

Hybrid Learning for Orchestrating Deep Learning Inference in Multi-user Edge-cloud Networks

Sina Shahhosseini¹, Tianyi Hu¹, Dongjoo Seo¹, Anil Kanduri², Bryan Donyanavard³, Amir M. Rahmani¹, Nikil Dutt¹

¹ University of California, Irvine, ² University of Turku, Finland, ³ San Diego State University
{sshahhos, tianyh7, dseo3, a.rahmani, dutt}@uci.edu, spakan@utu.fi, bdonyanavard@sdsu.edu

Abstract—Deep-learning-based intelligent services have become prevalent in cyber-physical applications including smart cities and health-care. 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). Identifying optimal orchestration considering the cross-layer opportunities and requirements in the face of varying system dynamics is a challenging multi-dimensional problem. While Reinforcement Learning (RL) approaches have been proposed earlier, they suffer from a large number of trial-and-errors during the learning process resulting in excessive time and resource consumption. We present a Hybrid Learning orchestration framework that reduces the number of interactions with the system environment by combining model-based and model-free reinforcement learning. Our Deep Learning inference orchestration strategy employs reinforcement learning to find the optimal orchestration policy. Furthermore, we deploy Hybrid Learning (HL) to accelerate the RL learning process and reduce the number of direct samplings. We demonstrate efficacy of our HL strategy through experimental comparison with state-of-the-art RL-based inference orchestration, demonstrating that our HL strategy accelerates the learning process by up to 166.6×.

I. INTRODUCTION

Deep-learning (DL) kernels provide intelligent end-user services in application domains such as computer vision, natural language processing, autonomous vehicles, and healthcare [1]. End-user mobile devices are resource-constrained and rely on cloud infrastructure to handle the compute intensity of DL kernels [2]. Unreliable network conditions and communication overhead in transmitting data from end-user devices affect real-time delivery of cloud services [3]. Edge computing brings compute capacity closer to end-user devices, and complement the cloud infrastructure in providing low latency services [4]. Collaborative end-edge-cloud (EEC) architecture enables on-demand computational offloading of DL kernels from resource-constrained end-user devices to resourceful edge and cloud nodes [5], [6]. Orchestrating DL services in multi-layered EEC architecture primarily focus on i) selecting an edge node onto which a task can be offloaded, and ii) selecting an appropriate learning model to accomplish the DL task. Selection of the edge node for offloading a DL task is based on a combination of factors including i) the edge node’s compute capacity and core-level heterogeneity, ii) communication penalty incurred in offloading, iii) workload intensity of the task and accuracy constraints, and iv) run-

time variations in connectivity, signal strength, user mobility, and interaction. Selecting an appropriate model for a DL task depends on design time accuracy constraints and run-time latency constraints simultaneously. Different learning models for DL tasks expose a Pareto-space of accuracy-compute intensity, such that higher accuracy models consume longer execution time [2].

Orchestrating DL tasks by finding the appropriate edge node for offloading, and configuring the learning model for DL task, while minimizing the latency under unpredictable network conditions makes orchestration a multi-dimensional optimization problem [7]. Therefore, the orchestration problem requires an intelligent run-time management to search through a wide configuration Pareto-space. Brute force search, heuristic, rule-based, and closed-loop feedback control solutions for orchestration require longer periods of time before converging to optimal decisions, making them inefficient for real-time orchestration [8]. Reinforcement Learning (RL) approaches have been adopted for orchestrating DL tasks in multi-layered end-edge-cloud systems [7] to address these limitations. Orchestration strategies using RL can be classified into *model-free* and *model-based* approaches.

Model-free RL techniques operate with no assumptions about the system’s dynamic or consequences of actions required to learn a policy. Model-free RL builds the policy model based on data collected through trial-and-error learning over epochs [8]. Existing *Model-free* RL strategies have used Deep Reinforcement Learning (DRL) algorithms for minimizing the latency for multi-service nodes in end-edge-cloud architectures [9]. AdaDeep [10] proposes a resource-aware DL model selection using optimal learning. AutoScale [7] proposes an energy-efficient computational offloading framework for DL inference. Originated from trial-and-error learning, *model-free* RL requires significant exploratory interactions with the environment [8]. Many of these interactions are impractical in distributed computer systems, since execution for each configuration is expensive and leads to higher latency and resource consumption during the learning process [8].

Model-based RL uses a predictive internal model of the system to seek outcomes while avoiding the consequence of trial-and-error in real-time. Existing approaches have modeled the computation offloading problem and then use deep reinforcement learning to find an optimal orchestration solution [11], [12] [13]. Model-based RL approach is computationally efficient and provides better generalization and significantly less number of real full system execution runs before con-

TABLE I: State-of-the-art reinforcement learning-based orchestration frameworks for deep learning inference in end-edge-cloud networks. Approach- Model-free (MF) and Hybrid Learning (HL). Algorithm- Q-learning (QL), DeepQ (DQL), DeepDynaQ (DDQ). Actuation knobs- *CO*: computation offloading, *HW*: hardware knobs, *APP*: application layer knobs.

Technique	Approach	Algorithm	Workload	Knobs
AutoScale [7]	MF	QL	Inference	CO,HW
AutoFL [14]	MF	QL	Training	CO,HW
AdaDeep [10]	MF	DQL	Inference	APP
Ours	HL	DDQ	Inference	CO,APP

verging to the optimal solution [8]. However, Model-based RL is sensitive to model bias and suffers from model errors between the predicted and actual behavior leading to sub-optimal orchestration decisions.

A hybrid learning approach integrating the advantages of both *model-free* and *model-based* RL is efficient for orchestrating DL tasks on end-edge-cloud architectures [15], [16]. In this work, we adapt such hybrid learning strategy for orchestrating deep learning tasks on distributed end-edge-cloud architectures. We model the end-edge-cloud system dynamics online, and design an RL agent that learns orchestration decisions. We incorporate the system model into the RL agent, which simulates the system and predicts the system behaviors. We exploit the hybrid *Deep Dyna-Q* [15], [16] model to design our RL agent, which requires fewer number of interactions with the end-to-end computer system, making the learning process efficient. Existing efforts to orchestrate DL inference/training over the network are summarized in Table I. We compare our framework with AutoScale [7] and AdaDeep [10] to demonstrate our agent’s performance since they employ RL to optimally orchestrate DL inference at Edge. Our main contributions are:

- A run-time reinforcement learning based orchestration framework for DL inference services, to minimize inference latency within accuracy constraints.
- A hybrid learning approach to accelerate the RL learning process and reduce direct sampling during learning.
- Experimental results demonstrating acceleration of the learning process on an end-edge-cloud platform by up to $166.6\times$ over state-of-the-art *model-free* RL-based orchestration.

II. ONLINE LEARNING FOR DL INFERENCE ORCHESTRATION

We begin by formulating the orchestration of DL inference on EEC architecture as an optimization problem, with the constraints of minimizing latency within an acceptable prediction accuracy. We design a RL agent to solve the orchestration problem, within the latency and accuracy constraints.

A. Problem Formulation

Consider an end-edge-cloud architecture, represented as (S,E,C), where *S*, *E*, *C* represents sensory device, edge, and cloud nodes respectively. Each edge and cloud node can

TABLE II: 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

service multiple sensory end device nodes. In our model, we define n as the number of sensory end-device such that $S = \{S_1, S_2, \dots, S_n\}$, representing n end-device nodes. Each of the *S*, *E*, *C* nodes locally stores a pool of optimized inference models with different levels of computing intensity and model accuracy. Each end-device node runs an application that requires a DL inference task periodically. All end-device node resources are represented as a tuple $\{P_i, M_i, B_i\}$, where P_i represents processor utilization of the node i ; M_i represents available memory at the node i ; B_i represents network connection status between the end-device node i and edge and cloud nodes in higher layers. The computation offloading decision determines whether an end-device node should offload an inference task to a resourceful edge or cloud node, or perform the computation locally. The offloading decision for each end-device node is represented by o_i , for each end-device node i . The inference model selection determines the implementation of the model deployed for each inference on each end-node device. Each end-device node can perform inference with one of l DL models $\{d_0, d_1, d_2, \dots, d_{l-1}\}$. In general, response time is the total time between making a request to a service and receiving the result. In our case, we define $T_{res}(o_i, d_k)$ as response time for a request from end-node device i with offload decision o_i and inference model d_k . Our objective is to minimize the average response time while satisfying the average accuracy constraint.

B. RL 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 of three major components: **(a) State Space:** State describes the current system dynamics in terms of available cores, memory, and network resources. These entities affect the inference performance, hence, our state vector is composed of CPU utilization, available memory, and bandwidth per each computing resource. Table II shows the discrete values for each component of the state. **(b) Action Space:** RL actions represent the orchestration knobs in the system. We define actions as the choice of compute node among available execution options at (S,E,C) layers, and choice of inference model to deploy. 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 $d_j \in \{d_1, d_2, \dots, d_l\}$. **(c) Reward Function:** A reward in RL is a feedback from the

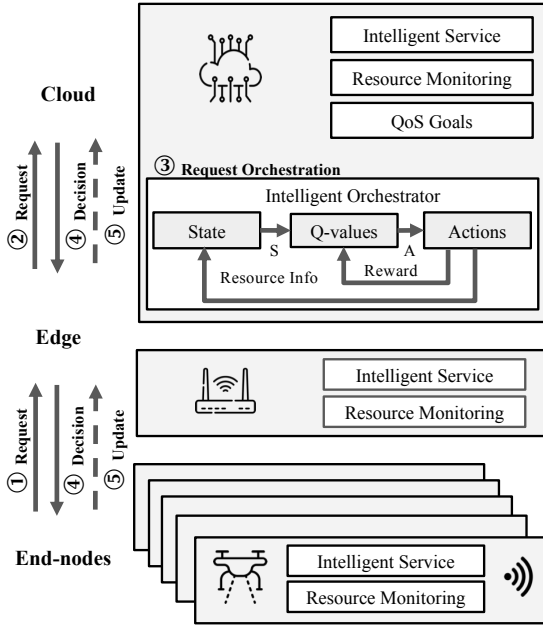


Fig. 1: Online learning framework for orchestrating DL inference.

environment to optimize objective of the system. In our work, the reward function is defined as the 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, we penalize the agent when the accuracy threshold is violated. On the other hand, when the selected action satisfies the constraint, the reward is the average response time.

C. DL Inference Orchestration Framework

Figure 1 shows our end-edge-cloud architecture framework, 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 end-user 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; and (ii) *Resource Monitoring* - a periodic service that collects the device's 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 *Resource Monitoring* components. The *Intelligent Orchestrator* acts as a centralized RL-agent that is hosted at the cloud layer for inference orchestration. The agent collects resource information including processor utilization, available memory, and network condition) from Resource Monitoring components throughout the network. The agent also gathers the reward value (i.e., response time) from the environment in order to learn an optimal policy. The *Quality of Service Goal* provides the required QoS for the system (i.e., the accuracy constraint). Figure 1 illustrates a step-wise procedure of the inference service in our framework. The end-device layer consists of resource-constrained devices that periodically make requests

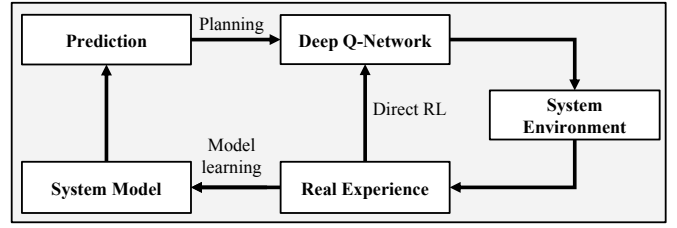


Fig. 2: Hybrid Learning Architecture.

for DL 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 send the available resource information including the processor utilization, available memory, and network condition to the cloud device (Step 5).

III. HYBRID LEARNING STRATEGY

Hybrid Learning is a combination of *model-free* and *model-based* RL. As illustrated in Figure 2, the architecture consists of *System Environment*, *System Model*, and *Policy Model*. The training process begins with an initial system model and an initial policy. The agent is trained in three phases viz., **Direct RL**, **System Model Learning**, **Planning**. Algorithm 1 defines the training process, which is composed of three major phases:

(1) Direct RL: In the Direct-RL phase, the agent interacts with the *System Environment* to collect Real Experience for training the Deep Q-Network (DQN) model. Every time the agent takes a *step*, the *real experience* is pushed into a prioritized replay buffer D_{direct} , and a random replay buffer D_{world} . We sample mini-batches from the buffer D_{direct} and update the DQN by Adam optimizer [17]. Then, we assign new priorities to the prioritized replay buffer D_{direct} .

(2) System Model Learning: We model our system to *Predict* the system's behavior for given pairs of (s_τ, a_τ) . *System Model Learning* starts with no assumption about the *System Environment*, and is learned and updated through real experiences. As the agent takes more steps with real experiences, the model ($System(\theta_s)$) learns the system from state-action pairs that have previously been experienced (in Direct RL). $System(s_\tau, a_\tau; \theta_s)$ predicts average response time r_τ and the next state $s_{\tau+1}$. In this phase, we train the model with mini-batches sampled randomly from the buffer D_{world} update the θ_s accordingly.

(3) Planning: During this phase, the agent uses the *System Model* to predict the system's behavior to improve the policy model $Q(s, a; \theta_Q)$. We train $Q(s, a; \theta_Q)$ with the predicted tuples $(s_\tau, a_\tau, r_\tau, s_{\tau+1})$ in a replay buffer D_{plan} . Given a current state s , we use $System(s, a; \theta_s)$ to generate a set A of K actions $(a_i, i = 0, \dots, K - 1)$ that might yield promising rewards r . For each action a_i in the set A , if a_i does not exist in the buffer D_{plan} , agent will take a step a_i at state s to get a reward r and the next state s' . Then

we push the tuple (s, a_i, r, s') into the buffer D_{plan} . If the action a_i already exists in buffer D_{plan} , we only update its corresponding current state in the buffer with the new state. Then, we train the policy model $Q(s, a; \theta_Q)$ in the same way as the Direct RL process but with the generated data sampled from the buffer D_{plan} .

We define α as a parameter to control the portion of *Direct RL* and *Planning* during the training policy. Increasing α over time results in decreasing the number of Direct RL during the training. In this strategy, after sufficient real experience, the *System Model* can predict the system behavior. Therefore, the agent relies more on *System Model* prediction rather than real experiences which emphasizes *model-based RL*.

Algorithm 1: Hybrid Learning Algorithm.

Data: ϵ, C, T, N
Result: $Q(s, a; \theta_Q), System(s, a; \theta_s)$

- 1 initialize $Q(s, a; \theta_Q)$ and $Q'(s, a; \theta_{Q'})$ with $\theta_{Q'} \leftarrow \theta_Q$;
- 2 initialize $System(s, a; \theta_s)$;
- 3 initialize replay buffer $D_{direct}, D_{world}, D_{plan}$;
- 4 **for** ($epoch \leftarrow 1 : N$) {
- 5 $\alpha \leftarrow \frac{epoch}{N}$;
- 6 # **Direct Reinforcement Learning** —————;
- 7 **for** ($session \leftarrow 1 : (1 - \frac{\alpha}{2})N_{direct}$) {
- 8 **for** ($step \leftarrow 1 : T_{direct}$) {
- 9 with probability ϵ , choose random a , otherwise
- 10 $a \leftarrow \operatorname{argmin}_{a'} Q(s, a'; \theta_Q)$;
- 11 $r, s' \leftarrow \operatorname{take\ step}(s, a)$;
- 12 store $(s, a, r, s') \rightarrow D_{direct}, D_{world}$;
- 13 $s \leftarrow s'$;
- 14 sample prioritized minibatch $(S, A, R, S') \subset D_{direct}$;
- 15 update θ_Q via Adam on the minibatch;
- 16 update target model $Q'(s, a; \theta_{Q'})$ by $\theta_{Q'} \leftarrow \theta_Q$;
- 17 # **System Model** —————;
- 18 **for** ($session \leftarrow 1 : (1 - \frac{\alpha}{2})N_{world}$) {
- 19 sample random minibatch $(S, A, R, S') \subset D_{world}$;
- 20 update θ_s ;
- 21 # **Planning** —————;
- 22 **for** ($session \leftarrow 1 : (\frac{\alpha+1}{2})N_{suggest}$) {
- 23 **for** ($step \leftarrow 1 : T_{suggest}$) {
- 24 $a \leftarrow \operatorname{argmin}_{a'} System(s, a'; \theta_s)$;
- 25 $s' \leftarrow System(s, a; \theta_s)$;
- 26 # **Choose best K actions for current state s** ;
- 27 $A \leftarrow \{a | a \in \operatorname{argsort}_{a'}(System(s, a'; \theta_s))[0:K]\}$;
- 28 **for** ($a' \in A$) {
- 29 **if** $a' \notin D_{plan}$ **then**
- 30 $r, s' \leftarrow \operatorname{take\ step}(s, a')$;
- 31 store $(s, a', r, s') \rightarrow D_{plan}$;
- 32 **else**
- 33 update $(s'', a', r, s') \in D_{plan}$ by $s'' \leftarrow s$;
- 34 $s \leftarrow s'$;
- 35 **for** ($session \leftarrow 1 : (\frac{\alpha+1}{2})N_{plan}$) {
- 36 sample prioritized minibatch from D_{plan} ;
- 37 update θ_Q via Adam on the minibatch;
- 38 update target model $Q'(s, a; \theta_{Q'})$ by $\theta_{Q'} \leftarrow \theta_Q$;

IV. EVALUATION

A. Experimental Setup

In this subsection, we describe the experimental setup for our proposed framework. First, we explain the DL workloads

TABLE III: MobileNet Models [18]

#	Model	Million MACs	Type	Accuracy (%)
$d0$	1.0 MobileNetV1-224	569	FP32	89.9
$d1$	0.75 MobileNetV1-224	317	FP32	88.2
$d2$	0.5 MobileNetV1-224	150	FP32	84.9
$d3$	0.25 MobileNetV1-224	41	FP32	74.2
$d4$	1.0 MobileNetV1-224	569	Int8	88.9
$d5$	0.75 MobileNetV1-224	317	Int8	87.0
$d6$	0.5 MobileNetV1-224	150	Int8	83.2
$d7$	0.25 MobileNetV1-224	41	Int8	72.8

TABLE IV: Experiment Environment Setup. R and W represent *Regular* and *Weak* network condition, respectively.

Exp	S1	S2	S3	S4	S5	E
A	R	R	R	R	R	R
B	R	W	R	W	R	W
C	W	W	W	R	R	R
D	W	W	W	W	W	W

as benchmarks in our evaluations. Then, we explain the scenarios to evaluate our framework. Finally, we describe the platform setup. For DL workloads, we consider the MobileNet image classification application as the benchmark [18]. We consider eight different MobileNet models [18] ($d0$ through $d7$) with varying levels of accuracy and performance. Table III summarizes the MobileNet models $d0$ through $d7$, with each model having different response time and accuracy levels. Our framework supports up to five end-device nodes, networked with edge and cloud layers. Each end-user 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 orchestration. 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 platform consists of five AWS a1.medium instances with single ARM-core (as the end-device nodes), connected to an AWS a1.large instance with two CPUs (as edge device), and an AWS a1.xlarge instance with four CPUs (as cloud node). We perform the training process using NVIDIA RTX 5000 at the cloud node. In this work, we conduct experiments under four unique scenarios with varying network conditions (See Table IV). Each scenario represents a combination of regular (R) and weak (W) network signal strength over five end-user devices (S1-S5) and 1 edge device (E). The regular network has no transmission delay, while we add 20ms delay to all outgoing packets for the weak connection.

B. Results

We demonstrate the efficacy of our hybrid learning strategy for RL-based inference orchestration compared to two state-of-the-art RL-based inference orchestration [10], [7]: AdaDeep [10] employs the DQL algorithm to optimally orchestrate DL model selection based on available resources; AutoScale [7] applies QL algorithm to optimally orchestrate DL inference in end-edge architecture (See Table I). We evaluate the performance of the agent on a multi-user end-edge-cloud framework (see Section II). We also report the overhead

incurred by the agent in our learning process, compared with AdaDeep [10] and AutoScale [7].

1) *Agent Performance*: We demonstrate our proposed hybrid learning agent’s performance in finding optimal orchestration decisions. At design time, we determine the true optimal configuration for orchestrating a DL task under any given condition of workloads, network, and number of active users using a brute force search. This is used for comparing the orchestration decisions made by our proposed approach and DQL against the true optimal configuration. Both Deep-Q Learning (DQL) and Hybrid Learning (HL) algorithms have yielded a 100% prediction accuracy in comparison with the true optimal configuration. Thus, RL-based orchestration decisions always converge with the optimal solution. We investigate the agent’s ability to find the optimal orchestration decisions under different scenarios of varying network conditions. Table IV summarizes the experimental scenarios A-D with different combinations of regular (R) and weak (W) networks. For example, in experimental scenario A, all the nodes are connected with a regular network, whereas in scenario B, nodes $S1$, $S3$, and $S5$ have regular connections and the rest have weak connections. Orchestration decisions made by our proposed hybrid agent over four different experimental scenarios (A-D) are shown in Table V. For each end-user sensor device node ($S1$ - $S5$) within each experimental scenario (A-D), we present the orchestration decisions viz., choice of execution node (among local device L, edge node E, cloud node C) and inference model ($d0$ - $d7$, in decreasing order of accuracy) made by our hybrid agent. Table V also shows the average response time (ART, in ms) and average accuracy with the selected model (AA, in %), along with the constraint (Cnst) on minimum accuracy requirement. Note that within each experimental scenario, the average response is lower as the accuracy threshold is relaxed. For instance, in experimental scenario A for device $S1$, models $d0$, $d4$, $d2$, $d7$ and $d7$ are selected respectively for accuracy thresholds ranging from *Max* through *Min*. Our proposed orchestrator explores the Pareto-optimal space of model selection and offloading choice of nodes to minimize latency within accuracy constraints. For instance in experimental scenario 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 $S1$ - $S5$, 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 predetermined tolerable level. For instance, by lowering the accuracy threshold by 4% (from 89% to 85%), the average response time can be reduced by 46% (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 to predict the optimal orchestration decisions.

2) *Training Overhead*: We evaluate the agent overhead during the training phase to demonstrate the efficiency of the hybrid learning algorithm in comparison with the state-of-the-art AutoScale [7] and AdaDeep [10]. To identify an optimal policy, we assess the number of steps required to interact

TABLE V: Results of the framework for different accuracy constraints for different experiments (five users). Cnst, ART, and AA represent constraint, average response time, and average accuracy, respectively. 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 execution using model $d0$ at the cloud.

Exp	End-node Devices						ART (ms)	AA (%)
	Cnst	S1	S2	S3	S4	S5		
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
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
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
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

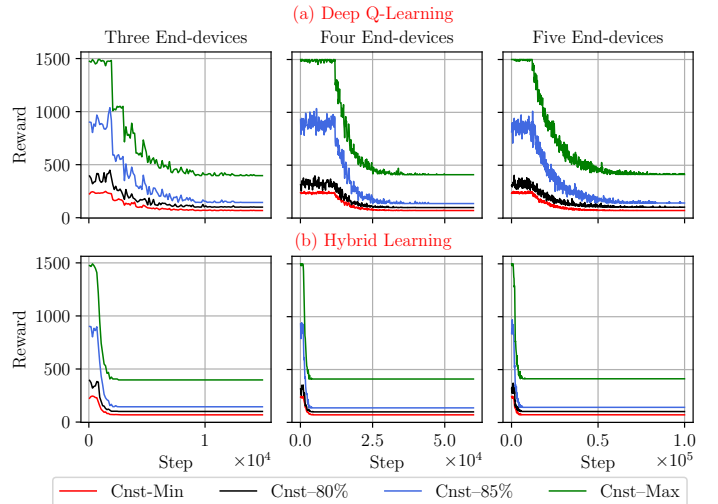


Fig. 3: Convergence time for up to five users within different constraints. Cnst represents constraint for each experiment. Deep Q-Learning refers to AdaDeep [10] work. The comparison with AutoScale [7] is mentioned in Table VI.

with the system environment under each approach. Figure 3 shows the training phases for different number of users under different accuracy constraints. Each subplot shows the training phase for the system with a different number of users using the DQL and HL algorithms. The agent is trained under different accuracy constraints, which results in converging to different optimal policies for the corresponding constraint (i.e., different converged reward values). Our evaluation shows that the HL algorithm accelerates the training steps up to 11.6× and

TABLE VI: Training overhead for Hybrid Learning algorithm compared with AdaDeep [10] and AutoScale [7]. Training overhead is presented as number of steps to achieve the optimal policy.

# of Users	Constraint	AutoScale	AdaDeep	Our
3	Min	0.7×10^4	0.1×10^5	0.2×10^4
	80%	0.5×10^4	0.1×10^5	0.2×10^4
	85%	0.3×10^4	0.1×10^5	0.2×10^4
	Max	0.7×10^4	0.1×10^5	0.2×10^4
4	Min	0.9×10^5	0.3×10^5	0.3×10^4
	80%	0.8×10^5	0.4×10^5	0.4×10^4
	85%	0.4×10^5	0.4×10^5	0.3×10^4
	Max	0.9×10^5	0.3×10^5	0.3×10^4
5	Min	0.1×10^7	0.6×10^5	0.6×10^4
	80%	0.1×10^7	0.6×10^5	0.6×10^4
	85%	0.6×10^6	0.7×10^5	0.6×10^4
	Max	0.1×10^7	0.7×10^5	0.6×10^4

166.6 \times in comparison with AdaDeep [10] and AutoScale [7] respectively. The convergence steps for different number of users are summarized in Table VI. Our result shows that the number of agent’s interactions with the system environment increases as we increase the dimension of the problem space (increasing number of users). However, our agent with the HL algorithm outperforms the state-of-the-art [7], [10] in the number of interactions with the system environment for the different number of users and under different accuracy constraints. The training time consists of experience time (i.e., time spent in interacting with the system environment to collect data) and computation time for learning the system and policy models. Table VII shows the overall training time for different number of users. Our evaluation shows that the HL algorithm converges up to 7.5 \times faster in comparison with AdaDeep, and 109.4 \times faster in comparison with A.. Further, we also present the overhead of the training agent in finding optimal orchestration decisions, with *experience time* and *computation time* metrics. *Experience time* is the total time to execute all taken steps (cost of interaction with the system environment) to identify an optimal policy, while *computation time* is the time to train the agent. The HL algorithm results in 4.4 \times and 9.4 \times speedup in comparison with AdaDeep for *Computation Time* and *Experience Time*, respectively. The computation time per step to train *System Model* and *Policy Model* with the HL algorithm is higher than the DQL and QL algorithms. However, our HL algorithm converges significantly faster than the algorithms (See Table VI and Table VII). Therefore, any additional computation cost per step is more than compensated by a significant reduction in the number of interactions to identify an optimal policy.

V. CONCLUSION

We presented a hybrid learning based framework for orchestrating deep learning tasks in end-edge-cloud architectures. Our proposed hybrid learning strategy requires fewer interactions with the real-time execution runs, converging to an optimal solution significantly faster than state-of-the-art model-free RL approaches. We deployed our proposed framework on enterprise AWS end-edge-cloud system for evaluating

TABLE VII: Training time (presented in minutes) for different number of users compared with AutoScale [7] and AdaDeep [10]. Comp and Exp represent *Computational Time* and *Experience Time*. *AutoScale employs the QL algorithm which has a low computational overhead.

# of Users	Time (min)	AutoScale*	AdaDeep	Ours
3	Comp	-	1	1
	Exp	1.5×10^2	6.8×10^1	2.6×10^1
	Total	1.5×10^2	6.9×10^1	2.7×10^1
4	Comp	0.3	1.0×10^1	3.6
	Exp	3.7×10^2	1.1×10^2	1.3×10^1
	Total	3.7×10^2	1.2×10^2	1.6×10^1
5	Comp	1.0×10^1	1.5×10^2	3.4×10^1
	Exp	5.8×10^3	1.8×10^2	1.9×10^1
	Total	5.8×10^3	3.3×10^2	5.3×10^1

MobileNet kernels. Our hybrid learning approach accelerates the training process by up to 166 \times in comparison with the state-of-the-art RL-based DL inference orchestration, while making optimal orchestration decisions after significantly early convergence. Our future work will explore cross-layer opportunities and more hardware-friendly RL algorithms.

REFERENCES

- [1] J. Schmidhuber, “Deep learning in neural networks: An overview,” *Neural networks*, 2015.
- [2] X. Wang *et al.*, “Convergence of edge computing and deep learning: A comprehensive survey,” *IEEE Communications Surveys & Tutorials*, 2020.
- [3] H. Khelifi *et al.*, “Bringing deep learning at the edge of information-centric internet of things,” *IEEE Communications Letters*, 2018.
- [4] A. Yousefpour *et al.*, “All one needs to know about fog computing and related edge computing paradigms: A complete survey,” *Journal of Systems Architecture*.
- [5] S. Shalhosseini *et al.*, “Exploring computation offloading in iot systems,” *Information Systems*.
- [6] S. Shalhosseini *et al.*, “Dynamic computation migration at the edge: Is there an optimal choice?,” in *GLSVLSI’19*.
- [7] Y. G. Kim *et al.*, “Autoscale: Energy efficiency optimization for stochastic edge inference using reinforcement learning,” *IEEE*, 2020.
- [8] R. S. Sutton *et al.*, *Reinforcement learning: An introduction*. MIT press, 2018.
- [9] H. Lu *et al.*, “Optimization of lightweight task offloading strategy for mobile edge computing based on deep reinforcement learning,” *Future Generation Computer Systems*, 2020.
- [10] S. Liu *et al.*, “Adadeep: A usage-driven, automated deep model compression framework for enabling ubiquitous intelligent mobiles,” 2020.
- [11] W. Zhan *et al.*, “Deep-reinforcement-learning-based offloading scheduling for vehicular edge computing,” *Internet of Things Journal*, 2020.
- [12] B. Lin *et al.*, “Computation offloading strategy based on deep reinforcement learning for connected and autonomous vehicle in vehicular edge computing,” *Journal of Cloud Computing*, 2021.
- [13] J. Wang *et al.*, “Computation offloading in multi-access edge computing using a deep sequential model based on reinforcement learning,” *Communications Magazine*, 2019.
- [14] Y. G. Kim and C.-J. Wu, “Autofl: Enabling heterogeneity-aware energy efficient federated learning,” in *MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, pp. 183–198, 2021.
- [15] B. Peng *et al.*, “Deep dyna-q: Integrating planning for task-completion dialogue policy learning,” *arXiv preprint arXiv:1801.06176*, 2018.
- [16] R. S. Sutton, “Integrated architectures for learning, planning, and reacting based on approximating dynamic programming,” in *Machine learning proceedings 1990*, 1990.
- [17] Z. Zhang, “Improved adam optimizer for deep neural networks,” in *IWQoS*, 2018.
- [18] A. G. Howard *et al.*, “Mobilenets: Efficient convolutional neural networks for mobile vision applications,” 2017.