Seek Common While Shelving Differences: Orchestrating Deep Neural Networks for Edge Service Provisioning

Lixing Chen and Jie Xu, *Member, IEEE*

***Abstract*— Edge computing (EC) platforms, which enable Application Service Providers (ASPs) to deploy applications in close proximity to users, are providing ultra-low latency and location-awareness to a rich portfolio of services. As monetary costs are incurred for renting computing resources on edge servers to enable service provisioning, ASP has to cautiously decide where to deploy the application and how much resources would be needed to deliver satisfactory performance. However, the service provisioning problem exhibits complex correlations with multifarious factors in EC systems, ranging from user behavior to computation offloading, which are difficult to be fully captured by mathematical modeling and also put off traditional machine learning techniques due to the induction of high-dimension state space. The recent success of deep learning (DL) underpins new tools for addressing our problem. While previous works provide valuable insights on applying DL techniques, e.g., distributed DL, deep reinforcement learning (DRL), and multi-agent DL, in EC systems, these techniques cannot solely handle the distributed and heterogeneous nature of EC systems. To address these limitations, we propose a novel framework based on multi-agent DRL, distributed neural network orchestration (N_2O), and knowledge distilling. The multi-agent DRL enables edge servers to learn deep neural networks that shelve distinct features learned from local edge sites and hence caters to the heterogeneity of EC systems. N_2O coordinates edge servers in a fully distributed manner toward a common goal of maximizing ASP's reward. It requires only local communications during execution and provides provable performance guarantees. The knowledge distilling is further utilized to distill the N_2O policy for reducing the communication overhead and stabilizing the decision-making. We also carry out systematic experiments to show the advantages of our method over state-of-the-art alternatives.**

***Index Terms*— Edge computing, deep reinforcement learning, multi-agent learning, distributed optimization.**

I. Introduction

EDGE computing [1], [2] is a promising solution to accommodate the explosive growth of Internet-connected devices and the huge amount of distributed data that they generate. Being physically close to the data sources and leveraging fast network technologies such as 5G, edge computing promises several benefits compared to the traditional cloud-based computing paradigm, including lower latency, higher energy efficiency, better privacy protection, reduced bandwidth consumption, and location/context awareness [1]. In fact, edge computing is no longer a mere version but becoming a reality. For example, Verizon and Amazon Web Services (AWS) are partnering to construct a cloud-like commercialized platform at the edge of Verizon's 5G network [3]. It is anticipated that application service providers (ASPs), e.g., developers and enterprise customers, will soon be able to rent computing resources on the shared edge computing platform and deploy their services close to end-users without building their own data center or trenching their own fiber. As renting computing resources incurs monetary costs, a realistic problem faced by ASPs is: how to deliver high-quality application services using the edge computing platform in a cost-effective manner? This brings up the edge service provisioning problem for ASPs: 1) where (i.e., at which edge sites) to deploy the application service, and 2) how much computing resources should be rented at these edge sites in order to maximize performance. Although service provisioning problems have been studied in the cloud computing context (e.g., [4]), the distributed and heterogeneous nature of edge computing systems, as well as the complicated user-edge interactions, calls for new approaches to address new challenges for efficient and cost-effective edge service provisioning.

A. Technical Challenges

Edge service provisioning is tightly intertwined with many other components in edge computing systems. Considering the vertical *user-edge* interaction, the computation offloading policy [5], [6] on the user-side determines the amount of service demand that is sent to edge servers, which are further influenced by how radio resources are scheduled by the wireless network [7], [8]. Considering the horizontal interactions between users or between edge servers, users may compete for radio/computing resources of an edge server [9] and edge servers may perform load balancing among each other [10]. All these factors affect directly or indirectly the edge service provisioning decisions and rewards of ASP, making it extremely difficult to characterize and solve the problem with traditional model-based approaches. Traditional learning-based approaches (e.g., reinforcement learning [11], multi-armed

Manuscript received July 15, 2020; revised September 26, 2020; accepted October 24, 2020. Date of publication November 9, 2020; date of current version December 16, 2020. This work was supported in part by the National Science Foundation under Award ECCS-2033681, Award ECCS-2029858, and Award CNS-2006630 and in part by the Army Research Office under Award W911NF-18-1-0343. *(Corresponding author: Lixing Chen.)*

The authors are with the Department of Electrical and Computer Engineering, University of Miami, Coral Gables, FL 33146 USA (e-mail: lx.chen@miami.edu; jiexu@miami.edu).

Color versions of one or more of the figures in this article are available online at https://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/JSAC.2020.3036953

bandit [12]) also quickly reach their bottlenecks due to the large state/action spaces for characterizing the complex edge computing system. The substantial breakthroughs of deep learning [13], [14] in recent years underpin new tools for solving the edge service provisioning problem. The complex correlations between service provisioning and other components in edge computing can be abstracted directly from data, thereby saving the effort of complicated system modeling. Previous works [5], [15]–[19] have considered exploiting deep learning in edge computing systems though not in the context of edge service provisioning. Among the existing works, two deep learning techniques, namely *Deep Reinforcement Learning* [20] and *Distributed Deep Learning* [21], are investigated most. Although these two techniques have their distinct advantages, they overlook certain crucial features in the edge computing for addressing the service provisioning problem effectively and efficiently. Below, we discuss their pros and cons in more detail.

1) Deep Reinforcement Learning: Deep reinforcement learning (DRL) is a model-free approach to learn a decision policy that captures the temporal decision dependency. It can work without an offline collected dataset and instead learn in an online fashion from its experience by interacting with the environment. These properties are desirable for solving our edge service provisioning problem, as well as many other decision problems in edge computing systems, e.g., computation offloading [6], [22], resource allocation [15], and caching [23]. However, existing works apply DRL to solve a single-agent decision problem, which is not suitable for edge service provisioning where decisions have to be made by *many* distributed edge sites. Because a single edge server is resource-constrained, cooperation among multiple edge servers is needed to accommodate geographically distributed and correlated service demand. Simply applying single-agent DRL requires collecting the state information of all edge servers [18], [23] by a centralized entity and hence incurs a high communication overhead. More importantly, centralized single-agent DRL requires training a big deep neural network to incorporate the state/action spaces of all edge servers, resulting in intolerably long training time and poor accuracy. This makes single-agent DRL infeasible in large-scale distributed edge computing systems.

2) Distributed Deep Learning: Distributed deep learning (DDL) is recently studied to train a global deep learning model in a distributed way using locally collected data [21], [24]. This idea has been applied in edge computing systems to derive computation offloading, service placement, and content caching policies [5], [16], [25]. With DDL, all learners (e.g., edge servers) eventually arrive at the same global model and hence, the same policy, after many rounds of the model integration process [21], [24]. However, because edge servers are different in terms of their geographical locations, user demographics, demand patterns, and computing capabilities, their policies are likely to be different. This requires edge servers to have distinct service provisioning policies while being coordinated to achieve a global performance goal.

Considering the distributed and heterogeneous nature of edge computing systems, multi-agent reinforcement learning (MA-RL) [26] is actually a better fit to address many edge computing decision problems but receives much less investigation. In MA-RL, agents learn their own policies tailored to their local environment, and work cooperatively to maximize the overall system performance. MA-RL has been applied to solve computation offloading [27] and resource allocation [28] problems in edge computing. More recently, multi-agent deep reinforcement learning (MA-DRL) [29]–[31] incorporates deep learning into MA-RL. In this literature, most existing works [29], [30] consider a cooperative setting where agents aim at maximizing the overall system reward and train their local DRL policies using the overall system reward feedback. Some other works [31] study the competitive setting (like a game) where agents aim at maximizing their own individual reward and train their local DRL policies using their individual reward feedback. These works all adopt the framework of *centralized training with decentralized execution* which will require a centralized entity. Our problem is different in that each agent uses its own individual reward feedback to train local edge service provisioning policies yet the goal is still to maximize the overall system reward. In particular, both training and execution of the edge service provisioning policies have to be fully distributed.

B. Novelties and Contributions

In this paper, we propose a novel framework for addressing the service provisioning problem in the edge computing system. Our method is based on multi-agent deep reinforcement learning (MA-DRL) and further incorporates distributed neural network orchestration and knowledge distilling, thereby overcoming the limitations of DDL, single-agent DRL, and existing MA-DRL solutions. The proposed framework respects the heterogeneity in edge computing systems and enables distributed policy learning and execution. While this paper uses edge service provisioning as a specific problem to illustrate the power of the proposed framework, it can also be applied to address many other decision problems in distributed and heterogeneous edge computing systems, e.g., service placement, content caching, computation offloading, etc. with proper adjustment. The main contributions are summarized as follows.

1) We formulate the edge service provisioning problem as a Markov Decision Process (MDP), and decompose it into multiple loosely connected local MDPs, one for each edge server. Each edge server trains a DRL policy based on local data for solving its local MDP. With this MA-DRL approach, training the service provisioning policy is fully distributed and the derived local policy fits the local environment.

2) Because the trained local service provisioning policies may have conflicted decisions, we then design a distributed orchestration scheme, called Neural Network Orchestration (N_2O), to coordinate the local policies to work towards the common goal of maximizing the overall system performance. N_2O is executed in a distributed manner and requires only local communication (i.e., information exchanges with only

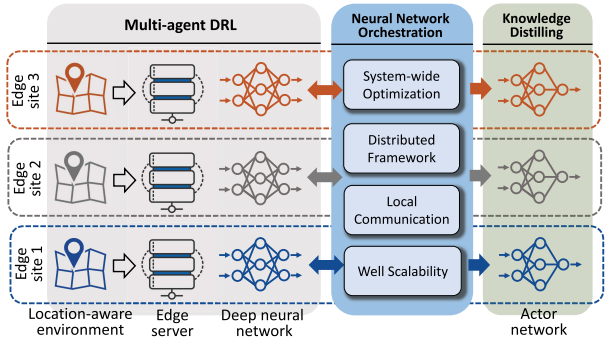

Fig. 1. Overall framework of the proposed method.

nearby edge servers) to derive a system-wide service provisioning decision. In particular, N_2O is able to handle the non-convexity of DNNs with a provable performance guarantee. In addition, N_2O works with stochastic communications, and its convergence rate is proven depending on the edge network topology.

3) To further reduce the overhead during policy orchestration and accelerate decision making, knowledge distilling [32] is utilized to distill the service provisioning policy based on the decisions made by N_2O. Its key idea is to train another DNN, called the actor network, at each edge server to approximate the service provisioning decisions derived by N_2O.

Fig. 1 depicts an overall framework for the proposed method. MA-DRL and knowledge distilling are carried out locally on each edge server, catering to distributed edge computing systems. The core innovation of our method is N_2O, which coordinates locally trained DNNs of distributed edge servers and provides data for training the distilled actor network, thereby connecting MA-DRL and knowledge distilling. The rest of the paper is organized as follows. Section II describes the edge computing system and defines the service provisioning problem. Section III formulates the service provisioning problem as a Markov decision process and presents two possible solutions, centralized deep Q-learning and multi-agent deep Q-learning. Section IV designs the neural network orchestration policy and theoretically analyzes its performance. Section V studies the knowledge distilling for the neural network orchestration policy. Section VI carries out the experiment, followed by the conclusion in Section VII.

II. Service Provisioning in Edge Computing Systems

A. Edge Computing System

We consider a typical edge computing scenario where an edge computing platform is constructed on a heterogeneous small-cell network [33]. The heterogeneous small-cell network consists of a set of small-cell base stations (SBSs), indexed by $\mathcal{N} = \{1, 2, \ldots, N\}$, and a macro base station (MBS), indexed by 0. These SBSs are expected to be densely deployed, reaching a density of $40 \sim 50$ SBSs/km^2 [34]. Each SBS is co-located with an edge server that possesses computing resources for supporting computing services. These edge servers provide platform-as-a-service to Application Service Provider (ASP), managing computing resources requested by ASP using virtualization techniques. Besides SBSs, there also exists an MBS that guarantees a ubiquitous service access with all-over radio coverage and connections to ASP's cloud. Users in the service area can offload computation tasks to either edge servers (via SBSs) or the cloud server (via MBS). Note that the SBSs/edge servers are densely deployed and hence the users may be in the coverage of multiple edge servers. A user uses a computation offloading policy to distribute the computation tasks among the local mobile device, reachable edge servers, and cloud server. Various computation offloading policies have been investigated in the existing literature [5], [6], and our method is compatible with most of these policies.

B. Service Provisioning

To enable service provisioning on the network edge, an ASP rents computing resources and deploys its application service on edge servers. As such, the edge servers charge the ASP for the amount of requested computing resources. The edge server uses virtualization techniques, e.g. *containerization* or *server virtualization*, to discretize the computing resources into containers or virtual machines. We let $\mathcal{A}_n := \{0, 1, 2, \ldots\}$ be the feasible set of resource rental decisions available on edge server n. Note that $\mathcal{A}_n$ contains `None` decision, denoted by 0, meaning that no computing resource is rented on the edge server. To adapt to the time-varying user population and service demand, ASP needs to change its rental decision across time. We discretize the operational timeline of ASP into time slots. At the beginning of each time slot t, the ASP determines its resource rental decisions on all N edge servers $\boldsymbol{a}_t := \{a_{n,t}\}_{n=1}^N$ where $a_{n,t} \in \mathcal{A}_n$ is the resource rental decision on edge server n. We call $\boldsymbol{a}_t \in \mathcal{A} := \mathcal{A}_1 \times \mathcal{A}_2 \times \ldots \mathcal{A}_N$ the system rental decision. The resource rental decision is fixed till the end of the time slot. We set the length of each time slot to be several minutes. We note that changing the rented computing resource will not incur high reconfiguration costs to edge servers. The state-of-the-art virtualization techniques [35] even enable virtual machines to be resized during the run-time of service applications. If $a_{n,t}$ is non-zero, then ASP is able to deploy its application service on edge server n and the users $\mathcal{M}_t = \{1, 2, \ldots, M_t\}$ can offload their tasks to edge server n under certain physical constraints. Fig. 2 provides an illustration of the edge computing system and service provisioning problem for a better understanding of the discussed system model.

C. Rewards of Application Service Provider

ASP derives rewards by provisioning service at edge servers. The improvement of service quality provided by edge computing can be multi-fold [36], e.g., latency reduction, energy saving, better privacy protection. Without loss of generality, we focus on the service delay reduction provided by edge computing because it is closely related to ASP's resource rental decisions. In general, the reward of ASP is determined by three factors: 1) the reduction of service delay provided by using edge computing service, 2) the amount of service demand processed on edge servers, and 3) the monetary cost

TABLE I
SUMMARY OF VARIABLES

Notation	Description	Notation	Description
$\mathcal{N}$	A set of edge servers (ESs)	$\mathcal{A}_n$	Feasible set of rental decisions on ES n
$a_{n,t}$	Rental decision on ES n in time slot t	r_t	ASP reward in time slot t
s_t	State of the edge system in time slot t	$o_{n,t}$	Local observation of ES n in time slot t
$\boldsymbol{\alpha}_n$	Localized rental decision of ES n	$\mathcal{B}_n$	One-hop neighbors of ES n
θ_n	Parameters of Q-network on ES n	W	Communication matrix
τ	Iteration index of N_2O	$\boldsymbol{a}'_n(\tau)$	Local copy of system rental decision on ES n
θ_n^μ	Parameters of actor network on ES n	$\mu(\cdot, \theta_n^\mu)$	Actor network on ES n
a_n^μ	Rental decision derived by actor networks	$\tilde{a}_n$	Rental decision derived by N_2O

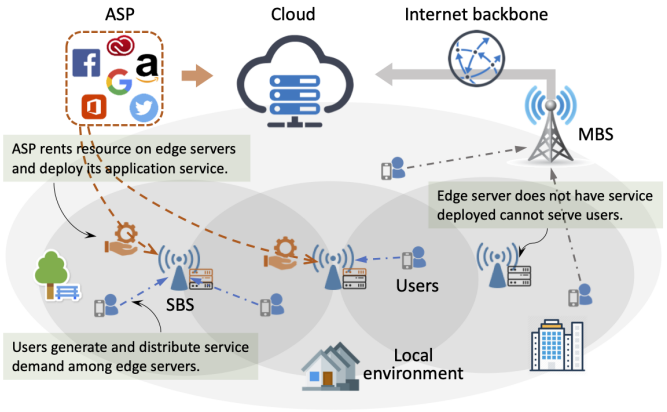

Fig. 2. Illustration of the edge computing system and service provisioning problem. The red part of edge servers denotes the amount of computing resources rented by the ASP. Users that cannot be served by the edge computing system can offload tasks to Cloud via MBS and Internet backbone.

for renting computing resources. In the following, we will show how the resource rental decisions interact with other components in the edge systems and affect the reward of ASP.

1) Service Delay Reduction: The service delay, denoted by $d = d^{\mathtt{tx}} + d^{\mathtt{com}}$, consists of the transmission delay $d^{\mathtt{tx}}$ and computation delay $d^{\mathtt{com}}$. The service delay reduction of edge computing is defined by the gap between the service delay of cloud computing and edge computing: $\Delta = d_{\mathtt{cloud}} - d_{\mathtt{edge}}$. Using edge computing, users can offload tasks to edge servers via one hop wireless link which is much faster than offloading tasks to the cloud server via congested Internet backbone. Therefore, the transmission delay of edge computing is much lower compared to the cloud computing, $d^{\mathtt{tx}}_{\mathtt{edge}} \ll d^{\mathtt{tx}}_{\mathtt{cloud}}$. If the rented computing resources on an edge server is non-zero, then ASP can deploy its application on the edge server and the reduction of transmission delay is realized. The reduction of transmission delay $\Delta_n^{\mathtt{tx}}$ provided by edge server n is $\Delta_n^{\mathtt{tx}}(a_n) = (d^{\mathtt{tx}}_{\mathtt{cloud}} - d^{\mathtt{tx}}_{\mathtt{edge}}) \cdot \mathbf{1}\{a_n \neq 0\}$ where $\mathbf{1}\{\cdot\}$ is an indicator function. The computation delay on an edge server is often a decreasing function of the amount of computing resources on the edge server. In most existing works [37], [38], the computation delay is formulated as an M/M/1 queuing system with expected delay $d^{\mathtt{com}}_{\mathtt{edge}}(a_n) = 1/(a_n \cdot u_{\mathtt{rate}} - \omega_n)$ (where $u_{\mathtt{rate}}$ is the unit processing and ω_n is task arriving rate at edge server n) or a constant-rate model with expected delay $d^{\mathtt{com}}_{\mathtt{edge}}(a_n) = 1/(a_n \cdot u_{\mathtt{rate}})$. Therefore, renting more computing resources on an edge server can provide a larger reduction of computation delay $\Delta_n^{\mathtt{com}}(a_n) = d^{\mathtt{com}}_{\mathtt{cloud}} - d^{\mathtt{com}}_{\mathtt{edge}}(a_n)$, which leads to lower service delay and higher rewards for ASP.

2) Service Demand Processed by Edge Servers: The service demand received by edge servers depends on the demand generation pattern on user-side and also how users distribute their service demand among edge servers.

a) User demand generation: The generation of service demand on user-side is relatively independent of ASP's rental decision. The service demand generated by a user follows a certain demand pattern that may depend on the demographic features of users (e.g., age and gender), the status of mobile devices (e.g., device type and battery level), and other external environmental factors (e.g., location, time, and events). We let $x_m \sim \mathcal{X}_m$ be the service demand generated by user $m \in \mathcal{M}_t$, where $\mathcal{X}_m$ is an *unknown* distribution parameterized by the above mentioned factors. Since the edge servers are geographically distributed, we shall expect that the user populations served by edge servers and their corresponding demand generation patterns are different.

b) Distributing service demand: Distributing service demand is a more important process that determines the amount of service demand received by edge servers. We abstract the distributing process into a mapping function $\mathcal{D} : \mathcal{X}_1 \times \cdots \times \mathcal{X}_{M_t} \to \Omega_1 \times \cdots \times \Omega_N$ from the service demands on user-side $\boldsymbol{x}_t = \{x_m\}_{m \in \mathcal{M}_t}, x_m \in \mathcal{X}_m$ to the service demand received on edge servers $\boldsymbol{\omega}_t = \{\omega_{n,t}\}_{n \in \mathcal{N}}, \omega_{n,t} \in \Omega_n$. Besides our service provisioning problem, quite a lot of other components in the edge computing, e.g., the user behavior, computation offloading, load balancing, radio resource scheduling, will affect the service demand distribution. We cannot precisely characterize the correlations among these components and mathematically model the mapping $\mathcal{D}$. Therefore, we only provide general discussions on the impact of resource rental decisions $\boldsymbol{a}_t$ on distributing the users' service demand. For example, in the computation offloading, users distribute their service demand aiming to minimize the service delay. Recall the impact of resource rental decisions on the service delay discussed previously (i.e., renting more computing resources leads to lower service delay), we could infer that users tend to offload more service demands to edge server n (i.e., a larger ω_n) if more computing resource is rented there (i.e., a larger a_n). Moreover, the users may not know

precisely the service delay the edge servers can provide, and hence they need to learn it from past experience for making offloading decisions [39]. In this case, users can be more willing to offload tasks to edge servers if low service delay is provided in the past, otherwise, users may reduce their reliance on the edge computing system. Therefore, we may need to include the previous rental decisions in the loop, which causes the temporal dependency between resource rental decisions.

3) Cost of Computing Resource Rental: The above discussion indicates that renting more computing resources on edge servers provides lower service delay and also attracts more service demand from users. However, renting more computing resources does not always mean higher rewards for ASP since higher monetary costs are also incurred at the same time. In the commercial edge computing system, the operator sets a price of its computing resources based on a resource pricing scheme whose goal is to maximize the profit of the edge computing system. The resource price is often time-varying depending on the total resource demand and also the competition among multiple service providers. Given the resource price, the rental cost for a service provider $\mathcal{C}: \mathcal{A} \rightarrow \mathbb{R}^+$ is often an *increasing* function of the amount of rented resources, and should be subtracted from the reward. It is possible sometime that the reward of providing edge computing service cannot cover the cost of renting computing resources. Therefore, ASP needs to judiciously decide resource rental decisions in order to maximize its reward.

Based on above discussions, we write the ASP reward, $r_t = \mathcal{R}(\{\boldsymbol{a}_\tau\}_{\tau=1}^t; \{\mathcal{X}_m\}_{m\in\mathcal{M}_t}, \Delta, \mathcal{D}, \mathcal{C})$, as a function of current and previous rental decisions $\{\boldsymbol{a}_\tau\}_{\tau=1}^t$ given users' service demand patterns $\{\mathcal{X}_m\}_{m\in\mathcal{M}_t}$, service delay reduction Δ, service demand distributing policy $\mathcal{D}$, rental cost function $\mathcal{C}$. The reward function is presented in a general form and many other elements in the edge computing system can be added to parameterized the reward function $\mathcal{R}$ based on the implementation scenario. As the rental decisions are temporal-dependent, the goal of ASP is to maximize the time-discounted reward by optimizing the resource rental decisions $\{a_t\}_{t=1}^\infty$:

$$\mathscr{P}1: \quad \max_{\{\boldsymbol{a}_t\}_{t=1}^\infty} \sum\nolimits_{t=1}^{\infty} \gamma^{t-1} r_t, \tag{1a}$$

$$\text{s.t. } r_t = \mathcal{R}\left(\{\boldsymbol{a}_\tau\}_{\tau=1}^t; \{\mathcal{X}_m\}_{m\in\mathcal{M}_t}, \Delta, \mathcal{D}, \mathcal{C}\right), \quad \forall t \tag{1b}$$

$$\boldsymbol{a}_t \in \mathcal{A}, \quad \forall t. \tag{1c}$$

where $\gamma \in [0, 1]$ is a discount factor. The key challenge for solving $\mathscr{P}1$ is the unknown reward function $\mathcal{R}(\cdot)$ and its uncertain parameters $\{\mathcal{X}_m\}_{m\in\mathcal{M}_t}, \Delta, \mathcal{D}, \mathcal{C}$. In particular, directly learning reward function is infeasible due to the time-dependency of resource rental decisions, e.g., when $t \rightarrow \infty$, the size of the input to reward function becomes infinity. In the next section, we will throw $\mathscr{P}1$ into a reinforcement learning problem and solve it with the proposed method.

Remarks on the Reward Function: Note that we do not provide a concrete reward function for ASP. There are two main reasons for this. First, it is extremely difficult to mathematically characterize an ASP reward function without simplifications on the user service demand generation and the complicated user-edge interactions. Second, our method is able to work with any reward functions that ASP may have. Actually, such reward functions are not required since the proposed method is designed based on the framework of deep reinforcement learning.

III. Edge Service Provisioning as a Markov Decision Process

Markov Decision Process (MDP) provides a solid mathematical framework for sequential discrete-time decision-making problems. We use MDP to characterize interactions between the edge computing environment and service provider. In each time slot t, a state is observed that reflects the current status of the edge computing system. The state is partially resulted from resource rental decisions taken before and thereby helping capture the temporal dependency of resource rental decisions. The example state includes the *time* and *location of edge servers* that can help infer the service demand pattern $\mathcal{X}_m$ of nearby users, *the number of connected users connected to SBSs* that affects the amount of service demand received by edge servers, and *the available bandwidth of SBSs* that affects the service delay reduction Δ and offloading policies $\mathcal{D}$. The examples are clearly not exhaustive, many other factors that affect ASP's reward can be included in the state. Although the large and continuous state space lays some difficulties in solving MDP, it is well-handled by *deep reinforcement learning* (DRL) [20]. Next, we first present a basic DRL-based framework that solves the service provisioning problem in a centralized manner.

A. Service Provisioning as a Centralized MDP

We first consider a centralized-MDP formulation by assuming the existence of a central controller that observes the state of all edge servers and the ASP reward. Let $s_t \in \mathcal{S}$ be the state of the edge computing system at the beginning of time slot t. The central controller proactively chooses a system rental decision $\boldsymbol{a}_t \in \mathcal{A}$ based on the observed state s_t and a policy $\pi: \mathcal{S} \rightarrow \mathcal{A}$. The reward of ASP with the system rental decision $\boldsymbol{a}_t \leftarrow \pi(s_t)$ is determined by an unknown reward function $r: \mathcal{S} \times \mathcal{A} \rightarrow \mathbb{R}$. When time slot t ends, the edge computing system transits to a new state s_{t+1} according to a transition $\mathcal{T}: \mathcal{S} \times \mathcal{A} \times S \rightarrow [0, 1]$, $\mathcal{T}(s, \boldsymbol{a}, s') = \Pr(s_{t+1} = s' \mid s_t = s, \boldsymbol{a}_t = \boldsymbol{a})$. Note that the reward function r and transition $\mathcal{T}$ of MDP are given in a general form and hence can represent real-world cases. The goal is to maximize an expectation over the discounted rewards $R_t = r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots$, which is the same to the objective in $\mathscr{P}1$.

1) Centralized Deep Q-Learning: Q-learning is a model-free reinforcement learning algorithm that can be used to learning a decision-making policy for MDP. It defines a Q-function for policy π, $Q^\pi(s, \boldsymbol{a}) = \mathbb{E}[R_t \mid s_t = s, \boldsymbol{a}_t = \boldsymbol{a}]$, which obeys the Bellman equation:

$$Q^\pi(s, \boldsymbol{a}) = \mathbb{E}_{s'\sim\mathcal{T}, \boldsymbol{a}'\sim\pi}\left[r_t + \gamma Q^\pi(s', \boldsymbol{a}') \mid s_t = s, \boldsymbol{a}_t = \boldsymbol{a}\right].$$

Traditional Q-learning uses a Q-table to learn the Q-value for each possible state-action pair, which often suffers from the

notorious problem of *Curse of Dimensionality*. The proposal of Deep Q-Learning (DQL) [20] successfully handles the high-dimension and continuous state space. The core of DQL is to build a deep neural network $Q(s, \boldsymbol{a}; \theta)$, referred to as *Q-network*, to approximate the Q-function, where θ is the parameter vector of the Q-network. DQL uses a greedy policy $\pi(s;\theta) := \arg\max_{\boldsymbol{a}\in\mathcal{A}} Q(s, \boldsymbol{a}; \theta)$ to give the decisions. The training of Q-network $Q(s, \boldsymbol{a}; \theta)$ aims to adjust the parameters θ for reducing the mean-squared error $\mathcal{L}(\theta) = \mathbb{E}_{s,\boldsymbol{a},r,s'}[(y - Q(s, \boldsymbol{a}; \theta)^2]$ where y is the optimal target Q-value. Since the optimal target Q-value is inaccessible, it is substituted with $y = r + \gamma \max_{a'} Q(s', a'; \theta^-)$ where θ^- are the parameters of a target network obtained previously. The training of Q-network follows the standard process of DQL and hence we omitted most details here. The pseudocode for the centralized DQL can be found in online Appendix A [40]. Interested readers are also referred to the reference [20].

2) Limitation of Centralized DQL: Although centralized DQL is theoretically sound, there are several issues to carry it out practically in edge computing systems. **1)** Running centralized DQL requires to collect the state of all edge servers in the edge system for making system rental decisions and training Q-networks. Doing so needs reliable global communications that may not be guaranteed in the distributed edge computing system. **2)** Even with available global communications, frequently sending experience (i.e., a 4-tuple of *state*, *action taken*, *rewards*, *new state*) of edge servers to the central controller for training the Q-network incurs extremely high communication overhead. This process occupies precious spectrum resources in the small-cell network and hence may degrade users' QoS. **3)** Centralized DQL does not scale well for large edge computing systems. The dimension of the state space for centralized MDP increases linearly and the number of system rental decisions increases exponentially with the number of edge servers in the edge computing system. When the number of edge servers becomes large, the central controller needs to build a huge Q-network to approximate the Q-function. This not only requires high-capacity hardware to carry out the training but also incurs long training time for Q-network to converge. To address these issues, we next introduce the multi-agent MDP and multi-agent DQL for service provisioning in edge computing systems.

B. Service Provisioning as Multi-Agent MDP

The multi-agent MDP is featured by partial observability and localized decision-making. Instead of letting ASP pick a system rental decision $\boldsymbol{a}_t \in \mathcal{A}$ in a centralized manner, multi-agent MDP employs a distributed decision-making process where ASP configures a Local Service Manager (LSM) on each edge server $n \in \mathcal{N}$ to decide the local resource rental decision a_n for edge server n. Multi-agent MDP of N edge servers is defined by a set of action spaces $\{\mathcal{A}_n\}_{n=1}^N$ and observation spaces $\{\mathcal{O}_n\}_{n=1}^N$, $\mathcal{O}_n \subseteq \mathcal{S}$, with $\mathcal{A}_n$ and $\mathcal{O}_n$ associated to LSM n. Each LSM n learns a policy $\pi_n : \mathcal{O}_n \to \mathcal{A}_n$. Although the resource rental decision on edge server n is determined independently by LSM n, the reward of LSM n is still correlated to rental decisions on other edge servers. Recall that edge servers share overlapped service area due to the dense deployment of SBSs. In this case, a user that falls in the overlapped area determines its offloading decision based on the rental decisions on all its reachable edge servers. To characterize this correlation, we model the edge computing system using a graph $G = (\mathcal{N}, \mathcal{E})$, where the edge servers $\mathcal{N}$ are the vertices, and there exists an edge $e \in \mathcal{E}$ between two edge servers if they have overlapped service area. The one-hop neighbors of edge server n in graph G are denoted by $\mathcal{B}_n$. Then, the reward r_n of LSM n is a function of the local observation, the rental decision on edge server n, and rental decisions on its neighbor edge servers $i \in \mathcal{B}_n$, $r_n : \mathcal{S} \times \mathcal{A}_n \times_{i\in\mathcal{B}_n} \mathcal{A}_i \to \mathbb{R}$. The reward $r_{n,t}$ on edge server n is only accessible to LSM n and LSMs do not send this information to other LSMs. The local observation evolves according to the transition function $\mathcal{T}_n : \mathcal{O}_n \times \mathcal{A}_n \times_{i\in\mathcal{B}_n} \mathcal{A}_i \times \mathcal{O}_n \to [0, 1]$.

1) Multi-Agent Deep Q-Learning (MA-DQL): In MA-DQL, each LSM n runs DQL independently to maximize its discounted reward, $R_{n,t} = r_{n,t} + \gamma^1 r_{n,t+1} + \gamma^2 r_{n,t+2} + \ldots$. A Q-network $Q_n(o_n, \boldsymbol{\alpha}_n; \theta_n)$ is learned locally by LSM n. The input of $Q_n(o_n, \boldsymbol{\alpha}_n; \theta_n)$ is the local observation o_n and *localized decision* $\boldsymbol{\alpha}_n := \{a_n \cup \{a_i\}_{i\in\mathcal{B}_n}\}$ of LSM n. The optimal localized decision is determined by the greedy policy $\boldsymbol{\alpha}_n^* := \arg\max_{\boldsymbol{\alpha}_n} Q_n(o_n, \boldsymbol{\alpha}_n; \theta_n)$. Although the localized decision of LSM n contains resource rental decisions of nearby LMSs in $\mathcal{B}_n$, LSM n can only control the resource rental decision a_n on edge server n. Therefore, the policy $\pi_n(o_n; \theta_n)$ of LSM n for determining the resource rental decision on edge server n is $\pi_n(o_n; \theta_n) := \left\{a_n \;\middle|\; a_n \in \boldsymbol{\alpha}_n^* = \arg\max_{\boldsymbol{\alpha}_n} Q_n(o_n, \boldsymbol{\alpha}_n; \theta_n)\right\}$. The pseudocode for MA-DQL can be found in online Appendix A [40].

2) Advantages of MA-DQL: MA-DQL addresses several limitations in centralized DQL. **1)** MA-DQL avoids the communication overheads for training Q-networks. In MA-DQL, the experiences $e_t := \{o_{n,t}, \boldsymbol{\alpha}_{n,t}, r_{n,t}, o_{n,t+1}\}$ stored by LSM n for training its Q-network $Q_n(\cdot, \cdot; \theta_n)$ includes the local observation $o_{n,t}, o_{n,t+1}$, reward $r_{n,t}$, and localized rental decision $\boldsymbol{\alpha}_{n,t}$. The local observations and rewards can be directly obtained, and LSM n only needs to observe the resource rental decisions taken by one-hop neighbors to obtain $\boldsymbol{\alpha}_{n,t}$. An LSM does not need experiences of other LSMs to complete training. **2)** MA-DQL scales well for large edge computing systems. The size of edge computing systems does not affect much the structure of Q-networks maintained by LSMs. The dimension of state space $|\mathcal{O}_n|$ is constant for each local Q-network and does not change with the total number of edge servers in the system. The number of actions for a local Q-network depends on the number of its neighbor edge servers, which is also a constant in expectation because edge servers/SBSs are often deployed with a certain density. **3)** MA-DQL has better adaptions to the change of edge computing system. Although the deployment of SBSs and edges servers are often fixed, it is still possible that new edge sites will be added or existing edge sites will be removed from the edge computing system. In such a case, the centralized DQL needs to reconstruct its Q-network and learns a new policy. By contrast, MA-DQL

only needs to modify the local Q-networks on edge servers that have overlapping areas with the added/removed edge server, which is much more efficient compared to centralized DQL.

3) Remarks on the Performance of MA-DQL: In MA-DQL, the training process of Q-network on each edge server follows the standard DQL except that local Q-networks output localized decisions and observe decisions taken by one-hop neighbors as part of experiences. Therefore, the sample complexity, computational complexity, and stability of MA-DQL on local edge servers are similar to that of standard DQL. We give detailed discussions on learning performances of MA-DQL and standard DQL in online Appendix B [40]).

Notice that MA-DQL is only halfway through the full solution. Now, each LSM learns its policy to maximize its own reward instead the reward of ASP (defined as a sum of LSMs' rewards $r_t = \sum_{n=1}^{N} r_{n,t}$). In this case, LSMs actually compete with each other like players in a non-cooperative game, and the rental decisions taken by LSMs resemble a Nash-equilibrium. In the next section, we proposed an orchestration scheme to coordinate locally learned Q-networks, such that LSMs could work cooperatively toward the maximization of ASP reward.

IV. Distributed Neural Networks Orchestration

Let $Q_n(o_n, \boldsymbol{\alpha}_n, \theta_n)$ be the Q-network learned by LSM n. For ease of the exposition, the time index t is omitted because our orchestration process is confined in a single time slot. Based on the definition of Q-values and the objective defined in $\mathscr{P}1$, the goal of LSMs in each time slot is collaboratively optimizing the system rental decision $\boldsymbol{a}$ to maximize the sum of local Q-values:

$$\mathscr{P}2: \quad \max_{\boldsymbol{a}=\{a_1,\ldots,a_N\}} \ \frac{1}{N}\sum\nolimits_{n=1}^{N} Q_n(o_n, \boldsymbol{\alpha}_n; \theta_n), \tag{2a}$$

$$\text{s.t. } \boldsymbol{\alpha}_n = \{a_n \cup \{a_i\}_{i\in\mathcal{B}_n}\}, \quad a_n \in \mathcal{A}_n, \ \forall n \in \mathcal{N}. \tag{2b}$$

Our goal is to solve $\mathscr{P}2$ with communication constraints imposed by the graph G — LSM n has its access to only the Q-network $Q_n(\cdot,\cdot,\theta_n)$ learned locally and communicate only with its immediate neighbors $i \in \mathcal{B}_n$. We propose a distributed orchestration scheme for deep neural networks, called Neural Network Orchestration (N_2O), to offer a distributed solution to $\mathscr{P}2$.

N_2O is inspired by distributed dual averaging [41] which is originally designed for distributed optimization of convex functions. However, Q-Networks are non-convex in most cases, and N_2O is particularly designed for coordinating non-convex Q-networks in a distributed manner. It utilizes the structural information of graph G and scales well for large edge computing systems.

A. Neural Network Orchestration (N_2O)

In N_2O, each LSM keeps a copy of the system rental decision $\boldsymbol{a}'_n = \{a'_{n,i}\}_{i\in\mathcal{N}}$, and the copy $\boldsymbol{a}'_n$ is only accessible to LSM n. The algorithm runs in an iterative manner, at each iteration τ, there are N pairs of vectors $(\boldsymbol{a}'_n(\tau), \boldsymbol{z}_n(\tau)) \in \mathcal{A}\times\mathbb{R}^N$ with the n-th pair associated with LSM n. To update the vector pair $(\boldsymbol{a}'_n(\tau), \boldsymbol{z}_n(\tau))$, each LSM n computes the partial differentiation $g_n(\tau) = -\partial Q_n(o_n, \boldsymbol{\alpha}'_n(\tau), \theta_n)/\partial \boldsymbol{a}'_n(\tau)$ of the local Q-function, where $\boldsymbol{\alpha}'_n(\tau) = \{a'_{n,n}(\tau) \cup \{a'_{n,i}(\tau)\}_{i\in\mathcal{B}_n}\}$, and receives information about the parameter $\boldsymbol{z}_i(\tau), i \in \mathcal{B}_n$ associated with LSM i in its neighborhood $\mathcal{B}_n$. These parameters are combined through a weighting process. Let $W \in \mathbb{R}^{N\times N}$ be a matrix of non-negative weights that respects the structure of graph G. For $m, n \in \mathcal{N}$, and $(m,n) \in \mathcal{E}$, we have $W_{m,n} > 0$. We let W be a doubly stochastic matrix, meaning that $\sum_{m\in\mathcal{N}} W_{m,n} = \sum_{m\in\mathcal{B}_n} W_{m,n} = 1, \forall n \in \mathcal{N}$, and $\sum_{n\in\mathcal{N}} W_{m,n} = \sum_{n\in\mathcal{B}_m} W_{m,n} = 1, \forall m \in \mathcal{N}$. With these variables, LSM n updates $(\boldsymbol{a}'_n(\tau), \boldsymbol{z}_n(\tau))$ as:

$$\boldsymbol{z}_n(\tau+1) = \sum\nolimits_{m\in\mathcal{B}_n} W_{m,n}\boldsymbol{z}_m(\tau) + g_n(\tau) \tag{3a}$$

$$\boldsymbol{a}'_n(\tau+1) = \Pi_{\mathcal{A}}^{\psi_n}(\boldsymbol{z}_n(\tau+1), \beta(\tau)) \tag{3b}$$

Each LSM n first computes $\boldsymbol{z}_n(\tau+1)$ from a weighted average of its own gradient $g_n(\tau)$ and the variables $\{\boldsymbol{z}_m(\tau)\}_{m\in\mathcal{B}_n}$ of its neighbors. Then $\boldsymbol{a}'_n(\tau+1)$ is computed by a projection $\Pi_{\mathcal{A}_n}^{\psi_n}$ with a positive stepsize $\beta(\tau) > 0$. The sequence $\{\beta(\tau)\}_{\tau=0}^{\infty}$ should be non-increasing, and the projection $\Pi_{\mathcal{A}}^{\psi_n}$ for each LSM n is defined by:

$$\Pi_{\mathcal{A}}^{\psi_n}(\boldsymbol{z}, \beta) = \arg\min_{\boldsymbol{a}\in\mathcal{A}} \left\{ \langle \boldsymbol{z}, \boldsymbol{a}\rangle + \frac{1}{\beta}\psi_n(\boldsymbol{a}) \right\} \tag{4}$$

where $\psi_n : \mathcal{A} \to \mathbb{R}$ is a convex auxiliary function that satisfies the following requirements: 1) $\psi_n(\cdot) \geq 0$ over $\mathcal{A}$; 2) $\psi_n(\boldsymbol{a})$ and $\nabla\psi_n(\boldsymbol{a})$ is bounded over $\mathcal{A}$, i.e., $\psi_n(\boldsymbol{a}) \leq \psi_n^{\max}$ and $\nabla\psi_n(\boldsymbol{a}) \leq \psi_n'^{,\max}, \forall \boldsymbol{a} \in \mathcal{A}$; 3) $-Q_n(o_n, \boldsymbol{\alpha}_n, \theta_n) + \frac{1}{\beta(0)}\psi_n(\boldsymbol{a}), \boldsymbol{\alpha}_n \subseteq \boldsymbol{a}, \forall \boldsymbol{a} \in \mathcal{A}$, is strongly convex. These requirements are not strict, such auxiliary functions can be easily constructed. A simple example that satisfies all the above requirement is $\psi_n = \frac{\gamma_n}{2}\|\boldsymbol{a} - \boldsymbol{c}_n\|^2$ with a positive constant γ_n and a constant vector $\boldsymbol{c}_n$ vector. Clearly, $\psi_n(\cdot) \geq 0$ holds true, and the second requirement is also satisfied considering a finite action set $\mathcal{A}$. For the third requirement, if the constant γ_n is chosen large enough, we are able to make $Q_n(o_n, \boldsymbol{\alpha}, \theta_n) + \psi_n(\boldsymbol{a})$ strongly convex. The pseudo-code of N_2O is presented in Algorithm 1. Running N_2O only requires local communications.

Algorithm 1 Neural Network Orchestration (N_2O)

Input: Local Q-Networks $Q_n(\cdot,\cdot,\theta_n), \forall n$, communication matrix W, auxiliary functions $\psi_n(\cdot), \forall n$, local observations $o_n, \forall n$.
Initialization: $\boldsymbol{z}_n(1) = \boldsymbol{0}, \forall n$, $\beta(1) = 1$, $\boldsymbol{a}'_n(1) = \Pi_{\mathcal{A}}^{\psi_n}(\boldsymbol{z}_n(1), \beta(1))$
for $\tau = 1, 2, \ldots, T$ **do**
  for each LSM $n \in \mathcal{N}$ **do**
    Calculate the partial differentiation: $g_n(\tau) = -\partial Q_n(o_n, \boldsymbol{\alpha}'_n(\tau), \theta_n)/\partial \boldsymbol{a}'_n(\tau)$;
    Receive $\boldsymbol{z}_m(\tau)$ from one-hop neighbors $m \in \mathcal{B}_n$.
    Update $\boldsymbol{z}_n(\tau+1) = \sum_{m\in\mathcal{B}_n} W_{m,n}\boldsymbol{z}_m(\tau) + g_n(\tau)$ and $\boldsymbol{a}'_n(\tau+1) = \Pi_{\mathcal{A}}^{\psi_n}(\boldsymbol{z}_n(\tau+1), \beta(\tau))$
    Broadcast $\boldsymbol{z}_n(\tau+1)$ to its one-hop neighbors
  end for
end for

Remarks on Information Exchange of N_2O During Implementation: Recall that our edge computing system is constructed on small-cell networks, where an edge server is collocated a small-cell base station. The wireless message passing between base stations often exists to gather information regarding the arrangement of nearby base stations for facilitating user handovers, spectrum allocation, and coverage optimization. Therefore, we do not need a dedicated communication scheduling component for N_2O, the information to be exchanged for running N_2O can be included in messages that are commonly transmitted between base stations.

B. Performance Analysis

To carry out the performance analysis of N_2O, we define the *L-Lipschitz* condition of Q-networks with respect to the same norm $\|\cdot\|$, i.e., $\forall \boldsymbol{\alpha}_n, \tilde{\boldsymbol{\alpha}}_n, \forall n$.

$$|Q_n(o_n, \boldsymbol{\alpha}_n; \theta_n) - Q_n(o_n, \tilde{\boldsymbol{\alpha}}_n; \theta_n)| \leq L\|\boldsymbol{\alpha}_n - \tilde{\boldsymbol{\alpha}}_n\|, \quad (5)$$

holds true. The *L-Lipschitz* condition implies that for any $\boldsymbol{\alpha}_n$ and any gradient $g_n = \partial Q_n(o_n, \boldsymbol{\alpha}_n; \theta_n)/\partial \boldsymbol{a}_n$, we will have $\|g_n\|_* \leq L$, where $\|\cdot\|_*$ denotes the dual norm to $\|\cdot\|$, defined by $\|v\|_* := \sup_{\|u\|=1}\langle v, u\rangle$. The *L-Lipschitz* condition exists for deep neural networks and its parameter L can be measured using techniques in [42]. Note that the *L-Lipschitz* condition is only used for analyzing the performance of N_2O and we do not need the parameters in *L-Lipschitz* condition to run our algorithm. In the sequel, we show the theoretical performance guarantees of N_2O.

1) Basic Convergence Result: We first give the basic convergence result of local decision sequence $\{\boldsymbol{a}'_n(\tau)\}_{\tau=1}^T$ to the optimum of $\mathscr{P}_2$ via the *running average*, $\bar{\boldsymbol{a}}'_n(T) = \frac{1}{T}\sum_{\tau=1}^T \boldsymbol{a}'_n(\tau)$. This value is locally defined at each LSM n and can be computed in a distributed manner. The convergence result of N_2O provides a decomposition of the error into an optimization error term, a cost associated with network communications, and a penalty caused by the non-convexity of Q-networks. To state the theorem, we define the *average dual variable* $\bar{\boldsymbol{z}}(\tau) := \frac{1}{N}\sum_{n=1}^N \boldsymbol{z}_n(\tau)$.

Theorem 1 (Basic Convergence): Let $\{\boldsymbol{a}'_n(\tau)\}_{\tau=0}^T$ and $\{\boldsymbol{z}(\tau)\}_{\tau=0}^T$ be the sequences generated according the updates defined in (3). For any $\boldsymbol{a}^* \in \mathcal{A}$ and for any decision sequence $\{\boldsymbol{a}'_i(\tau)\}_{\tau=1}^T$ of LSM $i \in \mathcal{N}$, we have

$$\frac{1}{N}\sum_{n=1}^N Q_n(o_n, \boldsymbol{\alpha}^*; \theta_n) - \frac{1}{N}\sum_{n=1}^N Q_n(o_n, \bar{\boldsymbol{\alpha}}'_i(T); \theta_n) \leq \textsc{Opt} + \textsc{Comm} + \textsc{Nonc},$$

with OPT, COMM, and NONC defined as,

$$\textsc{Opt} = \frac{1}{T\beta(T)}\psi(\boldsymbol{a}^*) + \frac{L^2}{2T}\sum_{\tau=1}^T \beta(\tau-1),$$

$$\textsc{Comm} = \frac{L}{T}\sum_{\tau=1}^T \beta(\tau) \cdot \left[\frac{2}{N}\sum_{n=1}^N \|\bar{\boldsymbol{z}}(\tau) - \boldsymbol{z}_n(\tau)\|_* + \|\bar{\boldsymbol{z}}(\tau) - \boldsymbol{z}_i(\tau)\|_*\right],$$

$$\textsc{Nonc} = \frac{2}{N\beta(T)}\sum_{n=1}^N \psi_n^{\max} + \frac{d^{\max}}{N\beta(T)}\sum_{n=1}^N \psi_n^{\prime,\max}.$$

where $d^{\max} := \arg\max_{\boldsymbol{a},\boldsymbol{a}'} \|\boldsymbol{a} - \boldsymbol{a}'\|_*, \forall \boldsymbol{a}, \boldsymbol{a}' \in \mathcal{A}$.

Proof: See Appendix C in supplementary materials [40]. □

The above theorem indicates that after running the N_2O algorithm for T iterations, every LSM n has access to a locally defined $\bar{\boldsymbol{a}}'_n(T)$ which guarantees that the difference $\frac{1}{N}\sum_{n=1}^N [Q_n(o_n, \boldsymbol{\alpha}^*; \theta_n) - Q_n(o_n, \bar{\boldsymbol{\alpha}}'_i(T); \theta_n)], \forall i \in \mathcal{N}$ is upper bounded by a sum of three terms. The first term OPT is the optimization error caused by gradient based algorithms. The second term COMM is the error caused by the different decision copies maintained at different LSMs. The third term NONC is caused by the non-convexity of Q-networks. As long as the bound of deviation $\|\bar{\boldsymbol{z}}(\tau) - \boldsymbol{z}_i(\tau)\|_*$ is tight enough and $\beta(\tau)$ is appropriately chosen, the error of $\bar{\boldsymbol{a}}'_i(T)$ is small uniformly across all LSMs. Next, we will provide a more precise statement of its convergence rates.

2) Convergence Rate and Network Topology: We next show how the network topology affects the convergence rates of N_2O. Let us first consider static network topology where the communication occurs via a fixed doubly stochastic weight matrix W at every iteration. The following result shows that the convergence rate of N_2O is determined by the *spectral gap* $1 - \sigma_2(W)$ of the matrix W, where $\sigma_2(W)$ is the second largest singular[1] value of W.

Theorem 2 (Convergence Rates): Given the conditions and definitions in Theorem 1, setting the step size $\beta(\tau) = \tau^{-1/2}$, the convergence rate of N_2O is $\mathcal{O}\left(\frac{L^2\log(T\sqrt{N})}{\sqrt{T}(1-\sigma_2(W))}\right)$.

Proof: See Appendix D in supplementary materials [40]. □

Theorem 2 establishes a connection between the convergence rate of N_2O and the spectral properties of the underlying graph G. The inverse dependence on the the spectral gap $1 - \sigma_2(W)$ is natural since it is well-known to determine the rates of mixing in random walks on graph [44], and the propagation of information in N_2O is tied to the random walk on underlying graph with transition probabilities specified by W (more detailed explanations are given in the proof). Using Theorem 2, we can derive explicit convergence rate for several types of graphs that can be used to model the edge computing network.

Corollary 1: Under the condition of Theorem 2, we have following convergence rates of N_2O:
a) For k-connected $\sqrt{N}$-by-$\sqrt{N}$ grids: $\mathcal{O}\left(\frac{L^2}{\sqrt{T}}\frac{N\log(T\sqrt{N})}{k^2}\right)$.
b) For random geometric graphs with connectivity radius $r = \Omega\left(\sqrt{\log^{1+\epsilon} N/N}\right)$ for any $\epsilon > 0$, with high probability: $\mathcal{O}\left(\frac{L^2}{\sqrt{T}}\frac{N\log(T\sqrt{N})}{\log N}\right)$.

Proof: See Appendix E in supplementary materials [40]. □

Up to the logarithmic factor, the convergence rate is of the order $L^2/\sqrt{T}$, and the remaining terms vary depending on the size and topology of the underlying graph G.

C. N_2O With Stochastic Communication Links

Next, we consider running N_2O with the stochastic communication. Such stochastic communication is of interest for

[1]The largest singular value $\sigma_1(W)$ is 1 since W is doubly stochastic [43].

many reasons. For example, the network operator may want to reduce the spectrum usage for information exchange during a certain time; or the communication links may sometimes fail during the execution of N_2O. The stochastic communication is characterized by a time-varying and random communication matrix $W(\tau)$ — the matrix $W(\tau)$ is potentially different for each iteration τ and randomly chosen.

The following theorem provides a convergence result for the case of time-varying random communication matrices. In particular, it applies to sequence $\{\boldsymbol{a}'_n(\tau)\}_{\tau=0}^{\infty}$ and $\{\boldsymbol{z}_n(\tau)\}_{\tau=0}^{\infty}$ generated by update (3) with step size $\beta(\tau)_{\tau=0}^{\infty}$, but in which W is replaced with $W(\tau)$.

Theorem 3: Let $\{W(\tau)\}_{\tau=0}^{\infty}$ be an independent and identically distributed sequence of double stochastic matrices. For any $\boldsymbol{a}^* \in \mathcal{A}$, with probability at least $1 - 1/T$, we have

$$\begin{aligned}\frac{1}{N}\sum\nolimits_{n=1}^{N}&\left[Q_n(o_n,\boldsymbol{\alpha}^*;\theta_n) - Q_n(o_n,\bar{\boldsymbol{\alpha}}'_i(T);\theta_n)\right]\\ &\leq \frac{1}{T\beta(T)}\psi(\boldsymbol{a}^*) + \frac{L^2}{2T}\sum\nolimits_{\tau=1}^{T}\beta(\tau-1)\\ &\quad + \frac{3L^2}{T}\left(\frac{6\log(T^2N)}{1-\lambda_2} + \frac{1}{T\sqrt{N}} + 2\right)\sum\nolimits_{\tau=1}^{T}\beta(\tau)\\ &\quad + \frac{2}{N\beta(T)}\sum\nolimits_{n=1}^{N}\psi_n^{\max} + \frac{d^{\max}}{N\beta(T)}\sum\nolimits_{n=1}^{N}\psi_n'^{,\max}.\end{aligned}$$

where λ_2 is the second largest eigenvalue of $\mathbb{E}[W(\tau)^\top W(\tau)]$.

Proof: See in Appendix F in supplementary materials [40]. □

Based on the result stated in Theorem 3, if we let the stepsize $\beta(\tau) = \tau^{-\frac{1}{2}}$, the convergence rate becomes $\mathcal{O}\left(\frac{L^2}{\sqrt{T}}\frac{\log(T\sqrt{N})}{1-\lambda_2}\right)$. The convergence rate for the stochastic communication is directly comparable to the convergence rate for the fixed communication matrices.

V. Knowledge Distilling for Neural Network Orchestration

N_2O requires LSMs to iteratively communicate with each other for deriving resource rental decisions in a distributed manner. The information exchanged during this process incurs non-negligible communication overhead which is unfavorable for spectrum saving and fast decision-making. In this section, we employ *knowledge distilling* technique to avoid the communication overhead. Knowledge distilling is originally proposed for deep neural network (DNN) compression which aims to transfer the knowledge from a cumbersome DNN to a smaller DNN that is less computation-prohibitive [45]. It is also utilized for facilitating DNN ensembles where the knowledge acquired by a large ensemble of DNNs is extracted and squeezed into a small DNN that is suitable for deployment [32]. In our problem, we aim to distill the knowledge generated by N_2O and store it in a DNN. To be specific, an actor network will be trained locally for each LSM whose output approximates the decision derived by N_2O. The actor network of LSM n takes local observation o_n as input and directly infers a localized decision for the LSM without information exchange with nearby LSMs.

A. Actor Neural Network

We use $\tilde{\mu}\left(s;\{\theta_n\}_{n\in\mathcal{N}}\right) : \mathcal{S} \rightarrow \mathcal{A}$ to abstract the process of N_2O. Note that although the formulation of $\tilde{\mu}\left(s;\{\theta_n\}_{n\in\mathcal{N}}\right)$ indicates the accessibility to all Q-networks $\{\theta_n\}_{n\in\mathcal{N}}$ and the system state $s \in \mathcal{S}$, each LSM actually only accesses its partial observation and local Q-network when running N_2O. The actor network for LSM n is denoted by $\mu_n(o_n;\theta_n^\mu)$, where θ_n^μ is the parameter vector of the actor network. The resource rental decision decided by the actor network is denoted by $a_n^\mu = \mu_n(o_n;\theta_n^\mu), \forall n \in \mathcal{N}$. Let $\tilde{\boldsymbol{a}} = \{\tilde{a}_n\}_{n=1}^N, \tilde{a}_n \in \mathcal{A}_n, \forall n$ be the distributed solution derived by N_2O, i.e., $\tilde{\boldsymbol{a}} = \tilde{\mu}(s;\{\theta_n\}_{n\in\mathcal{N}})$. An "ideal" actor is expected to give a decision a_n^μ that is same to $\tilde{a}_n$. However, it is unlikely that an ideal actor can be trained due to the fact that each LSM n learns only with its partial observation o_n while N_2O allows LSMs to communicate with each other and negotiate an optimal solution based on the observations of all LSMs. This information inequality determines that the actor trained solely by an LSM cannot precisely duplicate the decision of N_2O. Nevertheless, to maintain a distributed framework of our method, we stick to knowledge distilling with partial observations. The effectiveness of such an approach is reasoned by the correlation of LSMs' observations. The correlation of observations is a very mild assumption that often holds true especially for nearby LSMs. This is because the edge servers, on which these LSMs are configured, share overlapped service areas due to the dense deployment, thereby making the user population and service demand highly correlated. With correlated observations, an LSM may, to some extent, infer the decisions of nearby LSMs and pick a rental decision for maximizing the system reward.

B. Orchestration Policy Distilling

The overall framework for knowledge distilling is illustrated in Fig. 3. The essence of the actor network is a regression module that predicts the output of N_2O based on the local observation. The training of actor network is realized by the stochastic gradient decent with online experience collection. Suppose in time slot t, LSMs run N_2O and derive the distributed solution $\tilde{\boldsymbol{a}}_t$. Then, each LSM n collects experience $e_t = (o_{n,t}, \tilde{a}_{n,t})$ of time slot t in dataset $\mathcal{V}_n = \mathcal{V}_n \cup \{e_t\}$ ($\mathcal{V}_n$ is stored locally on edge server n). In the actor network training, we apply updates of θ_n^μ on samples (mini-batch) of experience $(o_n, \tilde{a}_n) \sim U(\mathcal{V}_n)$, drawn uniformly at random from $\mathcal{V}_n$. The objective of the parameter update is to minimize the loss function:

$$L_n(\theta_n^\mu) = \mathbb{E}_{(o_n,\tilde{a}_n)\sim U(\mathcal{V}_n)}\left[(\tilde{a}_n - \mu_n(o_n,\theta_n^\mu))^2\right] \quad (6)$$

The update of the actor parameter can be calculated as $\theta^\mu = \theta^\mu + \delta\nabla_{\theta^\mu}L(\theta^\mu)$, where δ is the learning rate. The pseudo-code for training actor networks is given in the online Appendix A [40]. During the training of actors, N_2O is still performed to acquire the target solution $\tilde{\boldsymbol{\alpha}}$, and therefore knowledge distillation does not help reduce the communication overhead in this phase. Once actors are trained, LSMs can directly use actors to give a local resource rental decision, and in this case, the communication overhead for running N_2O is avoided.

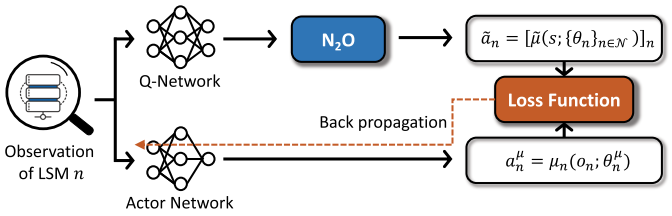

Fig. 3. Illustration of orchestration policy distilling.

Our original goal is to acquire actors that have comparable performance to that of N_2O. After using the knowledge distillation, we surprisingly find that the distilled policy achieves even higher rewards than N_2O. The rationale behind this seemingly abnormal phenomenon is that the trained actor network is more stable to the gradient oscillation. To be specific, the Q-network, as an approximation of the Q-function, is never smooth. It often has jagged surfaces that lead to gradient oscillations. It is possible that for a certain observation o_n, the gradient of Q-network, i.e., $g_n = \partial Q_n(o_n, \boldsymbol{\alpha}_n; \theta_n)/\partial \boldsymbol{a}_n$, oscillates across the localized decisions. Relying on the heavily oscillating gradients, N_2O may fail to converge to a good solution. During knowledge distilling, the actors are trained based on the experience of N_2O. There can be a few samples that N_2O does not converge well due to gradient oscillation at certain observations. However, the impact of these samples is alleviated by nearby samples (i.e., samples with similar observations) that have relatively smooth gradients and converge well. The generalization ability of deep neural networks makes the actor more robust to the gradient oscillation.

VI. Experiments and Results

A. Experimental Setup

1) Edge Computing System: The simulated environment of the edge computing system is provided in the supplementary materials [40]. We implement our method on edge computing systems with different sizes, ranging from 4 to 25 edge servers. These edge servers are deployed in a grid layout with a grid interval of 60m. The maximum communication radius of an edge server is 85m and therefore the service areas of edge servers are overlapped. The length of each time slot (i.e., the decision cycle of service provisioning) is 10 minutes.

The service demand generated at users is affected by two main factors, *the location of users* and *the time of the day*. The service areas are categorized into four types: residential zone, school zone, commercial zone, and public zone. The users in different types of areas have different demand patterns across the time of the day. For example, the expected service demand for a user in the residential zone from 8 p.m. to 9 p.m. is 22.5 tasks while at the same time the expected service demand for a user in the school zone is 3.0 tasks. The users in the edge system move based on a random walk process with existing users disappearing and new users appearing randomly. The expected number of users connected to an edge server in a time slot is 30. A user can offload tasks to edge servers within the communication range. The offloading policy of users aims to minimize the expected service delay, i.e., a sum of transmission delay and computation delay. The transmission delay depends on the wireless channel condition which is modeled by the free space path-loss with Rayleigh fading. The users estimate the expected computation delay of an edge server by averaging the previously experienced computation delays. The computation delay on an edge server is modeled as an M/M/1 queuing system: $d_{\texttt{edge}}^{\texttt{comp}} = 1/(a \cdot u_{\texttt{rate}} - \omega)$, where a is the resource rental decision on the edge server, $u_{\texttt{rate}}$ is the processing rate of unit computing resource, and ω is the amount of service demand received by the edge server. There are 3 resource rental decisions available on each edge server $\mathcal{A}_n = [0, 1, 2], \forall n$. The cost of the rented computing resources on edge server n is determined by the function $\texttt{cost} = p_n^{\text{unit}} \cdot a_n$ where p_n^{unit} is the unit price randomly picked from $[20, 40]$ in each time slot t.

The state of an edge server includes: 1) time of the day, 2) the number of users connected to the edge server, 3) available spectrum at the edge server, 4) previous computation delay of the edge server, 5) unit price of the computing resource. These states can be easily acquired and very related to the reward of edge servers. The time of the day determines the service demand generated by users. The number of connected users and the available spectrum together determine the bandwidth assigned to users which affects channel condition and transmission delay. The previous computation delay influences users' offloading decisions (if the previously experienced delay is large, then the user will offload less tasks to the edge server) and affects the amount of service demand received by edge servers.

2) Hyper-Parameters of Q-Networks: We use manual search for determining the hyper-parameters of Q-networks. For centralized DQL, we construct a 5-layer Q-network. The input layer (first layer) of centralized Q-network includes observations of all LSMs s ($5 \cdot N$ nodes, the length of the state vector for each edge server is 5). The output layer (last layer) outputs the Q-values of all possible actions (3^N nodes). Layer 2 to layer 4 are fully-connected layers with 256, 256, and 128 nodes, respectively. In multi-agent DQL, each LSM $n \in \mathcal{N}$ learns a local Q-network which is also a 5-layer deep neural network but much smaller than the centralized Q-network. The input layer of local Q-network n includes the observation of edge server n, o_n (5 nodes), and the resource rental decisions of LSM n and its one-hop neighbors, $\boldsymbol{\alpha}_n$ ($|\mathcal{B}_n| + 1$ nodes). The output of local Q-network is the Q-value (one node) given the partial observation o_n and localized rental decision $\boldsymbol{\alpha}_n$. Three fully-connected layers of the local Q-network (layer 2 to layer 4) have 8, 32, 8 nodes, respectively.

The auxiliary function used by N_2O is $\psi_n = \frac{\gamma_n}{2}\|\boldsymbol{a} - \boldsymbol{c}_n\|^2$, where $\gamma_n = 10, \forall n$ and $\boldsymbol{c}_n$ is the resource rental decision determined by Q-network n locally before running N_2O.

B. Results and Evaluations

1) Heterogeneity of Edge Sites: We first show the service provisioning policy learned by multi-agent DQL. Fig. 4 depicts resource rental decisions taken by four edge servers under various system states (two kinds of state information, the time of the day and unit resource price, are used in the figure).

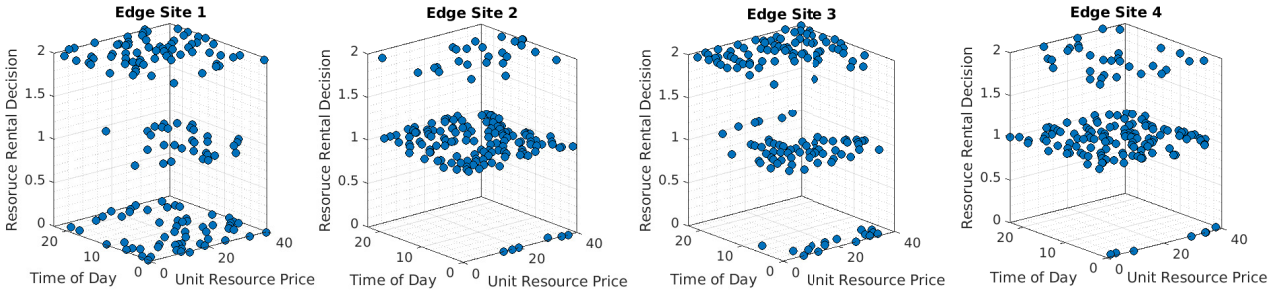

Fig. 4. Resource rental decisions at different edge sites.

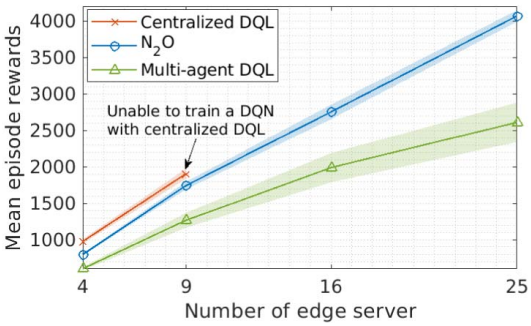

Fig. 5. Comparison on mean episode rewards.

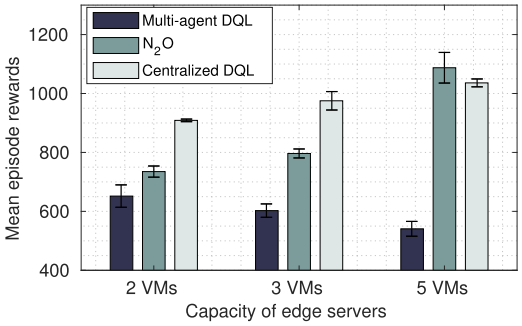

Fig. 6. Impact of resource rental decisions.

We can see that resource rental decisions at different edge sites exhibit noticeable differences, and therefore considering the heterogeneity of edge sites is very necessary.

2) Comparison on Mean Episode Rewards: We first compare the ASP reward $R_t = \sum_{n=1}^{N} r_{n,t}$ achieved by centralized DQL, multi-agent DQL, and N_2O. The results are shown in Fig. 5 which depicts the mean episode (each episode contains 6 time slots) reward of ASP $1/T \sum_{t=1}^{T} R_t$ and the standard value. As expected, the centralized DQL achieves the highest reward since it collects the system-wide state and learns to maximize ASP's reward over the edge computing system. The mean episode reward of multi-agent DQL is 62.06% of that achieved by centralized DQL. By running our orchestration algorithm N_2O, we are able to increase the mean episode reward to 91.39% of the reward achieved by centralized DQL.

It is worth noticing that although centralized DQL is able to achieve the highest system reward, it is not applicable in all cases even with a central controller and global communications. As can be observed in Fig. 5 that when the number of edge servers is larger than 9, we are not able to train a centralized Q-network due to technical difficulties. For example, if there are 16 edge servers with each edge server having 3 available resource rental decisions, then the number of possible actions for centralized DQL becomes $3^{16} = 43,046,721$ which is too large to be included in a single Q-network. Fig. 6 shows the mean episode reward achieved by centralized DQL, multi-agent DQL, and N_2O with different computing capacities on edge servers. The number of edge servers is 4 in this experiment setting and the number of virtual machines (VMs) on each edge server varies from 2 to 5. For centralized DQL and N_2O, they are able to achieve higher rewards when the computing capacity of edge servers is larger. This is because larger computing capacity provides more available rental decisions, which makes the service provisioning on the edge server more flexible for ASP. For multi-agent DQL, the reward decreases with the increase of computing capacity on edge servers. This is because increasing the flexibility of resource rental for each LSM makes the competition among LSMs more intense and therefore causing the reduction of total reward. It is worth noticing that when the number of available rental decisions for each edge server becomes 5, N_2O can even achieve higher rewards than centralized DQL. This is because centralized DQL cannot learn well when the action space is too large while N_2O is still efficient due to its good scalability.

3) Analysis of Algorithm Complexity: Table II further compares the complexity of centralized DQL and N_2O in terms of state space, action space, communication overhead, and memory requirement. These values are given with the configuration of 9 edge servers and 3 available rental actions per edge server. The dimension of state space is $5 \cdot 9 = 45$ for centralized DQL and 5 for N_2O and multi-agent DQL. The dimension of state space increases linearly with the number of edge servers for centralized DQL and stays constant for N_2O. The linear dependency of the state space dimension on the number of edge servers is still manageable for centralized DQL. The key difficulty for centralized DQL is that the number of actions increases exponentially with the number of edge servers in the edge computing system. By contrast, the number of actions for N_2O is only related to the number of its neighbor edge servers which is often small (3 neighbor edge servers in this experiment setting). A large action space not only poses a high resource requirement for training Q-networks but also causes slow convergence during training. As can be observed

TABLE II
COMPARISON OF CENTRALIZED DQL AND N_2O

Metrics \ Methods	Centralized DQL	N_2O
State space dimension	45	5
Number of actions	19,683	81
Communication overhead	504 Byte (global comm.)	450 Byte (local comm.)
Memory requirement	106 MB	4.2 MB

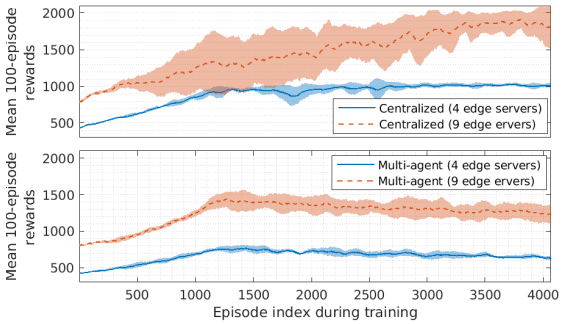

Fig. 7. Comparison of the training convergence.

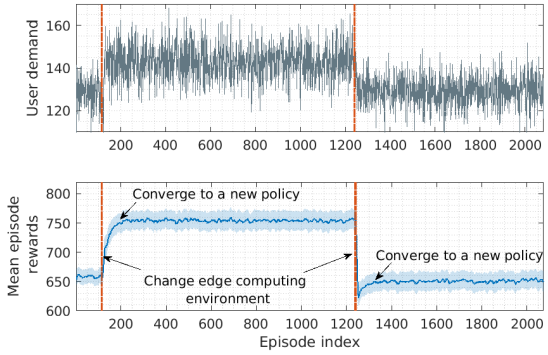

Fig. 8. Adaption to edge environment change.

in Fig. 7, when the number of edge servers is 9, the training of centralized DQL cannot converge fast. By contrast, increasing the number of edge servers does not affect the convergence of N_2O/multi-agent DQL.

The communication overheads for centralized DQL and N_2O given in Table II are both small (around 500 Byte). However, centralized DQL requires global communication where messages are transmitted over multi-hop wired connections. This often incurs much higher delay (due to multi-hop routing) compared to local transmission used by N_2O. Global communication also causes heavier congestion in the backhaul link. In addition, running N_2O requires 4.2 MB, only 3% of that used by centralized DQL.

4) Adaption to Environment Changes: The readers may argue based on the result in Fig. 7 that DQL takes a long time (1,200 episodes) to learn a Q-network for decision-making and therefore when the underlying environment changes the learner may need to learn a new Q-network, which becomes extremely

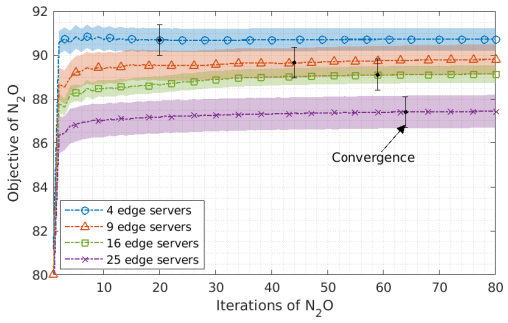

Fig. 9. N_2O convergence v.s. Number of edge servers.

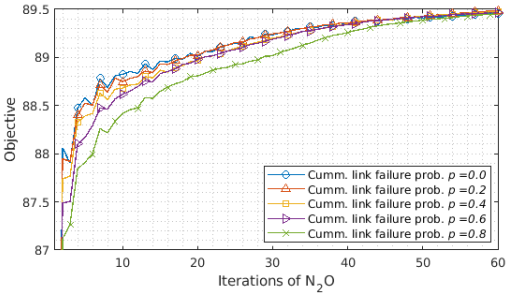

Fig. 10. N_2O convergence with stochastic communications.

inefficient. We would like to mention that the 1,200 episodes do not necessarily mean the learning is slow since we set a long exploration phase (30% of the total number of episodes) in the training process. One may shorten the exploration phase to reduce reward loss. Second, in Fig. 7, the learner learns the Q-network from scratch. However, when you already have a learned Q-network, adapting the Q-network to a new environment can be faster. Fig. 8 shows the adaption process of multi-agent DQL when the edge computing environment changes. We randomly change the users' service demand pattern at 270-th and 1400-th episode. It can be observed that N_2O is able to adapt its Q-networks to the new environment quickly.

5) Convergence Analysis of N_2O : Next, we analyze the convergence performance of N_2O. Fig. 9 shows the convergence of N_2O with different numbers of edge servers in the edge computing system. Overall, we see that the number of edge servers does not have a significant influence on the convergence of N_2O. This result can be expected as it has been shown in the theoretical analysis (Theorem 2) that the convergence rate is only $\log(\sqrt{N})$-dependent on the number of edge servers N. However, we can still see that increasing the number of edge servers slightly delays the convergence of N_2O.

Fig. 10 shows the convergence of N_2O with stochastic communication links. We let the communication link to fail with a certain probability in each iteration of N_2O. The failure probability varies from 0.2 to 0.8 and Fig. 10 depicts the evolution of objective values (objective of $\mathscr{P}2$) when running N_2O. In general, we see that N_2O is able to converge to the same optimal value with different link failure probabilities. In addition, N_2O converges slower with a larger link failure

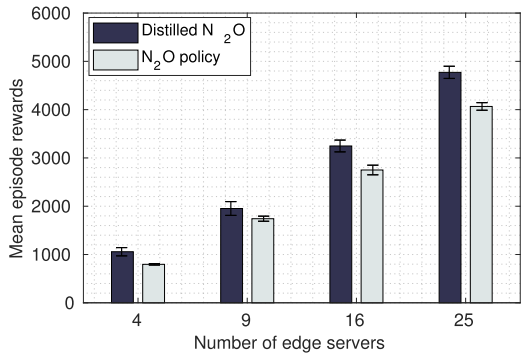

Fig. 11. Orchestration policy distillation.

probability. The impact of link failure on the convergence performance is not significant. N_2O is able to converge within 50 iterations even with the link failure probability 0.8.

6) Knowledge Distilling for N_2O : Fig. 11 shows the performance of knowledge distilling for N_2O. The actor for each LSM n uses a 5-layer network. The input layer has 5 nodes for feeding the local observation and the output layer has 1 node for outputting the local rental decision. The other three layers are fully-connected layers with 16, 32, 16 nodes, respectively. Fig. 11 gives the mean episode rewards of N_2O and distilled policy. We can see that the distilled policy achieves higher rewards since it helps eliminate the impact of gradient oscillation as discussed in V-B. We also change the number of edge servers in the edge computing system. We can see clearly that our method can achieve larger rewards with more edge servers. This is simply because more edge servers provide more available computing resources on the network edge, and therefore more user service demands can be accommodated and lower service delay can be delivered.

VII. Conclusion

In this paper, we proposed a novel distributed DRL method based on multi-agent DQL, neural network orchestration (N_2O), and knowledge distilling. The proposed method fits extremely well for distributed edge computing systems. It captures complicated interactions between the users and edge computing system using deep learning techniques. The multi-agent DQL effectively address the heterogeneity of edge sites, allowing each edge site to have a distinct Q-network that works well locally. N_2O coordinates local Q-networks to maximize the system-wide performance. Knowledge distilling is further applied to avoid the communication overhead incurred by N_2O. We exemplify the efficacy of the proposed method on a service provisioning problem for edge computing systems. The proposed method has a general framework that can be applied to a variety of issues in edge computing systems, e.g., computation offloading, service placement, service migration, and resource allocation. It is also suitable for many other systems featured by the distributed and heterogeneous nature. There are still many works can be done to improve the performance of our method. For example, episodic rewards can be used to improve the sample efficiency, and adversary training can be applied to improve the robustness of multi-agent DQL.

References

[1] Y. Mao, C. You, J. Zhang, K. Huang, and K. B. Letaief, "A survey on mobile edge computing: The communication perspective," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 4, pp. 2322–2358, 4th Quart., 2017.

[2] T. Taleb, K. Samdanis, B. Mada, H. Flinck, S. Dutta, and D. Sabella, "On multi-access edge computing: A survey of the emerging 5G network edge cloud architecture and orchestration," *IEEE Commun. Surveys Tuts.*, vol. 19, no. 3, pp. 1657–1681, 3rd Quart., 2017.

[3] Verizon. *Verizon and AWS: The Cutting-Edge of Edge*. Accessed: Nov. 10, 2020. [Online]. Available: https://enterprise.verizon.com/business/learn/edge-computing

[4] H. Zhao, M. Pan, X. Liu, X. Li, and Y. Fang, "Exploring fine-grained resource rental planning in cloud computing," *IEEE Trans. Cloud Comput.*, vol. 3, no. 3, pp. 304–317, Jul. 2015.

[5] J. Ren, H. Wang, T. Hou, S. Zheng, and C. Tang, "Federated learning-based computation offloading optimization in edge computing-supported Internet of Things," *IEEE Access*, vol. 7, pp. 69194–69201, 2019.

[6] X. Chen, H. Zhang, C. Wu, S. Mao, Y. Ji, and M. Bennis, "Optimized computation offloading performance in virtual edge computing systems via deep reinforcement learning," *IEEE Internet Things J.*, vol. 6, no. 3, pp. 4005–4018, Jun. 2019.

[7] F. Guo, L. Ma, H. Zhang, H. Ji, and X. Li, "Joint load management and resource allocation in the energy harvesting powered small cell networks with mobile edge computing," in *Proc. IEEE Conf. Comput. Commun. Workshops (INFOCOM WKSHPS)*, Apr. 2018, pp. 299–304.

[8] C. Huang, A. Zappone, G. C. Alexandropoulos, M. Debbah, and C. Yuen, "Reconfigurable intelligent surfaces for energy efficiency in wireless communication," *IEEE Trans. Wireless Commun.*, vol. 18, no. 8, pp. 4157–4170, Aug. 2019.

[9] X. Chen, L. Jiao, W. Li, and X. Fu, "Efficient multi-user computation offloading for mobile-edge cloud computing," *IEEE/ACM Trans. Netw.*, vol. 24, no. 5, pp. 2795–2808, Oct. 2016.

[10] L. Chen, S. Zhou, and J. Xu, "Computation peer offloading for energy-constrained mobile edge computing in small-cell networks," *IEEE/ACM Trans. Netw.*, vol. 26, no. 4, pp. 1619–1632, Aug. 2018.

[11] J. Xu, L. Chen, and S. Ren, "Online learning for offloading and autoscaling in energy harvesting mobile edge computing," *IEEE Trans. Cognit. Commun. Netw.*, vol. 3, no. 3, pp. 361–373, Sep. 2017.

[12] P. Dai *et al.*, "Multi-armed bandit learning for computation-intensive services in MEC-empowered vehicular networks," *IEEE Trans. Veh. Technol.*, vol. 69, no. 7, pp. 7821–7834, Jul. 2020.

[13] C. Zhang, P. Patras, and H. Haddadi, "Deep learning in mobile and wireless networking: A survey," *IEEE Commun. Surveys Tuts.*, vol. 21, no. 3, pp. 2224–2287, 3rd Quart., 2019.

[14] L. Deng and D. Yu, "Deep learning: Methods and applications," *Found. Trends Signal Process.*, vol. 7, nos. 3–4, pp. 197–387, Jun. 2014.

[15] T. Yang, Y. Hu, M. C. Gursoy, A. Schmeink, and R. Mathar, "Deep reinforcement learning based resource allocation in low latency edge computing networks," in *Proc. 15th Int. Symp. Wireless Commun. Syst. (ISWCS)*, Aug. 2018, pp. 1–5.

[16] Y. Qian, L. Hu, J. Chen, X. Guan, M. M. Hassan, and A. Alelaiwi, "Privacy-aware service placement for mobile edge computing via federated learning," *Inf. Sci.*, vol. 505, pp. 562–570, Dec. 2019.

[17] J. Konečný, H. B. McMahan, F. X. Yu, P. Richtárik, A. T. Suresh, and D. Bacon, "Federated learning: Strategies for improving communication efficiency," 2016, *arXiv:1610.05492*. [Online]. Available: http://arxiv.org/abs/1610.05492

[18] Z. Ning, P. Dong, J. J. P. C. Rodrigues, F. Xia, and X. Wang, "Deep reinforcement learning for vehicular edge computing: An intelligent offloading system," *ACM Trans. Intell. Syst. Technol.*, vol. 10, no. 6, pp. 1–24, 2019.

[19] C. Huang, R. Mo, and C. Yuen, "Reconfigurable intelligent surface assisted multiuser MISO systems exploiting deep reinforcement learning," *IEEE J. Sel. Areas Commun.*, vol. 38, no. 8, pp. 1839–1850, Aug. 2020.

[20] V. Mnih *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, Feb. 2015.

[21] J. Dean *et al.*, "Large scale distributed deep networks," in *Proc. Adv. Neural Inf. Process. Syst.*, 2012, pp. 1223–1231.

[22] L. Huang, S. Bi, and Y.-J.-A. Zhang, "Deep reinforcement learning for online computation offloading in wireless powered mobile-edge computing networks," *IEEE Trans. Mobile Comput.*, vol. 19, no. 11, pp. 2581–2593, Nov. 2020.

[23] Y. He, N. Zhao, and H. Yin, "Integrated networking, caching, and computing for connected vehicles: A deep reinforcement learning approach," *IEEE Trans. Veh. Technol.*, vol. 67, no. 1, pp. 44–55, Jan. 2018.
[24] I. Adamski, R. Adamski, T. Grel, A. Jędrych, K. Kaczmarek, and H. Michalewski, "Distributed deep reinforcement learning: Learn how to play Atari games in 21 minutes," in *Proc. Int. Conf. High Perform. Comput.* Cham, Switzerland: Springer, 2018, pp. 370–388.
[25] Z. Yu *et al.*, "Federated learning based proactive content caching in edge computing," in *Proc. IEEE Global Commun. Conf. (GLOBECOM)*, Dec. 2018, pp. 1–6.
[26] L. Busoniu, R. Babuska, and B. De Schutter, "Multi-agent reinforcement learning: An overview," in *Innovations in Multi-Agent Systems and Applications-1*. Berlin, Germany: Springer, 2010, pp. 183–221.
[27] M. G. R. Alam, Y. K. Tun, and C. S. Hong, "Multi-agent and reinforcement learning based code offloading in mobile fog," in *Proc. Int. Conf. Inf. Netw. (ICOIN)*, Jan. 2016, pp. 285–290.
[28] X. Liu, J. Yu, and Y. Gao, "Multi-agent reinforcement learning for resource allocation in IoT networks with edge computing," 2020, *arXiv:2004.02315*. [Online]. Available: http://arxiv.org/abs/2004.02315
[29] J. Foerster, I. A. Assael, N. De Freitas, and S. Whiteson, "Learning to communicate with deep multi-agent reinforcement learning," in *Proc. Adv. Neural Inf. Process. Syst.*, 2016, pp. 2137–2145.
[30] R. Lowe, Y. Wu, A. Tamar, J. Harb, O. P. Abbeel, and I. Mordatch, "Multi-agent actor-critic for mixed cooperative-competitive environments," in *Proc. Adv. Neural Inf. Process. Syst.*, 2017, pp. 6379–6390.
[31] S. Li, Y. Wu, X. Cui, H. Dong, F. Fang, and S. Russell, "Robust multi-agent reinforcement learning via minimax deep deterministic policy gradient," in *Proc. AAAI Conf. Artif. Intell.*, vol. 33, 2019, pp. 4213–4220.
[32] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," 2015, *arXiv:1503.02531*. [Online]. Available: http://arxiv.org/abs/1503.02531
[33] L. Chen and J. Xu, "Budget-constrained edge service provisioning with demand estimation via bandit learning," *IEEE J. Sel. Areas Commun.*, vol. 37, no. 10, pp. 2364–2376, Oct. 2019.
[34] X. Ge, S. Tu, G. Mao, and C. X. Wang, "5G ultra-dense cellular networks," *IEEE Trans. Wireless Commun.*, vol. 23, no. 1, pp. 72–79, Feb. 2016.
[35] D. Breitgand *et al.*, "Dynamic virtual machine resizing in a cloud computing infrastructure," U.S. Patent 9 858 095, Jan. 2, 2018.
[36] D. Ardagna, G. Casale, M. Ciavotta, J. F. Pérez, and W. Wang, "Quality-of-service in cloud computing: Modeling techniques and their applications," *J. Internet Services Appl.*, vol. 5, no. 1, p. 11, Dec. 2014.
[37] J. Xu, L. Chen, and P. Zhou, "Joint service caching and task offloading for mobile edge computing in dense networks," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, Apr. 2018, pp. 207–215.
[38] R. Beraldi, A. Mtibaa, and H. Alnuweiri, "Cooperative load balancing scheme for edge computing resources," in *Proc. 2nd Int. Conf. Fog Mobile Edge Comput. (FMEC)*, May 2017, pp. 94–100.
[39] Z. Zhu, T. Liu, Y. Yang, and X. Luo, "BLOT: Bandit learning-based offloading of tasks in fog-enabled networks," *IEEE Trans. Parallel Distrib. Syst.*, vol. 30, no. 12, pp. 2636–2649, Dec. 2019.
[40] L. Chen. *Online Supplementary Material*. Accessed: Nov. 10, 2020. [Online]. Available: https://github.com/chenlx-um/neural-network-orchestration.git
[41] J. C. Duchi, A. Agarwal, and M. J. Wainwright, "Dual averaging for distributed optimization: Convergence analysis and network scaling," *IEEE Trans. Autom. Control*, vol. 57, no. 3, pp. 592–606, Mar. 2012.
[42] A. Virmaux and K. Scaman, "Lipschitz regularity of deep neural networks: Analysis and efficient estimation," in *Proc. Adv. Neural Inf. Process. Syst.*, 2018, pp. 3835–3844.
[43] R. A. Horn and C. R. Johnson, *Matrix Analysis*. Cambridge, U.K.: Cambridge Univ. Press, 2012.
[44] D. A. Levin and Y. Peres, *Markov Chains and Mixing Times*, vol. 107. Providence, RI, USA: American Mathematical Society, 2017.
[45] N. Srivastava, G. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: A simple way to prevent neural networks from overfitting," *J. Mach. Learn. Res.*, vol. 15, no. 1, pp. 1929–1958, 2014.

Lixing Chen received the B.S. and M.S. degrees from the College of Information and Control Engineering, China University of Petroleum (East China), Qingdao, China, in 2013 and 2016, respectively. He is currently pursuing the Ph.D. degree with the College of Engineering, University of Miami. His research interests include mobile edge computing, game theory, and machine learning for networks.

Jie Xu (Member, IEEE) received the B.S. and M.S. degrees in electronic engineering from Tsinghua University, Beijing, China, in 2008 and 2010, respectively, and the Ph.D. degree in electrical engineering from UCLA in 2015. He is currently an Assistant Professor with the Department of Electrical and Computer Engineering, University of Miami. His research interests include mobile edge computing, machine learning for networks, and network security.