural network by minimizing the lost functions (line 12) [15] by using the *Stochastic Gradient Descent* algorithm that is the engine of most deep learning algorithms. The parameters of the target network $\hat{\mathcal{Q}}$ are only updated with the Q-network parameters every N steps and are remained fixed between individual updates. In this way, the correlations between the target and estimated Q-values are significantly reduced, thereby stabilizing the algorithm. It is important to note that $\mathcal{V}(\mathbf{s}; \beta)$ and $\mathcal{G}(\mathbf{s}, a_{\mathbf{s}}; \alpha)$ are estimated automatically without any extra supervision or modifications in the algorithm.

Algorithm 1 Deep Dueling Network Based Resources Slicing Algorithm

1: Initialize replay memory with capacity D.
2: Initialize the primary network $\mathcal{Q}$ including two fully-connected layers with random weights α and β.
3: Initialize the target network $\hat{\mathcal{Q}}$ as a copy of the primary Q-network with weights $\hat{\alpha} = \alpha$ and $\hat{\beta} = \beta$.
4: **for** *episode=1 to T* **do**
5: Based on the ϵ-greedy algorithm, with probability ϵ select a random action $a_{\mathbf{s}_t}$, otherwise select $a_{\mathbf{s}_t} = \arg\max \mathcal{Q}^*(\mathbf{s}_t, a_{\mathbf{s}_t}; \alpha, \beta)$
6: Perform action $a_{\mathbf{s}_t}$ and observe reward r_t and next state $\mathbf{s}_{t+1}$
7: Store transition $(\mathbf{s}_t, a_{\mathbf{s}_t}, r_t, \mathbf{s}_{t+1})$ in the replay memory
8: Sample random minibatch of transitions $(\mathbf{s}_j, a_{\mathbf{s}_j}, r_j, \mathbf{s}_{j+1})$ from the replay memory
9: Combine the value function and advantage functions as follows:

$$\begin{aligned} \mathcal{Q}(\mathbf{s}_j, a_{\mathbf{s}_j}; \alpha, \beta) =& \mathcal{V}(\mathbf{s}_j; \beta) + \big(\mathcal{G}(\mathbf{s}_j, a_{\mathbf{s}_j}; \alpha) \\ & - \frac{1}{|\mathcal{A}|} \sum_{a_{\mathbf{s}_j}} \mathcal{G}(\mathbf{s}_j, a_{\mathbf{s}_j}; \alpha)\big). \end{aligned} \tag{21}$$

10: $y_j = r_j + \gamma \max_{a_{\mathbf{s}_{j+1}}} \hat{\mathcal{Q}}(s_{j+1}, a_{\mathbf{s}_{j+1}}; \hat{\alpha}, \hat{\beta})$
11: Perform a gradient descent step on $(y_j - \mathcal{Q}(\mathbf{s}_j, a_{\mathbf{s}_j}; \alpha, \beta))^2$
12: After N episodes reset $\hat{\mathcal{Q}} = \mathcal{Q}$
13: **end for**

VI. Performance Evaluation

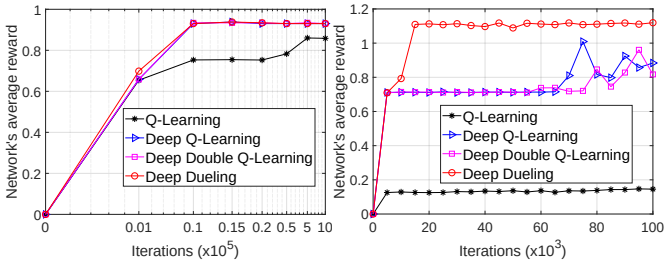
A. Parameter Setting

We consider three common classes of slices, i.e., utilities (class-1), automotive (class-2), and manufacturing (class-3). Unless otherwise stated, the arrival rates μ_c of requests from class-1, class-2, and class-3 are set at 12, 8, and 10 requests/hour, respectively. The completion rates λ_c of requests from class-1, class-2, and class-3 are set at 3 requests/hour. The immediate reward r_c for each accepted request from class-1, class-2, and class-3 are 1, 2, and 4, respectively. These parameters will be varied to evaluate the impacts of the immediate reward on the decisions of the RMO. Each slice request requires 1 GB of storage resources, 2 CPUs for computing, and 100 Mbps of radio resources [14]. For the Q-learning algorithm, we set the discount factor at 0.9 [13]. To evaluate the performance of the proposed deep dueling network, we will compare its performance with other deep reinforcement learning algorithms, i.e., deep Q-learning [15] and deep double Q-learning [16]. The architecture of the deep neural network greatly affects the convergence of the algorithm. Intuitively, the complexity of the algorithm will increase when the number of hidden layers increases. However, when the number of hidden

layers is small, the algorithm may not converge to the optimal policy. Similarly, the algorithm will need more time to estimate the Q-function when the size of hidden layers and mini-batch size are big. In our experiment, we use two fully-connected hidden layers together with input and output layers. The size of the hidden layers and the mini-batch size are set at 64. Both the Q-learning algorithm and the deep reinforcement learning algorithms use ϵ-greedy algorithm with the initial value of ϵ is 1, and its final value is 0.1 [13]. The maximum size of the experience replay buffer is 10,000, and the target Q-network is updated every 1,000 iterations [15].

B. Simulation Results

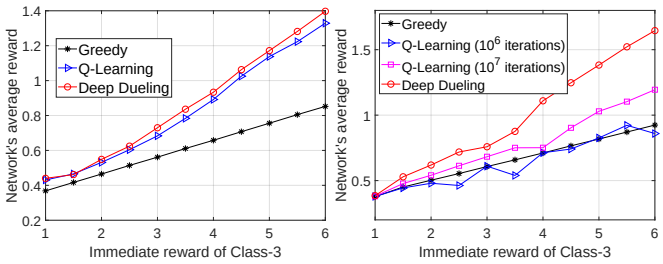
1) Convergence of Deep Reinforcement Learning Approaches: First, we show the learning process through the convergence of proposed deep reinforcement learning approaches, i.e., deep Q-learning, deep double Q-learning, and deep dueling, in different scenarios. As shown in Fig. 3(a), when the maximum radio, computing, storage resources are 400 Mbps, 8 CPUs, and 4 GB, respectively (*small-scale system*), the convergence rates of the three deep Q-learning algorithms are considerably higher than that of the Q-learning algorithm. Specifically, while the deep reinforcement learning approaches converge to the optimal value within 10,000 iterations, the Q-learning need more than 10^6 iterations to obtain the optimal policy. This is stemmed from the fact that in the system under consideration, the state space is dimensional and the system dynamically changes over time. By implementing the neural network with fully-connected layers, the deep reinforcement algorithms can efficiently reduce the curse of dimensionality, thereby improving the convergence rate. We then evaluate



(a) (b)

Fig. 3: The convergence of reinforcement learning algorithms in (a) the *small-scale system* and (b) the *large-scale system*.

the performance of the proposed algorithms in a *large-scale system*. We increase the radio, storage, computing resources to 2 Gbps, 20 GB, and 40 CPUs, respectively. The arrive rates of classes are increased by 4 times, i.e., $\mu_1 = 48$, $\mu_2 = 32$, and $\mu_3 = 40$ requests/hour, while the completion rates are equal to 2 requests/hour for all classes. We then observe the convergence rate of the proposed deep reinforcement algorithms as shown in Fig. 3(b). As the system becomes more complicated with a large state space, the performance of proposed deep dueling outperforms all other deep reinforcement learning techniques. In particular, the deep dueling algorithm only needs less than 20,000 iterations to converge to the optimal policy. Significantly, after 20,000 iterations, the average reward obtained by the proposed deep dueling is approximately 1.5 times greater than those of deep Q-learning and deep double Q-learning, and 6.5 times greater than that of the conventional Q-learning algorithm. The reason is that by decoupling the neural network into two streams, the deep dueling algorithm can significantly reduce the overestimation of the optimizer.



(a) (b)

Fig. 4: The average reward of the system when the immediate reward of class 3 is varied with (a) the *small-scale system* and (b) the *large-scale system*.

2) Average Reward and Network Performance: Next, we compare the performance of the proposed solution, i.e., deep dueling algorithm, with other methods, i.e., Q-learning [5] and greedy algorithms [17], [18], in terms of average reward and the number of requests running in the system.

For the *small-scale system*, Fig. 4(a) shows the average reward of the system obtained by three algorithms as the reward of slices in class-3 increases from 1 to 6 units. As observed, with the increasing for the reward of slices from class-3, the average reward of the system is increased. However, the average reward obtained by the reinforcement learning algorithms, i.e., deep dueling and Q-learning, is significantly higher than that of the greedy algorithm. It is worth noting that the achieved reward of the Q-learning algorithm is not as good as the reward obtained by the deep dueling algorithm even with small-scale scenarios. This is due to the fact that the Q-learning algorithm has a slow convergence rate due to the curse of high dimensionality problem. This observation is more pronounced when we increase the size of the system in the next simulation.

We then observe the performance of the proposed solutions, i.e., the Q-learning and the deep dueling network, in the *large-scale system*. Fig. 4(b) shows that the average reward obtained by the deep dueling algorithm is much higher than those of the greedy and Q-learning algorithms. This is because of the slow convergence of the Q-learning algorithm to optimality. Specifically, after 10^6 iterations, the performance of the Q-learning algorithm is just as the same as that of the greedy algorithm. The performance of the Q-learning algorithm can be improved after 10^7 iterations, but it is still way inferior to that of the deep dueling algorithm. These results verify that the Q-learning algorithm, despite of its optimality, requires much longer time to converge, compared with our proposed deep dueling algorithm, and thus the Q-learning algorithm may not appropriate to implement in practical large-scale network slicing systems.

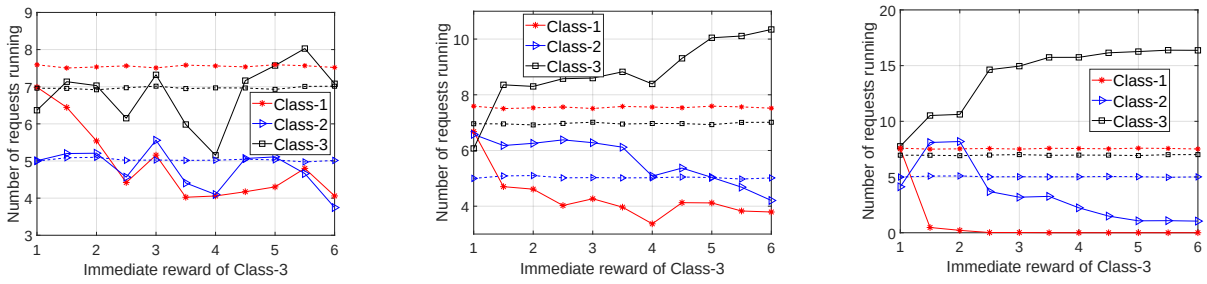


Fig. 5: Number of request running in the system obtained by (a) the Q-learning algorithm (after 10^6 iterations), (b) the Q-learning algorithm (after 10^7 iterations), and (c) deep dueling algorithm (after 20,000 iterations) when the immediate reward of class-3 is varied. The dash lines are results of the greedy algorithm.

Fig. 5 shows that the deep dueling and Q-learning algorithms reserve resources for slices which have high immediate rewards. However, the deep dueling algorithm achieves better performance compared with the Q-learning algorithm thanks to the novel dueling network. For example, when the immediate reward of slices from class-3 is 6, the number of requests running in the systems is about 16 requests and 11 requests for the deep dueling and the Q-learning algorithms, respectively.

In summary, the proposed deep dueling algorithm can achieve an outstanding performance in terms of long-term average reward and the convergence rate compared with other reinforcement learning algorithms, e.g., the Q-learning, deep Q-learning and deep double Q-learning. Importantly, the proposed deep dueling algorithm provides an effective tool for the network provider to deal with the practical large-scale problem under uncertainty of users' demands.

VII. Conclusion

In this paper, we develop a dynamic network resource management framework which allows the network provider to jointly slice multiple combinatorial resources to different requests in a real-time manner. To maximize the long-term average revenue under the uncertainty of slice service demands, we employ an advanced deep learning architecture, called deep dueling. Extensive simulations show that the proposed framework yields up to 40% higher long-term average revenue with few thousand times faster, compared with state of the art network slicing approaches.

References

[1] A. Manzalini, C. Buyukkoc, P. Chemouil, S. Kuklinski, F. Callegati, A. Galis, M. -P. Odini, C. -L. I, J. Huang, M. Bursell, N. Crespi, E. Healy, and S. Sharrock, "Towards 5g software-defined ecosystems," *IEEE SDN White Paper*, 2016.

[2] M. Jiang, M. Condoluci, and T. Mahmoodi, "Network slicing management & prioritization in 5G mobile systems," *22th European Wireless Conference*, Oulu, Finland, Finland, May 2016.

[3] H. M. Soliman, and A. Leon-Garcia, "QoS-aware frequency-space network slicing and admission control for virtual wireless networks," *IEEE GLOBECOM*, Washington, DC, USA, Dec. 2016.

[4] V. Sciancalepore, K. Samdanis, X. Costa-Perez, D. Bega, M. Gramaglia, and A. Banchs, "Mobile traffic forecasting for maximizing 5G network slicing resource utilization," *IEEE INFOCOM*, Atlanta, GA, USA, May 2017.

[5] D. Bega, M. Gramaglia, A. Banchs, V. Sciancalepore, K. Samdanis, and X. Costa-Perez, "Optimising 5G infrastructure markets: The business of network slicing," *IEEE INFOCOM*, Atlanta, GA, USA, May 2017.

[6] A. Aijaz, "Radio resource slicing in a radio access network," *U.S. Patent Application 15/441,564*, filed November 2, 2017.

[7] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. Cambridge, MA, USA: MIT Press, 1998.

[8] X. Foukas, G. Patounas, A. Elmokashfi, and M. K. Marina, "Network slicing in 5g: Survey and challenges," *IEEE Communications Magazine*, vol. 55, no. 5, May 2017, pp. 94-100.

[9] M. Puterman, *Markov Decision Processes: Discrete Stochastic Dynamic Programming*, Hoboken, NJ: Wiley, 1994.

[10] Z. Wang, T. Schaul, M. Hessel, H. V. Hasselt, M. Lanctot, and N. D. Freitas, "Dueling network architectures for deep reinforcement learning," [Online]. Available: arXiv:1511.06581.

[11] 5G Network Slicing for Vertical Industries, *Global mobile Suppliers Association*. Available Online: https://www.huawei.com/minisite/5g/img/5g-network-slicing-for-vertical-industries-en.pdf

[12] R. G. Gallager, *Discrete stochastic processes*, Kluwer Academic Publishers, London, 1995.

[13] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Mach. Learn.*, vol. 8, no. 34, pp. 279292, 1992.

[14] D. Sattar and A. Matrawy, "Optimal Slice Allocation in 5G Core Networks," [Online]. Available: arXiv:1802.04655.

[15] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, et al., "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, Feb. 2015, pp. 529-533.

[16] H. V. Hasselt, A. Guez, and D. Silver, "Deep Reinforcement Learning with Double Q-Learning," in *AAAI*, 2016.

[17] A. Aijaz, "Hap SliceR: A Radio Resource Slicing Framework for 5G Networks With Haptic Communications," *IEEE Systems Journal*, no. 99, Jan. 2017, pp. 1-12.

[18] B. Han, J. Lianghai, and H. D. Schotten, "Slice as an Evolutionary Service: Genetic Optimization for Inter-Slice Resource Management in 5G Networks," *IEEE Access*, Jun. 2018.