Chapter 9
Deep Reinforcement Learning for Mobile Edge Computing Systems

Ming Tang and Vincent W. S. Wong

9.1 Introduction

Recently, humans use mobile devices to accomplish many computational intensive tasks, such as artificial intelligence, distributed data analysis, virtual reality, and augmented reality. Despite the fact that mobile devices have become increasingly powerful, they may not be capable of processing all their tasks locally and meeting the delay requirements of the tasks. Mobile edge computing (MEC) [1], also known as multi-access edge computing [2] and fog computing [3], is emerging as a promising architecture. In MEC systems, edge nodes equipped with processing and storage resources are deployed close to the mobile devices. Thus, mobile devices can offload their computational intensive tasks to the edge nodes for processing. When compared with cloud computing systems [4], MEC systems can provide mobile devices with a faster response and hence a low task latency. Mao et al. in [1], Qiu et al. in [5], and Ranaweera et al. in [6] provided comprehensive surveys in the area of MEC.

The environment in MEC systems may involve time-varying and complex system dynamics, such as time-varying task arrivals, device mobility, wireless channel variation, and the interaction among mobile devices. Meanwhile, the operators of the MEC systems, mobile devices, and edge nodes may not be aware of these system dynamics a priori. Such unknown system dynamics impose challenges on addressing the deployment, management, and scheduling problems in MEC systems. On the other hand, conventional network optimization approaches (e.g., online optimization [7], game-theoretic approach [8]) always rely on the modeling

M. Tang · V. W. S. Wong (✉)
Department of Electrical and Computer Engineering, The University of British Columbia, Vancouver, BC, Canada
e-mail: mingt@ece.ubc.ca; vincentw@ece.ubc.ca

L. Cai et al. (eds.), *Broadband Communications, Computing, and Control for Ubiquitous Intelligence*, Wireless Networks,
https://doi.org/10.1007/978-3-030-98064-1_9

of the environment in MEC systems. When the modeling of the environment does not match the practical systems, these conventional approaches may fail to provide a satisfactory performance. In addition, it may be challenging for these approaches to address time-varying environment in MEC systems.

Deep reinforcement learning (DRL) [9] is a promising technique to address the unknown and complex system dynamics in MEC systems [10]. With DRL, an agent (e.g., a mobile device, an edge node, or a network operator in MEC systems) can learn to make decisions (e.g., in terms of deployment, management, and scheduling) by interacting with the environment. During the learning process, the agent continuously gathers its experience from the interaction with the environment and gradually learns the decision-making policy that optimizes its objective. In comparison to conventional reinforcement learning (RL) approaches [11], DRL employs deep learning techniques to tackle the curse of dimensionality issue. Due to the strong capability of deep learning for analyzing and abstracting data, DRL is capable of handling complex systems with large state spaces [9].

In this chapter, we aim at providing an overview on how DRL techniques can benefit the MEC systems. In the rest of this chapter, we first present an overview of DRL in Sect. 9.2. In Sect. 9.3, we demonstrate the application of DRL techniques in MEC systems with a case study, which focuses on the task offloading problem. Finally, in Sect. 9.4, we outline several challenges and future research directions. For notation, let $\mathbb{Z}_{++}$ denote the set of positive integers.

9.2 Overview of Deep Reinforcement Learning

In this section, we first introduce the general DRL problem formulation. Then, we present the main idea for obtaining the optimal policy using DRL algorithms. Finally, we summarize some existing DRL algorithms.

9.2.1 DRL Problem Formulation

In general, a DRL problem can be formulated as a discrete time stochastic control process [9]. In this problem, an agent interacts with the environment. Suppose there are a set of time slots $\mathcal{T}$. At the beginning of time slot $t \in \mathcal{T}$, given the state of the environment $s(t)$, the agent gathers an observation $o(t)$. Based on the observation, the agent chooses an action $a(t)$. After that, the agent obtains a reward $r(t)$. The environment then transits to the next state $s(t+1)$, and the agent gathers the next observation $o(t+1)$. Through such an interaction with the environment, the agent aims at learning an optimal policy that maximizes the expected long-term reward.

For the sake of simplicity, let us consider a fully observable system. In this system, the agent can observe the actual state of the environment, i.e., $o(t) = s(t)$ for all $t \in \mathcal{T}$. Meanwhile, suppose the transition of the state follows Markovian stochastic control processes. The scenario with partially observable system and non-Markovian environment can be found in [9, Section 10]. Let $\mathcal{S}$ and $\mathcal{A}$ denote the state and action spaces. We denote π as the policy of the agent. Note that there are two types of policy: deterministic policy $\pi(s) : \mathcal{S} \to \mathcal{A}$ and stochastic policy $\pi(s, a) : \mathcal{S} \times \mathcal{A} \to [0, 1]$, where $\pi(s, a)$ is the probability that action a is selected given current state s. Given any state $s \in \mathcal{S}$, let $V^{\pi}(s) : \mathcal{S} \to \mathbb{R}$, also called the value function, denote the expected long-term reward in state s under policy π. For example, under the scenario with an infinite time horizon $\mathcal{T} = \{1, 2, \ldots\}$, the value function $V^{\pi}(s) \triangleq \mathbb{E}[\sum_{i=0}^{\infty} \gamma^i r(t+i) \mid s(t) = s, \pi]$. That is, given any $s(t) = s$, the value function $V^{\pi}(s)$ is equal to the expected cumulative future discounted reward, where $\gamma \in (0, 1]$ is the discount factor. The value function under various scenarios can be found in [11, Section 3.5]. The objective of the agent is to find a policy π^* that maximizes $V^{\pi}(s)$. On the other hand, as an alternative to the value function, Q-value function $Q^{\pi}(s, a) : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ is sometimes considered, which is the expected long-term reward of choosing action $a \in \mathcal{A}$ in state s under policy π.

9.2.2 Determine the Optimal Policy with Deep Learning

To find the optimal policy, in conventional RL approaches, the agent estimates one or multiple of the following components during the learning process [9, Section 3.2]:

(a) Value function $V^{\pi}(s)$ or Q-value function $Q^{\pi}(s, a)$
(b) Policy $\pi(s)$ or $\pi(s, a)$
(c) The model of the environment, i.e., the transition function (from current state to the next state) and the reward function (from state and action to the reward)

For the RL approaches requiring the estimation of components (a) and (b), they are called model-free algorithms. When component (c) is considered, the associated algorithm is called a model-based algorithm. Furthermore, the algorithms that estimate (a) and (b) are called value-based and policy-based algorithms, respectively. With the estimation of those components (a), (b), or (c), the agent can obtain the optimal policy accordingly. For example, in a model-free value-based Q-learning algorithm, the agent estimates the Q-value function and exploits a policy of choosing the action with the maximum Q-value given each state. Due to the definition of Q-value function, such a policy can maximize the expected long-term reward.

On the other hand, most real-world problems are very complex, e.g., the state and action space can be high-dimensional and continuous. Thus, estimating the value function, policy, and model can be challenging and requires a huge amount of computational resource and memory. To address this issue, in DRL algorithms, deep learning techniques are used for estimating the value function, policy, and model. In particular, deep learning essentially relies on neural networks to estimate a mapping from some input to some output, i.e., $f : \mathcal{X} \to \mathcal{Y}$. The mapping is characterized by the parameters of the neural network, denoted by θ, and can be represented by

$y = f(x \mid \theta)$. DRL employs neural networks to estimate certain mappings, such as value function, policy, and model. During the learning process in DRL, based on the gathered experience (i.e., state, action, next state, reward), the agent gradually updates the parameters of the neural network (i.e., θ) using deep learning techniques in order to make the neural network achieve an accurate approximation of the actual mapping (e.g., the actual value function, policy, and model). Since neural networks are capable of addressing complicated mappings with large input and output spaces, they provide DRL the capability of addressing complex real-world environment.

9.2.3 Existing DRL Algorithms

For value-based DRL algorithms, deep Q-learning (DQL) [12] aims at estimating the Q-value function using neural networks. In addition to DQL, double deep Q-network (DQN) [13] can handle the issue of overestimation in DQL. Meanwhile, dueling DQN [14] offers a more accurate estimation of the Q-value function by separately learning the value resulting from the state and action. Instead of estimating the expected reward in the value function as in DQL, distributional DQN [15] aims at estimating the distribution of the cumulative reward under each state. Such a distribution characterizes the randomness of the reward in the system.

Policy gradient algorithms belong to policy-based DRL algorithms and are commonly used. These algorithms directly learn an optimal policy that maximizes the expected long-term reward using gradient ascent. In the area of DRL, deep deterministic policy gradient (DDPG) [16] learns the representation of a deterministic policy, where this approach is applicable for continuous action space. Distributed distributional DDPG [17] is a variant of DDPG that can be run in a distributional fashion. Asynchronous advantage actor-critic (A3C) [18] exploits the approximation of both policy and value function using neural networks. Built upon A3C, actor-critic with experience replay [19] exploits the experience replay, which can decrease the data correlation and increase the sample efficiency. Soft actor-critic [20] encourages policy exploration by maximizing the entropy of the policy and the expected long-term reward simultaneously. Twin delayed deep deterministic [21] exploits double DQN to address the overestimation issue in actor-critic methods. In addition, trust region optimization [22] introduces the idea of trust region to bound the change in policy to guarantee monotonic reward improvement. As a variant of trust region optimization, proximal policy optimization [23] introduces a penalty term in the objective function to alleviate the change in policy, and it is easy for implementation. In comparison to those value-based DRL methods, policy gradient algorithms are capable of handling continuous action space and stochastic policy.

For model-based DRL algorithms, the agent either knows the model of the environment a priori or uses the gathered experience to predict the model. The model will be used for planning, i.e., taking the model as an input, the agent finds the optimal policy for the interaction with the environment. For discrete action space, lookahead search can be used by building a decision tree and exploring the potential trajectories in the tree [9, Section 6.1]. Monte Carlo tree search [24] is a typical family of approaches to lookahead search. Such methods have been incorporated with deep learning for addressing real-world complex problem, e.g., the game of Go [25]. For continuous action space, trajectory optimization can be used. For a function that is differentiable, the agent can optimize the policy along trajectories using gradient ascent. Plaat et al. in [26] and Franccois et al. in [9, Section 6] provided comprehensive surveys for model-based DRL algorithms. In comparison to model-free DRL algorithms, model-based DRL algorithms are more sample efficient. That is, with the learned model, model-based algorithms can converge with fewer samples (or gathered experience) than model-free algorithms.

9.3 Case Study: Deep Q-Learning for Task Offloading in MEC

Due to the huge potential of addressing complex and dynamics systems, DRL has been applied in various problems in MEC systems, such as task offloading, mobility management, and edge maintenance. Luong et al. in [27, Section IV] and Wang et al. in [10, Section VIII] surveyed the existing works using DRL in MEC systems.

In this section, we present a case study on the task offloading problem in MEC systems, which is based on our earlier work [28]. In particular, in MEC systems, edge nodes may have limited amount of processing capacity. The tasks offloaded by different mobile devices at an edge node will share the processing capacity of the edge node. Thus, from the perspective of a mobile device, if many other mobile devices choose to offload to a particular edge node, then this mobile device may choose not to offload to the same edge node in order to reduce the task delay. We refer to the number of concurrent mobile devices offloading to an edge node as the *load level* of the edge node. Such a load level depends on the task arrivals and offloading decisions of all mobile devices. Thus, it is time varying and unknown to the mobile devices a priori. This makes it challenging for a mobile device to make the task offloading decision (i.e., whether to offload or not, and if yes, which edge node to choose). On the other hand, although the challenge can be mitigated by letting a centralized entity make a decision for the mobile device, such a centralized decision making may require global information of the system and incur a high signaling overhead.

To address the unknown load level dynamics, we propose a distributed DQL-based task offloading algorithm. The proposed algorithm is a model-free value-based approach that enables each mobile device to make its offloading decision without knowing the task models and offloading decisions of other mobile devices.

In the following subsections, we first present the system model and task offloading problem, respectively. Then, we propose the DQL-based algorithm for MEC systems. Finally, we evaluate the performance of our proposed algorithm.

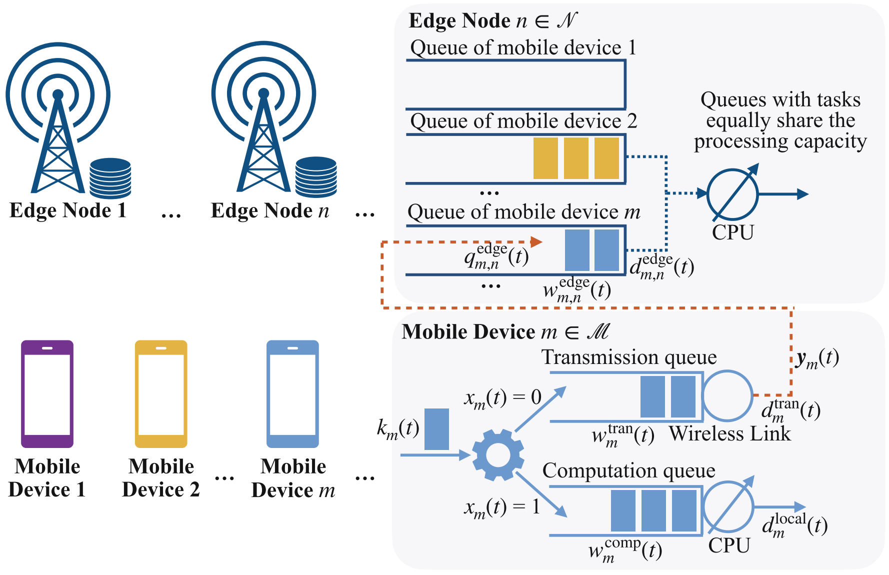

Fig. 9.1 An illustration of an MEC system with edge nodes and mobile devices

9.3.1 System Model

We consider an MEC system that has a set of mobile devices $\mathcal{M} = \{1, 2, \ldots, M\}$ and a set of edge nodes $\mathcal{N} = \{1, 2, \ldots, N\}$. We consider a time-slotted system with a set of time slots $\mathcal{T} = \{1, 2, \ldots, T\}$. Let Δ (in seconds) denote the duration of each time slot. An illustration of an MEC system is shown in Fig. 9.1.

In the following, we first present the task model and task offloading decisions. Then, we discuss the local processing model and edge node offloading model.

9.3.1.1 Task Model

At the beginning of time slot $t \in \mathcal{T}$, mobile device $m \in \mathcal{M}$ either may have a new computational task to be processed or does not have any new task arrival. As in some existing works (e.g., [29]), we assume that the mobile device has a new task arrival at time slot t with a certain probability. If mobile device m has a new task, then we refer to this task using an index $k_m(t) \in \mathbb{Z}_{++}$. For presentation simplicity, if mobile device m does not have any new task, we set $k_m(t) = 0$.

Let $\lambda_m(t)$ (in bits) denote the size of task $k_m(t)$. For presentation simplicity, we set $\lambda_m(t) = 0$ if $k_m(t) = 0$. We set the size of task $k_m(t)$ to be from a discrete set $\Lambda \triangleq \{\lambda_1, \lambda_2, \ldots, \lambda_{|\Lambda|}\}$, where $|\Lambda|$ denotes the cardinality of set Λ. We consider a setting where any task of mobile device m has a deadline τ_m (in time slots). That is, task $k_m(t)$ will be dropped if it has not been completely processed within τ_m

time slots. Moreover, as in some existing works (e.g., [30, 31]), we assume that the number of CPU cycles required for processing a task is proportional to the size of the task. Some examples satisfying this assumption include file compression, video segment encoding and decoding, and object detection in video streaming. Let ρ_m (in CPU cycles per bit) denote the processing density of any task of mobile device m. Thus, $\rho_m \lambda_m(t)$ CPU cycles are required for processing a task of mobile device m with size $\lambda_m(t)$.

9.3.1.2 Task Offloading Decision

If mobile device $m \in \mathcal{M}$ has a new task at the beginning of time slot $t \in \mathcal{T}$, then it needs to decide *whether to process task* $k_m(t)$ *locally or offload it to an edge node*. We use binary variable $x_m(t) \in \{0, 1\}$ to denote this decision. We set $x_m(t) = 1$ if the task is processed locally and set $x_m(t) = 0$ if the task is offloaded to an edge node.

If mobile device m decides to offload task $k_m(t)$ to an edge node, then it needs to decide *which edge node to choose*. We use binary variable $y_{m,n}(t) \in \{0, 1\}$ to denote whether mobile device m chooses edge node $n \in \mathcal{N}$ to offload task $k_m(t)$ or not. We set $y_{m,n}(t) = 1$ if mobile device m chooses edge node n and set $y_{m,n}(t) = 0$ otherwise. Note that exactly one edge node can be chosen to offload task $k_m(t)$, i.e.,

$$\sum_{n \in \mathcal{N}} y_{m,n}(t) = \mathbb{1}(x_m(t) = 0), \;\; m \in \mathcal{M}, t \in \mathcal{T}, \tag{9.1}$$

where indicator function $\mathbb{1}(x_m(t) = 0) = 1$ if $x_m(t) = 0$, and $\mathbb{1}(x_m(t) = 0) = 0$ otherwise. Let vector $\boldsymbol{y}_m(t) = (y_{m,n}(t), n \in \mathcal{N})$.

9.3.1.3 Local Processing Model

If mobile device m decides to process task $k_m(t)$ locally (i.e., $x_m(t) = 1$), then it will place task $k_m(t)$ in the computation queue for local processing. The tasks in the computation queue are processed in a first-in first-out (FIFO) manner. We use f_m^{device} (in CPU cycles) to denote the processing capacity of mobile device $m \in \mathcal{M}$. We assume that f_m^{device} does not change across time slots. Meanwhile, we assume that if a task has been processed in a time slot, then the processing of the next task in the computation queue will start at the beginning of the next time slot.

For mobile device $m \in \mathcal{M}$, at the beginning of time slot $t \in \mathcal{T}$, let $w_m^{\text{comp}}(t)$ (in time slots) denote the remaining number of time slots until all the tasks placed in the computation queue before time slot t have been either processed or dropped. Note that if task $k_m(t)$ is to be placed in the computation queue, then $w_m^{\text{comp}}(t)$ corresponds to the number of time slots that task $k_m(t)$ will wait in the computation queue for processing, i.e., the queuing delay of task $k_m(t)$ at the computation queue.

Here, we use notation w for the short form of "wait." The expression of $w_m^{\text{comp}}(t)$ is derived as follows. For mobile device $m \in \mathcal{M}$, $w_m^{\text{comp}}(t) = 0$ for $t = 1$, and

$$w_m^{\text{comp}}(t) = \min\left\{\left[w_m^{\text{comp}}(t-1) + \left\lceil \frac{\lambda_m(t-1)x_m(t-1)}{f^{\text{device}}\Delta/\rho_m} \right\rceil - 1\right]^+, \tau_m - 1\right\}, \quad t \in \mathcal{T} \setminus \{1\}, \tag{9.2}$$

where $\lceil\cdot\rceil$ is the ceiling function, and operator $[z]^+ = \max\{z, 0\}$. Specifically, for $t \in \mathcal{T} \setminus \{1\}$, if task $k_m(t-1)$ was placed in the computation queue, then given $w_m^{\text{comp}}(t-1)$, the first term in the min operator corresponds to the remaining number of time slots until task $k_m(t-1)$ has been processed since the beginning of time slot t. The second term corresponds to the remaining number of time slots until task $k_m(t-1)$ has been dropped. On the other hand, if either $\lambda_m(t-1) = 0$ or $x_m(t-1) = 0$, then we have $w_m^{\text{comp}}(t) = [w_m^{\text{comp}}(t-1) - 1]^+$. The second term in the min operator is canceled out, because $w_m^{\text{comp}}(t) < \tau_m$ holds for $t \in \mathcal{T}$. In this case, no task was placed in the computation queue in time slot $t-1$. Thus, from time slot $t-1$ to t, the associated remaining number of time slots is decremented by one.

Suppose task $k_m(t)$ is processed locally (i.e., $x_m(t) = 1$). Let $d_m^{\text{local}}(t)$ denote the corresponding delay for local processing, i.e., the number of time slots required to process task $k_m(t)$. The expression of $d_m^{\text{local}}(t)$ is derived as follows:

$$d_m^{\text{local}}(t) = \left\lceil \frac{\lambda_m(t)}{f^{\text{device}}\Delta/\rho_m} \right\rceil, m \in \mathcal{M}, t \in \{t' \mid t' \in \mathcal{T}, k_m(t') \neq 0, x_m(t') = 1\}. \tag{9.3}$$

Let $\text{Delay}_m(t)$ denote the delay of task $k_m(t)$. Recall that the tasks in the computation queue are processed in an FIFO manner. If task $k_m(t)$ is processed locally, then the delay of task $k_m(t)$ can be derived as follows:

$$\text{Delay}_m(t) = \min\left\{w_m^{\text{comp}}(t) + d_m^{\text{local}}(t), \tau_m\right\}, \quad m \in \mathcal{M}, t \in \{t' \mid t' \in \mathcal{T}, k_m(t') \neq 0, x_m(t') = 1\}. \tag{9.4}$$

Specifically, the delay of task $k_m(t)$ is equal to its queuing delay at the computation queue, i.e., $w_m^{\text{comp}}(t)$, plus the processing delay, i.e., $d_m^{\text{local}}(t)$. Note that if the delay exceeds τ_m time slots, then the task will be dropped immediately. Without loss of generality, if task $k_m(t)$ has been dropped, then we set the value of $\text{Delay}_m(t)$ to be τ_m.[1]

[1] This setting is for the simplicity of mathematical presentation. For any task $k_m(t)$ that has been dropped, the value of $\text{Delay}_m(t)$ will not be taken into account in our proposed algorithm according

9.3.1.4 Edge Node Offloading Model

If mobile device $m \in \mathcal{M}$ decides to offload task $k_m(t)$ to an edge node (i.e., $x_m(t) = 0$), then it places task $k_m(t)$ in the transmission queue. Tasks in the transmission queue are sent in an FIFO manner. After task $k_m(t)$ has been sent to the chosen edge node based on decision $\boldsymbol{y}_m(t)$, it will be processed by the edge node.

Transmission to an Edge Node The tasks in the transmission queue will be forwarded to the chosen edge node using a wireless network interface. We assume that the mobile devices transmit using orthogonal channels. Thus, there is no interference between mobile devices. Let $|h_{m,n}|^2$ denote the channel gain from mobile device $m \in \mathcal{M}$ to edge node $n \in \mathcal{N}$. Let P denote the transmission power of the mobile device. Thus, the transmission rate from mobile device m to edge node n can be computed as follows:

$$r_{m,n}^{\text{tran}} = W \log_2\left(1 + \frac{|h_{m,n}|^2 P}{\sigma^2}\right), \quad m \in \mathcal{M}, n \in \mathcal{N}, \tag{9.5}$$

where W denotes the bandwidth allocated to the channel, and σ^2 denotes the received noise power at the edge node. We assume that the transmission rate $r_{m,n}^{\text{tran}}$ does not change across time slots. If a task has been sent in a time slot, then the next task in the transmission queue will be sent at the beginning of the next time slot.

For mobile device $m \in \mathcal{M}$, at the beginning of time slot $t \in \mathcal{T}$, let $w_m^{\text{tran}}(t)$ (in time slots) denote the remaining number of time slots until all the tasks placed in the transmission queue before time slot t have been either processed or dropped. If task $k_m(t)$ is to be offloaded to an edge node, then $w_m^{\text{tran}}(t)$ also corresponds to the number of time slots that task $k_m(t)$ will wait in the transmission queue, i.e., the queuing delay of task $k_m(t)$ at the transmission queue. The expression of $w_m^{\text{tran}}(t)$ is given as follows. For mobile device $m \in \mathcal{M}$, $w_m^{\text{tran}}(t) = 0$ for $t = 1$, and

$$w_m^{\text{tran}}(t) = \min\left\{\left[w_m^{\text{tran}}(t-1) + \left\lceil \sum_{n \in \mathcal{N}} \frac{\lambda_m(t-1) y_{m,n}(t-1)}{r_{m,n}^{\text{tran}} \Delta} \right\rceil - 1\right]^+, \tau_m - 1\right\},$$
$$t \in \mathcal{T} \setminus \{1\}. \tag{9.6}$$

The interpretation of $w_m^{\text{tran}}(t)$ is similar to that of $w_m^{\text{comp}}(t)$. If there is no task arrival in time slot $t-1$ (i.e., $\lambda_m(t-1) = 0$), then $w_m^{\text{tran}}(t) = [w_m^{\text{tran}}(t-1)-1]^+$. Meanwhile, if task $k_m(t-1)$ was placed in the computation queue (i.e., $x_m(t-1) = 1$), then $y_{m,n}(t-1) = 0$ for all $n \in \mathcal{N}$ according to (9.1), and hence $w_m^{\text{tran}}(t) = [w_m^{\text{tran}}(t-1) - 1]^+$.

to Sects. 9.3.2 and 9.3.3. Meanwhile, the delay of a dropped task is not accounted when we evaluate the average delay of the tasks with our proposed algorithm and benchmark methods in Sect. 9.3.4.

If task $k_m(t)$ is offloaded to an edge node, then the number of time slots required to send task $k_m(t)$ to the edge node, denoted by $d_m^{\text{tran}}(t)$, is computed as follows:

$$d_m^{\text{tran}}(t) = \left\lceil \sum_{n \in \mathcal{N}} \frac{\lambda_m(t) y_{m,n}(t)}{r_{m,n}^{\text{tran}} \Delta} \right\rceil, m \in \mathcal{M}, t \in \{t' \mid t' \in \mathcal{T}, k_m(t') \neq 0, x_m(t') = 0\}. \tag{9.7}$$

Processing at an Edge Node At any edge node $n \in \mathcal{N}$, the tasks from different mobile devices are placed in different queues. We refer to the queue that stores the tasks of mobile device $m \in \mathcal{M}$ as the queue of mobile device m. We assume that when a task has been sent to an edge node in a time slot, the edge node places the task into the associated queue at the beginning of the next time slot.

At edge node $n \in \mathcal{N}$, let $q_{m,n}^{\text{edge}}(t)$ (in bits) denote the occupancy of the queue of mobile device $m \in \mathcal{N}$ at the end of time slot $t \in \mathcal{T}$. Let $k_{m,n}^{\text{edge}}(t)$ denote the index of the task placed in the queue of mobile device m at the beginning of time slot t. Specifically, if task $k_m(t')$ is offloaded to edge node n in time slot $t-1$ (i.e., $t' + w_m^{\text{tran}}(t') + d_m^{\text{tran}}(t') - 1 = t - 1$ and $y_{m,n}(t') = 1$), then $k_{m,n}^{\text{edge}}(t) = k_m(t')$. If there does not exist such a task, then we set $k_{m,n}^{\text{edge}}(t) = 0$. We denote $\lambda_{m,n}^{\text{edge}}(t)$ (in bits) as the size of task $k_{m,n}^{\text{edge}}(t)$. If $k_{m,n}^{\text{edge}}(t) = 0$, then we set $\lambda_{m,n}^{\text{edge}}(t) = 0$. In time slot t, we refer to the queue of mobile device m as an *active queue* if either the queue is non-empty or there exists a new task arrival at the queue. Thus, the set of active queues at edge node n in time slot t, denoted by $\mathcal{B}_n(t)$, is defined as

$$\mathcal{B}_n(t) = \left\{ m \;\middle|\; q_{m,n}^{\text{edge}}(t-1) > 0 \text{ or } \lambda_{m,n}^{\text{edge}}(t) > 0, m \in \mathcal{M} \right\}. \tag{9.8}$$

Let $B_n(t)$ denote the number of active queues, i.e., $B_n(t) = |\mathcal{B}_n(t)|$.

Let f_n^{edge} (in CPU cycles per second) denote the processing capacity of edge node n. Within each time slot $t \in \mathcal{T}$, the active queues in set $\mathcal{B}_n(t)$ equally share the processing capacity of edge node $n \in \mathcal{N}$. This is the generalized processor sharing (GPS) model with equal processing capacity sharing [32]. Note that the number of active queues, i.e., $B_n(t)$, varies across time slots and is unknown to the mobile devices and edge nodes a priori. This corresponds to the unknown load level dynamics at the edge nodes and leads to the associated uncertain processing delay.

At the beginning of time slot t, let $w_{m,n}^{\text{edge}}(t)$ (in time slots) denote the remaining number of time slots until all the tasks placed in the queue of mobile device m at edge node n before time slot t have been either processed or dropped. Due to the unknown load level dynamics at the edge nodes, the mobile devices and edge nodes are unaware of the value of $w_{m,n}^{\text{edge}}(t)$ before all those tasks have been either processed or dropped. Let $d_{m,n}^{\text{edge}}(t)$ denote the number of time slots required to process task $k_{m,n}^{\text{edge}}(t)$. For presentation simplicity, we set $d_{m,n}^{\text{edge}}(t) = 0$ if $k_{m,n}^{\text{edge}}(t) = 0$, and $w_{m,n}^{\text{edge}}(1) = 0$. The values of $w_{m,n}^{\text{edge}}(t)$ and $d_{m,n}^{\text{edge}}(t)$ satisfy the following constraints:

$$w_{m,n}^{\text{edge}}(t) = \min\left\{\left[w_{m,n}^{\text{edge}}(t-1) + d_{m,n}^{\text{edge}}(t-1) - 1\right]^+, \tau_m - 1\right\},$$
$$m \in \mathcal{M}, n \in \mathcal{N}, t \in \mathcal{T} \setminus \{1\}, \tag{9.9}$$

$$\sum_{t'=t+w_{m,n}^{\text{edge}}(t)}^{t+w_{m,n}^{\text{edge}}(t)+d_{m,n}^{\text{edge}}(t)-1} \frac{\mathbb{1}\left(m \in \mathcal{B}_n(t')\right) f_n^{\text{edge}} \Delta}{\rho_m B_n(t')} \geq \lambda_{m,n}^{\text{edge}}(t),\ m \in \mathcal{M}, n \in \mathcal{N}, t \in \mathcal{T}, \tag{9.10}$$

$$\sum_{t'=t+w_{m,n}^{\text{edge}}(t)}^{t+w_{m,n}^{\text{edge}}(t)+d_{m,n}^{\text{edge}}(t)-2} \frac{\mathbb{1}\left(m \in \mathcal{B}_n(t')\right) f_n^{\text{edge}} \Delta}{\rho_m B_n(t')} < \lambda_{m,n}^{\text{edge}}(t),\ m \in \mathcal{M}, n \in \mathcal{N}, t \in \mathcal{T}. \tag{9.11}$$

The intuition of (9.9) is similar to those for $w_m^{\text{comp}}(t)$ in (9.2) and $w_m^{\text{tran}}(t)$ in (9.6). In inequality (9.10), $t + w_{m,n}^{\text{edge}}(t)$ and $t + w_{m,n}^{\text{edge}}(t) + d_{m,n}^{\text{edge}}(t) - 1$ correspond to the time slots when the processing of task $k_{m,n}^{\text{edge}}(t)$ starts and ends, respectively. Thus, inequalities (9.10) and (9.11) ensure that the processing of task $k_{m,n}^{\text{edge}}(t)$ can be accomplished by time slot $t+w_{m,n}^{\text{edge}}(t)+d_{m,n}^{\text{edge}}(t)-1$ and cannot be accomplished by $t + w_{m,n}^{\text{edge}}(t) + d_{m,n}^{\text{edge}}(t) - 2$.

For a task $k_m(t)$ that was offloaded to edge node n, it is placed in the associated queue of edge node n at the beginning of time slot $\phi_m(t) \triangleq t + w_m^{\text{tran}}(t) + d_m^{\text{tran}}(t)$. Thus, its queuing delay at edge node n is $w_{m,n}^{\text{edge}}(\phi_m(t))$, and its processing time is $d_{m,n}^{\text{edge}}(\phi_m(t))$. Thus, the delay of task $k_m(t)$ is derived as follows:

$$\text{Delay}_m(t) = \min\Bigg\{w_m^{\text{comp}}(t) + d_m^{\text{local}}(t)$$
$$+ \sum_{n \in \mathcal{N}} y_{m,n}(t)\left(w_{m,n}^{\text{edge}}(\phi_m(t)) + d_{m,n}^{\text{edge}}(\phi_m(t))\right), \tau_m\Bigg\},$$
$$m \in \mathcal{M}, t \in \{t' \mid t' \in \mathcal{T}, k_m(t) \neq 0, x_m(t) = 0\}. \tag{9.12}$$

Note that the above expressions (9.2)–(9.4), (9.6), (9.7), and (9.9)–(9.12) are used for presenting our system model. In practical systems, each mobile device can directly observe the delay of the tasks $\text{Delay}_m(t)$ after those tasks have either been processed or dropped.

9.3.2 Task Offloading Problem

We consider a fully observable system, where the mobile devices can observe the actual values of the state (e.g., queue information, task size). In particular, at the beginning of time slot $t \in \mathcal{T}$, mobile device $m \in \mathcal{M}$ observes its state. If mobile device m has a new task to be processed, then it will choose an action for the task, i.e., whether to offload the task or not, and which edge node to offload the task to. The state and action will result in a cost. We define the cost as the delay of the task if the task has been processed, and define it as a penalty if the task has been dropped. The objective is to find an optimal policy, i.e., a mapping from state to action, that minimizes the expected long-term cost.

9.3.2.1 State

Let $\boldsymbol{H}(t)$ denote the historical load level dynamics of the edge nodes within the previous T^{step} time slots. It is a matrix with size $T^{\text{step}} \times N$ and contains the number of active queues of all edge nodes from time slot $t - T^{\text{step}}$ to $t - 1$. In particular, element (i, j) of matrix $\boldsymbol{H}(t)$ is denoted by $\{\boldsymbol{H}(t)\}_{i,j}$, which corresponds to the number of active queues at edge node j in time slot $t - T^{\text{step}} + i - 1$. That is, $\{\boldsymbol{H}(t)\}_{i,j} = B_j(t - T^{\text{step}} + i - 1)$. We assume that the edge nodes will broadcast the number of active queues at the end of each time slot. The number of active queues can be represented by a maximum of $\lfloor \log_2 M \rfloor + 1$ bits. For example, if $M = 1000$, then a maximum of 10 bits are needed.

At the beginning of time slot $t \in \mathcal{T}$, each mobile device $m \in \mathcal{N}$ observes the task size $\lambda_m(t)$, the number of time slots required to wait for processing and offloading (i.e., $w_m^{\text{comp}}(t)$ and $w_m^{\text{tran}}(t)$), the queue occupancy at all the edge nodes $\boldsymbol{q}_m^{\text{edge}}(t - 1) \triangleq (q_{m,n}^{\text{edge}}(t - 1), n \in \mathcal{N})$, and the historical load level $\boldsymbol{H}(t)$. The state can be represented as follows:

$$\boldsymbol{s}_m(t) = \left(\lambda_m(t), w_m^{\text{comp}}(t), w_m^{\text{tran}}(t), \boldsymbol{q}_m^{\text{edge}}(t - 1), \boldsymbol{H}(t)\right). \tag{9.13}$$

Mobile device m can compute $\boldsymbol{q}_m^{\text{edge}}(t - 1)$ locally based on the tasks that have been offloaded to the edge nodes and the number of active queues at the edge nodes. Let $\mathcal{S}$ denote the finite and discrete space of the state. That is, $\mathcal{S} \triangleq \Lambda \times \{0, 1, \ldots, T\}^2 \times \mathcal{Q} \times \{0, 1, \ldots, M\}^{T^{\text{step}} \times N}$, where $\mathcal{Q}$ denotes the set of available queue occupancy at the edge nodes within T time slots.

9.3.2.2 Action

After mobile device m observes state $\boldsymbol{s}_m(t)$ at the beginning of time slot t, if there is a new task arrival (i.e., $\lambda_m(t) > 0$), then the mobile device will choose an action for the task, denoted by $a_m(t)$. We consider an action space $\mathcal{A} = \{0\} \cup \mathcal{N}$. If the mobile device chooses to process the task locally, then $a_m(t) = 0$. If the mobile device offloads the task to edge node $n \in \mathcal{N}$, then $a_m(t) = n$. That is,

$$a_m(t) = \begin{cases} 0, & x_m(t) = 1, \\ n \text{ such that } y_{m,n}(t) = 1, & x_m(t) = 0. \end{cases} \tag{9.14}$$

Note that we represent the task offloading decisions $x_m(t)$ and $\boldsymbol{y}_m(t)$ using $a_m(t)$ for the presentation simplicity of the algorithm.

9.3.2.3 Cost

Given the state $\boldsymbol{s}_m(t)$ and action $a_m(t)$, mobile device m will observe a cost after task $k_m(t)$ has either been processed or being dropped due to the task deadline. If task $k_m(t)$ has been processed, then we define the cost as the delay of task $k_m(t)$, i.e., $\text{Delay}_m(t)$. If the task $k_m(t)$ has been dropped, then we set the cost to be a constant penalty C_m, where C_m is larger than the maximum delay τ_m. Specifically, cost function $c_m(\boldsymbol{s}_m(t), a_m(t))$ is defined as follows:

$$c_m(\boldsymbol{s}_m(t), a_m(t)) = \begin{cases} \text{Delay}_m(t), & \text{if task } k_m(t) \text{ has been processed,} \\ C_m, & \text{if task } k_m(t) \text{ has been dropped.} \end{cases} \tag{9.15}$$

In the rest of this chapter, we will use the short form $c_m(t)$ to denote $c_m(\boldsymbol{s}_m(t), a_m(t))$.

Note that we focus on the unknown load level dynamics at the edge nodes. That is, a mobile device does not know the number of tasks offloaded by other mobile devices to an edge node a priori. Thus, when a mobile device makes an offloading decision for a task, it does not know the cost for choosing that decision. This leads to the necessity of using DRL to address the unknown and complex system dynamics.

9.3.2.4 Problem Formulation

For each mobile device $m \in \mathcal{M}$, the objective is to find an optimal policy $\pi_m^* : \mathcal{S} \to \mathcal{A}$ that minimizes the expected long-term cost. That is,

$$\pi_m^* = \arg\text{minimize}_{\pi_m} \quad \mathbb{E}\left[\sum_{t \in \mathcal{T}} \gamma^{t-1} c_m(t) \,\middle|\, \pi_m \right]$$
$$\text{subject to} \quad \text{constraints (9.1)–(9.4), (9.6), (9.7), (9.9)–(9.12),} \tag{9.16}$$

where the parameter $\gamma \in (0, 1]$ is a discount factor. This discount factor captures the discounted cost in the future. The expectation $\mathbb{E}[\cdot]$ is with respect to the time-varying parameters, e.g., the task arrivals and the task offloading decisions of other mobile devices.

9.3.3 *Deep Q-Learning-Based Algorithm*

In this section, we propose a DQL-based task offloading algorithm, under which the mobile devices can make their offloading decisions without knowing the system dynamics a priori. In this algorithm, each mobile device aims at learning a Q-value for each action given each state. The Q-value reveals the expected long-term cost of the mobile device given the state by selecting the associated action. The mapping from the state to the Q-value of each action is characterized by a neural network. With such a mapping, each mobile device can minimize its expected long-term cost by selecting the action with the minimum Q-value under its state.

In the following, we first present the neural network. Then, we propose the DQL-based algorithm.

9.3.3.1 Neural Network

For each mobile device, we use a neural network to characterize the mapping from each state to the Q-value of each action. The neural network contains six layers, as shown in Fig. 9.2. For the neural network of mobile device $m \in \mathcal{M}$, we use $\boldsymbol{\theta}_m$ to denote the parameter vector. This vector contains the biases of all neurons and the weights of all connections from the input layer to A&V layer (see Fig. 9.2).[2] In the following, we present each layer in detail.

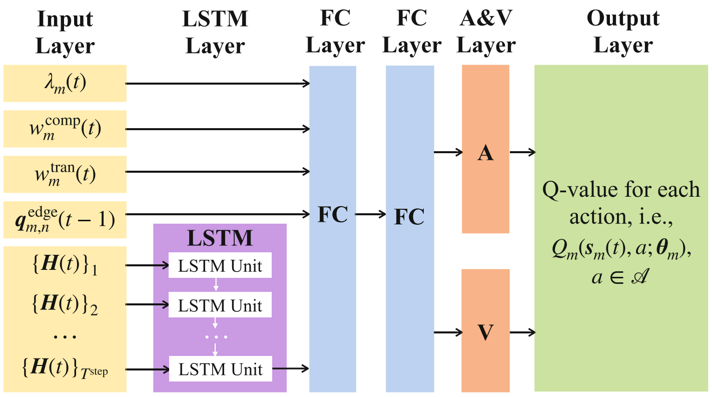
Input Layer
LSTM Layer
FC Layer
FC Layer
A&V Layer
Output Layer
$\lambda_m(t)$
$w_m^{\text{comp}}(t)$
$w_m^{\text{tran}}(t)$
$\boldsymbol{q}_{m,n}^{\text{edge}}(t-1)$
$\{\boldsymbol{H}(t)\}_1$
$\{\boldsymbol{H}(t)\}_2$
$\cdots$
$\{\boldsymbol{H}(t)\}_{T^{\text{step}}}$
LSTM
LSTM Unit
LSTM Unit
LSTM Unit
FC
FC
A
V
Q-value for each action, i.e., $Q_m(\boldsymbol{s}_m(t), a; \boldsymbol{\theta}_m)$, $a \in \mathcal{A}$

Fig. 9.2 The neural network of mobile device $m \in \mathcal{M}$

[2] The weights of the connections between the A&V layer and the output layer as well as the bias of the neurons in the output layer are given and fixed. Hence, we do not include them in the network parameter vector $\boldsymbol{\theta}_m$, as the vector $\boldsymbol{\theta}_m$ includes the parameters that are adjustable through learning in the DQL-based algorithm.

Input Layer In the neural network of any mobile device $m \in \mathcal{M}$, an input layer is responsible for taking the state (i.e., $\boldsymbol{s}_m(t)$) as input to the neural network.

Long Short-Term Memory (LSTM) Layer After the input layer, the LSTM is in charge of predicting the load level dynamics at the edge nodes in the short-term future. In the LSTM layer, there is an LSTM network that has T^{step} LSTM units. The LSTM units are connected in sequence. Let $\{\boldsymbol{H}_m(t)\}_i$ denote the ith row of the historical load level dynamics matrix $\boldsymbol{H}_m(t)$. The ith LSTM unit takes $\{\boldsymbol{H}_m(t)\}_i$ as an input. These LSTM units can record the variations of the load level dynamics at the edge nodes from $\{\boldsymbol{H}_m(t)\}_1$ to $\{\boldsymbol{H}_m(t)\}_{T^{\text{step}}}$. The output of the last LSTM unit is connected to the next layer in the neural network. This output can reveal the information indicating the load level dynamics in the short-term future.

Fully Connected (FC) Layer There are two FC layers after the LSTM layer. These layers are in charge of mapping from the learned load level dynamics and the state information to the Q-value of each action. Each FC layer has a set of neurons with rectified linear unit (ReLU). In the first FC layer, each neuron has full connections to all neurons (except those related to $\boldsymbol{H}(t)$) in the input layer and the output of the LSTM network. In the second FC layer, each neuron is connected to all neurons in the first FC layer.

A&V Layer and Output Layer We include an A&V layer after the FC layers. This is inspired by dueling DQN technique [14]. The main idea is to separately estimate the *state-value* (i.e., the part of Q-value resulting from the state) and the *advantage-value* for each action (i.e., the part of Q-value resulting from the action). The Q-value of each action given a state is the combination of the associated state-value and the advantage-value of the action.

In particular, in the A&V layer, there are two networks, i.e., network A and network V. Network A contains $|\mathcal{A}|$ neurons, where $|\mathcal{A}| = 1 + N$ is the number of available actions. This network is in charge of estimating the advantage-value for each action $a \in \mathcal{A}$. Recall that $\boldsymbol{\theta}_m$ denotes the biases and weights from the input layer to the A&V layer. Given parameter vector $\boldsymbol{\theta}_m$, let $A_m(\boldsymbol{s}_m(t), a; \boldsymbol{\theta}_m)$ denote the advantage-value of action $a \in \mathcal{A}$ under state $\boldsymbol{s}_m(t) \in \mathcal{S}$. Network V contains one neuron. It is responsible to estimate the state-value. Given parameter vector $\boldsymbol{\theta}_m$, we denote $V_m(\boldsymbol{s}_m(t); \boldsymbol{\theta}_m)$ as the state-value of state $\boldsymbol{s}_m(t)$. Note that parameter vector $\boldsymbol{\theta}_m$ needs to be trained during the DQL-based algorithm.

The output layer determines the Q-value of each action $a \in \mathcal{A}$ given state $\boldsymbol{s}_m(t) \in \mathcal{S}$. Such a Q-value can be determined as follows [14]:

$$\begin{aligned} Q_m(\boldsymbol{s}_m(t), a; \boldsymbol{\theta}_m) = V_m(\boldsymbol{s}_m(t); \boldsymbol{\theta}_m) + \Bigg(& A_m(\boldsymbol{s}_m(t), a; \boldsymbol{\theta}_m) \\ & - \tfrac{1}{|\mathcal{A}|} \textstyle\sum_{a' \in \mathcal{A}} A_m(\boldsymbol{s}_m(t), a'; \boldsymbol{\theta}_m) \Bigg). \end{aligned} \tag{9.17}$$

Specifically, the Q-value of an action is equal to the summation of the corresponding state-value and the additional advantage-value of the action, where the additional advantage-value is with reference to the average advantage-value among all actions.

9.3.3.2 Algorithm Design

We now present our proposed DQL-based task offloading algorithm. To reduce the computational loads at the mobile devices, we consider a setting where the edge nodes help mobile devices to perform training of the neural network. In particular, let $n_m \in \mathcal{N}$ denote the edge node that helps mobile device $m \in \mathcal{M}$ for training. This edge node can be the one that has the maximum transmission capacity with mobile device m. For edge node $n \in \mathcal{N}$, let $\mathcal{M}_n \subset \mathcal{M}$ denote the set of mobile devices for which the edge node performs training, i.e., $\mathcal{M}_n = \{m \mid n_m = n, m \in \mathcal{M}\}$.

Mobile device $m \in \mathcal{M}$ and edge node $n \in \mathcal{N}$ execute Algorithms 1 and 2, respectively. In particular, mobile device m collects experience $(\boldsymbol{s}_m(t), a_m(t), c_m(t), \boldsymbol{s}_m(t+1))$ for $t \in \mathcal{T}$ through the interaction with the MEC system. The associated edge node n_m maintains an experience replay D_m, which stores the experience of mobile device m. For presentation simplicity, we use the experience t of mobile device m to refer to $(\boldsymbol{s}_m(t), a_m(t), c_m(t), \boldsymbol{s}_m(t+1))$. Edge node n_m learns the mapping from each state to the Q-value of each action using the experience. Specifically, edge node n_m keeps two neural networks for mobile device m, including an evaluation network Net_m and a target network $Target\_Net_m$. Both neural networks follow the structure presented in Sect. 9.3.3.1. Nevertheless, they have different parameter vectors, i.e., $\boldsymbol{\theta}_m$ for Net_m and $\boldsymbol{\theta}_m^-$ for $Target\_Net_m$, and different functionalities. Edge node n_m aims at training the evaluation network Net_m to characterize the mapping from state to Q-values. During the training process, edge node n_m uses Net_m for action selection. It uses the target network $Target\_Net_m$ to approximate the expected long-term cost of each action given any state. We call the output of the target network as target Q-value. Edge node n_m will update the parameter vector of Net_m by minimizing the gap between the target Q-value and the Q-value under Net_m.

Algorithm 1 at Mobile Device $m \in \mathcal{M}$ The algorithm iterates for E episodes. At the beginning of each episode, mobile device $m \in \mathcal{M}$ initializes the state, i.e.,

$$\boldsymbol{s}_m(1) = (\lambda_m(1), w_m^{\text{comp}}(1), w_m^{\text{tran}}(1), \boldsymbol{q}_m^{\text{edge}}(0), \boldsymbol{H}(1)). \tag{9.18}$$

We set $q_{m,n}^{\text{edge}}(0) = 0$ for all $n \in \mathcal{N}$ and set $H(1)$ as a zero matrix with size $T^{\text{step}} \times N$.

At the beginning of time slot $t \in \mathcal{T}$, if mobile device m has a new task arrival $k_m(t)$, then it will request the recent parameter vector of network Net_m, i.e., $\boldsymbol{\theta}_m$, through sending a parameter_request to edge node n_m. Note that in practical systems, the mobile device can choose not to request the parameter vector in every time slot with new task arrivals in order to reduce the signaling overhead. Although reducing such a frequency may degrade the convergence rate of the proposed

Algorithm 1 Deep Q-learning-based algorithm at mobile device $m \in \mathcal{M}$

```
1: for episode = 1, 2, ..., $E$ do
2:     Initialize state $\boldsymbol{s}_m(1)$;
3:     for time slot $t \in \mathcal{T}$ do
4:         if mobile device $m$ has a new task arrival $k_m(t)$ then
5:             Send a parameter_request to edge node $n_m$;
6:             Receive network parameter vector $\boldsymbol{\theta}_m$;
7:             Select an action $a_m(t)$ according to (9.19);
8:         end if
9:         Observe the next state $\boldsymbol{s}_m(t+1)$;
10:        Observe a set of costs $\{c_m(t'), t' \in \widetilde{\mathcal{T}}_{m,t}\}$;
11:        for each task $k_m(t')$ with $t' \in \widetilde{\mathcal{T}}_{m,t}$ do
12:            Send $(\boldsymbol{s}_m(t'), a_m(t'), c_m(t'), \boldsymbol{s}_m(t'+1))$ to edge node $n_m$;
13:        end for
14:    end for
15: end for
```

algorithm, we have evaluated empirically that the degradation of the convergence rate is minimal if the frequency is maintained within a certain range. For example, with the system setting in Sect. 9.3.4, requesting the parameter vector every 100 time slots leads to a similar convergence rate as requesting it every time slot.

With a probability of ϵ, mobile device m randomly selects an action in set $\mathcal{A}$. With a probability of $1-\epsilon$, given the recent state $\boldsymbol{s}_m(t)$, it selects an action that leads to the minimum Q-value according to $\boldsymbol{\theta}_m$. That is,

$$a_m(t) = \begin{cases} \text{a random action from } \mathcal{A}, & \text{with a probability of } \epsilon, \\ \arg\min_{a \in \mathcal{A}} Q_m(\boldsymbol{s}_m(t), a; \boldsymbol{\theta}_m), & \text{with a probability of } 1-\epsilon. \end{cases} \tag{9.19}$$

Then, at the beginning of time slot $t+1$, mobile device m can observe the next state $\boldsymbol{s}_m(t+1)$. Note that in our system setting, the processing of task $k_m(t)$ does not need to be accomplished within time slot t. Thus, at the beginning of time slot $t+1$, the cost $c_m(t)$ associated with task $k_m(t)$ may have not been observed. Due to the same reason, mobile device m may observe a set of costs associated with some tasks $k_m(t')$ arrived in time slot $t' \le t$. Thus, we denote $\widetilde{\mathcal{T}}_{m,t} \subset \mathcal{T}$ as the set of time slots such that the tasks associated with those time slots have been either processed or dropped within time slot t. That is,

$$\widetilde{\mathcal{T}}_{m,t} = \{t' \mid t' = 1, 2, \ldots, t,\ \lambda_m(t') > 0,\ t' + \text{Delay}_m(t') - 1 = t\}, \tag{9.20}$$

where set $\widetilde{\mathcal{T}}_{m,t}$ can be an empty set for some $m \in \mathcal{M}$ and $t \in \mathcal{T}$. At the beginning of time slot $t+1$, mobile device m can observe a set of costs $\{c_m(t'), t' \in \widetilde{\mathcal{T}}_{m,t}\}$. For each task $k_m(t')$ with $t' \in \widetilde{\mathcal{T}}_{m,t}$, mobile device m sends the associated experience $(\boldsymbol{s}_m(t'), a_m(t'), c_m(t'), \boldsymbol{s}_m(t'+1))$ to edge node n_m. To reduce the signaling overhead, we consider a setting where mobile device m does not send matrices $\boldsymbol{H}(t')$ and $\boldsymbol{H}(t'+1)$ in states $\boldsymbol{s}_m(t')$ and $\boldsymbol{s}_m(t'+1)$. This is feasible because

Algorithm 2 Deep Q-learning-based algorithm at edge node $n \in \mathcal{N}$

```
1: Initialize neural network $Net_m$ with random $\boldsymbol{\theta}_m$ for $m \in \mathcal{M}_n$;
2: Initialize neural network $Target\_Net_m$ with random $\boldsymbol{\theta}_m^-$ for $m \in \mathcal{M}_n$;
3: Initialize experience replay $D_m$ for $m \in \mathcal{M}_n$ and Count $\leftarrow 0$;
4: while True do
5:     if receive a parameter_request from $m \in \mathcal{M}_n$ then
6:         Send the recent parameter vector $\boldsymbol{\theta}_m$ to mobile device $m$;
7:     end if
8:     if receive an experience $(\boldsymbol{s}_m(t), a_m(t), c_m(t), \boldsymbol{s}_m(t+1))$ from $m \in \mathcal{M}_n$ then
9:         Store $(\boldsymbol{s}_m(t), a_m(t), c_m(t), \boldsymbol{s}_m(t+1))$ in experience replay $D_m$;
10:        Sample a set of experiences (denoted by $\mathcal{I}$) from $D_m$;
11:        for each experience $i \in \mathcal{I}$ do
12:            Obtain experience $(\boldsymbol{s}_m(i), a_m(i), c_m(i), \boldsymbol{s}_m(i+1))$;
13:            Compute $\hat{Q}_{m,i}^{\text{Target}}$ according to (9.23);
14:        end for
15:        Set vector $\hat{\boldsymbol{Q}}_m^{\text{Target}} \leftarrow (\hat{Q}_{m,i}^{\text{Target}}, i \in \mathcal{I})$;
16:        Update $\boldsymbol{\theta}_m$ to minimize $L(\boldsymbol{\theta}_m, \hat{\boldsymbol{Q}}_m^{\text{Target}})$ in (9.21);
17:        Count $\leftarrow$ Count $+ 1$;
18:        if mod(Count, Replace_Threshold) $= 0$ then
19:            $\boldsymbol{\theta}_m^- \leftarrow \boldsymbol{\theta}_m$;
20:        end if
21:    end if
22: end while
```

we have assumed that the edge nodes broadcast their load level dynamics in each time slot.

Algorithm 2 at Edge Node $n \in \mathcal{N}$ Edge node $n \in \mathcal{N}$ first initializes neural networks Net_m and $Target\_Net_m$ and experience replay D_m for mobile device $m \in \mathcal{M}_n$. Then, it will wait for the messages from the mobile devices.

If edge node n receives a parameter_request from mobile device $m \in \mathcal{M}_n$, then it will forward the recent parameter vector $\boldsymbol{\theta}_m$ to mobile device m. If edge node n receives an experience from mobile device $m \in \mathcal{M}_n$, then it will store the experience in the experience replay D_m. After that, edge node n will update the parameter vector $\boldsymbol{\theta}_m$ of network Net_m according to steps 10–20 in Algorithm 2. The edge node first randomly samples I experiences from D_m. Let $\mathcal{I}$ denote the set of sampled experiences. For each experience $i \in \mathcal{I}$, edge node n will compute a target Q-value $\hat{Q}_{m,i}^{\text{Target}}$ (to be explained in the next paragraph) and update $\boldsymbol{\theta}_m$ by minimizing the following loss function:

$$L\left(\boldsymbol{\theta}_m, \hat{\boldsymbol{Q}}_m^{\text{Target}}\right) = \frac{1}{I} \sum_{i \in \mathcal{I}} \left(\hat{Q}_{m,i}^{\text{Target}} - Q_m(\boldsymbol{s}_m(i), a_m(i); \boldsymbol{\theta}_m) \right)^2, \tag{9.21}$$

where $\hat{\boldsymbol{Q}}_m^{\text{Target}} = (\hat{Q}_{m,i}^{\text{Target}}, i \in \mathcal{I})$. This loss function captures the difference between the target Q-value and the output of network Net_m for each experience $i \in$

$\mathcal{I}$. The edge node minimizes the loss function using backpropagation (see Section 6 in [33]).

Edge node n determines the target Q-value $\hat{Q}_{m,i}^{\text{Target}}$ for $i \in \mathcal{I}$ using double DQN technique [13]. This technique can improve the approximation of the expected long-term cost when compared with the conventional method (e.g., [12]). To determine the target Q-value, we denote a_i^{Next} as the action that leads to the minimum Q-value given the next state $\boldsymbol{s}_m(i+1)$ under network Net_m, i.e.,

$$a_i^{\text{Next}} = \arg\min_{a \in \mathcal{A}} Q_m(\boldsymbol{s}_m(i+1), a; \boldsymbol{\theta}_m). \tag{9.22}$$

The target Q-value $\hat{Q}_{m,i}^{\text{Target}}$ is computed as follows:

$$\hat{Q}_{m,i}^{\text{Target}} = c_m(i) + \gamma Q_m\left(\boldsymbol{s}_m(i+1), a_i^{\text{Next}}; \boldsymbol{\theta}_m^-\right). \tag{9.23}$$

Intuitively, the target Q-value is equal to the summation of the cost in experience i and the discount factor multiplied by the Q-value of action a_i^{Next} given the next state $\boldsymbol{s}_m(i+1)$ under network *Target_Net*$_m$. This value essentially approximates the expected long-term cost of action $a_m(i)$ given state $\boldsymbol{s}_m(i)$.

To keep the parameter vector $\boldsymbol{\theta}_m^-$ of *Target_Net*$_m$ up-to-date, edge node n updates $\boldsymbol{\theta}_m^-$ every several number of training rounds by copying the parameter vector $\boldsymbol{\theta}_m$ of network Net_m. Such updates make the target Q-value (which is derived based on *Target_Net*$_m$) a more accurate approximation of the expected long-term cost. We use Replace_Threshold to denote the corresponding number of training rounds, where mod$(\cdot)$ is the modulo operator in step 18 in Algorithm 2.

Discussion on Convergence Despite that we are able to prove the convergence of Q-learning algorithm, the convergence of a DQL-based algorithm is still an open problem. This is because the neural network is essentially an approximation of the mapping from state to Q-values. Due to such an approximation, the convergence may no longer be guaranteed. In this chapter, we empirically evaluate the convergence performance of the proposed algorithm in Sect. 9.3.4.

9.3.4 Performance Evaluation

We consider five edge nodes and 50 mobile devices. Table 9.1 shows the parameter settings of the MEC system and the hyperparameters of the proposed algorithm. As shown in Table 9.1, we consider a setting where each task has a deadline $\tau_m = 10$ time slots regardless of the task size. For example, for a video segment decoding task in live streaming, a video segment should always be decoded before a certain deadline to avoid rebuffering, where the deadline is independent of the number of image frames in the video segment. We set the penalty for dropped tasks C_m to 20

Table 9.1 Parameter settings of the MEC system

Parameter	Value	Parameter	Value
Δ	0.1 second	$f_m^{\text{device}}, m \in \mathcal{M}$	2.5 GHz [34]
$\lambda_m(t), m \in \mathcal{M}, t \in \mathcal{T}$	Discrete uniform distribution over set $\{2.0, 2.1, \ldots, 5.0\}$ Mbits [35]	$r_{m,n}^{\text{tran}}, m \in \mathcal{M}, n \in \mathcal{N}$	14 Mbps [36]
$\rho_m, m \in \mathcal{M}$	0.297 Gigacycles per Mbits [35]	$f_n^{\text{edge}}, n \in \mathcal{N}$	41.8 GHz [34]
$\tau_m, m \in \mathcal{M}$	10 time slots (i.e., 1 second [37])	$C_m, m \in \mathcal{M}$	20
Task arrival probability	0.3	Discount factor	0.9
Learning rate	0.001	Batch size	16
ϵ	Decrement from 1 to 0.01		

for $m \in \mathcal{M}$, where this value is twice as large as the maximum delay of a processed task (i.e., 10 time slots). Such a penalty setting makes the cost of a dropped task be always larger than the cost of a processed task, under which the proposed DQL-based algorithm will avoid tasks being dropped by optimizing the task offloading decision. In addition, we consider a setting where the task arrival probability is a constant value [29]. Despite that the task arrival probability is fixed across time, the number of tasks offloaded to an edge node can be time varying and is unknown to any particular mobile device, due to the time-varying and unknown offloading decisions of other mobile devices. This leads to the unknown load level dynamics at the edge nodes and the necessity of using our proposed DQL-based approach.

In the following, we evaluate the algorithm convergence. Then, we compare the performance of our proposed algorithm with some existing algorithms.

9.3.4.1 Algorithm Convergence

Figure 9.3 shows the convergence of the average cost among mobile devices of our proposed algorithm under different hyperparameters. For comparison, we show the random policy, where mobile devices randomly select their actions.

Figure 9.3(a) shows the algorithm convergence under different values of batch size, i.e., the number of experience sampled in one training round (i.e., I). When the batch size is increased from 2 to 8, the average cost converges to a lower value. Further increasing the batch size to 32 does not make a significant difference. Thus, a small batch size (e.g., 8) is sufficient for achieving a satisfactory convergence.

Figure 9.3(b) shows the algorithm convergence under different values of learning rate, which is the step size for updating network Net_m. As shown in the figure, when the learning rate is equal to 0.001, the average cost converges relatively fast and converges to a smaller value when compared with the other values of learning rate.

On the other hand, in practical systems, the task arrival probability can be non-stationary. Under such a scenario, once the environment has changed, the proposed

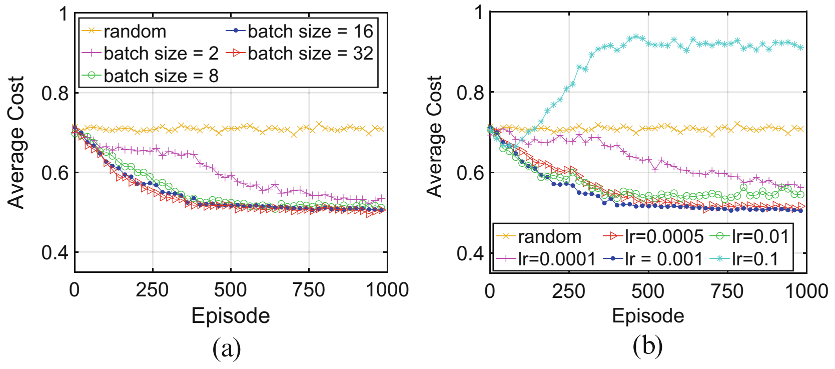

Fig. 9.3 Algorithm convergence under different: (**a**) batch size and (**b**) learning rate (denoted by "lr")

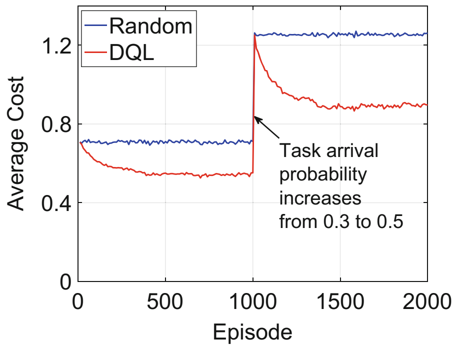

Fig. 9.4 An illustration of the algorithm performance with non-stationary task arrival probability

algorithm can adapt to it by resetting the probability of random exploration to be one in order to enable the random exploration again. Figure 9.4 shows an example of the performance of the proposed DQL-based algorithm with non-stationary task arrival probability. In this simulation, at around 1000 episodes, the task arrival probability increases from 0.3 to 0.5. Hence, the average cost of both the random policy and our proposed DQL-based algorithm are changed accordingly. As the episodes proceed, our proposed algorithm gradually adapts to the change of task arrival probability and converges again. We will leave it as future work to design an efficient algorithm for addressing frequent environmental changes. Candidate approaches include concept drift detection [38] and non-stationary reinforcement learning [39, 40].

9.3.4.2 Method Comparison

We compare our proposed algorithm with several benchmark methods. These include no offloading (denoted by No Offl.), random offloading (denoted by R.

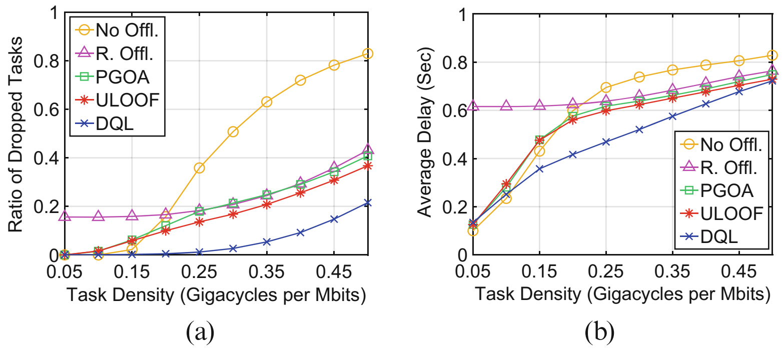

Fig. 9.5 Performance with different task densities: (**a**) ratio of dropped tasks and (**b**) average delay

Offl.), potential game-based offloading algorithm (PGOA) in [31], and user-level online offloading framework (ULOOF) in [34]. With PGOA and ULOOF, mobile devices make their offloading decisions based on a best response algorithm for potential game and the capacity estimation with historical observations, respectively. In the simulation results, we use "DQL" to refer to our proposed DQL-based algorithm.

In the simulations, we consider two performance metrics. The first is the ratio of dropped tasks. This is the ratio of the number of dropped tasks to the total number of tasks. The second is the average delay of the tasks that have been processed.

Task Density In Fig. 9.5, as the task density increases, the ratio of dropped tasks and the average delay of each method increase. This is because a larger task density implies a higher computational requirement of each task. As the task density increases from 0.05 to 0.25 Gigacycles per Mbits, the ratio of dropped tasks and average delay of our proposed algorithm increase less drastically than those of the benchmark methods. When the density is 0.25 Gigacycles per Mbits, our proposed algorithm maintains a ratio of dropped tasks of around 0.01 and an average delay of 0.47 second. As the task density further increases to 0.5 Gigacycles per Mbits, although all methods have a similar average delay, our proposed algorithm can reduce the ratio of dropped tasks by 41.4%–74.1% when compared with the benchmark methods.

Processing Capacity of Edge Node As shown in Fig. 9.6, under various values of the processing capacity of each edge node, our proposed algorithm can reduce the ratio of dropped tasks and the average delay when compared with the benchmark methods. The reduction of the ratio of dropped tasks is especially significant when the processing capacity of each edge node is small. When the processing capacity is 15 GHz, the proposed algorithm reduces the ratio of dropped tasks by at least 57.0% and reduces the average delay by at least 9.4% when compared with the benchmark methods. As the processing capacity further increases, both performance metrics converge, because further increasing the capacity does not reduce the delay of those

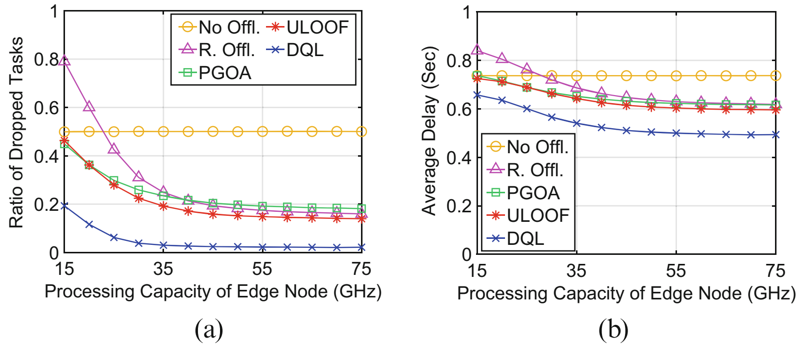

Fig. 9.6 Performance with different processing capacities of edge nodes: (**a**) ratio of dropped tasks and (**b**) average delay

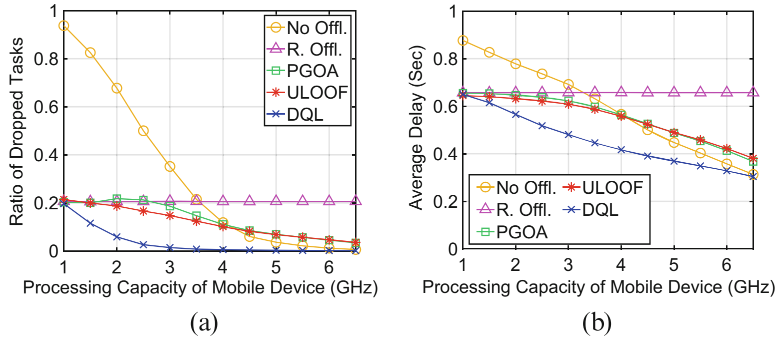

Fig. 9.7 Performance with different processing capacities of mobile devices: (**a**) ratio of dropped tasks and (**b**) average delay

tasks offloaded due to the limited transmission capacity. The converged ratio of dropped tasks and the average delay of our proposed algorithm is at least 84.3% and 17.2% less than those of the benchmark methods, respectively.

Processing Capacity of Mobile Devices In Fig. 9.7, as the processing capacity of each mobile device increases, our proposed algorithm has a more significant decrease in terms of the ratio of dropped tasks and average delay when compared with PGOA and ULOOF. When the processing capacity increases to 3.5 GHz, our proposed algorithm achieves a ratio of dropped tasks of 0.007, which is 93.9%–96.5% lower than those of the benchmark methods. Meanwhile, our algorithm achieves an average delay that is 31.4% and 29.4% lower than those of PGOA and ULOOF, respectively. As the processing capacity of each mobile device further increases, processing a task locally becomes optimal. Thus, our proposed algorithm tends to choose local processing and achieves a similar performance as no offloading.

9.4 Challenges and Future Directions

Despite the fact that DRL can effectively address the unknown and time-varying system dynamics, there are still several remaining challenges and future research directions for the deployment of DRL algorithms in MEC systems.

First, the training process of DRL algorithms may require substantial computational resource consumption. Meanwhile, under the scenario where the training process is offloaded to some devices with sufficient computational resources, the offloading may lead to communication resource consumption for neural network transmission. As a result, the scalability of DRL algorithms in MEC systems may be a concern. To address these issues, we may include both offline phase and online phase for DRL algorithms, where the offline phase is performed offline with powerful devices (e.g., cloud server). Transfer learning techniques [41] may be utilized for the online phase to make the neural network quickly adapt to the real-world environment. Moreover, deep compression techniques [42], such as network pruning (i.e., reducing the number of weights in neural networks) and quantization (i.e., reducing the number of bits for representing a weight), may be used to reduce the communication resource requirement.

Second, the environment in MEC systems may be time varying. Thus, DRL algorithms should be able to detect the change of the environment and quickly adapt to the new environment after changing. Methods for concept drift detection [38] are applicable for detecting the change of the environment. In addition, techniques for lifelong learning [43] may be applicable for handling the non-stationary environments. Meanwhile, existing works, e.g., [39, 40], proposed DRL algorithms for non-stationary reinforcement learning, which may be applicable to MEC systems.

Third, in MEC systems with a large number of mobile devices and edge nodes, there are potentials for the mobile devices and edge nodes to cooperate to learn the optimal policy through their interaction with the environments. In other words, the mobile devices and edge nodes may cooperatively train the neural networks in the DRL algorithms. Such a collaboration can alleviate the requirements for computational resources and improve the resource efficiency. Federated learning techniques [44] can enable the collaboration among mobile devices and edge nodes for neural network training and hence may be incorporated in DRL algorithms.

9.5 Conclusion

In this chapter, we provided an overview of the DRL algorithms for MEC systems. We introduced DRL fundamentals and then presented a case study on task offloading in MEC systems. In this case study, we focused on the unknown and time-varying load level dynamics at the edge nodes and proposed a DQL-based algorithm that enables the mobile devices to make task offloading decisions in a decentralized fashion. We conducted simulations and showed that the proposed algorithm can reduce the task delay and ratio of dropped tasks. Finally, we outlined the challenges and future research directions for DRL algorithms in MEC systems.

References

1. Y. Mao, C. You, J. Zhang, K. Huang, K.B. Letaief, A survey on mobile edge computing: the communication perspective. IEEE Commun. Surv. Tutorials **19**(4), 2322–2358, Fourth quarter (2017)
2. P. Porambage, J. Okwuibe, M. Liyanage, M. Ylianttila, T. Taleb, Survey on multi-access edge computing for Internet of things realization. IEEE Commun. Surv. Tutorials **20**(4), 2961–2991 (2018)
3. F. Bonomi, R. Milito, J. Zhu, S. Addepalli, Fog computing and its role in the Internet of things, in *Proc. ACM SIGCOMM Workshop on Mobile Cloud Computing (MCC), Helsinki*, August 2012
4. Z. Sanaei, S. Abolfazli, A. Gani, R. Buyya, Heterogeneity in mobile cloud computing: taxonomy and open challenges. IEEE Commun. Surv. Tutorials **16**(1), 369–392, First quarter (2014)
5. T. Qiu, J. Chi, X. Zhou, Z. Ning, M. Atiquzzaman, D.O. Wu, Edge computing in industrial Internet of things: architecture, advances and challenges. IEEE Commun. Surv. Tutorials **22**(4), 2462–2488, Fourth quarter (2020)
6. P. Ranaweera, A.D. Jurcut, M. Liyanage, Survey on multi-access edge computing security and privacy. IEEE Commun. Surv. Tutorials **23**(2), 1078–1124, Second quarter (2021)
7. T. Chen, Q. Ling, G.B. Giannakis, An online convex optimization approach to proactive network resource allocation. IEEE Trans. Signal Process. **65**(24), 6350–6364 (2017)
8. H. Shah-Mansouri, V.W.S. Wong, Hierarchical fog-cloud computing for IoT systems: a computation offloading game. IEEE Internet Things J. **5**(4), 3246–3257 (2018)
9. V. François-Lavet, P. Henderson, R. Islam, M.G. Bellemare, J. Pineau, An introduction to deep reinforcement learning. Found. Trends Mach. Learn. **11**(3–4), 219–354 (2018)
10. X. Wang, Y. Han, V.C.M. Leung, D. Niyato, X. Yan, X. Chen, Convergence of edge computing and deep learning: a comprehensive survey. IEEE Commun. Surv. Tutorials **22**(2), 869–904, Second quarter (2020)
11. R.S. Sutton, A.G. Barto, *Reinforcement Learning: An Introduction*, 2nd edn. (MIT Press, Cambridge, MA, 2018)
12. V. Mnih, K. Kavukcuoglu, D. Silver, A.A. Rusu, J. Veness, M.G. Bellemare, A. Graves, M. Riedmiller, A.K. Fidjeland, G. Ostrovski, Human-level control through deep reinforcement learning. Nature **518**(7540), 529–533 (2015)
13. H. van Hasselt, A. Guez, D. Silver, Deep reinforcement learning with double Q-learning, in *Proc. AAAI Conf. on Artificial Intelligence, Phoenix, AZ*, May 2016
14. Z. Wang, T. Schaul, M. Hessel, H. van Hasselt, M. Lanctot, N. de Freitas, Dueling network architectures for deep reinforcement learning, in *Proc. Int'l Conf. on Machine Learning (ICML), New York City, NY*, June 2016
15. M.G. Bellemare, W. Dabney, R. Munos, A distributional perspective on reinforcement learning, in *Proc. Int'l Conf. on Machine Learning (ICML), Sydney*, August 2017
16. T.P. Lillicrap, J.J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, D. Wierstra, Continuous control with deep reinforcement learning. Preprint, arXiv:1509.02971, July 2019
17. G. Barth-Maron, M.W. Hoffman, D. Budden, W. Dabney, D. Horgan, T.B. Dhruva, A. Muldal, N. Heess, T. Lillicrap, Distributed distributional deterministic policy gradients, in *Proc. Int'l Conf. on Learning Representations (ICLR), Vancouver*, April 2018
18. V. Mnih, A.P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, K. Kavukcuoglu, Asynchronous methods for deep reinforcement learning, in *Proc. Int'l Conf. on Machine Learning (ICML), New York City, NY*, June 2016

19. Z. Wang, V. Bapst, N. Heess, V. Mnih, R. Munos, K. Kavukcuoglu, N. de Freitas, Sample efficient actor-critic with experience replay. Preprint, arXiv:1611.01224, July 2017
20. T. Haarnoja, A. Zhou, P. Abbeel, S. Levine, Soft actor-critic: off-policy maximum entropy deep reinforcement learning with a stochastic actor, in *Proc. Int'l Conf. on Machine Learning (ICML), Stockholm*, June 2018
21. S. Fujimoto, H. Hoof, D. Meger, Addressing function approximation error in actor-critic methods, in *Proc. Int'l Conf. on Machine Learning (ICML), Stockholm*, June 2018
22. J. Schulman, S. Levine, P. Abbeel, M. Jordan, P. Moritz, Trust region policy optimization, in *Proc. Int'l Conf. on Machine Learning (ICML), Lille*, June 2015
23. J. Schulman, F. Wolski, P. Dhariwal, A. Radford, O. Klimov, Proximal policy optimization algorithms. Preprint, arXiv:1707.06347, August 2017
24. C.B. Browne, E. Powley, D. Whitehouse, S.M. Lucas, P.I. Cowling, P. Rohlfshagen, S. Tavener, D. Perez, S. Samothrakis, S. Colton, A survey of Monte Carlo tree search methods. IEEE Trans. Comput. Intel. AI **4**(1), 1–43 (2012)
25. D. Silver et al., Mastering the game of Go with deep neural networks and tree search. Nature **529**(7587), 484–489 (2016)
26. A. Plaat, W. Kosters, M. Preuss, Model-based deep reinforcement learning for high-dimensional problems, a survey. Preprint, arXiv:2008.05598, December 2020
27. N.C. Luong, D.T. Hoang, S. Gong, D. Niyato, P. Wang, Y.-C. Liang, D.I. Kim, Applications of deep reinforcement learning in communications and networking: a survey. IEEE Commun. Surv. Tutorials **21**(4), 3133–3174, Fourth quarter (2019)
28. M. Tang, V.W.S. Wong, Deep reinforcement learning for task offloading in mobile edge computing systems. IEEE Trans. Mobile Comput. **21**(6), 1985–1997 (2020)
29. J. Liu, Y. Mao, J. Zhang, K.B. Letaief, Delay-optimal computation task scheduling for mobile-edge computing systems, in *Proc. IEEE Int'l Symp. on Information Theory (ISIT), Barcelona*, July 2016
30. X. Lyu, W. Ni, H. Tian, R.P. Liu, X. Wang, G.B. Giannakis, A. Paulraj, Distributed online optimization of fog computing for selfish devices with out-of-date information. IEEE Trans. Wirel. Commun. **17**(11), 7704–7717 (2018)
31. L. Yang, H. Zhang, X. Li, H. Ji, V. Leung, A distributed computation offloading strategy in small-cell networks integrated with mobile edge computing. IEEE/ACM Trans. Netw. **26**(6), 2762–2773 (2018)
32. A.K. Parekh, R.G. Gallager, A generalized processor sharing approach to flow control in integrated services networks: the single-node case. IEEE/ACM Trans. Netw. **1**(3), 344–357 (1993)
33. I. Goodfellow, Y. Bengio, A. Courville, *Deep Learning* (MIT Press, Cambridge, 2016)
34. J.L.D. Neto, S.-Y. Yu, D.F. Macedo, M.S. Nogueira, R. Langar, S. Secci, ULOOF: a user level online offloading framework for mobile edge computing. IEEE Trans. Mobile Comput. **17**(11), 2660–2674 (2018)
35. C. Wang, C. Liang, F.R. Yu, Q. Chen, L. Tang, Computation offloading and resource allocation in wireless cellular networks with mobile edge computing. IEEE Trans. Wirel. Commun. **16**(8), 4924–4938 (2017)
36. Speedtest Intelligence, Speedtest global index: Canada average mobile upload speed based on March 2021 data, https://www.speedtest.net/reports/canada/. Accessed 25 June 2021
37. X. Lyu, H. Tian, W. Ni, Y. Zhang, P. Zhang, R.P. Liu, Energy-efficient admission of delay-sensitive tasks for mobile edge computing. IEEE Trans. Commun. **66**(6), 2603–2616 (2018)
38. J. Lu, A. Liu, F. Dong, F. Gu, J. Gama, G. Zhang, Learning under concept drift: a review. IEEE Trans. Knowl. Data Eng. **31**(12), 2346–2363 (2019)
39. A. Xie, J. Harrison, C. Finn, Deep reinforcement learning amidst lifelong non-stationarity. Preprint, arXiv:2006.10701, June 2020
40. V. Lomonaco, K. Desai, E. Culurciello, D. Maltoni, Continual reinforcement learning in 3D non-stationary environments, in *Proc. IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops*, June 2020

41. F. Zhuang, Z. Qi, K. Duan, D. Xi, Y. Zhu, H. Zhu, H. Xiong, Q. He, A comprehensive survey on transfer learning. Proc. IEEE **109**(1), 43–76 (2021)
42. S. Han, H. Mao, W.J. Dally, Deep compression: compressing deep neural networks with pruning, trained quantization and Huffman coding, in *Proc. Int'l Conf. on Machine Learning (ICML), New York City, NY*, June 2016
43. Z. Chen, B. Liu, Lifelong machine learning. Synth. Lect. Artif. Intell. Mach. Learn. **12**(3), 1–207 (2018)
44. W.Y.B. Lim, N.C. Luong, D.T. Hoang, Y. Jiao, Y.-C. Liang, Q. Yang, D. Niyato, C. Miao, Federated learning in mobile edge networks: a comprehensive survey. IEEE Commun. Surv. Tutorials **22**(3), 2031–2063, Third quarter (2020)