2023 IEEE 24th International Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM) | 979-8-3503-3165-3/23/$31.00 ©2023 IEEE | DOI: 10.1109/WoWMoM57956.2023.00034

Context-Aware Heterogeneous Task Scheduling for Multi-Layered Systems

Sharon L.G. Contreras and Marco Levorato
Computer Science Dept., University of California at Irvine, United States
Email: {sladrond,levorato}@uci.edu

***Abstract*—Machine learning is becoming an increasingly integral component of mobile applications. However, the execution of compute-heavy neural models (e.g., for computer vision tasks) on resource-constrained devices is challenging due to their limited computing power, memory, and energy reservoir. While edge computing mitigates these issues, the transfer of information-rich signals over capacity-limited and time-varying wireless channels may result in large latency and latency variations. Herein, we propose a methodology to route heterogeneous tasks across the resources and layers of systems composed of mobile devices and edge servers. Different from prior work, we consider aspects of real-world systems, such as context switching, task accumulation, and the interplay between communications and computing components of the overall pipeline, that are rarely captured in abstract models. To optimize the task flow, we use a deep reinforcement learning agent trained on real-world data collected using a system we developed. The agent uses an articulate definition of state drawing features from several logical blocks of the system. Results indicate that the agent adapts the routing of tasks to parameters controlling their heterogeneity, as well as the hardware setup and the state of the wireless channel.**

***Index Terms*—Resource allocation, Task offloading, Context-aware control, Edge computing.**

I. Introduction

Machine learning is becoming an increasingly crucial component of mobile applications. Emerging applications, such as augmented reality (AR), often produce streams of data analysis tasks (e.g., image analysis or speech-to-text) that not only are computationally intense, but also heterogeneous in nature. While the "local" execution of the tasks onboard mobile platforms is inherently challenging due to the limitations of these platforms, emerging distributed computing strategies over layered systems - i.e., mobile-edge-cloud - suffer from the need to transfer information-rich data and signals over wireless and wireline channels with finite and time-varying capacity, as well as from the limitations of infrastructure-level servers (and especially edge servers) and the inevitable need to share such resources across multiple mobile devices.

In this context, the allocation and optimization of computing resources across the layers of articulate systems is an extremely important problem. Recent contributions present frameworks that optimize performance metrics such as energy consumption [1], task performance [2]–[4], and latency [5] individually. However, most existing work tends to base their solutions on abstract models of the system's operations that often fail to capture the intricate behavior of real-world deployments, while also typically considering a homogeneous streams of tasks (e.g., object detection).

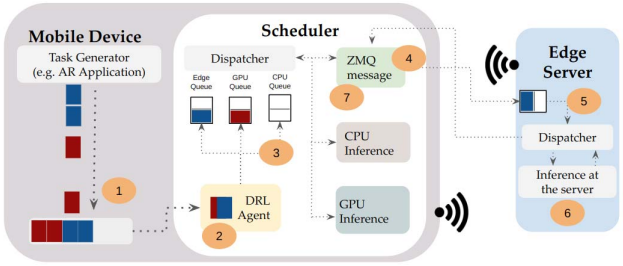

Fig. 1: Illustration of system model considered in this paper: a multi-layered mobile-edge system collaborates toward the timely execution of a heterogeneous stream of computing tasks. Each layer has multiple resources, e.g., CPUs and GPUs.

The focus of this paper is the management of streams of heterogeneous tasks over layered systems composed of mobile devices (MD) and edge servers (ES) or ES and cloud servers (CS), where each unit embeds multiple resources: i.e., *Central Processing Units* (CPUs) and *Graphical Processing Units* (GPUs). The system is illustrated in Fig. 1. Different from most prior work, we consider the influence on the system dynamics of context switching, task accumulation, and the interdependence between communication layer timing and computing performance, which all have a considerable impact on the system behavior and performance. In this context, we propose a predictive framework that controls in real-time (*i*) the routing of individual tasks across the resources and layers of the system, and (2) the machine learning model assigned to each task to minimize latency, energy consumption while maximizing task performance.

The engine of the framework we propose takes the form of a deep reinforcement learning algorithm (DRL), whose input features include multiple core system components such as CPU, GPU, memory usage, and wireless channel statistics. In order to train and test the framework, we developed a real-world system composed of a task generator, the DRL agent that receives and allocates the input queue of tasks, and a dispatcher that routes the tasks received based on the actions chosen by the DRL agent. Based on this setup, we collect a comprehensive dataset, described in detail in Section VII.

To summarize, this paper makes the following contributions:

979-8-3503-3165-3/23/$31.00 ©2023 IEEE
DOI 10.1109/WoWMoM57956.2023.00034

(1) We develop a framework for the management of heterogeneous computing tasks' flow across multi-layered systems that accounts for real-world effects such as context switching, task accumulation, and intricate interdependencies between communication and computing components of the system.

(2) We build a multi-layer system empowered with the ability to dynamically route tasks across layers and computing components. Notably, in contrast with contributions focusing computing on one of the layers (e.g., edge computing), all the layers of the system collaborate toward the efficient and timely completion of the incoming tasks. We perform dynamic allocation of heterogeneous machine learning tasks. We leverage the allocation and model decisions based on the status of each layer and implement a multi-objective performance function that maximizes the resources of each layer.

(3) We design and train a deep reinforcement learning algorithm that controls task routing and allocation taking as input statistics of computing and wireless resources, the queue of incoming tasks, the status of the models, and tasks dispatched. Training is performed on a comprehensive dataset that we pledge to release to the community. The source code of our framework and the dataset is in https://tinyurl.com/heterogeneousTaskScheduling

(4) We evaluate our solution and show empirically that different settings and characteristics of the system (e.g. task arrival distribution, hardware configuration) result in different decisions regarding the allocation policy. In a setting where the MD is an embedded platform with relatively low computing capabilities (e.g., an NVIDIA Jetson Nano) the optimal policy privileges allocation of tasks to the CPU to avoid a large delay penalty associated with context switching – while also using low-complexity neural models to contain the increase in execution time with respect to allocating the task to the GPU. If a more powerful embedded platform is used (e.g., an NVIDIA Xavier AGX in our experiments) the controller expresses a more pronounced preference toward using larger neural models and performing inference locally on the GPU. We also show that fixed policies such as traditional edge computing (that is, all tasks are routed to the edge server) incurs a degraded latency up to 5x times, in the worst case, compared to a dynamic optimized policy.

The rest of this paper is organized as follows. In section II, we illustrate the impact of context-switching on end-to-end delay when inference is performed in multiple processors and platforms. Next, in section III we summarize relevant contributions to the tasks allocation and routing in multi-layered systems and compare them with our work. We propose a system model and its performance metrics in section IV. In section V, we provide details about the hardware setting and the software implementation of our system. We describe our solution in Section VI. Sections VII and VIII describe the dataset collected to perform experiments as well as the results obtained using our proposed framework. Finally, Section IX concludes the paper.

II. Motivation

In this section, we illustrate the impact of context switching - which to the best of our knowledge is not considered in existing literature on task management in collaborative multi-layered systems. In order to support heterogeneous tasks, unless sufficient resources (e.g., memory) are available, the system will need to switch context as the current task changes over time. For instance, a neural network model needs to be loaded in the memory of a GPU in order to be executed. This preparation phase may take a considerable amount of time (e.g., a large fraction of a second), and may impair the continuity of execution of the stream of tasks.

To illustrate the impact of model loading, we consider a setting where two platforms from the NVIDIA Jetson family – particularly the Jetson Nano, and Xavier AGX – are used as mobile devices. We note that they differ in terms of computing capabilities, the internal configuration of resources, and energy consumption. The NVIDIA Jetson Nano (NVIDIA JN) architecture has a shared physical memory between the CPU and GPU. The key advantage of this architecture is that data transfers are performed less frequently compared to an architecture with separate discrete memory per processor (that is, CPU memory isolated from GPU memory). However, the small size of the memory prevents the program from opening multiple processes to decouple the loading from the inference in neural network models. The latter architecture platform used as MD, the NVIDIA Xavier AGX (NVIDIA JXA), has high memory bandwidth – the speed we can read or store data into the memory, deep learning accelerators, and a cache coherence between CPU and GPU which helps reduce latency overhead and bandwidth usage. The cache coherence [6] is important in our case as a data tensor or neural model can be stored and retrieved directly from it instead of the main memory through the CPU. Our framework automatically adapts the management of the computing tasks across the layers of the system and their components to these factors without the need for the designer to hard-code a platform-specific policy.

We illustrate our observations by means of simple experiments over the setting we will describe in detail later in the paper. In these experiments, we generate a stream of 200 tasks composed of two distinct classes, that is, image classification and image segmentation, each associated with a neural model (SegFormer-B1 [7] and EfficientNet-B1 [8]). The number of tasks of the same class is distributed uniformly such that when there are 2 context changes in a stream of 200 tasks, we have 100 image classification tasks and 100 image segmentation tasks. As shown in Fig. 3 and 2, context switching has a computing platform-dependent impact on the execution timeline. In the figures, we can see that the CPU – a general-purpose processing unit that fetches instructions out-of-order but executes them sequentially – is not optimized to perform matrix operations associated, for instance, with the execution of neural network models and tends to have a stabilized task delay even against context switching. Conversely, the GPU, a specific purpose processing

unit that performs parallel thread execution, requires data transfer from the main memory at least once, and potentially at every context switch. This can have a considerable impact on task delay when a new task is allocated to a specific unit. In addition to the expected differences among processors, there are some differences in the task delay dynamics which depend on the platform architecture. As shown in the pictures, if the NVIDIA JN is used, the CPU inference delay increases linearly as the frequency of context switching increases while the GPU inference increases close to exponentially due to their memory architecture.

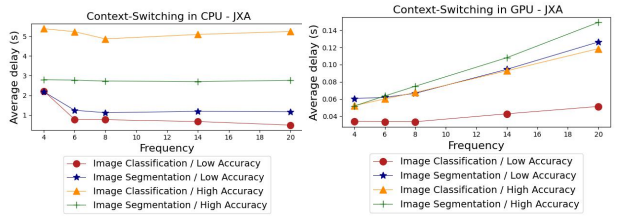

Fig. 2: Loading model and inference latency in NVIDIA Jetson Xavier AGX for different context-change values across different ML tasks and processing units.

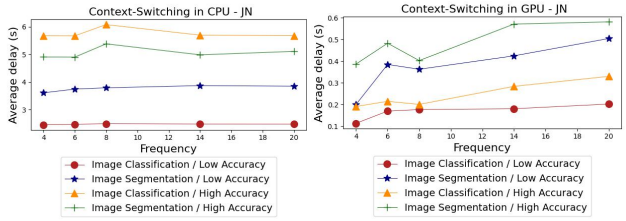

Fig. 3: Model loading and inference latency in the Jetson Nano for different context-change frequencies across different ML tasks and processing units.

We note that context switching is influenced by the order of task arrivals into the system. We also observe how the timing of task arrivals, which at a fine grain is also influenced by the communication layer as well if tasks are allocated to the edge server – has an impact on batch processing on the GPUs: a critical component of inference acceleration.

III. Related Work

To the best of our knowledge, there are no available studies on context-switching and inference performance interdependence focused on multi-layered systems. The closest class of contributions in the literature targets the optimization of inference performance by task offloading in one or more layers (e.g. mobile device to edge server or edge server to cloud). We summarize this work and contrast it with the contributions of this paper in the following.

Task offloading in the context of edge computing – often referred to as Mobile Edge Computing Systems (MECS) – has been extensively studied [9]. We can categorize this problem by the execution methodology: *full offloading* - where the whole computing task is offloaded to the edge server, *partial offloading* - where portions of the computing task are offloaded and others are executed locally on the mobile device. Guo *et al.* [4] and Fresa *et al.* [3] are examples of full offloading where the objective is to maximize the accuracy of inference tasks. Matsubara *et al.* [10] and Zhao *et al.* [11] perform partial offloading to optimize the balance between end-to-end delay, energy consumption and task performance using techniques such as split computing, early exit, and data compression [12].

Mobile-Edge-Cloud cooperation frameworks have been developed. For instance, Hong *et al.* and Zhang *et al.* [2], [13] present methodologies to allocate data tasks on the available resources optimizing customer performance metrics. These frameworks showed promising results in simulated data and were controlled under the assumption of homogeneous tasks and hardware setups.

Task scheduling and placement in MECS has been largely studied to optimize multiple performance metrics such as end-to-end delay and available resources (e.g CPU, GPU) [14]–[16], and [17] among others. In particular, Shu *et al.* [15] consider explicit task dependencies and heterogeneous servers to reduce the overall task delay. Zhang *et al.* [17] also focuses on task delay and tasks dependencies, however, the proposed solution is evaluated on real-world heterogeneous edge servers (i.e. multiple CPU, GPU, and network configurations).

IV. System Model and Optimization Objectives

In this section, we first describe the model of the system we consider and then define performance measures and optimization objectives.

A. System Model

We consider a multi-layered system where a MD and an ES collaborate to complete a stream of tasks generated by the MD. Formally, we define the task arrival process $\mathbf{T} = \{T_i\}_{i=1,2,\ldots}$, where $T_i{=}(t_i, c_i)$ is the i–th task arrival described by the arrival time t_i and $c_i{\in}\{1,\ldots,K\}$ is the task class. The MD and ES are wirelessly connected and composed of multiple computing resources - namely GPUs and CPUs - characterized by different computing power. We denote resources at the MD as R_n^{MD}, $n{=}\{1,\ldots,N^{\mathrm{MD}}\}$ and at the ES as R_n^{ES}, $n{=}\{1,\ldots,N^{\mathrm{ES}}\}$, and group them in the set R_n, $n{=}\{1,\ldots,N^{\mathrm{MD}}+N^{\mathrm{ES}}\}$.

At a high level, in this work, a task corresponds to a set of data to be analyzed using a deep neural network. When a task arrives at the MD, it enters the finite queue Q^{MD}. Tasks are then processed sequentially by the management function $(n_i, m_i){=}A(T_i)$, where n_i points to a resource in $\{1,\ldots,N^{\mathrm{MD}}+N^{\mathrm{ES}}\}$. The control variable m_i determines the neural model used for the execution of the task. Multiple queues in the MD are used to route the incoming tasks to a particular resource queue (See Fig.1).

If n_i corresponds to a resource on the MD, then the task is sent to that computing unit - which has its internal task queue. Conversely, if n_i is a resource on the ES, the task is sent to the radio interface and will eventually enter the ES' queue Q^{ES} - where the entry will also contain the pair (n_i, m_i).

B. Performance Measures

We define a set of relevant performance measures that will guide our optimization rationale.

Task-Level: We define two key metrics at the task level: *task performance* and *task latency*. The former is primarily associated with the selection of the model used to complete the tasks. For instance, a more complex neural model will most likely achieve a better average performance (where the average is over the data samples acquired by the MD) compared to a lower complexity model. We then define $p_c(c, m)$ as the expected normalized task performance associated with a task of class c when model m is used.

Moreover, we denote h_i as the time at which task i is completed and define the task latency as $l_i = h_i - t_i$. We note that latency is the sum of multiple components, including queueing time, execution time, communication time (if the task is executed on the ES), and context switching. While it is indeed possible to abstract these components, in real-world systems they are determined by complex software-hardware effects and interdependencies that are difficult to capture using available modeling strategies, also due to the influence of temporal patterns of the system state.

System-Level: At the system level, we focus on *power consumption* at the MD. We define the instantaneous - normalized - power consumption time τ as w_τ. Intuitively, and similarly to the latency, the power consumption also is the result of a rather complex definition of system state and its pattern that makes it hard to use simple abstractions.

C. Optimization Objectives

The objective of the framework we propose is to jointly maximize the expected task performance

$$P = E\left[\sum_i p_{c_i}(c_i, m_i)\right], \tag{1}$$

while minimizing the expected task latency

$$L = E\left[\sum_i l_i\right] \tag{2}$$

and average power consumption

$$W = E\left[\int_\tau w(\tau) d\tau\right]. \tag{3}$$

The expectations are computed over realizations of the stochastic process describing the system behavior. We remark that due to the complexity of the system we resort to optimization and evaluation based on datasets collected experimentally.

V. System Architecture

In this section, we describe in detail the system we developed. An overview of the system is provided in Fig. 4. The software we developed has a modular structure and includes the following modules: task generator, dispatcher, and logger. Note that the latter is not only used to evaluate performance, but also to compose the real-time state used by the DRL agent to select the actions. Our task generator initiates a background process that generates a stream of tasks according to a particular distribution. Our dispatcher manages the reception of tasks and the execution of the tasks using torch libraries such as multiprocessing [18] and cuda [19], when tasks are allocated locally, the dispatcher is able to perform loading (when required) and inference. If the tasks are allocated to the Edge, the dispatcher prepares and sends the message to be transmitted using the wireless interface of the MD. Our loggers are hardware customized, where we use the Tegra Utility [20] integrated within the Jetson platforms to obtain the state of the hardware.

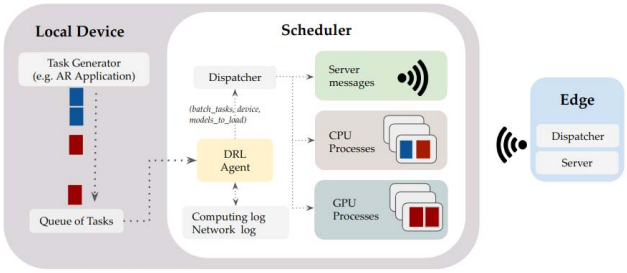

Fig. 4: Schematics of the system implementation.

A. Computing Tasks

We developed a tasks generator module that defines a sequence of tasks to be dispatched. This module creates distributions of tasks based on three parameters: (1) the set of tasks, (2) the distribution of the time between tasks, and (3) the probability of switching from one task to another in the arrival sequence.

For the sake of simplicity, we focus our attention on computer vision, specifically on image classification and semantic segmentation. This choice is motivated by the availability of families of models designed for resource-constrained systems providing performance proportional to their complexity [7], [8]. We consider two families of models: EfficientNet (which we could identify as m_0 and m_1 for effficientnet-b0 and efficientnet-b1) [8] for image classification and SegFormer (which we could identify as m_2 and m_3 for segformer-b0 and segformer-b1) [7] semantic segmentation. We use pre-trained models from the HuggingFace [21] repository.

In general, image classification models tend to have lower complexity compared to those designed for the much more difficult image segmentation task. This leads to different memory usage and, in general, task execution times for the two tasks. For instance, model m_1 or efficientnet-b1 has 19M

parameters takes 0.3s in JN and 0.071s in JXA while m_3 or segformer-b1 with 50.3M parameters takes $0.5s$ in JN and 0.075s in JXA.

B. Task Dispatching

We implement the task dispatching process using batches, where a batch is a minimum number of tasks needed to initiate the allocation process. We use the multiple producer-consumer paradigms to establish communication between the task generator and the scheduler. The input queue is a process-safe data structure that shares data between processes allowing the tasks to be passed from the Task Generator module to the DRL agent. Once the tasks are received by the DRL agent, we obtain the most recent network and computing features. The computing and network logs are hardware-customized modules running in the background to provide an accurate state of the mobile device.

C. Hardware-Software Setup

The MD and ES hardware setup differ in capacity, but also in the OS implementation, which has implications for how we extract the real-time state of the hardware and the system operations. Our framework is built to run on multiple NVIDIA boards as MD and multiple Desktop-Linux computers as ES (See Tables I, II).

The wireless antenna adapter used in the JN platform was used to transmit messages at 2.4GHz, the antenna gain is 5dbi. We use the same frequency to transmit messages from the JXA platform, however, the antenna gain is slightly higher (6dBi). We empirically observed a neglectable change in the RTT for the proposed scenarios.

	Mobile Device NVIDIA Jetson Nano	**Edge Server Laptop**
CPU	Quad-core ARM A57 @ 1.43 GHz	11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz
GPU	128-core NVIDIA Maxwell™	NVIDIA GeForce RTX 3050 Ti
Memory	4GB 64-bit LPDDR4 25.6GB/s	32 GB DDR4 at 3200 MHz (2 x 16 GB), dual channel
AI Performance	472 GFLOPS	1.69TFLOPS
Network	Wifi 802.11 ac	Wifi 802.11 ac

TABLE I: Hardware Settings I

VI. Deep Reinforcement Learning Framework

As mentioned earlier, we optimize task allocation across the layers and resources of the system as well as the model used to complete the task. In the following, we define the structure of decision-making, how the state is composed, and how we transpose the abstract performance measures we defined earlier into the real-world system we deployed.

We adopt a double deep Q network (DDQN) [22] algorithm as the core engine of our framework because it is a sequential decision-making problem. DDQN is model-free

	Mobile Device Jetson Xavier AGX	**Edge Server Jetson Orin**
CPU	8-core NVIDIA Carmel Armv8.2 64-bit	12-core Arm Cortex-A78AE v8.2 64-bit
GPU	NVIDIA Volta architecture with 512 NVIDIA cores	NVIDIA Ampere architecture with 2048 NVIDIA CUDA cores
Memory	64GB 256-bit, 136.5GB/s	32GB 256-bit, 204.8 GB/s
AI Performance	22TOP	275 TOPS
Wifi	Wifi 802.11 ac	Wifi 802.11 ac

TABLE II: Hardware Settings II

DRL that leverages the greedy policy implemented by deep Q Network (DQN) algorithms while reducing overestimation. In the DDQN algorithm, the greedy policy is evaluated using a second neural network. We refer the interested reader to [22] for a thorough description of the algorithm, which we summarize at a high level in Algorithm 1.

A. Decision Timing

We define a time step as the interval between a task arriving at the Q^{MD} (i.e. input queue) in the DRL agent and the result of the inference being obtained. The input queue is separated into a queue per resource available (Q^{R_i}) which enables asynchronous inference execution processes and consequently stores the individual real-time latency. This management of tasks facilitates the differentiation between the delay of the task and the total time in the time step. In fact, the time step is an upper bound of the latency.

B. State and Action space

The state space represents the available resources and interfaces in the MD, which is readily observable by the agent without the need to exchange information over the wireless link. In particular, we consider blocks of computing, task, model, and network features. We then define the global state as $S = \{Q^{MD}, C, T, M, N\}$ composed of 43 parameters, where Q^{MD} is the number of tasks in the input queue, C is the state of the computing resources in the MD, T, and M are the statistics of the recent tasks dispatched and the neural network models recently used, and N is a set of features from the IP, TCP, and wireless interface. The specific variables used in the implementation are given in Section VII.

Actions are applied to batches of tasks, and the action space is $A = \{a_0, .., a_N\}$, where $a_i = (m_j, p_k)$ is the pair composed of the neural model to be used to perform inference and the resource (e.g. CPU, GPU) chosen to perform inference at the current batch of tasks. Therefore $N = M \times P$, where M is the maximum number of neural network models per task and P is the maximum number of processors that can be used based on the hardware specifications.

C. Reward Function

As indicated in Section IV, we define a multi-objective optimization objective that includes task delay, normalized task

Algorithm 1 Multi-Layered System with Double DQN

Initialize Agent and Neural Network parameters: primary network Q_θ, target network $Q_{\theta'}$, and replay buffer$D$
Initialize Framework parameters: dispatcher configuration, task generator, computing feature logger, network feature logger processes
Initialize Environment parameters: action space, size of the batch of tasks, number of classes of tasks, the wireless connection configuration
for episode $e = 1, N$ **do**
  while dispatched tasks $<$ total tasks
  & size(input queue) > 0 **do**
    Select a random action a_t with probability ε.
    Otherwise, select $a_t = \arg\max_a Q(s_t, a; \theta)$
    Execute action a_t, collect reward r_{t+1} and observe next state s_{t+1}
    Log the experience $(\text{timestamp}, s_t, rt + 1, s_{t+1})$
    Store the transition $(s_t, a_t, r_{t+1}, s_{t+1})$ in $\mathcal{D}$
    if s is terminal state **then**
      break
    end if
  end while
end for
for each update state **do**
  sample e_t = $(s_t, a_t, r_{t+1}, st + 1)$ from $\mathcal{D}$
  Compute target Q value:
  $Q^*(s_t, a_t; \theta) \approx r_t + \gamma Q_\theta(s_{t+1}, argmax_{a'} Q'(s_t, a'))$
  Do gradient descent step on $(Q^*(s_t, a_t) - Q_\theta(s_t, a_t))^2$
  Update target network parameters:
  $\theta' \leftarrow \theta$
end for
Terminate all processes initialized by the Framework

accuracy, and power consumption (measured over a time step). Our reward function reflects the objectives using a sigmoid function that penalizes the performance metrics to be outside a desired region. The overall performance is calculated as the sum of the weighted average of the normalized optimization objectives. This implies that the normalized accuracy is used as inverse since the more accurate, the smaller the value of the overall performance is and consequently, the reward is higher.

We compute the expectation of the performance as the average normalized accuracy of the task class based on the model selected. Since we use trained neural network models, we consider the documented accuracy to normalize the values in an interval [0,1] such that they are comparable. The latency is measured in real-time, we timestamp tasks at arrival time and retrieve the total time when the results are available to the MD. This latency is averaged among the tasks dispatched in each batch. Finally, power consumption is obtained as an instantaneous measure from the kernel variables. We use the interval duration to extract an approximate energy consumption within the time step. Energy consumption also is normalized.

Then, the overall reward function is defined as follows:

$$R_t = \begin{cases} S & \text{if resources available at time } t \\ -1 & \text{otherwise} \end{cases}, \qquad (4)$$

where $S = 1/(1 + e^x)$ and $x = \sum_{i=1}^{N} w_i \dot{\delta}^*$ and δ represents the vector of the normalized performance metrics, in our case: end-to-end delay, accuracy and power consumption.

D. Architecture

The architecture of the proposed DDQN implementation relies on custom modules such as the environment, task generator, dispatcher, and loggers. The main characteristic of the DDQN algorithm is the experience replay and target network deployment to stabilize the optimization. In our case, to build the online neural network model we adopted a sequential structure starting with 3 layers of 1-dimensional convolutions and Relu as an activation function, a flattened layer, and 2 lineal layers with Relu as an activation function. Our inputs are state-action pairs, which is the 1-dimensional vector and the output is the Q-value that corresponds to the selected action.

VII. Dataset

In order to train the neural network embedded in the DDQN, we collected a dataset that captures the experiences of our DRL agent in a complex environment with the aim to test and evaluate schemes build to optimize computing resources.

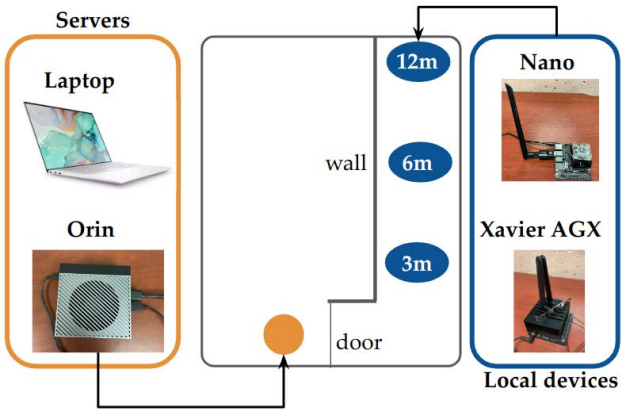

Fig. 5: Experimental setup: Indoors space with a wall and a door between the Edge Server and the Mobile Device. The mobile device was located in a hallway at different distances from the Edge server.

We implemented two types of policies in the experiments, the baseline policies which are CPU, GPU, and EDGE ONLY. The CPU policy refers to performing inference of all tasks generated on the CPU regardless of the context change, the same rule applies to the GPU ONLY and EDGE ONLY policies. The latter type of policy implemented is random which provides a non-biased sample of the action space. We log the state of the MD in the computer, network, tasks, and model features for the time the experiments ran. Finally, we log the output

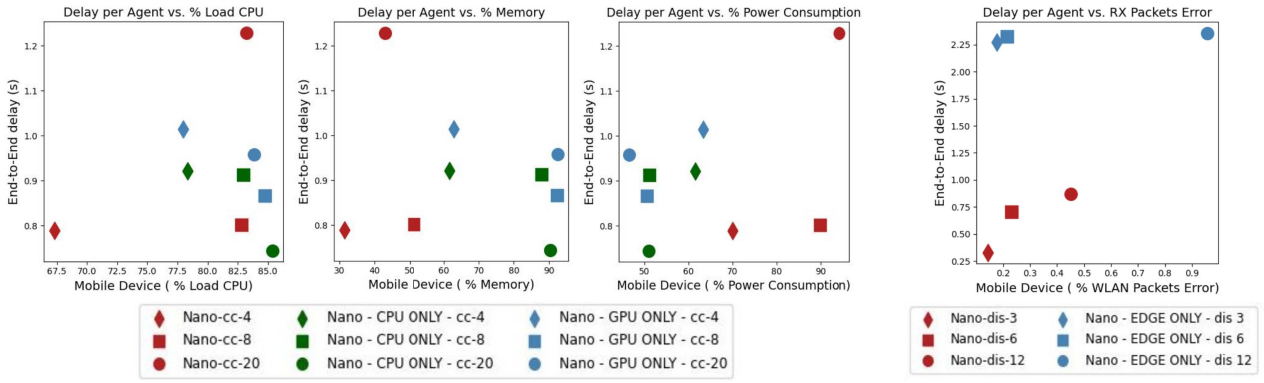

Fig. 6: End-to-end tasks delay vs. state features for NVIDIA Jetson Nano. The left plots show mobile device features with the agents trained to adapt to the context changes. The right plot shows the wireless representative feature with the agents trained to adapt to the changes in distances

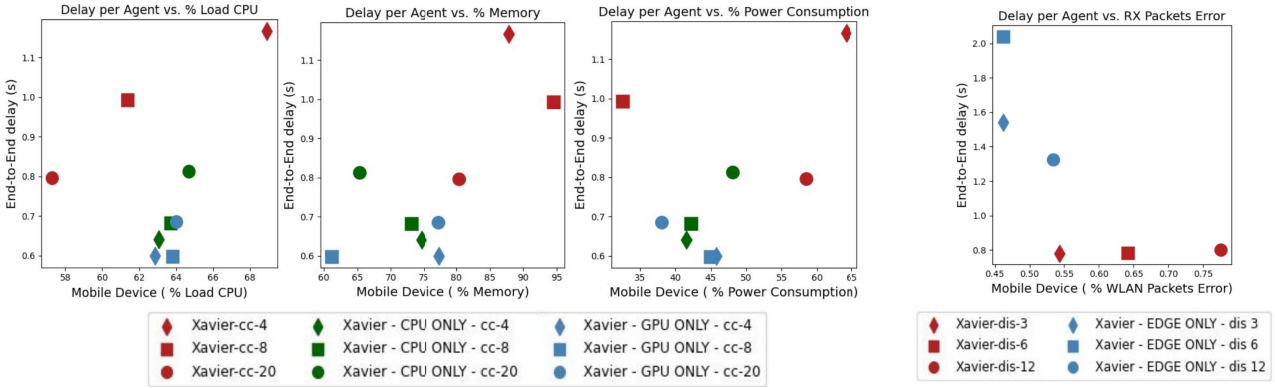

Fig. 7: End-to-end tasks delay vs. state features for Jetson Xavier AGX. The left plots show mobile device features with the agents trained to adapt to the context changes. The right plot shows the wireless representative feature with the agents trained to adapt to the changes in distances

or reward of our system implementation. As part of modeling a complex environment, we use two scenarios composed of two hardware settings to show that variation in computing and wireless capacities enables variation in the optimal allocation policy. The hardware settings are listed in I and II. The first setting uses an NVIDIA JN board as MD: an embedded computer with limited computing capabilities. This device is designed for simple machine-learning models. Moreover, the CUDA memory is shared with the CPU, which results in some limitations if we decouple the loading from the inference in machine learning tasks. The wireless interface is not integrated into this board, we have added a USB Wifi antenna adapter that facilitates wireless communication. In the first setting, we use a Laptop - Linux as the ES. This latter platform not only possesses a larger computing power but also uses a standard architecture. The second setting uses an NVIDIA JXA board as MD: an embedded platform that reduces task latency and memory bandwidth due to the zero-copy functionality active for the CUDA memory. For this board, we acquired a Wifi antenna, with a larger antenna gain than the JN. As ES, we use an NVIDIA Orin board, which has 10x the computing power and embeds inference accelerators [23]. We define experiments based on the following parameters: a hardware setting, a specific distance between the ES and the MD, a distribution of the heterogeneous tasks and a policy. We recorded information from 9 experiments per hardware setting and per policy (see details in Table III). The dataset is composed of a collection

Parameter	Values
Hardware settings	Table I and Table II
ES-MD distance	3, 6, and 12 mts
Tasks distributions	4, 8, 20 context changes
Policies	Random, EDGE ONLY, CPU ONLY, and GPU ONLY
Actions configuration (processors)	GPU, CPU and EDGE accordingly
Actions configuration (models)	Semantic segmentation: SegFormer B0 and B1 [7]. Image Classification: EfficientNet B0 and B1 [8]

TABLE III: Experiment parameters

of the DRL agent experiences per experiment, an experience

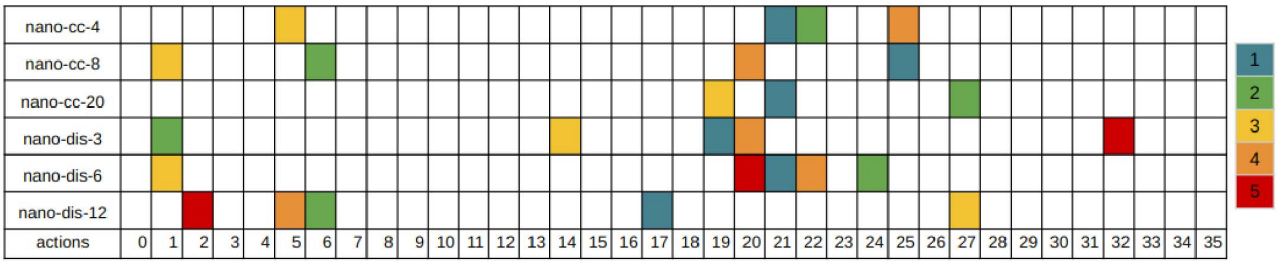

Fig. 8: Ranking of actions per trained agent within scenario I

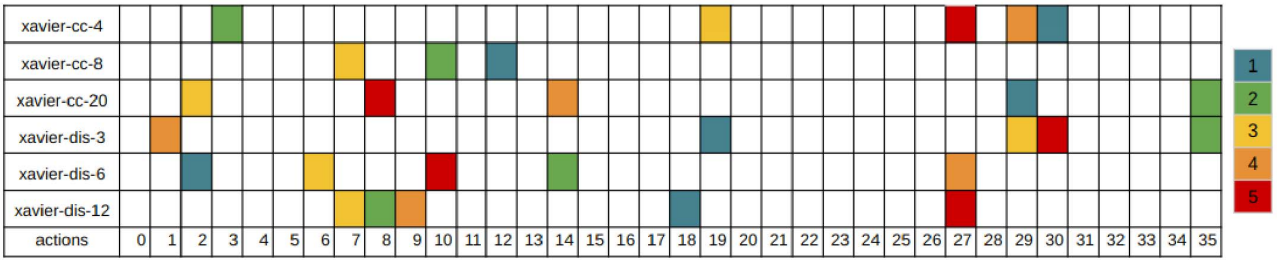

Fig. 9: Ranking of actions per trained agent within scenario II

contains the timestamp, the state, the chosen action, and the next state. The state is formed by the computing, tasks, model, and network features.

VIII. Results

Based on the dataset described in the previous section, we train the DRL agent and deploy it on the considered scenarios. We name the scenarios in terms of MD, policy, and frequency of context switching. For instance, the policy deployed by the agent in an environment when the context changes 4 times and uses the information of the NVIDIA JN is defined as *nano-cc-4*. The weights of latency, accuracy, and energy consumption we use in the training phase are $[0.6, 0.3, 0.1]$, respectively.

In Fig. 6 and Fig 7, we show the average end-to-end task delay for selected state features of each hardware setting. The % of CPU load, memory, and power consumption are features of the state of the mobile device, while the % of lost WLAN packets corresponds to changes in link quality. Policies that allocate all the tasks to one processor emphasize the impact of context change, which tends to dominate the other delay components as we increase the degree of heterogeneity of the tasks. For instance, under the CPU ONLY policy, the overall latency in the various environments tends to saturate as the frequency of context switching increases. The GPU ONLY policy tends to consume less power and more memory compared to CPU ONLY policy in all environments. As expected, the EDGE ONLY policy suffers from the impairments of the wireless channel, especially as the MD-ES distance increases.

The policy adopted from the environment where the MD has an NVIDIA JN hardware setting is shown in Fig. 6 (nano-cc-4, nano-cc-8, and nano-cc-20). In Fig. 6, we observe that the DRL agent achieves smaller delays than the baseline policies – CPU ONLY and GPU ONLY – when the context change frequency is smaller than 20. This is because the DRL agent policy is prioritizing delay over power consumption of the MD. Further, the number of consecutive tasks of the same class is larger compared to the nano-cc-20 environment, and then there is more room for adaptation. Consequently, the improvement of the end-to-end delay when the context change is 8 is higher compared to 4. On the other hand, the DRL agent deployed in the environments where the distance between the MD and ES varies (nano-dis-3, nano-dis-6, nano-dis-12) show a superior reduction of the end-to-end delays (60% of the time) as the DRL agent policy is choosing to perform some tasks locally.

The policy obtained from the environment where the NVIDIA JXA is the MD, showed in table II (xavier-cc-4, xavier-cc-8, xavier-cc-20), only improves the end-to-end delay compared to the baseline policies – CPU ONLY and GPU ONLY – for the scenario when the context change occurs 20 times in a sequence of tasks. This is because the hardware is powerful enough to repeatedly use complex models, and therefore the policy is prioritizing local computing with a stable delay for the low frequency context change scenario. Additionally, we note that the end-to-end delay achieved by our policy is 2.5x times smaller compared to the EDGE ONLY policy.

In figures 6 and 7, we show the relationship between the state of the MD and one of the metrics used to optimize the trained agent (e.g. end-to-end delay). Importantly, the actions are taken by the DRL agent to achieve such performance show that different policies apply to different scenarios. To illustrate this effect, we show the ranking of actions used in each of the policies adopted by the DRL agent (see Fig. 8 and Fig. 9). The most selected actions for variations in context

change frequency for the NVIDIA JXA setting are a low complexity model in CPU for the image classification task and either high or low complexity in ES for the segmentation task; which shows that diversification of resources for tasks allocation enables performance improvements. This policy is reasonable due to the convergence in delay that occurs when the model is placed in the CPU instead of the GPU.

The policy obtained from the environment that varies the hardware of the MD and ES follows a pattern based on the context-change frequency; for instance, in the NVIDIA JN setting the most selected action for the scenario with the least context-change frequency is the same as the one with the highest context-change frequency (nano-cc-4 and nano-cc-20 respectively). This is the option with a high-complexity model for the image classification tasks and a low-complexity model for the image segmentation tasks placed in the CPU. Only the nano-cc-8 environment's first choice places the models in the GPU and the ES with low and high complexity respectively.

The policy obtained from the environment that varies in MD to ES distance (e.g. nano-dis-3, nano-dis-6, nano-dis-12) in NVIDIA JXA and NVIDIA JN implements different actions. NVIDIA JXA agents privilege actions that select resources onboard the ES. For instance, the second in the ranking of actions chosen in the NVIDIA JXA environment includes dispatching all image classification tasks to the ES, where the high complexity is preferred in the closest distance and the low complexity is preferred when the distance between the mobile device and the ES increases. Conversely, the second choice in the ranking of the choices in the NVIDIA JN setting shows a preference for dispatching tasks locally where the GPU is used at the same time as the CPU. For actions 1, 24, and 6 (See Fig. 9) the image segmentation tasks are always dispatched to the GPU. This empirically shows that the policy is adapting to the changes in hardware in each scenario as well as to the changes in the environment.

IX. Conclusions

In this paper, we presented a framework for the control of task routing across the layers and computing units of edge computing systems. Different from most prior work, we resort to the use of a real-world deployment to capture important effects, such as context switching, that greatly impact performance in systems characterized by streams of heterogeneous tasks. We train a DRL agent to control the allocation of tasks to the system devices and computing units in response to a complex definition of the state that incorporates features observable by the mobile device and demonstrate that the agent applies different policies to different hardware settings, task arrival settings, and other general features of the system.

Acknowledgement

This work was supported by the Intel Corporation and the NSF grant MLWiNS-2003237 and CCF-2140154

References

[1] T. Baker, M. Asim, H. Tawfik, B. Aldawsari, and R. Buyya, "An energy-aware service composition algorithm for multiple cloud-based iot applications," *Journal of Network and Computer Applications*, vol. 89, pp. 96–108, 2017.

[2] T. Zhang, Z. Li, Y. Chen, K.-Y. Lam, and J. Zhao, "Edge-cloud cooperation for dnn inference via reinforcement learning and supervised learning," in *2022 IEEE International Conferences on Internet of Things (iThings) and IEEE Green Computing & Communications (GreenCom) and IEEE Cyber, Physical & Social Computing (CPSCom) and IEEE Smart Data (SmartData) and IEEE Congress on Cybermatics (Cybermatics)*. IEEE, 2022, pp. 77–84.

[3] A. Fresa and J. P. Champati, "Offloading algorithms for maximizing inference accuracy on edge device under a time constraint," *arXiv preprint arXiv:2112.11413*, 2021.

[4] H. Guo, J. Liu, and J. Lv, "Toward intelligent task offloading at the edge," *IEEE Network*, vol. 34, no. 2, pp. 128–134, 2019.

[5] D. Callegaro, M. Levorato, and F. Restuccia, "Seremas: Self-resilient mobile autonomous systems through predictive edge computing," in *2021 18th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*. IEEE, 2021, pp. 1–9.

[6] A. Bhat and S. Dasadhikari, "Search — nvidia on-demand," 2018. [Online]. Available: https://on-demand.gputechconf.com/gtc/2018/presentation/s8868-cuda-on-xavier-what-is-new.pdf

[7] E. Xie, W. Wang, Z. Yu, A. Anandkumar, J. M. Alvarez, and P. Luo, "Segformer: Simple and efficient design for semantic segmentation with transformers," *Advances in Neural Information Processing Systems*, vol. 34, pp. 12 077–12 090, 2021.

[8] M. Tan and Q. Le, "Efficientnet: Rethinking model scaling for convolutional neural networks," in *International conference on machine learning*. PMLR, 2019, pp. 6105–6114.

[9] P. Mach and Z. Becvar, "Mobile edge computing: A survey on architecture and computation offloading," *IEEE communications surveys & tutorials*, vol. 19, no. 3, pp. 1628–1656, 2017.

[10] Y. Matsubara, D. Callegaro, S. Singh, M. Levorato, and F. Restuccia, "Bottlefit: Learning compressed representations in deep neural networks for effective and efficient split computing," *arXiv preprint arXiv:2201.02693*, 2022.

[11] Z. Zhao, K. Wang, N. Ling, and G. Xing, "Edgeml: An automl framework for real-time deep learning on the edge," in *Proceedings of the International Conference on Internet-of-Things Design and Implementation*, 2021, pp. 133–144.

[12] Y. Matsubara, M. Levorato, and F. Restuccia, "Split computing and early exiting for deep learning applications: Survey and research challenges," *ACM Computing Surveys*, vol. 55, no. 5, pp. 1–30, 2022.

[13] Z. Hong, W. Chen, H. Huang, S. Guo, and Z. Zheng, "Multi-hop cooperative computation offloading for industrial iot–edge–cloud computing environments," *IEEE Transactions on Parallel and Distributed Systems*, vol. 30, no. 12, pp. 2759–2774, 2019.

[14] Z. Tang, W. Jia, X. Zhou, W. Yang, and Y. You, "Representation and reinforcement learning for task scheduling in edge computing," *IEEE Transactions on Big Data*, 2020.

[15] C. Shu, Z. Zhao, Y. Han, and G. Min, "Dependency-aware and latency-optimal computation offloading for multi-user edge computing networks," in *2019 16th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*. IEEE, 2019, pp. 1–9.

[16] T. X. Tran, K. Chan, and D. Pompili, "Costa: Cost-aware service caching and task offloading assignment in mobile-edge computing," in *2019 16th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON)*. IEEE, 2019, pp. 1–9.

[17] W. Zhang, S. Li, L. Liu, Z. Jia, Y. Zhang, and D. Raychaudhuri, "Hetero-edge: Orchestration of real-time vision applications on heterogeneous edge clouds," in *IEEE INFOCOM 2019-IEEE Conference on Computer Communications*. IEEE, 2019, pp. 1270–1278.

[18] P. Multiprocessing, "Multiprocessing package - torch.multiprocessing." [Online]. Available: https://pytorch.org/docs/stable/multiprocessing.html

[19] P. Cuda, "Torch.cuda." [Online]. Available: https://pytorch.org/docs/stable/cuda.html

[20] "tegrastats utility."

[21] A. Community, "Hugging face – the ai community building the future." [Online]. Available: https://huggingface.co/huggingface

[22] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the AAAI conference on artificial intelligence*, vol. 30, no. 1, 2016.
[23] M. Ditty, "Nvidia orin system-on-chip," in *2022 IEEE Hot Chips 34 Symposium (HCS)*. IEEE Computer Society, 2022, pp. 1–17.