Online Learning for Orchestration of Inference in Multi-User End-Edge-Cloud Networks

SINA SHAHHOSSEINI, University of California, Irvine
DONGJOO SEO, University of California, Irvine
ANIL KANDURI, University of Turku
TIANYI HU, University of California, Irvine
SUNG-SOO LIM, Kookmin University
BRYAN DONYANAVARD, San Diego State University
AMIR M. RAHMANI, University of California, Irvine
NIKIL DUTT, University of California, Irvine

Deep-learning-based intelligent services have become prevalent in cyber-physical applications including smart cities and health-care. Deploying deep-learning-based intelligence near the end-user enhances privacy protection, responsiveness, and reliability. Resource-constrained end-devices must be carefully managed in order to meet the latency and energy requirements of computationally-intensive deep learning services. Collaborative end-edge-cloud computing for deep learning provides a range of performance and efficiency that can address application requirements through computation offloading. The decision to offload computation is a communication-computation co-optimization problem that varies with both system parameters (e.g., network condition) and workload characteristics (e.g., inputs). On the other hand, deep learning model optimization provides another source of tradeoff between latency and model accuracy. An end-to-end decision-making solution that considers such computation-communication problem is required to synergistically find the optimal offloading policy and model for deep learning services. To this end, we propose a reinforcement-learning-based computation offloading solution that learns optimal offloading policy considering deep learning model selection techniques to minimize response time while providing sufficient accuracy. We demonstrate the effectiveness of our solution for edge devices in an end-edge-cloud system and evaluate with a real-setup implementation using multiple AWS and ARM core configurations. Our solution provides 35% speedup in the average response time compared to the state-of-the-art with less than 0.9% accuracy reduction, demonstrating the promise of our online learning framework for orchestrating DL inference in end-edge-cloud systems.

1 INTRODUCTION

Deep-learning (DL) is advancing real-time and interactive user services in domains such as autonomous vehicles, natural language processing, healthcare, and smart cities [31]. Due to user device resource constraints, deep learning kernels are often deployed on cloud infrastructure to meet computational demands [2]. However, unpredictable network constraints including signal strength and delays affect real-time cloud services [14]. Edge computing has emerged to complement cloud services, bringing compute capacity closer to the user-end devices [42]. A collaborative end-edge-cloud architecture is essential to provide deep-learning-based services with acceptable latency to user-end devices [26]. The edge paradigm increases offloading opportunities for resource-constrained user-end devices. Offloading DL services in a 3-tier end-edge-cloud architecture is a complex optimization problem considering: (i) diversity in system parameters including heterogeneous computing resources, network constraints, and application characteristics, and (ii) dynamicity of DL service environment including workload arrival rate, user traffic, and multi-dimensional performance requirements (e.g., application accuracy, response time) [8].

Existing offloading strategies for DL tasks are based on the assumptions that (i) all DL tasks have similar compute intensity and require similar communication bandwidth, (ii) offloading improves performance, and (iii) latency is guaranteed with offloaded tasks. However, these assumptions do not hold in practice due to dynamically varying application and network characteristics, where the

computation-communication and accuracy-performance tradeoffs are inconsistent and nontrivial [8, 33, 37]. Under varying system dynamics, such offloading strategies limit the gains from using the edge and cloud resources. Further, model optimization techniques such as quantization and pruning can reduce the computation complexity of DL tasks by sacrificing the model accuracy [6, 35]. Considering model optimization techniques in conjunction with offloading provides opportunities to influence the computation-communication trade-off [36]. This exposes an alternative to offloading in resource constrained devices executing DL inference. Finding the optimal choice between offloading the DL tasks to edge and cloud layers and using optimized models for inference at local devices results in a high-dimensional optimization problem.

Understanding the underlying system dynamics and intricacies among computation, communication, accuracy, and latency is necessary to orchestrate the DL services on multi-level edge architectures. Reinforcement learning is an effective approach to develop such an understanding and interpret the varying dynamics of such systems [25, 28]. Reinforcement learning allows a system to identify complex dynamics between influential system parameters and make a decision online to optimize objectives such as response time [34]. We propose to employ online reinforcement learning to orchestrate DL services for multi-users over the end-edge-cloud system. Our contributions are:

- Runtime orchestration scheme for DL inference services on multi-user end-edge-cloud networks. The orchestrator uses reinforcement learning to perform platform offloading and DL model selection at runtime to optimize response time provided accuracy requirements.
- Implementation of our online learning solution on a real end-edge-cloud test-bed and demonstration of its effectiveness in comparison with state-of-the-art [32] edge orchestration strategies.

2 BACKGROUND

In this section, we present the relevant background and significance of orchestrating DL workloads on end-edge-cloud architecture.

2.1 Offloading DL Workloads in End-Edge-Cloud Architecture

Computation offloading techniques offload an application (or a task within an application) to an external device such as cloud servers [21]. Offloading is typically done in order to improve performance or efficiency of devices [2]. DL workloads on end-devices are conventionally offloaded to cloud servers, but delay-sensitive services for distributed systems rely on performing inference at the edge as an alternative [42]. Inference at the edge can provide cloud-like compute capability closer to the user devices, reducing data transmission and network traffic load. Edge offloading can provide relatively predictable and reliable performance compared to cloud offloading, as there is less workload and network variance [14] [12]. In the context of the end-edge-cloud paradigm, computation offloading techniques partition workloads and distribute tasks among multiple layers (local device, edge device, cloud servers) such that the performance and efficiency objectives are met.

The collaborative end-edge-cloud architecture provides execution choices such that each workload can be executed on the device, on the edge, on the cloud, or a combination of these layers. Each execution choice effects the performance and energy consumption of the user end device, based on the system parameters such as hardware capabilities, network conditions, and workload characteristics. A distributed end-edge-cloud system consists of the following layers:

- **application layer**: provides user level access to a set of services to be delivered by computing nodes

- **platform layer**: provides a set of capabilities to connect, monitor and control end/edge/cloud nodes in a network
- **network layer**: provides connectivity for data and control transfer between different physical devices across multiple
- **hardware layer**: provides hardware capabilities for computing nodes in the system

Each layer presents a diverse set of requirements, constraints, and opportunities to tradeoff performance and efficiency that vary over time. For example, the application layer focuses on the user's perception of algorithmic correctness of services, while the platform layer focuses on improving system parameters such as energy drain and data volume migrated across nodes. Both application and platform layers have different measurable metrics and controllable parameters to expose different opportunities that can be exploited for meeting overall objectives. In the case of DL inference, different DL model structures present opportunities in the application layer, and different computation offloading decisions in a collaborative end-edge-cloud system present opportunities in the platform layer, both for optimizing the execution while meeting required model accuracy.

2.2 Intelligence for Orchestration

Runtime system dynamics affect orchestration strategies significantly in addition to requirements and opportunities. Sources of runtime variation across the system stack include workload of a specific computing node, connectivity and signal strength of the network, mobility and interaction of a given user, etc. Considering cross-layer requirements, opportunities, and runtime variations provide necessary feedback to make appropriate choices on system configurations such offloading policies. Identifying optimal orchestration considering the cross-layer opportunities and requirements in the face of varying system dynamics is a challenging problem. Making the optimal orchestration choice considering these varying dynamics is an NP-hard problem, while brute force search of a large configuration space is impractical for real-time applications. Understanding the requirements at each level of the system stack and translating them into measurable metrics enables appropriate orchestration decision making. Heuristic, rule-based, and closed-loop feedback control solutions are not efficient until reaching convergence, which requires long periods of time [34]. To address these limitations, reinforcement learning approaches have been adapted for the computation offloading problem [32]. Reinforcement learning builds specific models based on data collected over initial epochs, and dramatically improves the prediction accuracy [34].

3 MOTIVATION

This section presents a comprehensive investigation of DL inference for multi-users in end-edge-cloud systems. We examine the scenario using a real setup including five AWS a1.medium instances with single ARM-core as end-node devices connected to an AWS a1.large instance as edge device and an AWS a1.xlarge instance as cloud node. We conduct experiments for DL inferences with the MobileNetV1 model while varying (i) network connection, (ii) number of active users, and (iii) accuracy requirement. We consider three possible execution choices: (i) on device, (ii) on edge, and (iii) on cloud. The device, edge, and cloud execution choices represent executing the inference completely on the local device, on the edge, and on the cloud respectively. The detailed specifications for the end-edge-cloud setup appear in Section 5.3.

3.1 Impact of System Dynamics on Inference Performance

Network. We consider two possible levels of network connections: (i) a low-latency (regular) network that has the signal strength for better connectivity, and (ii) a high-latency (weak) network that has a weaker signal with poor connectivity. Figure 1 (a) shows the response time of MobileNet

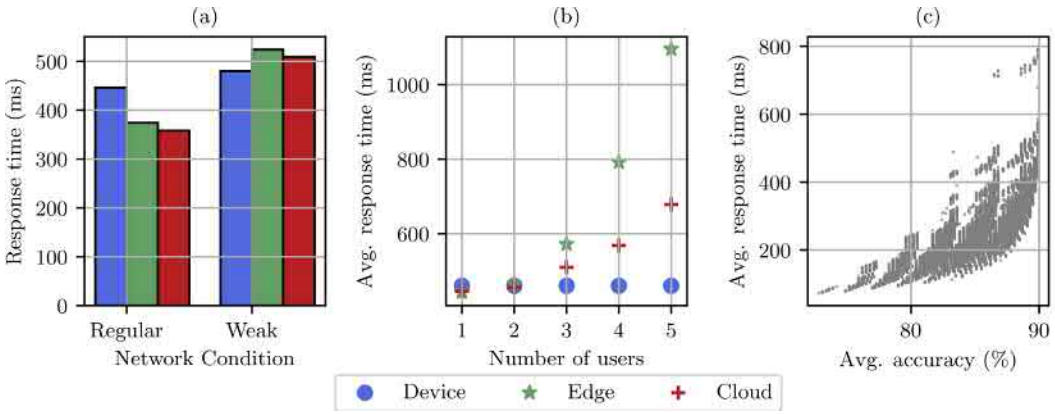

Fig. 1. Impact of varying system and application dynamics on performance for MobileNet application. (a) Response time on user-end device, edge and cloud layers with regular and weak network conditions. (b) Average response time with varying number of active users for different computing schemes. (c) Average response time achieved with varying levels of average accuracy.

application on user device, edge, and cloud layers with regular and weak networks. With a regular network, the response time is highest for executing the application on the user end device. The response time decreases as the computation is offloaded to edge and cloud layers, with the higher computational resources. With a weak network, the response time of the edge and cloud layers is higher, as the poor signal strength adds delay. The response time of the edge node in this case is higher than the cloud layer, given the lower compute capacity of the edge node. Performance of the user end device is independent of the network connection, resulting in lowest response time. This demonstrates the spectrum of response times achievable with compute nodes at different layers, under varying network constraints. For example, the best execution choice with a regular network is the cloud layer, whereas it is the local execution with a weak network.

Users. We examine user variability by considering multiple simultaneously active users ranging from 1 to 5. Figure 1 (b) shows the average response time with varying number of users. The average response time remains constant when running the application on a user end device, i.e., each user executes the application on their local device. When offloaded to the edge layer, the average response time increases significantly as the number of users increase. This is attributed to the increased network load with multiple simultaneously active users as well as limited resources at the edge layer to handle several user requests concurrently. The average response time also increases when offloaded to the cloud layer as the number of simultaneous users increases. However, the response time is lower when compared to the edge layer, since the cloud layer has a larger volume of resources to handle multiple simultaneous user requests.

Accuracy. We demonstrate the impact of varying DL models on performance under different system dynamics. We select between eight models with Top-5 accuracy between %72.8 and %89.9, while also considering all three layers for execution, and between 1 and 5 simultaneously active users. Figure 1 (c) shows the average response time achieved with varying levels of average accuracy over a multi-dimensional space of different execution choice and different number of users. Each point in Figure 1 (c) represents a unique case of an execution choice (among device, edge, and cloud), number of active users (among 1 to 5), and accuracy level. We present the average response time achieved with different levels of accuracy. As expected, the response time increases with

Table 1. Reinforcement Learning Based Works. *CO* represents the computation offloading technique. *HW* and *APP* represents knobs belong to the hardware and application layer, respectively.

Related Works	Real System Evaluation	Multi-User	End-to-End	Actions
[3, 5, 23, 29, 40]	✗	✗	✗	CO
[15]	✓	✗	✗	CO, HW
[1, 4, 13, 16, 20, 32, 38]	✗	✓	✗	CO
Ours	✓	✓	✓	CO, APP

increase in model accuracy. However, we observe tradeoffs among different response times between accuracy and number of active users. For instance, it is possible to support multiple users within the response time of servicing a single user, by lowering the model accuracy.

Considering the three major sources of variations in number of users, network conditions, and model accuracy, finding an optimal choice of execution for end-edge-cloud architectures at runtime is challenging. As such architectures scale in the number of users and edge nodes, the accuracy-performance Pareto-space becomes increasingly cumbersome for finding an optimal configuration among the fine-grained choices. Brute force and smart search algorithms do not offer practically feasible solutions to orchestrate applications in real-time. While machine learning algorithms can identify near-optimal configuration choices, they require exhaustive training, considering continuously varying system dynamics. We propose to employ online reinforcement learning to understand the volatility of system dynamics and make near-optimal orchestration decisions in real-time to improve the response time of DL inferencing on end-edge-cloud architectures.

3.2 Related Work

We categorize research related to optimally deploying DL services at the edge in two ways: (i) work related to deploying DL inference tasks over the end-edge-cloud collaborative architecture, and (ii) work related to adopting reinforcement learning methods to optimally offload tasks.

DL Inference in End-edge-cloud Networks. Prior works propose frameworks to decompose DL inference into tasks and perform distributed computations. In these works, a DL model can be partitioned vertically or horizontally along the end-edge-cloud architecture. Generally, DL models are partitioned according to the compute cost of model layers and required bandwidth for each layer to be distributed among the end-edge-cloud [11, 12, 30, 41]. These works find the optimal partition points based on traditional optimization techniques and offer design-time optimal solutions. Some efforts try to reduce the computation overhead of DL tasks through various model optimization methods such as quantization. These methods transform or re-design models to fit them into resource-constrained edge devices with little loss in accuracy [7, 9, 22]. AdaDeep [18] proposes a Deep Reinforcement Learning method to optimally select from a pool of compressed models according to available resources. However, AdaDeep relies only on the model selection technique while our work combines computation offloading and model selection techniques to achieve the optimal response time.

Learning-based Offloading. Prior works address the offloading problem to optimize different objectives including latency and energy consumption. Most of the works formulate the offloading problem with limited number of influential parameters and adopt online learning techniques with numerical evaluation [1, 3–5, 13, 16, 23, 29, 38, 40]. Lu et. al. [20] propose a Deep Recurrent Q-Learning algorithm based on Long Short Term Memory network to minimize the latency for multi-service nodes in large-scale heterogeneous MEC and multi-dependence in mobile tasks. The

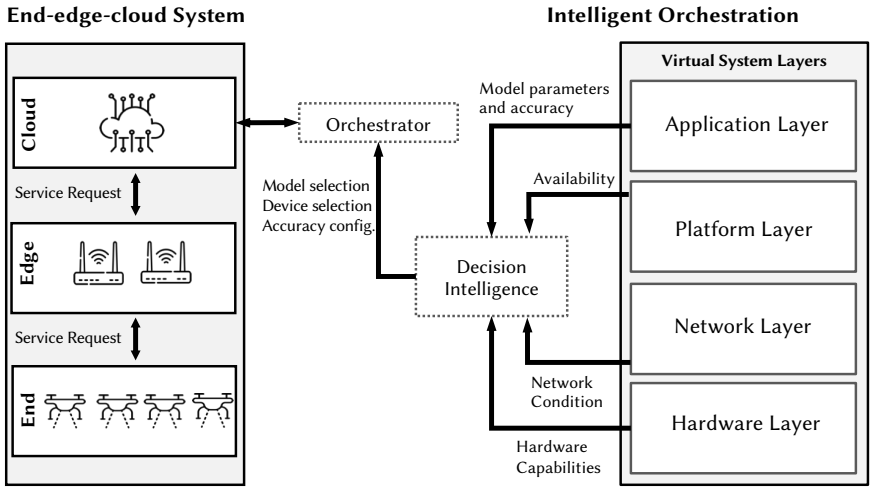

Fig. 2. Intelligent orchestration of DL inference in end-edge-cloud architectures.

algorithm is evaluated in iFogSim simulator with Google Cluster Trace. [32] proposes a Q-Learning based algorithm to minimize energy by considering various parameters in task characteristics and resource availability. Young Geun et al. [15] propose a reinforcement learning based offloading technique for energy efficient deep learning inference in the edge-cloud architecture. The work focuses on the learning for heterogeneous systems and lacks a comprehensive solution for multi-users end-edge-cloud systems. Table 1 positions our work with respect to state-of-the-art solutions. Our solution uses RL to optimally orchestrate DL inference in multi-user networks considering offloading and DL model selection techniques combined together.

3.3 Contributions

The ideal DL inference deployment provides maximum inference accuracy and minimum response time. Figure 2 shows an abstract overview of our target multi-layered architecture for online computation offloading of DL services. We consider three layers viz., user-end device, edge and cloud. Further, we classify this architecture into virtual system layers that include application, platform, network and hardware layers. Each of the virtual system layers provide sensory inputs for monitoring system and application dynamics such as DL model parameters, accuracy requirements, availability of devices for execution, network characteristics, and hardware capabilities. The *Decision Intelligence* component in Figure 2 periodically monitors resource availability from all virtual system layers to determine appropriate execution choice and DL models to achieve the required QoS (e.g, accuracy, response time). *Decision Intelligence* analyzes the system parameters to make orchestration decisions in terms of model selection, accuracy configuration, and offloading choices. The orchestrator is a software component that is hosted at the cloud layer and enforces the orchestration decisions upon receiving a service request from the user-end devices.

Finding an optimal computation policy including offloading and model selection to optimize objectives (e.g., accuracy, response time) is considered an NP-hard problem. The problem generally can be solved using traditional optimization techniques such as heuristic-based methods, meta-heuristic methods, or exact solutions. Due to slow convergence time, traditional optimization techniques for high-dimensional problems are not good candidates for runtime decision-making to

Table 2. Notation descriptions

Notation	Description
S	end-node device
E	edge device
C	cloud device
P	processor utilization
M	memory utilization
B	network condition
o	offloading decision
o_i^j	offloading decision for end-node i to resource j
N	number of end-node devices
d_k	DL model k
l	number of available DL models
T_{res}^j	response time for offloading DL task to resource j
α	learning rate
γ	discount factor

optimize objectives. Modeling an unexplored high-dimensional system is feasible using reinforcement learning techniques [34]. In this work, we use reinforcement learning to deploy DL inference at the edge by considering offloading and model selection. Some works have been proposed to address the computation offloading problem using online techniques [1, 3–5, 13, 15, 23, 29, 38, 40]. However, there is no relevant work to investigate the integration of online learning with DL inference deployment. Therefore, the literature suffers from some shortcomings that are summarized as follows:

- ***Cross-layer Optimization*:** online solutions have not previously coordinated offloading and model optimizations together. As Table 1 shows, all related work relies on only computation offloading (CO). To the best of our knowledge, for the first time, this paper considers both computation offloading and application-level adjustment (APP) together in order to achieve required QoS.
- ***Real System Evaluation*:** most RL-based solutions in the literature are numerically evaluated. Some have been proposed and evaluated with simulators. As Table 1 shows, the literature lacks a real hardware implementation for online learning framework. This paper implements the online system on real hardware devices which leads to realistic evaluation of online agent's overhead.
- ***End-to-End Solution*:** end-to-end solution considers a service from the moment a request is issued from the end-node device to delivering results to itself. Table 1 illustrates that the literature lacks an end-to-end solution.

4 ONLINE LEARNING FRAMEWORK

Our goal is to make offloading decisions and inference model selections in order to minimize inference latency while achieving acceptable accuracy. To do so, we first define the optimization problem, then we propose a reinforcement learning agent to solve the problem. Table 2 defines the notation used for the problem definition.

4.1 System Model and Problem Formulation

All computing devices in the end-edge-cloud system are represented by (S,E,C) where $S = \{S_1, S_2, .., S_n\}$ represents a set of end-node devices whose number is N; E represents the edge layer (in our case,

a single device); C represents the cloud layer. Each end-node device requires a DL inference periodically. The inference model is selected from a pool of optimized models where each model has different characteristics including computational complexity and model accuracy. All device resources are represented in a tuple $\{P_i, M_i, B_i\}$ where P_i represents processor utilization of device i; M_i represents available memory for device i; B_i represents network's connection condition between the device i and upper layer's node.

The computation offloading decision determines whether each end-node device should offload an inference to higher-layer computing resources, or perform computation locally. The offload decision for each end-node device is represented by a tuple $o_i = \{o_i^S, o_i^E, o_i^C\}$ where o_i^j represents offloading decision to layer j. If end-node device i executes at layer $j \in \{S, E, C\}$, then $o_i^j = 1$; otherwise it must be zero. For a given end-node device i, the sum of all offloading decisions $\sum_j^{\{S,E,C\}} o_i^j$ must equal 1. $o = \{o_1, o_2, ..., o_n\}$ represents the offloading decision vector for all end-node devices. The inference model selection determines the implementation of the model deployed for each inference on each end-node device. Each end-node device S_i can perform inference with one of l DL models $\{d_1, d_2, d_3, ..., d_l\}$.

In general, response time is the total time between making a request to a service and receiving the result [27]. In our case, response time is the sum of the round trip transmission time from an end-node device to the node that performs the computation, plus the computation time. Response time T_{res} for a request from end-node device i with offload decision tuple $o_i = \{o_i^S, o_i^E, o_i^C\}$ can be summarized as follows:

$$T_{res_i} = o_i^S.T_{res}^S + o_i^E.T_{res}^E + o_i^C.T_{res}^C \tag{1}$$

Our objective is to minimize the average response time while satisfying the average accuracy constraint. The problem is formulated in the following formula:

$$\begin{aligned} \textbf{P1:} \min \quad & \frac{1}{N}\sum_{i=1}^{N} T_{res_i}(o_i, d_k) \\ \text{s.t.} \quad & \overline{accuracy} > threshold \end{aligned} \tag{2}$$

where $\overline{accuracy}$ is the spatial average accuracy for simultaneous DL inferences.

4.2 Reinforcement Learning Agent

Reinforcement learning (RL) is widely used to automate intelligent decision making based on experience. Information collected over time is processed to formulate a policy which is based on a set of rules. Each rule consists three major components viz., (a) state, (b) action, and (c) reward. Among the various RL algorithms [34], Q-learning has low execution overhead, which makes it a good candidate for runtime invocation. However, it is ineffective for large space problems. There are two main problems with Q-learning for large space problems [24]: (a) required memory to save and update the Q-Values increases as the number of actions and state increases. (b) required time to populate the table with accurate estimates is impractical for the large Q-table. In our case, increasing number of users will increase the problem's space dimension. The reason is more number of users leads to more number of rows and columns in the Q-table. Therefore, it takes more time to explore every state and update the Q-values. Due to the curse of dimensionality, function approximation is more appealing [24]. The Deep Q-Learning (DQL) algorithm combines the Q-Learning algorithm with deep neural networks. DQL uses Neural Network architecture to estimate the Q-function by replacing the need for a table to store the Q-values. In this work, we build an RL agent using two reinforcement learning algorithms: (a) an epsilon-greedy **Q-Learning** and (b) a **Deep Q-Learning** algorithms. We evaluate the RL agent with the mentioned algorithms considering different problem

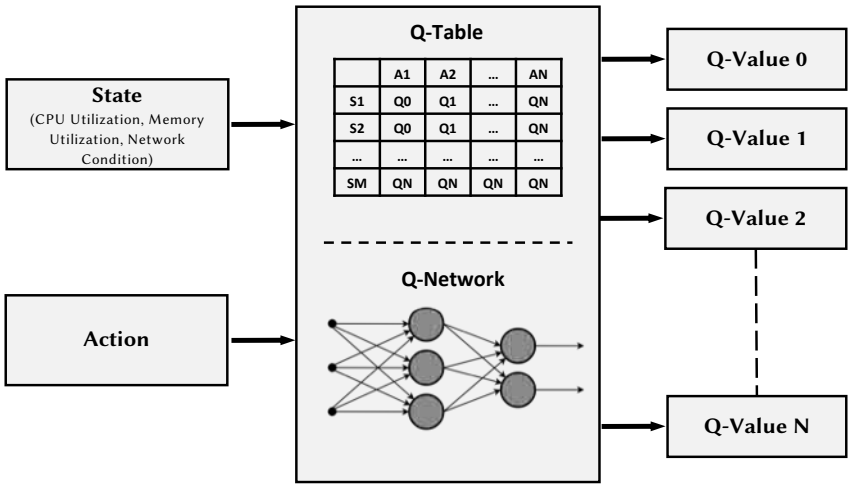

Fig. 3. Proposed reinforcement learning agent with Q-Learning and Deep Q-Learning algorithms. Q-Learning uses a Q-Table to store $Q(S, A)$ values, Deep Q-Learning estimates Q-Values with a neural network architecture.

complexities. Figure 3 depicts high-level black diagram for our agent. The RL agent is invoked at runtime for intelligent orchestration decisions. In general, the agent is composed as follows:

State Space: Our state vector is composed of CPU utilization, available memory, and bandwidth per each computing resource. Table 3 shows the discrete values for each component of the state. The state vector at time step τ is defined as follows:

$$S_\tau = \{P^E, M^E, B^E, P^C, M^C, B^C, P^{S_1}, M^{S_1}, B^{S_1}, ..., P^{S_n}, M^{S_n}, B^{S_n}\} \tag{3}$$

Action Space: The action vector consists of which inference model to deploy, and which layer to assign the inference. We limit the edge and cloud devices to always use the high accuracy inference model, and the end-node devices have a choice of l different models. Therefore, the action space is defined as $a_\tau = \{o^i, d_j\}$ where $i \in \{S, E, C\}$ and $d_j \in \{d_1, d_2, ..., d_l\}$.

Table 3. State Discrete Values

State	Discrete Values	Description
P^{S_i}	Available, Busy	End-node CPU Utilization
M^{S_i}	Available, Busy	End-node Memory Utilization
B^{S_i}	Regular, Weak	End-node Available Bandwidth
P^E	Nine discrete levels	Edge CPU Utilization
M^E	Available, Busy	Edge Memory Utilization
B^E	Regular, Weak	Edge Available Bandwidth
P^C	Nine discrete levels	Cloud CPU Utilization
M^C	Available, Busy	Cloud Memory Utilization
B^C	Regular, Weak	Cloud Available Bandwidth

Reward Function: The reward function is defined as the negative average response time of DL inference requests. In our case, the agent seeks to minimize the average response time. To ensure the agent minimizes the average response time while satisfying the accuracy constraint, the reward

R is calculated as follows:

$$\begin{aligned} &\text{if } \overline{accuracy} > \text{threshold:} \\ &\qquad R_\tau \leftarrow -Average\ Response\ Time \\ &\text{else:} \\ &\qquad R_\tau \leftarrow -Maximum\ Response\ Time \end{aligned} \tag{4}$$

To apply the accuracy constraint, the minimum possible reward is assigned when the accuracy threshold is violated. On the other hand, when the selected action satisfies the average accuracy constraint, the reward is negative average response time.

4.2.1 Q-Learning Algorithm. Q-Learning algorithm is a model-free reinforcement learning algorithm to learn the value of an action in a particular state. The algorithm does not require a model of the environment where it can handle problems with stochastic transitions and rewards without requiring adaptations. The Q-Learning algorithm stores data in a Q-table. The structure of a Q-Learning agent is a table with the states as rows and the actions as the columns. Each cell of the Q-table stores a Q-value, which estimates the cumulative immediate and future reward of the associated state-action pair. Epsilon-greedy is a common enhancement to Q-Learning that helps avoid getting stuck at local optima [34]. Algorithm 1 defines our agent's logic with the epsilon-greedy Q-Learning:

Line	**Description**
3:	First the agent determines the current system state from the resource monitors.
4-8:	Next, either the state-action pair (S_τ, A_τ) with the highest Q-value is identified to choose the next action to take, or a random action is selected with probability ϵ.
9-10:	The selected action is applied and normal execution resumes. After all inferences are completed, the reward R_τ for the execution period is calculated based on measured response time.
11-12:	Based on the resource monitors, the new state $A_{\tau+1}$ is identified, along with the state-action pair with highest Q-value.
13:	The Q-value of the previous state-action pair is updated.
14:	The current state is updated, and the loop continues.

4.2.2 Deep Q-Learning Algorithm. Q-Learning has been applied to solve many real-world problems. However, it is unable to solve high-dimensional problems with many inputs and outputs [24] as it is impractical to represent the Q-function as a Q-table for large pair of S and A. In addition, it is unable to transverse $Q(S, A)$ pairs. Therefore, a neural network is used to estimate the Q-values. The network uses the state of the environment as an input and the output is the Q-value for each actions. The neural network approximation is capable of handling high dimensional space problems [39]. One of the main problems with Deep Q-Learning is stability [24]. In order to reduce the instability caused by training on correlated sequential data, we improve the DQL algorithm with *replay buffer* technique [17]. During the training, we calculate the loss and its gradient using a mini-batch from the buffer. Every time the agent takes a *step* (moves to the next state after choosing an action), we push a *record* into the buffer. Algorithm 2 defines Deep Q-Learning algorithm which is described below:

Line	**Description**
4:	First the agent determines the current system state from the resource monitors.
4:9	Next, either the state-action pair S_τ, A_τ with the highest Q-value estimated by neural network (θ) is identified to choose the next action to take, or a random action is selected with probability ϵ.

Algorithm 1 Q-Learning Algorithm

Initialization in design time:
  τ represents time step
  S_τ represents state at τ
  A_τ represents action at τ
while system is on **do**
  From *Resource Monitoring*:
    $S_\tau \leftarrow$ State at step τ
  if $RAND < \epsilon$ **then**
    Choose random action A_τ
  else
    Choose action A_τ with largest $Q(S_\tau, A_\tau)$
  end if
  Monitor the response time for each devices
  Calculate reward R_τ
  From *Resource Monitoring*:
    $S_{\tau+1} \leftarrow$ State at step $\tau + 1$
  Choose action $A_{\tau+1}$ with the largest $Q(S_{\tau+1}, A_{\tau+1})$
  To *Updating Qtable*:
    $Q(S_\tau, A_\tau) \leftarrow Q(S_\tau, A_\tau) + \alpha[R_\tau + \gamma.Q(S_{\tau+1}, A_{\tau+1}) - Q(S_\tau, A_\tau)]$
  $S_\tau \leftarrow S_{\tau+1}$
end while

10:11 The selected action is applied and normal execution resumes. After all inferences are completed, the reward R_τ for the execution period is calculated based on measured response time.

12: At each time step, each record (S_τ,A_τ,R_τ,$S_{\tau+1}$) is added to a circular buffer D called the *replay buffer*.

13: We randomly sample *Batch Size records* from the buffer and then feed it to the network as mini-batch.

14: We calculate the temporal difference loss on the mini-batch and perform a gradient descent calculation to update the network. The *temporal difference loss* function calculates the *mean-square error* of the predicted and target Q-values as the loss of the mini-batch.

15: The current state is updated, and the loop continues.

5 FRAMEWORK SETUP

In this section we describe our proposed framework for dynamic computation offloading based on online learning, targeted at multi-layered end-edge-cloud architecture.

5.1 Framework Architecture

Figure 4 shows our proposed framework for end-edge-cloud architecture, integrating service requests, resource monitoring, and intelligent orchestration. The *Intelligent Orchestrator* (IO) acts as an RL-agent for making computation offloading and model selection decisions. The end-device layer consists of multiple user-end devices. Each end-device has two software components: (i) *Intelligent Service* - an image classification kernel with DL models of varying compute intensity and prediction accuracy; (ii) *Resource Monitoring* - a periodic service that collects devices' system parameters including CPU and memory utilization, and network condition, and broadcasts the information to the edge and cloud layers. Both the edge and cloud layers also have the *Intelligent Service* and

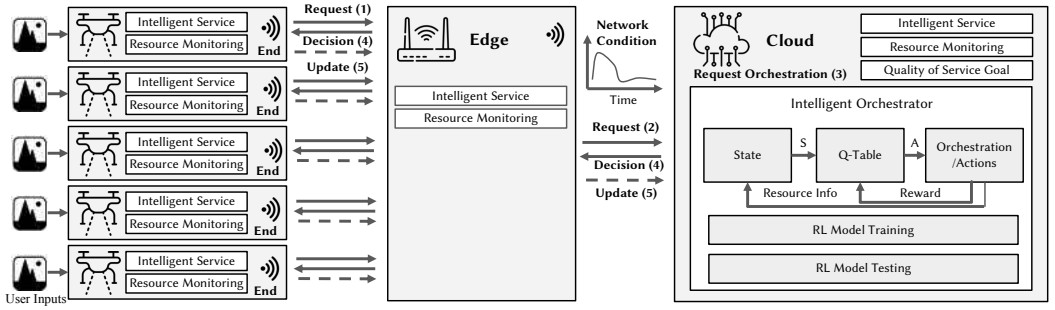

Fig. 4. Orchestration framework with online learning for orchestrating DL inference.

Resource Monitoring components. The *Intelligent Orchestrator* acts a centralized RL-agent that is hosted at the cloud layer for inference orchestration. The agent collects resource information (e.g., processor utilization, available memory, available bandwidth) from Resource Monitoring components throughout the network. The agent also gathers the reward information (i.e., response time) from the environment in order to learn an optimal policy. The agent builds the Q-function based on the RL algorithm. It builds a Q-Table for Q-Learning algorithm and a Q-Network for Deep Q-Learning algorithm based on cumulative reward obtained from the environment over time. *Quality of Service Goal* provides the required QoS for the system (i.e., the accuracy constraint).

Figure 4 illustrates the procedure step-wise of the inference service in our framework. The end-device layer consists of resource-constrained devices that periodically make requests to a DL

Algorithm 2 Deep Q-Learning Algorithm with Experience Replay

Initialization in design time:
  τ represents time step
  S_τ represents state at τ
  A_τ represents action at τ
  Initialize replay buffer D to capacity N
  Initialize action-value function Q with random weight θ
for epoch = 1, Epochs **do**
  for episode = 1, Episodes **do**
    From *Resource Monitoring*:
    $S_\tau \leftarrow$ State at step τ
    if $RAND < \epsilon$ **then**
      Choose random action A_τ
    else
      Choose action A_τ with largest $Q_\theta(S_\tau, A_\tau)$
    end if
    Monitor the response time for each devices
    Calculate reward R_τ
    Store the record (S_τ,A_τ,R_τ,$S_{\tau+1}$) into buffer D
    Sample random mini-batch of records from buffer D
    To *Updating Q-Network*:
    Compute temporal difference loss with respect to the network parameter θ
    $S_\tau \leftarrow S_{\tau+1}$
  end for
end for

inference service (step 1). The requests are passed through the edge layer (step 2) to the cloud device to be processed by *Intelligent Orchestrator* (step 3). The agent determines where the computation should be executed, and delivers the *Decision* to the network (step 4). Each device updates the agent after it performs an inference with the response time information of the requested service (step 5). In addition, all devices in the framework send the available resource information including the processor utilization, available memory, and network condition to the cloud device (step 5).

5.2 Benchmarks and Scenarios

MobileNets are small, low-latency deep learning models trained for efficient execution of image classification on resource-constrained devices [10]. For DL workloads, we consider MobileNetV1 image classification application as the benchmark [10]. We deploy the MobileNetV1 service for end-node classification. We consider eight different MobileNet models ($d0$ through $d7$) with varying levels of accuracy and performance. Each model among $d0$ through $d7$ has varying number of Multiply-Accumulate units (MACs), MAC width and data format (e.g., FP32 and Int8), exposing models with different accuracy-performance trade-offs. Table 4 summarizes the MobileNet models we consider, detailing the number of Multiply-Accumulates (MACs), MAC width and data formats (e.g., FP32 and Int8). The multiplier width is used to reduce a network's size uniformly at each layer. For a given layer and multiplier width, the number of input channels and the number of output channels is decreased and increased, respectively, by a factor of the width multiplier. During the orchestration phase, we select an appropriate model from $d0$-$d7$ to achieve the target level of classification accuracy while maximizing the performance.

Our framework supports multiple end-devices, networked with edge and cloud layers. For evaluation purposes, we set the maximum number of simultaneously active user devices to five. Each user-end device is connected to a single edge device, and can request a DL inference service to the cloud layer. The cloud layer hosts the IO that contains the RL agent, which handles the inference service requests. Upon on each service request, the RL agent is invoked to determine: (i) where the request should be processed and (ii) what DL model should be executed for the corresponding request. The RL agent's goal is to minimize average response time for all end-node devices while satisfying the accuracy constraint. This enforces quality control by imposing a strict threshold on the average DL model accuracy. In this work, we conduct experiments under four unique scenarios with varying network conditions. Each scenario represents a combination of regular (R) and weak (W) network signal strength over five user-end devices (S1-S5) and 1 edge device (E). The experimental scenarios are summarized in Table 5. The regular network has no transmission delay, while we add 20ms delay to all outgoing packets to emulate the weak connection behavior.

Table 4. MobileNet Models [10]

#	Model	Million MACs	Type	Top-1 Accuracy (%)	Top-5 Accuracy (%)
$d0$	1.0 MobileNetV1-224	569	FP32	70.9	89.9
$d1$	0.75 MobileNetV1-224	317	FP32	68.4	88.2
$d2$	0.5 MobileNetV1-224	150	FP32	63.3	84.9
$d3$	0.25 MobileNetV1-224	41	FP32	49.8	74.2
$d4$	1.0 MobileNetV1-224	569	Int8	70.1	88.9
$d5$	0.75 MobileNetV1-224	317	Int8	66.8	87.0
$d6$	0.5 MobileNetV1-224	150	Int8	60.7	83.2
$d7$	0.25 MobileNetV1-224	41	Int8	48.0	72.8

Table 5. Experiment Environment Setup. *R* and *W* represent *Regular* and *Weak* network condition, respectively.

Experiment	S1	S2	S3	S4	S5	E
EXP-A	R	R	R	R	R	R
EXP-B	R	W	R	W	R	W
EXP-C	W	W	W	R	R	R
EXP-D	W	W	W	W	W	W

Each experimental scenario in Table 5 shows the network condition of the specific device. Putting together the five different user devices and one edge device forms a unique combination of varying network conditions per each experimental scenario.

5.3 Experimental Setup

The platform consists of five AWS a1.medium instances with single ARM-core as end-devices connected to an AWS a1.large instance as edge device and an AWS a1.xlarge instance as cloud node. Table 6 summarizes device specifications in details. DL model inferences are executed on processor cores on all nodes using ARM-NN SDK [19]. The inference engine is a set of open-source Linux software tools that enables machine learning workloads on ARM-core-based devices. The framework's message passing protocol is implemented using web services deployed at each node. Section 7.2 provides our analysis on framework's setup overhead.

5.4 Hyper-parameters and RL Training

An RL agent has a number of hyper-parameters that impact its effectiveness (e.g., learning rate, epsilon, discount factor, and decay rate). The ideal values of parameters depend on the problem complexity, which in our case scales with the number of users (i.e., active end-node devices). Table 7 shows the different problem configurations we used to determine the hyper-parameters. We train the agent with two different learning algorithms (See Section 4.2). Our Q-Learning agent initializes a Q-table with Q-values of zero, and chooses actions using an $\epsilon - greedy$ policy where ϵ is the exploration rate. We initially set $\epsilon = 1$, meaning the agent selects a random action with probability 1, otherwise it selects an action that gives the maximum future reward (i.e., Q-value) in the given state. Although we perform probabilistic exploration continuously, we decay the exploration by epsilon decay parameter (See Table 7) per agent invocation. The Deep Q-Learning agent uses different neural network structure for different number of users as the problem complexity changes. We train DNN models with two fully connected layers where the hidden layers have 48, 64, 128 neurons for three, four, and five devices, respectively. We implement the experience replay as a FIFO buffer with size equal to 1000. In order to update the network, at each step, we randomly sample 64 records from the buffer and then use them as a mini-batch. We use $\epsilon - greedy$ policy to train the Deep Q-network, where we initially set the ϵ equal to 1.

Table 6. Device Specification

Node Type	vCPUs	Memory (GiB)	Frequency (GHz)	Bandwidth (Gbps)	Architecture
End	1	2	2.3	Up to 10	aarch64
Edge	2	4	2.3	Up to 10	aarch64
Cloud	4	8	2.3	Up to 10	aarch64

Table 7. Hyper-parameter values

	Q-Learning		Deep Q-Learning	
Number of Users	**Learning Rate (α)**	**Epsilon Decay**	**Learning Rate (α)**	**Epsilon Decay**
1	0.9	1e-1	–	–
2	0.9	1e-2	–	–
3	0.9	1e-2	1e-3	0.4
4	0.9	1e-3	1e-3	0.7
5	0.9	1e-4	1e-3	0.9

6 EVALUATION RESULTS AND ANALYSIS

In this section, we demonstrate the effectiveness of our online learning based inference orchestration. We evaluate our approach on the multi-layered end-edge-cloud framework, described in Section 4. Our approach features online reinforcement learning for intelligent orchestration, DL inference services and end-edge-cloud architectures, targeting DL inference performance. [32] presents state-of-the-art machine learning based orchestration for end-edge-cloud architecture baseline. For a fair comparison, we evaluate our approach against the strategy proposed in [32], which integrates the aforementioned features of our approach.

6.1 Performance Analysis

We evaluate our agent's ability to identify the optimal orchestration decision at each invocation. Through reinforcement learning, the agent predicts orchestration decisions including offloading policy and DL model configuration to maximize performance and meet the accuracy threshold. At design time, we determine the true optimal configuration in any given conditions of workloads, network, and number of active users using a brute force search. First, we compare our reinforcement learning based Intelligent Orchestrator's (IO) prediction accuracy against this true optimal configuration. Our proposed approach with both ***Q-Learning*** and ***Deep Q-Learning*** algorithm has yielded a 100% prediction accuracy in comparison with the true optimal configuration. Thus, our reinforcement learning based orchestration decisions always converge with the optimal solution. Next, we evaluate our agent's efficacy by comparing it with a representative state-of-art [32] baseline in terms of performance and accuracy. To implement the baseline policy into our framework, we limit the agent to actions that specify offloading decisions $a_\tau = \{o^i\}$, using the most accurate DL model. We additionally compare fixed orchestrations for points of reference. The fixed solution is limited to configurations where all end-devices either (a) perform the most accurate DL inference execution locally, (b) offload to the edge, or (c) offload to the cloud. In the following subsections, we demonstrate the efficacy of our proposed agent to find the optimal configuration in presence of different number of users (up to five). Then, we investigate its ability to adapt to network variations and evaluate its overhead. We explain the impact of varying DL models on the performance under different system dynamics and elaborate how the proposed agent follows the defined constraints.

6.1.1 User Variability. To evaluate the user variability, we consider up to five simultaneously active user-end devices, keeping the network constraints constant. We consider five different levels of accuracy thresholds viz., *Min*, *80%*, *85%*, *89%*, and *Max*. *Min* refers to the accuracy threshold for computing where no constraint is applied to the learning algorithms (See Equation 4) and *Max* represents the accuracy threshold for computing where the average accuracy constraint is set to 89.9%. We present the average response time and average accuracy for each of these thresholds using

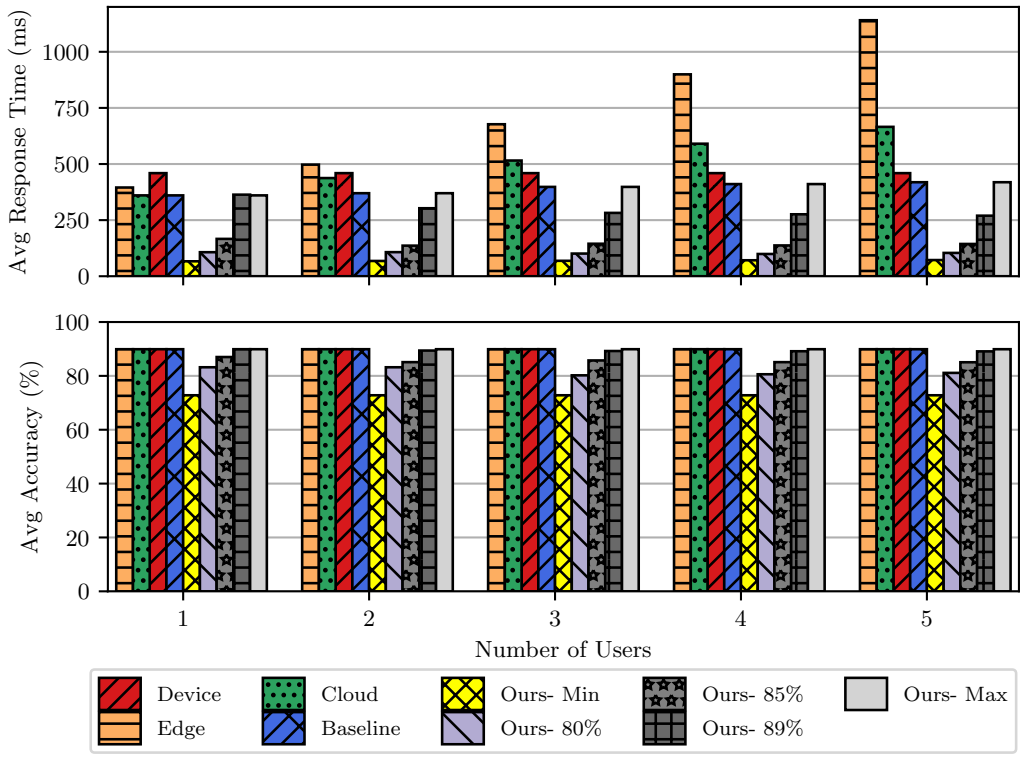

Fig. 5. Results of the framework within Exp-A for different number of active users.

our proposed approach. For evaluation, we also present the average response time and accuracy metrics achieved with the state-of-the-art baseline approach [32], and three fixed orchestration decisions viz., device only, edge only and cloud only.

Fixed Strategies Figure 5 shows the average response time and accuracy for different numbers of active users for regular network conditions (represented by scenario Exp-A in Table 5), using different orchestration strategies. The x-axis represents the number of active users. Each bar represents a different orchestration decision made by using the corresponding orchestration strategy. With the device only strategy, each user-end device executes the inference service on the local device. Thus, varying number of users has no effect on the average response time in this case. With the edge and cloud only strategies, simultaneous requests contend for edge and cloud resources. This increases the average response time significantly, as the number of users increase. For instance, the fixed edge only strategy with five active users leads to an average response time of 1140ms, while it is 665*ms* with cloud only strategy. Higher volume of available resources at the cloud layer results in relatively better average response time in comparison with the edge only strategy. On the other hand, the average response time with the device only strategy is 459*ms*, representing the optimal case.

Baseline With the baseline [32] approach, the average response time remains constant until the number of users is two. This is due to the orchestration decision of distributing the services across edge and cloud layers. As the number of users increase to three, the service requests contend for resources, leading to an increase in the average response time. With the number of users

increasing from three through five, the average response time increases, but at a relatively lower rate, exhibiting efficient utilization of the edge and cloud resources. As the number of users increase, the efficiency of the baseline approach over the fixed strategies is more prominent. Both the baseline and fixed strategies are agnostic to model selection and configuration, retaining the maximum prediction accuracy of the inference service. Thus the average accuracy remains constant with the aforementioned strategies, as shown in Figure 5.

Our proposed solution Our proposed solution achieves the same average response time in comparison with the baseline for the *Max* accuracy scenario. When the accuracy threshold is relaxed, our reinforcement learning based intelligent orchestrator selects appropriate models (among $d0$-$d7$) to improve the average response time. As the number of users increase, our solution leverages the model selection combined with offloading technique to address the potential increase in response time. With appropriate model selection, our approach reduces the compute intensity, and consequently maintains a lower average response time even with the increasing number of users. Trivially, the average response time with our approach is lower as the accuracy threshold is reduced. However, it should be noted that we enforce the boundaries on tolerable loss of accuracy with our model selection decisions. Figure 5 shows the average response time and average accuracy with our solution over different scenarios of accuracy thresholds and varying number of users. Our solution provides up to 35% improvement in the average response time in comparison with the baseline, within a tolerable loss of 0.9% accuracy. Table 8 shows the orchestration decisions of our agent for different numbers of active users, and also over four different experimental scenarios (Table 5). We present the orchestration decision and the average response time achieved with each decision, for the maximum accuracy threshold scenario.

6.1.2 Network variation. We consider two possible levels of network connection: (i) a regular network that has low latency, and (ii) a weak network that has high latency. We add 20ms delay to all outgoing packets to emulate the weak connection behavior. With varying network conditions, there is an increased delay with offloading decisions across the network. Both the baseline and fixed approaches are affected by the weak network conditions, resulting in a higher average response time. The fixed strategies employ the trivial device, edge and cloud only offloading decisions, suffering higher latency. The baseline approach is confined to only an intelligent offloading strategy, which also results in higher average response time inevitably. On the other hand, our proposed solution adapts to varying network conditions by opportunistically exploiting the accuracy trade-offs through model selection. This way, we address for the latency penalty levied by weak network conditions by reducing the compute intensity of the workloads, within the tolerable accuracy bounds.

Table 9 shows the orchestration decisions made by our intelligent orchestrator, average response time, and average accuracy achieved over varying networking conditions. Each experiment scenario (Exp-A through Exp-D) combines different network conditions for each node in the network (See Table 5). For example, in Exp-A, all the nodes are connected with regular network, whereas in Exp-B, nodes $S1$, $S3$, and $S5$ have regular connections and the rest have weak connections. We set the number of active users to five.

Model Selection Within each experiment scenario, the average response is lower as the accuracy threshold is relaxed. $d0$ through $d7$ represent models with different response time and accuracy levels. For instance, models $d0$, $d4$, $d2$, $d7$ and $d7$ are selected respectively for accuracy thresholds ranging from *Max* through *Min* in Exp-A. Our proposed orchestrator explores the Pareto-optimal space of model selection and offloading choice, combining the opportunities at application and platform layers simultaneously. For instance in Exp-A, maintaining an accuracy level of 89% results in an average response time of 269.8ms, by i) setting the models to $d4$, $d4$, $d4$, $d0$, and $d4$ on devices

Table 8. Detailed offloading decisions of our agent for different number of active users in all four experiments (Maximum Accuracy Threshold). For example, in Exp-A, the orchestrator offloads the most accurate DL inference execution ($d0$) to the cloud device ($d0, C$ for end-node $S1$). In the presence of five active users, the decisions are $\{d0, E\}$, $\{d0, L\}$, $\{d0, L\}$, $\{d0, C\}$, and $\{d0, L\}$ for end-nodes $S1$ to $S5$, respectively. In this case, $S1$, $S2$, and $S4$ perform DL inference execution of the $d0$ model locally (L). $S0$ and $S3$ offload inference execution of the $d0$ model to the edge (E) and cloud (C), respectively.

			End-node Devices					
	Experiments	**Number of Users**	**S1**	**S2**	**S3**	**S4**	**S5**	**Avg Res (ms)**
Decision	**Exp-A**	1	$d0, C$	–	–	–	–	363.47
		2	$d0, C$	$d0, E$	–	–	–	363.17
		3	$d0, C$	$d0, L$	$d0, E$	–	–	397.53
		4	$d0, L$	$d0, L$	$d0, E$	$d0, C$	–	410.35
		5	$d0, E$	$d0, L$	$d0, L$	$d0, C$	$d0, L$	418.91
	Exp-B	1	$d0, E$	–	–	–	–	403.30
		2	$d0, E$	$d0, C$	–	–	–	416.78
		3	$d0, E$	$d0, C$	$d0, L$	–	–	431.90
		4	$d0, L$	$d0, C$	$d0, E$	$d0, L$	–	457.96
		5	$d0, C$	$d0, E$	$d0, L$	$d0, L$	$d0, L$	472.88
	Exp-C	1	$d0, C$	–	–	–	–	471.65
		2	$d0, C$	$d0, E$	–	–	–	467.80
		3	$d0, C$	$d0, E$	$d0, L$	–	–	488.21
		4	$d0, C$	$d0, E$	$d0, L$	$d0, L$	–	480.70
		5	$d0, L$	$d0, L$	$d0, L$	$d0, C$	$d0, E$	464.59
	Exp-D	1	$d0, L$	–	–	–	–	585.68
		2	$d0, E$	$d0, C$	–	–	–	527.39
		3	$d0, L$	$d0, C$	$d0, E$	–	–	491.77
		4	$d0, L$	$d0, C$	$d0, E$	$d0, L$	–	501.07
		5	$d0, L$	$d0, C$	$d0, E$	$d0, L$	$d0, L$	506.62

S1-S5, and ii) device configurations to L (local device), L, L, E (edge) and L for S1-S5. However, the average response time can be improved by sacrificing the accuracy within a pre-determined tolerable level. For instance, by lowering the accuracy threshold by 4% (from 89% to 85%), the average response time can be reduced by 88% (from 269ms to 143ms) by i) setting the models to $d2$, $d6$, $d5$, $d6$, and $d5$ on devices S1-S5, and ii) device configurations to L (local device), L, L, L and L for S1-S5. With varying network conditions, our solution explores the offloading and model selection Pareto-optimal space at run-time to predict the optimal orchestration decisions.

For example, in Exp-D, our framework obtains 356.75ms on average response time with significantly weak network connectivity, while it can adapt to regular connectivity in Exp-A to obtain 269.80ms on average response time. In this case, the average accuracy is 89.1% which shows 0.8% error with the maximum average accuracy. The baseline [32] orchestrates the most accurate DL inference execution to obtain 506.62ms and 418.9ms average response time in Exp-D and Exp-A, respectively. Orchestration decisions of the baseline approach over different experimental scenarios is summarized in Table 10. Although our proposed framework and the baseline can adapt to network variability, our agent provides additional trade-off opportunities to deploy different models combined with offloading technique. This leads to up to 35% speedup while sacrificing less than 1% average accuracy.

Table 9. Results of the proposed framework for different accuracy constraints for different experiments (five users). For example, in Exp-D with 89% average accuracy constraint, our framework orchestrates $S1$, $S2$, $S3$, and $S4$ to execute DL inference using model $d4$ locally and offload inference execution using model $d0$ at the cloud. However, the baseline obtains the maximum accuracy by executing the most accurate DL inference locally for $S1$, $S4$, and $S5$ while offloading $d0$ to the edge and cloud for $S3$ and $S2$, respectively.

			End-node Devices						
	Experiments	**Constraint**	**S1**	**S2**	**S3**	**S4**	**S5**	**Avg Res (ms)**	**Avg Acc (%)**
Decision	**Exp-A**	Min	$d7, L$	$d7, L$	$d7, L$	$d7, L$	$d7, L$	72.08	72.80
		80%	$d7, L$	$d6, L$	$d6, L$	$d6, L$	$d6, L$	103.88	81.11
		85%	$d2, L$	$d6, L$	$d5, L$	$d6, L$	$d5, L$	143.81	85.06
		89%	$d4, L$	$d4, L$	$d4, L$	$d0, E$	$d4, L$	269.80	89.10
		Max	$d0, E$	$d0, L$	$d0, L$	$d0, C$	$d0, L$	418.91	89.90
	Exp-B	Min	$d7, L$	$d7, L$	$d7, L$	$d7, L$	$d7, L$	106.76	72.80
		80%	$d6, L$	$d3, L$	$d6, L$	$d6, L$	$d6, L$	139.92	83.23
		85%	$d5, L$	$d5, L$	$d6, L$	$d6, L$	$d2, L$	176.21	85.05
		89%	$d4, L$	$d4, L$	$d0, E$	$d4, L$	$d4, L$	303.50	89.10
		Max	$d0, C$	$d0, E$	$d0, L$	$d0, L$	$d0, L$	472.88	89.90
	Exp-C	Min	$d7, L$	$d7, L$	$d7, L$	$d7, L$	$d7, L$	119.28	72.80
		80%	$d6, L$	$d6, L$	$d7, L$	$d6, L$	$d6, L$	149.52	81.11
		85%	$d5, L$	$d6, L$	$d5, L$	$d6, L$	$d5, L$	190.76	85.47
		89%	$d4, L$	$d4, L$	$d4, L$	$d4, L$	$d0, C$	318.45	89.10
		Max	$d0, L$	$d0, L$	$d0, L$	$d0, C$	$d0, E$	464.59	89.90
	Exp-D	Min	$d7, L$	$d6, L$	$d7, L$	$d7, L$	$d7, L$	158.53	72.80
		80%	$d6, L$	$d6, L$	$d6, L$	$d7, L$	$d6, L$	182.53	81.12
		85%	$d2, L$	$d6, L$	$d6, L$	$d5, L$	$d5, L$	225.32	85.06
		89%	$d4, L$	$d4, L$	$d4, L$	$d4, L$	$d0, C$	356.75	89.10
		Max	$d0, L$	$d0, C$	$d0, E$	$d0, L$	$d0, L$	506.62	89.90

Table 10. Results of the state-of-the-art [32] in all four experiments.

		End-node Devices						
	Experiments	**S1**	**S2**	**S3**	**S4**	**S5**	**Avg Res (ms)**	**Avg Acc (%)**
Decision	**Exp-A**	$d0, E$	$d0, L$	$d0, L$	$d0, C$	$d0, L$	418.91	89.9
	Exp-B	$d0, C$	$d0, E$	$d0, L$	$d0, L$	$d0, L$	472.88	89.9
	Exp-C	$d0, L$	$d0, L$	$d0, L$	$d0, C$	$d0, E$	464.59	89.9
	Exp-D	$d0, L$	$d0, C$	$d0, E$	$d0, L$	$d0, L$	506.62	89.9

6.2 Overhead Analysis

Developing a global RL agent for optimal runtime orchestration decisions in an end-edge-cloud system incurs overhead to multiple sources. We evaluate the sources of overhead in both exploration and exploitation phases to demonstrate the feasibility of our proposed solution.

6.2.1 Exploration Overhead. We evaluate the time required by the proposed agent for the training phase to identify an optimal policy. Figure 6 shows the training phase for different numbers of end-devices under different accuracy constraints. We train the agent with ***Q-Learning*** and ***Deep Q-Learning*** algorithms under different accuracy constraints (See Figure 6.(a) and 6.(b), respectively). The convergence time for five devices with different policies are summarized in Table 11. Q-Learning

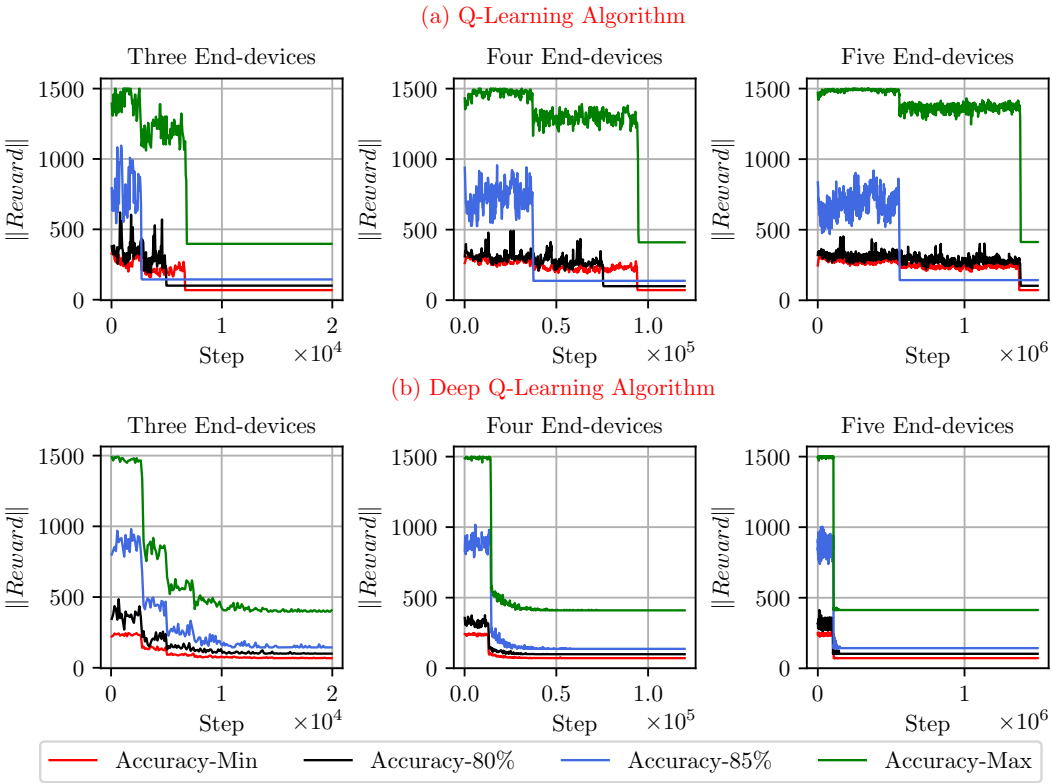

Fig. 6. Training overhead for multi-user networks with ***Q-Learning*** and ***Deep Q-Learning*** algorithms under different accuracy constraints (See Algorithm 1 and 2, respectively).

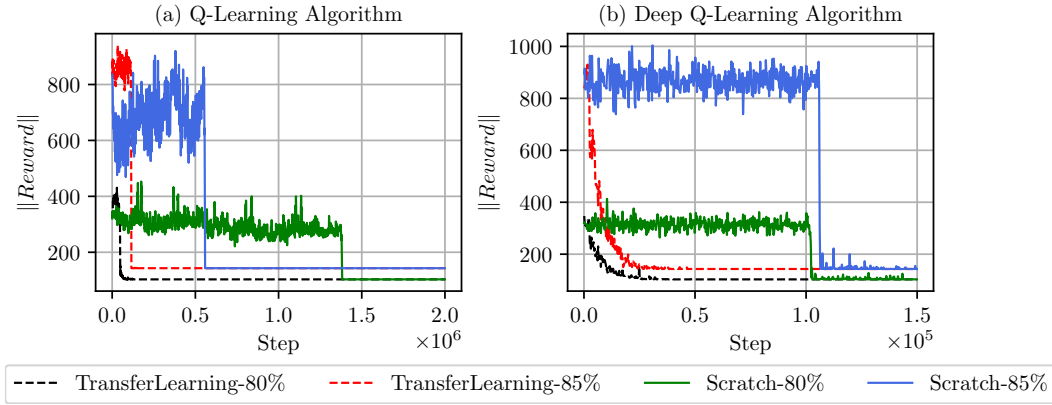

Fig. 7. Transfer learning strategy can be used to alleviate the convergence time. In our experiments, the strategy improves the convergence time up to 12.5× and 3.3× for Q-Learning and Deep Q-Learning for five End-devices, respectively. For example, the training phase for Q-Learning algorithm under 80% accuracy constraint converges at 10.5×10^5 steps. While, using the transfer learning it converges at 8.2×10^4 steps.

Table 11. Training convergence time for three, four , and five End-devices with Q-Learning and Deep Q-Learning algorithms (See Algorithm 1 and 2, respectively).

Number of Users	**Constraint**	**Q-Learning** (step #)	**Deep Q-Learning** (step #)
3	Min	6.6×10^3	15.5×10^3
	80%	1.8×10^3	14.5×10^3
	85%	0.8×10^3	16.1×10^3
	Max	6.7×10^3	18.1×10^3
4	Min	9.4×10^4	2.9×10^4
	80%	3.1×10^4	2.8×10^4
	85%	0.9×10^4	4.0×10^4
	Max	9.5×10^4	4.4×10^4
5	Min	10.5×10^5	1.0×10^5
	80%	10.5×10^5	1.0×10^5
	85%	5.6×10^5	1.0×10^5
	Max	10.5×10^5	1.0×10^5

agent converges faster than Deep Q-Learning agent for the three End-devices scenario. However, increasing the number of End-devices leads to the more complex problem. Deep Q-Learning agent converges up to 10.4× faster than Q-Learning agent for the five End-devices scenario. In other words, Deep Q-Learning algorithm converges faster for high-dimensional space problems.

In addition, we observe that the training phase can be accelerated by exploiting previous experiences in similar scenarios known as transfer learning strategy. Figure 7 shows that the strategy can alleviate the convergence up to 12.5× and 3.3× for Q-Learning and Deep Q-Learning algorithms, respectively. In the transfer learning strategy, we train a model with minimum accuracy threshold from scratch. Then, we initialize model with the trained model to reduce the convergence time. In conclusion, the Deep Q-Learning algorithm with the transfer learning strategy can speedup the convergence time up to 34× in comparison with Q-Learning algorithm for the five End-devices scenario.

6.2.2 Run-time Overhead. The agent is invoked periodically at runtime, imposing overhead on DL inference execution. We evaluate the following components individually:
(a) *Resource Monitoring:* A continuous resource monitoring service imposes runtime overhead in terms of DL inference response time. Figure 8 shows that the latency overhead for all layers is negligible (less than 0.8% of minimum response time overall).

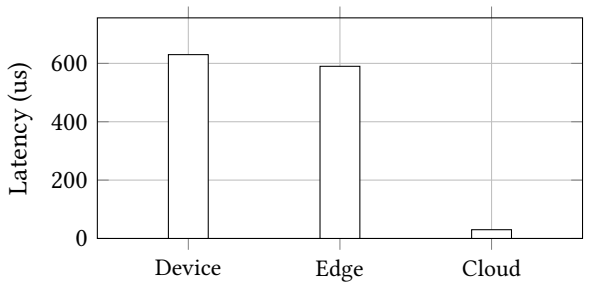

Fig. 8. Resource Monitoring Overhead

(b) *Message Broadcasting:* Sharing resource usage and orchestration decision information over the network potentially increases DL inference response time. Table 3 shows the additional network latency for different network conditions. The *request* is the latency required to send an input image to a higher layer, and dominates the sources of network overhead. We observe that the broadcasting, in total, does not impose more than 2% of overall response time.

(c) *Intelligent Orchestrator:* The Q-Learning agent's logic itself takes on average 0.6ms to execute in the cloud. While, the Deep Q-Learning agent's step takes 11*ms* on average to execute using NVIDIA RTX 5000 in cloud. During exploitation, our trained agent identifies the optimal orchestration decision within five invocations. We conclude that after an agent is trained, the improvements of 35% in average response time compared to prior art justifies the total overhead of our agent.

Table 12. Message Broadcasting Overhead

	Regular	**Weak**
Request	20 ms	137 ms
Update	0.4 ms	2 ms
Decision	1 ms	2 ms
Total	21.4 ms	141 ms

7 CONCLUSION

Cross-layer optimization that considers both model optimization and computation offloading together provides an opportunity to enhance performance while satisfying accuracy requirements. In this paper, for the first time, we proposed an online learning framework for DL inference in end-edge-cloud systems by coordinating tradeoffs synergistically at both the application and system layers. The proposed reinforcement learning-based online learning framework adopts model optimization techniques with computation offloading to find the minimum average response time for DL inference services while meeting an accuracy constraint. Using this method, we observed up to 35% speedup for average response time while sacrificing less than %0.9 accuracy on a real end-edge-cloud system when compared to prior art. Our approach shows that online learning can be deployed effectively for orchestrating DL inference in end-edge-cloud systems, and opens the door for further research in online learning for this important and growing area.

REFERENCES

[1] Md Golam Rabiul Alam, Mohammad Mehedi Hassan, Md ZIa Uddin, Ahmad Almogren, and Giancarlo Fortino. 2019. Autonomic computation offloading in mobile edge for IoT applications. *Future Generation Computer Systems* 90 (2019), 149–157.

[2] Marco V Barbera, Sokol Kosta, Alessandro Mei, and Julinda Stefa. 2013. To offload or not to offload? the bandwidth and energy costs of mobile cloud computing. In *2013 Proceedings Ieee Infocom*. IEEE, 1285–1293.

[3] Xianfu Chen, Honggang Zhang, Celimuge Wu, Shiwen Mao, Yusheng Ji, and Medhi Bennis. 2018. Optimized computation offloading performance in virtual edge computing systems via deep reinforcement learning. *IEEE Internet of Things Journal* 6, 3 (2018), 4005–4018.

[4] Zhao Chen and Xiaodong Wang. 2018. Decentralized computation offloading for multi-user mobile edge computing: A deep reinforcement learning approach. *arXiv preprint arXiv:1812.07394* (2018).

[5] BaiChuan Cheng, ZhiLong Zhang, and DanPu Liu. 2019. Dynamic Computation Offloading Based on Deep Reinforcement Learning. In *12th EAI International Conference on Mobile Multimedia Communications, Mobimedia 2019*. European Alliance for Innovation (EAI).

[6] Yu Cheng, Duo Wang, Pan Zhou, and Tao Zhang. 2017. A survey of model compression and acceleration for deep neural networks. *arXiv preprint arXiv:1710.09282* (2017).

[7] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. 2015. Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*. 3123–3131.

[8] Amir Erfan Eshratifar, Mohammad Saeed Abrishami, and Massoud Pedram. 2019. JointDNN: an efficient training and inference engine for intelligent mobile cloud computing services. *IEEE Transactions on Mobile Computing* (2019).

[9] Song Han, Jeff Pool, John Tran, and William Dally. 2015. Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*. 1135–1143.

[10] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[11] Hyuk-Jin Jeong, Hyeon-Jae Lee, Chang Hyun Shin, and Soo-Mook Moon. 2018. IONN: Incremental offloading of neural network computations from mobile devices to edge servers. In *Proceedings of the ACM Symposium on Cloud Computing*. 401–411.

[12] Yiping Kang, Johann Hauswald, Cao Gao, Austin Rovinski, Trevor Mudge, Jason Mars, and Lingjia Tang. 2017. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. *ACM SIGARCH Computer Architecture News* 45, 1 (2017), 615–629.

[13] Hongchang Ke, Jian Wang, Hui Wang, and Yuming Ge. 2019. Joint Optimization of Data Offloading and Resource Allocation With Renewable Energy Aware for IoT Devices: A Deep Reinforcement Learning Approach. *IEEE Access* 7 (2019), 179349–179363.

[14] Hakima Khelifi, Senlin Luo, Boubakr Nour, Akrem Sellami, Hassine Moungla, Syed Hassan Ahmed, and Mohsen Guizani. 2018. Bringing deep learning at the edge of information-centric internet of things. *IEEE Communications Letters* 23, 1 (2018), 52–55.

[15] Young Geun Kim and Carole-Jean Wu. 2020. Autoscale: Energy efficiency optimization for stochastic edge inference using reinforcement learning. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. IEEE, 1082–1096.

[16] Ji Li, Hui Gao, Tiejun Lv, and Yueming Lu. 2018. Deep reinforcement learning based computation offloading and resource allocation for MEC. In *2018 IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, 1–6.

[17] Long-Ji Lin. 1992. Self-improving reactive agents based on reinforcement learning, planning and teaching. *Machine learning* 8, 3-4 (1992), 293–321.

[18] Sicong Liu, Junzhao Du, Kaiming Nan, Atlas Wang, Yingyan Lin, et al. 2020. AdaDeep: A Usage-Driven, Automated Deep Model Compression Framework for Enabling Ubiquitous Intelligent Mobiles. *arXiv preprint arXiv:2006.04432* (2020).

[19] Arm Ltd. [n.d.]. IP Products: Arm NN. https://developer.arm.com/ip-products/processors/machine-learning/arm-nn

[20] Haifeng Lu, Chunhua Gu, Fei Luo, Weichao Ding, and Xinping Liu. 2020. Optimization of lightweight task offloading strategy for mobile edge computing based on deep reinforcement learning. *Future Generation Computer Systems* 102 (2020), 847–861.

[21] Pavel Mach and Zdenek Becvar. 2017. Mobile edge computing: A survey on architecture and computation offloading. *IEEE Communications Surveys & Tutorials* 19, 3 (2017), 1628–1656.

[22] Bradley McDanel, Surat Teerapittayanon, and HT Kung. 2017. Embedded binarized neural networks. *arXiv preprint arXiv:1709.02260* (2017).

[23] Minghui Min, Liang Xiao, Ye Chen, Peng Cheng, Di Wu, and Weihua Zhuang. 2019. Learning-based computation offloading for IoT devices with energy harvesting. *IEEE Transactions on Vehicular Technology* 68, 2 (2019), 1930–1941.

[24] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *nature* 518, 7540 (2015), 529–533.

[25] Seyed Sajad Mousavi, Michael Schukat, and Enda Howley. 2016. Deep reinforcement learning: an overview. In *Proceedings of SAI Intelligent Systems Conference*. Springer, 426–440.

[26] Burhan A Mudassar, Jong Hwan Ko, and Saibal Mukhopadhyay. 2018. Edge-cloud collaborative processing for intelligent internet of things: A case study on smart surveillance. In *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[27] M.R. Nakhkash et al. 2019. Analysis of Performance and Energy Consumption of Wearable Devices and Mobile Gateways in IoT Applications. In *COINS*.

[28] Jihong Park, Sumudu Samarakoon, Mehdi Bennis, and Mérouane Debbah. 2019. Wireless network intelligence at the edge. *Proc. IEEE* 107, 11 (2019), 2204–2239.

[29] Guanhua Qiao, Supeng Leng, and Yan Zhang. 2019. Online learning and optimization for computation offloading in D2D edge computing and networks. *Mobile Networks and Applications* (2019), 1–12.

[30] Xukan Ran, Haolianz Chen, Xiaodan Zhu, Zhenming Liu, and Jiasi Chen. 2018. Deepdecision: A mobile deep learning framework for edge video analytics. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*. IEEE,

1421–1429.
[31] Jürgen Schmidhuber. 2015. Deep learning in neural networks: An overview. *Neural networks* 61 (2015), 85–117.
[32] Tanmoy Sen and Haiying Shen. 2019. Machine Learning based Timeliness-Guaranteed and Energy-Efficient Task Assignment in Edge Computing Systems. In *2019 IEEE Conference on Fog and Edge Computing*. IEEE, 1–10.
[33] Sina Shahhosseini, Iman Azimi, Arman Anzanpour, Axel Jantsch, Pasi Liljeberg, Nikil Dutt, and Amir M Rahmani. 2019. Dynamic Computation Migration at the Edge: Is There an Optimal Choice?. In *Proceedings of the 2019 on Great Lakes Symposium on VLSI*. ACM, 519–524.
[34] Richard S Sutton and Andrew G Barto. 2018. *Reinforcement learning: An introduction*. MIT press.
[35] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. 2017. Efficient processing of deep neural networks: A tutorial and survey. *Proc. IEEE* 105, 12 (2017), 2295–2329.
[36] Ben Taylor, Vicent Sanz Marco, Willy Wolff, Yehia Elkhatib, and Zheng Wang. 2018. Adaptive deep learning model selection on embedded systems. *ACM SIGPLAN Notices* 53, 6 (2018), 31–43.
[37] Surat Teerapittayanon, Bradley McDanel, and Hsiang-Tsung Kung. 2017. Distributed deep neural networks over the cloud, the edge and end devices. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 328–339.
[38] Ziling Wei, Baokang Zhao, Jinshu Su, and Xicheng Lu. 2018. Dynamic edge computation offloading for Internet of Things with energy harvesting: A learning method. *IEEE Internet of Things Journal* 6, 3 (2018), 4436–4447.
[39] Phil Winder. 2020. *Reinforcement Learning*. O'Reilly Media.
[40] Jie Xu and Shaolei Ren. 2016. Online learning for offloading and autoscaling in renewable-powered mobile edge computing. In *2016 IEEE Global Communications Conference (GLOBECOM)*. IEEE, 1–6.
[41] Mengwei Xu, Feng Qian, Mengze Zhu, Feifan Huang, Saumay Pushp, and Xuanzhe Liu. 2019. Deepwear: Adaptive local offloading for on-wearable deep learning. *IEEE Transactions on Mobile Computing* 19, 2 (2019), 314–330.
[42] Ashkan Yousefpour, Caleb Fung, Tam Nguyen, Krishna Kadiyala, Fatemeh Jalali, Amirreza Niakanlahiji, Jian Kong, and Jason P Jue. 2018. All one needs to know about fog computing and related edge computing paradigms. (2018).