A Reinforcement Learning Approach for Online Service Tree Placement in Edge Computing

Yimeng Wang*, Yongbo Li*, Tian Lan*, Nakjung Choi†
*The George Washington University, †Nokia Bell Labs

***Abstract*—We consider the problem of optimally mapping an edge computing service that is modeled as a tree with multiple processing sub-tasks and data flows onto the underlying physical network. As new computing and data analytics applications require more complicated data processing structures, and different types of data (e.g., images, videos, and numbers) sensed at geographically distributed locations must be collected and processed to obtain a complex and comprehensive result, highly intelligent algorithms are needed to solve this challenging problem. In this paper, we propose a learning-based hierarchical service tree placement strategy that aims to optimize the net utility, defined as achieved utility minus network congestion. The key idea is to decouple a service tree into appropriate sub-trees each containing a single computing sub-task as well as associated data flows and to recursively leverage Q-learning to place each sub-tree while maintaining the dependencies of sub-tasks in the service tree structure. It enables a scalable solution for large networks with unknown arrival statistics and complex service structures. Numerical results show that our solution can significantly outperform baseline heuristics in online service tree placement.**

I. Introduction

The growing explosion of data generation and consumption at the network edge, together with new technologies such as Internet of Things (IoT) and software-defined networking (SDN), has prompted a rapid shift toward edge/fog computing. The network architecture of edge computing can be broadly described as a hierarchical model with each layer offering different combinations of computing/networking resources and performance tradeoffs. The problem of dividing computation between the different layers and optimally mapping a service request consisting of multiple processing and data flow elements onto the underlying physical network is a very challenging problem.

Consider a simple edge computing application where surveillance cameras upload sequences of images to the edge network for object identification, and the results are returned to different end-devices. Since the computed result (i.e., object tags) have a much smaller data rate than incoming image data flows, a greedy strategy to push the computation closer to the network edge where the images are captured may seem appealing. However, there are multiple factors to be considered. First, edge servers/nodes such as smart routers and set-top boxes often have limited capacities. In an online environment with dynamic computing request arrivals, such a greedy approach may quickly saturate (part of) the edge network, leaving no room to accommodate future latency-sensitive or traffic-intensive requests that are more crucial for edge processing. Second, the placement problem requires a joint optimization of both computing and network resource consumption. While

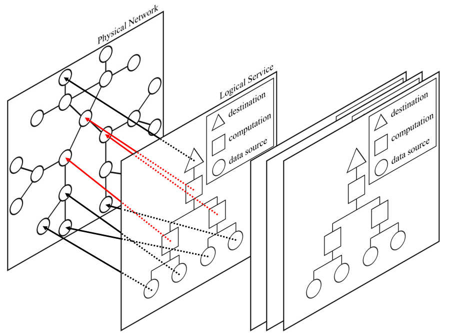


Fig. 1: Mapping service requests onto physical edge network.

pushing all computations to a small number of resource-contained edge nodes is not feasible, to spread computation across the network, we also need to carefully place the services to mitigate congestion. Finally, there also exist services with more complex topology and different data rate requirements, such as image upscaling [10] that generates higher output data rate than the input. A more intelligent service placement algorithm is necessary to handle these diverse service requests on the fly, with respect to resource availability in the physical network.

We consider the problem of online service placement in edge computing, to optimize overall net utility defined as the utility received from admitted requests minus resulting network congestion. Prior work on network service placement problems often optimize the placement focusing on single task [11], the chain structure [4], [16], [9], [7] and the tree structure [1], [8], using various model-based approaches, e.g., game theory [15] and virtual machine placements [2], [3]. A key feature of our solution is the use of deep reinforcement Q-learning, which has been successfully applied to address many network optimizations such as software-defined network (SDN) routing [6], transmission scheduling [17], caching [5], and edge network security [14]. Our solution leverages a Q-learning agent to interact with the system environment, select placement decisions (i.e., actions), and improve the decision makings of service placement according to the experience without knowing the stationary state transition stochastics in the edge network. Furthermore, it can also adopt to environment changes during the online training.

One of the challenges in applying Q-learning to the service placement problem is the need to deal with a huge action space. More precisely, new computing and data analytics applications require more complicated data processing structures [1]. Different types of data (e.g., images, videos, numbers, etc.) sensed from geographically distributed locations must

978-1-7281-2700-2/19/$31.00 2019 © IEEE

be collected and processed to obtain a complex and comprehensive result. Take intelligent traffic management as an example. The application may require multiple photos, traffic densities of multiple streets, inter-vehicle communications, and weather information. The graphic data can be processed first to recognize a vehicle, then combined with the traffic and weather information to produce a route prediction or driving guidance. In this paper, we model an edge computing service request as an arbitrary tree topology (denoted as a service tree), in which each node represents a computing sub-task and each edge a data flow. As shown in Figure 1, placing each service tree requires jointly mapping all sub-tasks and data flows in the tree to proper edge nodes and links respectively, in the underlying physical network. It is easy to see that for a network of n edge nodes and a service tree of m sub-tasks, the action/decision space has an exponential size $o(n^m)$ even without considering different routing strategies. Hence, a heuristic or model-based strategy - such as value iteration method of Markov decision process - either is faced with prohibitive complexity or cannot adapt to a wide range of service requirements with the large variety of today's applications. A highly intelligent placement strategy is expected to work in an unsupervised fashion.

To address this challenge, we propose a hierarchical service tree placement strategy. It (i) decouples a service tree into appropriate sub-trees each containing a single computing sub-task as well as associated data flows, and (ii) recursively leverages a Q-learning agent to place each sub-tree on the physical network while maintaining the dependencies of sub-tasks in the service tree structure. In particular, the sub-problems to place each sub-tree is efficiently computed using Q-learning and online training, and then popular routing strategies such as shortest path and minimum congestion are employed to solve the routing problem with known placement decisions. For a network of n edge nodes and a service tree of size m, our hierarchical service tree placement strategy can effectively reduce the action space size from $o(n^m)$ to $o(n \cdot m)$, since the sub-trees are iteratively placed. It makes our learning-based solution more scalable for large networks and more complex service structures. Numerical results show that our solution can significantly outperform baseline heuristics in terms of optimizing overall utility and congestion in online service tree placement.

The main contributions of this paper are summarized as follows:

- We consider the problem of online service tree placement in edge computing and propose a learning-based solution that can effectively solve the joint optimization of received utility and network congestion on the fly with dynamic service request arrivals.
- Our solution makes novel use of deep Q-learning and employs a hierarchical service tree placement strategy, which decouples the problem and recursively use Q-learning to place sub-trees. It can effectively reduce the action/decision space size from $o(n^m)$ to $o(n \cdot m)$, enabling scalable solutions.
- We implement the Q-learning agent and evaluate its performance using numerical simulations. The results show that our proposed solution outperforms the heuristics by up to 35.97% and 25.70% in terms of the total reward and network congestion.

II. System Model

Consider an edge network denoted as an undirected graph, $G = (\mathbf{V}, \mathbf{E})$, where each vertex $i \in \mathbf{V}$ represents an edge node (i.e., an edge server) and each edge $e \in \mathbf{E}$ represents a physical link in the network. Sensors and IoT devices that are connected to these edge nodes continuously generate data flows. Various service requests are then (dynamically) submitted to process (a set of) these data flows to produce results to be consumed by end-devices that are also located in the edge network. The service placement problem aims to map a structural request - often represented as a tree [1] [8] [12] or a chain [13] - onto the physical edge network, as depicted in Figure 1. More precisely, for a given set of sensors (i.e., sources) and a given end-device (i.e., destination), the service placement problem needs to jointly identify (i) a set edge nodes with the necessary computing resources to compute the request and (ii) a set of links to connect the edge nodes with the sources and destination for data transfer. It is easy to see that each edge computing request must be assigned with both network and computing resources, which are tightly coupled and lead to a challenging joint online optimization problem under dynamic request arrivals and departures.

In this paper, we model each edge computing request as a service tree T. As shown in Figure 2, the leaf nodes of each service tree denote the sources of related data flows, the root node denotes the destination of the request, all other nodes represent multiple intermediate computing steps (or sub-tasks, which we use interchangeably in this paper) to process the data and produce the final result for consumption, and each edge in the tree represents required data flow between sub-tasks. Each computing sub-task requires a certain amount of computing resources to process one or multiple incoming data flows and generate a single output data flow for the next processing step until reaching the final destination (i.e., the root node) of the request. We note that while each computing sub-task requires a separate bundle of computing resources, multiple sub-tasks can be mapped to the same physical edge node, as long as there is enough computing resource available, eliminating the need for cross-node data transfer as the traffic is contained inside the edge node. We consider a virtualized resource sharing strategy, where the computing resources available at each edge node, e.g., CPU, GPU, memory, are bundled into virtual resource containers (i.e., VMs). In particular, each edge node $i \in \mathbf{V}$ has a computing resource capacity constraint with C_i containers available. Sub-tasks can be assigned to node i only if there are free containers available.

We consider an online service tree placement problem, where edge computing requests (i.e., service trees) are submitted on the fly, not known *a priori*. Let u_T be the utility if a service tree T is accepted and successfully placed, which is predefined by the importance/urgency of service T. If T is rejected, then no utility is achieved. Then, under dynamic request arrivals, the goal of service tree placement is to map sub-tasks and data flows of the new service trees onto vertices and edges of the physical network on the fly, in order to maximize the net difference between total utility achieved and

TABLE I: Example of computing requests for different applications

Request	Inputs	Output	Structure
Video Captioning	625KB/s	625KB/s	**Line**, length 3, 1 source.
Image Upscaling	20KB/s	600KB/s	**Line**, length 3, 1 source.
Traffic Information	64KB/s 4KB/s	20KB/s	**Tree**, height 3, 2 sources.
Image Recognition	600KB/s 1000KB/s	1KB/s	**Tree**, height 3, 2 sources.
Driving Assistant	20KB/s 1KB/s 1KB/s	1KB/s	**Tree**, height 4, 5 sources, 3 sub-tasks.
Medical Implants	1KB/s 1KB/s	1KB/s	**Tree**, height 3, 2 sources.

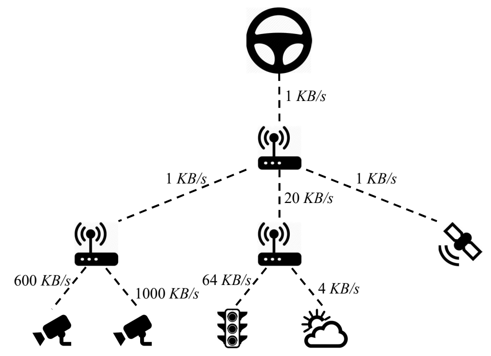


Fig. 2: Example request of the driving assistant service.

network congestion, under computing resource constraints at edge nodes.

Some examples of real-world edge requests are shown in Table I. Note that some computed results may have very low data rates (e.g., image recognition labels, etc), we use a default data rate of 1KB/s for those data flows. Consider the driving assistant application further illustrated in Figure 2. Two cameras/sources mounted on the vehicle provide raw input data flows (images or videos) at rates of 600KB/s and 1000KB/s respectively, which are processed by an image identification software (i.e., sub-task 1) to identify and label nearby objects, such as other vehicles, pedestrians, and obstacles. The resulting label is then sent via a much smaller data flow (e.g., integer indexes of the object labels) at a rate of only 1KB/s. At the same time, two data flows containing real-time traffic information at 64KB/s and weather information at 4KB/s are processed by another navigation software (i.e., sub-task 2) to calculate route recommendations at an output data rate of 20KB/s. Finally, by combining results from these sub-tasks, along with GPS location of the vehicle coming from another sensor at 1KB/s, a driving assistant software (i.e., sub-task 3) computes the final output to be displayed on end-device.

While an output data flow from data processing tasks often has lower data rate as in the driving assistant application, there also exist opposite examples, where data processing tasks can generate an output data flows at higher rates. Such examples include image upscaling [10] and video captioning applications. Therefore, we consider arbitrary data flow rates in our service tree model, so it is applicable to diverse edge applications in real-world.

III. PROBLEM FORMULATION

Our goal is to optimize the net utility, defined as total utilities achieved in the system minus (weighted) network congestion. To utilize reinforcement learning, the problem is formulated as a Markov decision process (MDP), in which the learning agent can interact with the environment, and learn from the experiences of its behaviors. The MDP is described as follows: In any *system state*, making an *action* will generate some *reward* (i.e., optimization objective), while the system will *transit* to another state as a result. The MDP formulation of the service tree placement problem is discussed next.

System States. A system state consists of current *link traffic loads* L_t, *remaining edge node capacities* C_t, and *target service tree* T_t to be placed. We consider a discrete-time model for the service tree placement problem, where time slots are sufficiently small, such that there is at most one request arrival in each slot. For a network with $n = |\mathbf{E}|$ edge nodes, we use a $n \times n$ symmetric matrix L_t to denote the link load matrix, where $L_{t,(i,j)}$ is the total traffic load on link (i, j) at time slot t. Next, we bundle multi-resources into containers on each edge node and use a vector C_t to denote the currently remaining containers on each edge node, available for placing new computing sub-tasks. Finally, the target service tree is defined by its topology as well as resource requirements associated with each vertex and edge.

Actions and State Transition. At state $s_t = (L_t, C_t, T_t)$, action a_t defines the service tree placement decision, i.e., a mapping from service tree T_t to the edge network $G = (\mathbf{V}, \mathbf{E})$. In particular, each node k' in the service tree beside the root and leafs (which denote known destination and sources of request T_t) is mapped to some vertex $v \in \mathbf{V}$, and each edge (i', j') in the service tree is mapped to some edge $(i, j) \in \mathbf{E}$. It is easy to see that for an edge network of size n and a service tree with m nodes, the action space may have size $o(n^m)$. Later, we will develop an efficient hierarchical placement strategy to significantly reduce the action space.

Once an action a_t is implemented, the system state immediately moves to $(L'_t, C'_t, 0)$, in which link loads L'_t and remaining computing capacity C'_t are both updated based on the service tree placement and the resource requirements of T_t. The new system state $s_{t+1} = (L_{t+1}, C_{t+1}, T_{t+1})$ at time slot $t + 1$ is further derived by removing all requests and their resource consumption that departs in time slot t. We note that action a_t is feasible only if there are enough computing resource available to support the placement of T_t. Otherwise, the request T_t is rejected with no changes in traffic load or remaining capacity, and no utility will be received. Existing solutions for the MDP problems, e.g., value iteration and policy iteration, require the quantification of transition probabilities for all state/action pairs. In this paper, we use deep Q-learning to solve the problem. The state transitions can be learned by the neural network, thus analytically quantifying the transition probabilities becomes unnecessary.

Rewards. The rewards of an MDP is a sum of the immediate reward and the discounted future reward. With a discount factor $\gamma < 1$, the reward function can be defined as

$$r_t = \sum_{\tau=0}^{\infty} \gamma^{\tau} r_{im}(t + \tau), \tag{1}$$

where the $r_{im}(t+\tau)$ represents the immediate reward obtained at future time slot $t + \tau$. The immediate reward at $\tau = 0$ can be calculated directly from system state s_t and action a_t, by taking into account the additional traffic load and computation resource requirement of placing T_t. Our goal is to optimize the net utility, defined as total utilities achieved in the system minus (weighted) network congestion. In particular, we consider the maximum link congestion:

$$w_1(t) = \max(L'_t), \tag{2}$$

and the sum utility collected from all active service trees during time slot t, i.e.,

$$w_2(t) = \sum_{\tau=0}^{t} u_{T_\tau} \cdot \mathbf{1}(T_\tau, t), \tag{3}$$

where the u_{T_τ} denotes the utility of request T_τ (defined in Section II), and the $\mathbf{1}(T_\tau, t)$ is an indicator function that is equal to 1 if request T_τ is actively served at time t and 0 otherwise. Thus, using some tradeoff factors $\alpha_1, \alpha_2 > 0$, the immediate reward (i.e., overall optimization objective) is defined as weighted net utility:

$$r_{im}(t) = -\alpha_1 w_1(t) + \alpha_2 w_2(t). \tag{4}$$

With the weighted reward function, additional measurements, such as the latency and energy consumption, can be jointly optimized during the placement. In this work, we only consider two performance tradeoffs. The existing metrics in Equations 2, 3 can also be extended to other metrics such as the mean congestion, relative load, etc. Since the VMs are normally well isolated, we ignore the congestion in computing resources.

IV. Service Tree Placement via Reinforcement Learning

We leverage deep Q-learning to solve the online service tree placement problem formulated as an MDP. A reinforcement learning agent is set up to interact with state evolution in the MDP model. By observing the system states s_t and making action decisions a_t, the agent can explore the action space, observe consequences, and then continuously self-improve according to the experience. It is easy to see that for an edge network of size n and a service tree with m nodes, the action space at each time t - which commonly equals to the size of the output layers of the neural network in Q-learning - has an exponential size of $o(n^m)$ since all sub-tasks in a service tree T_t are placed jointly. Next, we will develop an efficient hierarchical placement strategy to significantly reduce the action space. Our key idea is to iteratively place each sub-task and its associated data flows using Q-learning, until the entire service tree T_t is successfully placed. Then, the network reaches a new state and the corresponding reward is calculated, to update the Q-learning agent.

We start by considering a sub-task node k in T_t, whose children are only leaf nodes, i.e., data sources for sub-task k. Let Y_k denote the sub-tree consisting of node k and its leaf children. We note that sub-tree Y_k can be placed before any other decisions of T_t, because we only need to determine the

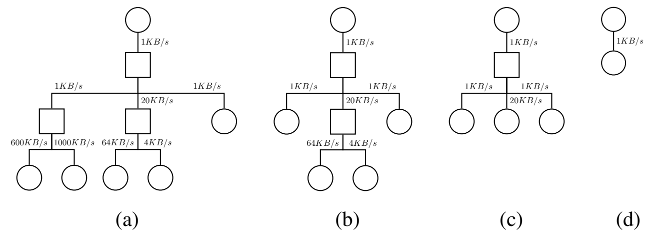


Fig. 3: Illustrating our hierarchically placement strategy.

placement of node k (i.e., to find an edge hosting the sub-task), while the data flow routing from given source nodes in Y_k to node k can be determined using simple heuristics such as minimum congestion or shortest path routing. Let c_k denote the computing resource requirement of sub-task k, $\lambda_{i,k}$ a vector of its incoming data flow rates from the source child nodes in Y_k, $\lambda_{o,k}$ the output data rate after data are processed by sub-task k. The placement of sub-tree Y_t is determined with respect to system state s_t and sub-task related parameters, which are represented as a seven-tuple $P_k = (Y_k, \lambda_{\mathbf{i},\mathbf{k}}, c_k, \lambda_{o,k}, r_{T_t}, d_{T_t}, u_{T_t})$, with r_{T_t}, d_{T_t} and u_{T_t} as the destination, duration and utility of the (parent) request.

For given s_t and P_k, Q-learning implements a neural network to calculate the Q-values (i.e., achievable discounted reward) for each action a_t and use them to guide the decision making. More precisely, for edge network of size $n = |E|$, s_t and P_k are fed into the input layer of the neural network, and $n+1$ Q-values are generated at the output layer, corresponding to n possible placements of sub-task k in Y_k as well as an additional action to reject Y_k. In Q-learning, the most common decision making policy is Epsilon-greedy – with a probability ϵ, the action having the highest Q-value will be chosen, and otherwise, it will pick a random action in the action space equally-likely. We note that a rejection may also occur if the assigned edge node i does not have enough computing resource to satisfy the requirement c_k. When an action is chosen, we will place sub-tree Y_k on the edge network and update link loads L_t and remaining computing capacities C_t accordingly.

Our hierarchical service tree placement strategy will iteratively identify such sub-trees using depth-first search and place them on the edge network. Once a sub-tree Y_k is placed, we remove Y_k from the service tree T_t and replace it by a virtual source node with rate $\lambda_{o,k}$. This is because as far as later placement decisions are concerned, Y_k simply generates a data flow at rate $\lambda_{o,k}$ that is fed into the next processing sub-task, as if it is a source node for the rest of service tree T_t. Next, considering the reduced service tree $T_t - Y_k$, we can recursively use the same procedure and Q-learning agent to identify another proper sub-tree (e.g., with depth-first search) and place it on the network, until either (i) the entire service tree is mapped onto the physical edge network, i.e., T_t reduces to a single source and a single destination with no more sub-tasks to consider, or (ii) no feasible placement can be found for some sub-tree Y_k, causing the entire service tree being rejected. Figure 3 demonstrates the hierarchical service tree placement strategy for the example discussed in Section II. We note that the proposed strategy effectively reduces the action space from $o(n^m)$ to $o(n \cdot m)$, because each sub-tree is now

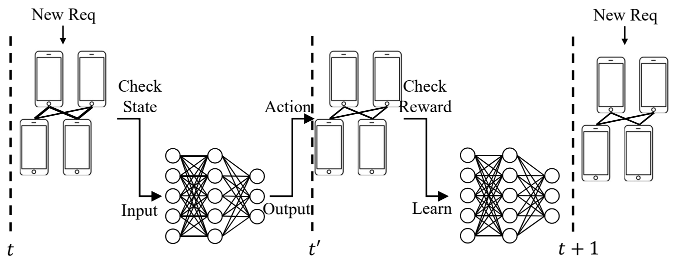


Fig. 4: Illustration of online service tree placement using Q-learning.

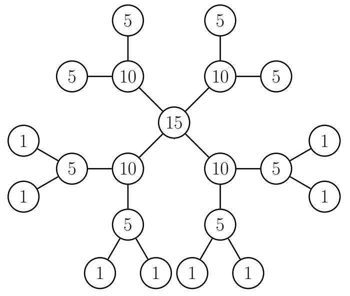


Fig. 5: An instance of edge network topology in our simulation. Computing capacities are marked on each node.

placed separately by recursively using Q-learning.

Once service tree T_t is processed, either placed or rejected, the system will reach state $(L'_t, C'_t, 0)$ and subsequently $(L_{t+1}, C_{t+1}, T_{t+1})$ at the next request arrival. Using Equations (2), (3), and (4), the immediate reward $r_{im}(t)$ can be calculated if T_t has been successfully placed. Otherwise, if any sub-tree of T_t is rejected, T_t is denied with all remaining sub-tasks removed, and zero utilities achieved. The Q-learning engine will continue to make placement decisions on the fly and adjust based on the achieved reward. In particular, when T_t is successfully placed, we evenly distribute its immediate reward $r_{im}(t)$ to all the state/action pairs that are selected during the iterative sub-tree placement process, since all the sub-tree actions together obtain this immediate reward. Finally, the neural network is trained by comparing the actual reward obtained from the environment, and the estimated Q-values from the output layer. The general process of online service tree placement using Q-learning and training is depicted in Figure 4.

V. Evaluation

We evaluate our learning-based service tree placement strategy using the requests listed in Table I and on a random edge network consisting of $n = 21$ edge nodes with one instance shown in Figure 5. For given system state s_t, we generate a new request T_t with equal probability, leverage the Q-learning module to make placement decisions a_t, and evaluate the resulting reward (i.e., utility gain and network congestion) through simulation. Note that in real-world networks, multiple requests may arrive in the same time slot. In this case, they will be assigned sequentially. In particular, we compare our proposed placement strategy, namely Q-learning, against two baseline strategies, Nearest and Random, i.e., **Nearest:** The computation sub-tasks are iteratively placed in nearest feasible edge nodes, so data are processed as close to network edge (and data sources) as possible. **Random:** We randomly place

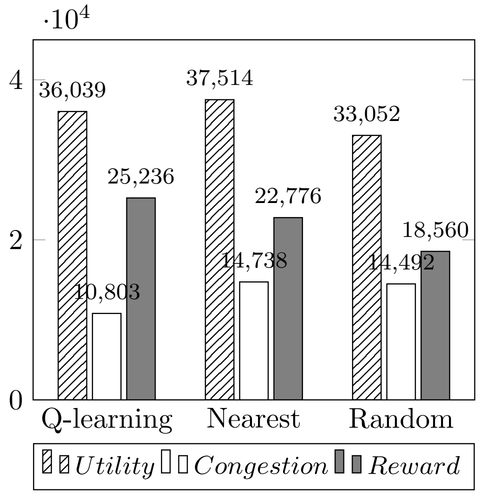


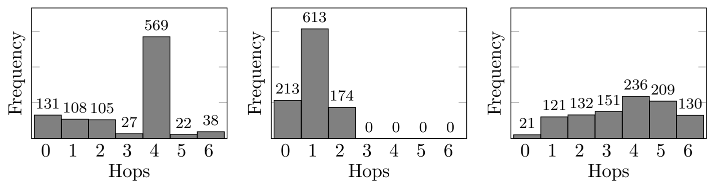
Fig. 6: Rewards and breakdown for different placement policies – Q-learning, Nearest, and Random.

sub-tasks anywhere between the sources and the destination using a uniform distribution. A sub-task is rejected if no nodes with sufficient resource are available.

We implement a Q-learning module with 6 hidden layers, including 1 convolutional layer, 1 pooling layer, and 4 fully connected layers. The learning rate is set to 0.001, and the discount factor for future reward is $\gamma = 0.9$. To calculate the reward, we set the weights for network congestion and received utility to $\alpha_1 = 10^3$ and $\alpha_2 = 1$, thus the total reward is the net difference between utility and congestion. For instance, assigning a sub-task with unit utility $u = 1$ gains zero rewards if the resulted congestion increases by $1KB/s$.

Figure 6 compares the average reward achieved by different policies, as well as the breakdown of reward into utility and network congestion. Our proposed Q-learning based service tree placement strategy achieves the highest reward, which is 10.80% and 35.97% higher than the two baseline strategies, nearest and random, respectively. Moreover, although the Nearest strategy achieves the highest utility since it tries to accommodate all service requests as close to the edge as possible, with the result of low rejection rate and high network congestion, our Q-learning based strategy can achieve similar utility value, but significantly reduce the network congestion by both rejecting resource-costly jobs when needed and placing all service trees in an prioritized manner. In fact, it is able to reduce network congestion by 26.70% comparing with the Nearest strategy. Hence, by jointly considering network congestion and utility gain, our Q-learning based strategy is able to optimize the net utility in the online service tree placement problem.

Next, to illustrate why our Q-learning based strategy achieves a much higher reward than the baselines, we depict and compare the placement decisions of the three different strategies. In particular, in Figure 7 we calculate the distance (measured by the number of hops) between each placed computing sub-task and its data sources on the network, and plot the distribution of these distances to visualize the locations of service tree placements. We run each strategy for 1000 requests and analyze the decisions made by the three strategies.



(a) Q-learning (b) Nearest (c) Random

Fig. 7: The histogram of the distance (in hops) between the assigned computing node and the sources.

Figure 7c spreads services trees across the whole network, resulting in longer travel distance for the data flows and thus high congestion. Depicted in Figure 7b, the Nearest policy allocates as many sub-tasks as possible close to the source nodes, with most of the sub-tasks placed at 1-2 hops away from the sources. However, since there is limited computing resource available at the network edge, it causes the links and servers at network edge to quickly saturate, which not only results in high network congestion and low computing resource availability at network edge, but also forces future service requests that could have gained more savings to move away from network edge.

In Figure 7a, the Q-learning based strategy also tends to place the tasks close to the source nodes, but in a more intelligent, prioritized fashion. Analyzing the placement decisions, we make a number of key observations: (i) Those service trees that do not have large input data flows can be placed in edge nodes farther away from the sources, without bringing much of a negative impact on network congestion. In an online setting, this helps release precious edge resources for future services trees that can benefit most. (ii) Some tasks such as video captioning and image upscaling have higher output data rates than their inputs. Thus, it is more beneficial to assign them to edge nodes close to the destinations, rather than the sources. Our Q-learning based strategy is able to distinguish these different traffic patterns and intelligently place each service tree to maximize reward.

Finally, to demonstrate the effect of neural network size, we run another set of experiment to compare the convergences of the Q-learning module with different number of layers. As shown in Figure 8, training a neural network with 2 or 3 layers have the similar convergence speed and optimal reward. Fewer or more layers will negatively affect the optimization results. Note that in real-world implementations, the training process can be carried offline. Thus, the convergence time will not affect the performance in real networks.

VI. Conclusion

In this paper, we propose a new, multi-step Q-learning method to solve the problem of online service tree placement in edge networks, to jointly optimize the received utility and network congestion. Our numerical results show that the proposed learning-based strategy outperforms the baselines by up to 35.97% and 26.70% in terms of the net utility achieved.

Our future work includes implementing the framework in an edge computing testbed and evaluating it with real-world applications.

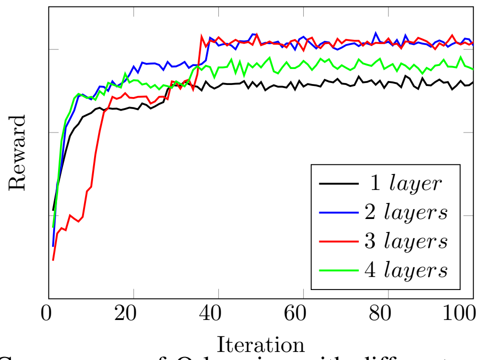


Fig. 8: Convergence of Q-learning with different numbers of layers.

References

[1] Z. Abrams and J. Liu. Greedy is good: On service tree placement for in-network stream processing. In *26th IEEE International Conference on Distributed Computing Systems (ICDCS'06)*, pages 72–72. IEEE, 2006.

[2] O. Ascigil, T. K. Phan, A. G. Tasiopoulos, V. Sourlas, I. Psaras, and G. Pavlou. On uncoordinated service placement in edge-clouds. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, pages 41–48. IEEE, 2017.

[3] S. Clayman, E. Maini, A. Galis, A. Manzalini, and N. Mazzocca. The dynamic placement of virtual network functions. In *2014 IEEE network operations and management symposium*, pages 1–9. IEEE, 2014.

[4] L. Guo, J. Pang, and A. Walid. Joint placement and routing of network function chains in data centers. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 612–620. IEEE, 2018.

[5] Y. He, N. Zhao, and H. Yin. Integrated networking, caching, and computing for connected vehicles: A deep reinforcement learning approach. *IEEE Transactions on Vehicular Technology*, 67(1):44–55, 2018.

[6] S.-C. Lin, I. F. Akyildiz, P. Wang, and M. Luo. Qos-aware adaptive routing in multi-layer hierarchical software defined networks: A reinforcement learning approach. In *2016 IEEE International Conference on Services Computing (SCC)*, pages 25–33. IEEE, 2016.

[7] S. Mehraghdam, M. Keller, and H. Karl. Specifying and placing chains of virtual network functions. In *2014 IEEE 3rd International Conference on Cloud Networking (CloudNet)*, pages 7–13. IEEE, 2014.

[8] A. Pathak and V. K. Prasanna. Energy-efficient task mapping for data-driven sensor network macroprogramming. *IEEE Transactions on Computers*, 59(7):955–968, 2010.

[9] Z. A. Qazi, C.-C. Tu, L. Chiang, R. Miao, V. Sekar, and M. Yu. Simple-fying middlebox policy enforcement using sdn. In *ACM SIGCOMM computer communication review*, volume 43, pages 27–38. ACM, 2013.

[10] S. Schulter, C. Leistner, and H. Bischof. Fast and accurate image upscaling with super-resolution forests. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3791–3799, 2015.

[11] O. Skarlat, M. Nardelli, S. Schulte, and S. Dustdar. Towards qos-aware fog service placement. In *2017 IEEE 1st international conference on Fog and Edge Computing (ICFEC)*, pages 89–96. IEEE, 2017.

[12] U. Srivastava, K. Munagala, and J. Widom. Operator placement for in-network stream query processing. In *Proceedings of the twenty-fourth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 250–258. ACM, 2005.

[13] M. Xia, M. Shirazipour, Y. Zhang, H. Green, and A. Takacs. Network function placement for nfv chaining in packet/optical datacenters. *Journal of Lightwave Technology*, 33(8):1565–1570, 2015.

[14] L. Xiao, X. Wan, C. Dai, X. Du, X. Chen, and M. Guizani. Security in mobile edge caching with reinforcement learning. *IEEE Wireless Communications*, 25(3):116–122, 2018.

[15] Q. Zhang, Q. Zhu, M. F. Zhani, R. Boutaba, and J. L. Hellerstein. Dynamic service placement in geographically distributed clouds. *IEEE Journal on Selected Areas in Communications*, 31(12):762–772, 2013.

[16] Y. Zhang, N. Beheshti, L. Beliveau, G. Lefebvre, R. Manghirmalani, R. Mishra, R. Patneyt, Shirazipour, et al. Steering: A software-defined networking for inline service chaining. In *2013 21st IEEE international conference on network protocols (ICNP)*, pages 1–10. IEEE, 2013.

[17] J. Zhu, Y. Song, D. Jiang, and H. Song. A new deep-q-learning-based transmission scheduling mechanism for the cognitive internet of things. *IEEE Internet of Things Journal*, 5(4):2375–2385, 2018.