國立清華大學電機資訊學院資訊工程研究所

碩士論文

Department of Computer Science

College of Electrical Engineering and Computer Science

National Tsing Hua University

Master Thesis

支援物與雲間連續部署深度神經網路的多用戶系統

A Multi-Tenant System for Deploying Deep Neural Networks in a Thing-to-Cloud Continuum

謝佳穎

Chia-Ying Hsieh

學號：109062538

Student ID:109062538

指導教授：徐正炘 博士

Advisor: Cheng-Hsin Hsu, Ph.D.

中華民國 111 年 10 月

October, 2022

中文摘要

基於深度神經網路的物聯網分析應用愈來愈普及。過去我們習慣將深度神經網路這種需要高強度運算的工作佈建到雲伺服器進行運算，然而，隨著物聯網設備的普及，他們產生的資料量也跟著增加，將資料全部送到雲伺服器進行運算反而會因網路壅塞增加延遲。因此，隨著物聯網設備的規格提升，學術界提出將深度神經網路切割並在不同設備上執行。在本篇論文中我們提出了一個提供多用戶在物到雲連續部署深度神經網路的系統。為了最大化服務的需求數量，我們使用了（一）多任務（multi-task）、（二）順帶服務（hitchhiking）、（三）可伸縮終點（early exit）與（四）重新調整（reconfiguration）四個功能。我們將深度神經網路的佈建決策過程分成規劃與執行兩階段，並在各階段提出演算法來解決問題。在規劃階段，我們根據當下的資源狀況決定佈建計畫；在執行階段，我們根據服務品質與當下資源狀況決定是否重新調整已佈建的模型的計畫來達到更好的服務品質。最後我們實作了一個實驗性平台來評估我們提出的系統。實驗結果表明，在規劃階段，我們的系統比起其他作法提升了 6.8 倍的服務數量；在執行階段，我們可以提升 35% 的服務滿意度。我們更觀察到（一）多任務和順帶服務提升了 5.4 倍的服務量、（二）可伸縮終點在不違反準確度需求的條件下降低了延遲、（三）我們的系統在高工作量的環境下有更好的表現。因此，我們建議在一般狀況下使用多任務、順帶服務與可伸縮終點功能，然後在環境變化大的狀況下使用重新調整功能。

Abstract

Deep Neural Networks (DNN) based IoT analytics is getting popular. With the growing amount of IoT sensor data, offloading the computation to the cloud becomes inefficient due to traffic congestion. With the improved capabilities of IoT devices, the concept of dividing DNN among IoT devices, edge servers, and cloud servers is proposed. In this thesis, we propose a multi-tenant system, called T2C, to dynamically choose, deploy, monitor, and control IoT analytics implemented via DNN in a thing-to-cloud continuum. T2C leverages (i) multi-task, (ii) hitchhiking, (iii) early exit, and (iv) reconfiguration to maximize the number of served user requests, satisfying the accuracy and latency requirements. We divide the deployment decision-making process into the planning and operation phases. In the planning phase, we make the deployment plan under the current resource status. In the operation phase, we check model's runtime Quality-of-Service (QoS) and decide if we should conduct a reconfiguration for better performance. We propose a suite of deployment planning and dynamic reconfiguration algorithms to dynamically deploy and migrate layers among the IoT device, edge server, and cloud server. We implement our proposed system in a prototype testbed. The results show that our system: (i) achieves a 6.8X throughput boost compared to baseline algorithms in the planning phase and (ii) improves the satisfied ratio by up to 35% in the operation phase. Furthermore, we observe that (i) multi-task and hitchhiking improve throughput by up to 5.4X, (ii) early exits reduce latency without violating accuracy requirements, and (iii) T2C leads to a higher performance boost under a higher workload. Hence, we suggest using T2C with multi-task, hitchhiking, and early exits in normal cases and enabling reconfiguration if the environment is highly dynamic.

致謝

我要感謝在研究所期間陪伴我並幫助我進行研究的所有人。 首先非常感謝我的家人提供我生活上的支持，並在我心情低落時開導我。感謝實驗室的學長姐、同學以及學弟妹提供我在實驗上的幫助，特別謝謝范孜亦，這兩年我們都一起在實驗室跑實驗、做作業、寫論文並互相給予研究建議，讓我能夠順利度過研究所生涯，也很謝謝她把我帶回排球的世界，讓我重回運動的懷抱。另外也特別感謝我的朋友林奕昀和廖敏君在研究所期間給我的開導和陪伴。 接著我要感謝加州大學爾灣分校的Nalini Venkatasubramanian教授、Praveen Venkateswaran、以及Tung-Chun Chang的幫助，研究所期間每週的會議讓我在英語、研究以及報告等方面都有大幅提升，在我的研究陷入瓶頸時也是他們的建議以及幫助讓我能夠順利完成研究。 最後非常感謝我的指導教授，徐正炘教授，教授總是在我遇到任何問題時不遺餘力的幫助我，在各方面都能夠指引我進步的方向，並提供我很多機會去接觸不同的人以及研究領域，讓我進步良多。感謝以上的所有人讓我能夠健康順利地完成學業並留下美好的碩班回憶。

Acknowledgments

Special thanks to all the accompany and help I received during the past two years. First, I appreciate my family for providing the environment to support my research. Thanks to all the lab mates for the help on my research, especially Tzu-Yi Fan. We worked together, played together, and provided comments to each other during the past two years. She also encouraged me to get back to playing volleyball, which made me healthier. I also want to thank Yi-Yun Lin and Min-Chun Liao for their mental support. Next, I would like to thank professor Nalini Venkatasubramanian, Praveen Venkateswaran, and Tung-Chun Chang from the University of California Irvine. The weekly meeting over the past few years helped me improve the ability of English, research, presentation, and so on. Last, many thanks to my advisor, professor Cheng-Hsin Hsu. He provided lots of help with my study, pointed out the right way for my research, and let me meet different people and research areas. I appreciate all the people mentioned above. I have been through a healthy and memorable research life.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

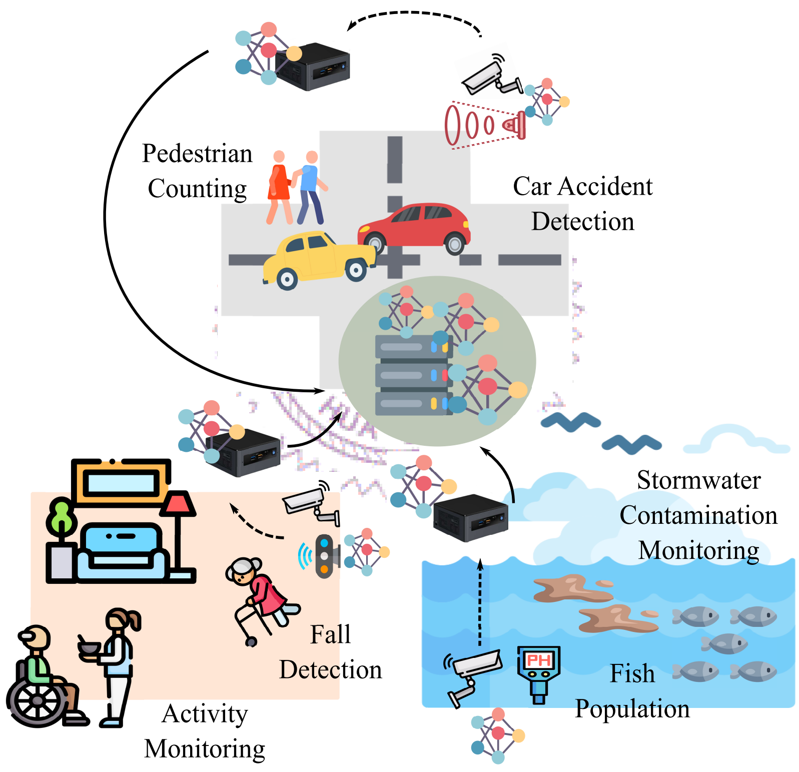


Figure 1.1: Sample applications of IoT analytics in smart environments.

Emerging Internet-of-Things (IoT) devices and Deep Neural Networks (DNNs) have been deployed in numerous domains [49], such as smart cities, healthcare, manufacturing, etc. Fig. 1.1 shows sample applications enabled by IoT and DNNs, such as traffic management, senior living, and water quality monitoring in smart environments. In these smart environments, IoT devices with sensors are deployed in different geographical areas and *IoT analytics* turn raw sensor data into useful and actionable information. A naive way to deploy these IoT analytics is to host all DNNs on IoT devices. However, doing so may lead to prohibitively long running times due to the limited computing power of

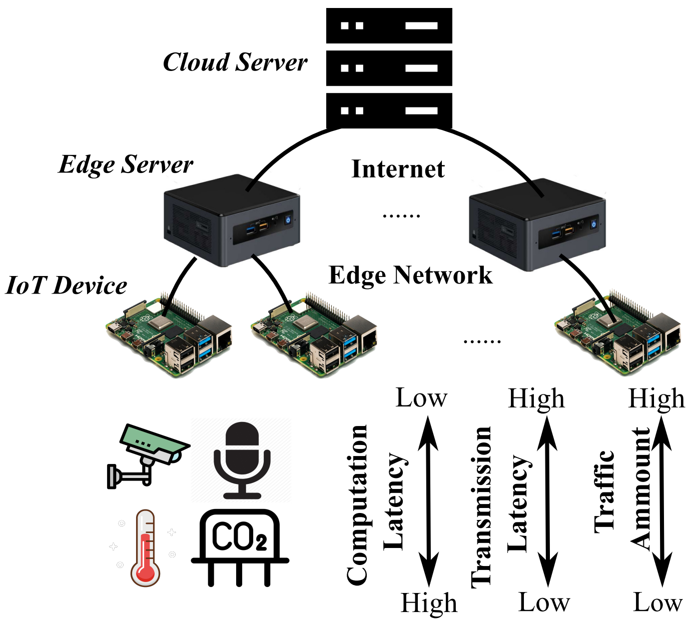


Figure 1.2: IoT devices are connected to edge and cloud servers.

IoT devices. A better way is to leverage *edge servers*, as well as *cloud servers*, which are connected to IoT devices via the edge network and Internet, respectively, as illustrated in Fig. 1.2. Decisions on where to host DNNs and perform inference across IoT devices, edge servers, and cloud servers can result in trade-offs between analytics accuracy, computation and transmission latencies, and network overhead, among others. Hence, the DNN deployment decision becomes a key research problem to guarantee the Quality-of-Service (QoS) of IoT analytics in terms of latency, accuracy, cost, etc. Jointly managing the resources across IoT devices, edge servers, and cloud servers is referred to as thing-to-cloud continuum [10] in the literature.

We make the following crucial observations that may allow us to serve IoT analytics more efficiently. Multiple analytics could share the same sensor data or even some common *prefix layers* of their DNNs. For example, Fig. 1.1 reveals that: (i) surveillance videos and LiDAR point clouds can be used for pedestrian counting and accident detection at intersections, (ii) surveillance videos and mmWave radar spectrograms can be used for activity recognition and fall detection in senior houses, and (iii) underwater cameras and pH sensors can be used for stormwater contamination and fish population monitoring in creeks. Hence, by consolidating the prefix layers of several DNNs, we may avoid duplicated data processing.

In this thesis, we develop a multi-tenant system for deploying DNNs in a thing-to-cloud continuum, called *T2C*. T2C allows the administrators of IoT infrastructures to dynamically choose, deploy, monitor, and control IoT analytics under resource constraints. Furthermore, it considers multi-tenant scenarios in that resources are shared by different analytics requests from all the users, trying to maximize the throughput of analytics requests. Designing T2C is not an easy task because of: (i) limited resources, (ii) diverse QoS requirements, (iii) different request arrival patterns, and (iv) dynamic environments. To cope with these challenges, our T2C system adopts the following features:

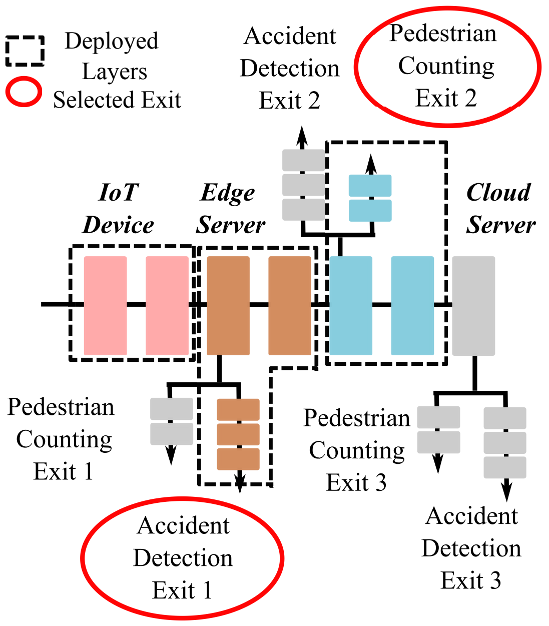
- *Multi-task networks* [15], where each DNN performs multiple tasks or analytics by leveraging shared prefix layers to avoid duplicated computations and lower resource consumption.
- *Early exit* [55], where some suffix layers can be skipped to reduce resource consumption while achieving acceptable QoS levels.
- *Hitchhiking*, where incoming user requests may be served by already-deployed DNNs to further consolidate the executions of DNNs.
- *Reconfiguration*, where layers are migrated among the IoT devices, edge servers, and cloud servers to meet QoS requirements in dynamic environments.

Fig. 1.3 shows sample neural networks demonstrating the four unique features of T2C. Fig. 1.3(a) shows a multi-task DNN model for both pedestrian counting and car accident detection tasks. Each task of this model has three exits with different accuracy levels. The circled exits are chosen based on the QoS requirements. The dashed boxes indicate the layers deployed on individual computing devices. Fig. 1.3(b) reveals that an incoming request hitchhikes on a deployed DNN if meeting the new request's QoS requirement. Fig. 1.3(c) presents the idea of reconfiguration: two layers are moved from the edge to the cloud server, when the workload of that edge server surges.

1.1 Contributions

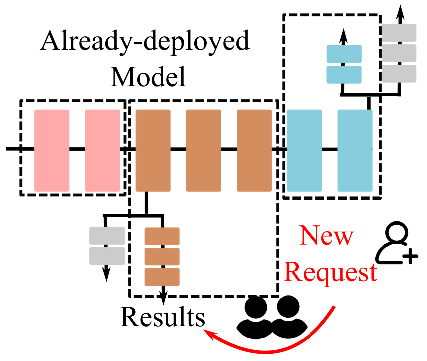
Our multi-tenant T2C system exercises the above mentioned four features to serve more users requests while maintaining the required QoS levels. We make the following contributions:

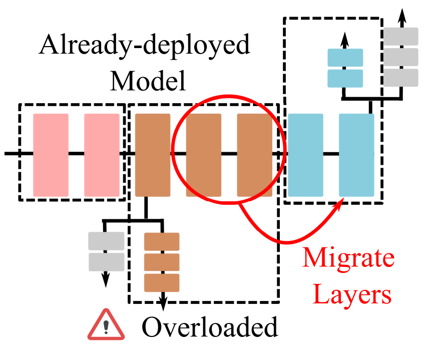
- We propose a suite of T2C algorithms to optimize the deployment of multiple DNNs across the IoT devices, edge servers, and cloud servers for maximum throughput under resource constraints.
- We build a prototype testbed with heterogeneous computing devices and network conditions to demonstrate the practicality and efficiency of our proposed system and algorithms.
- We conduct extensive evaluations and show that: (i) multi-task and hitchhiking can improve throughput by 5.4X, (ii) early exit can reduce latency by 7.68%, (iii) reconfiguration improves satisfied requests by up to 35%, and (iv) our system results in almost 6.8X throughput boost under a heavy workload.



Deployed Layers
Selected Exit
Accident Detection Exit 2
Pedestrian Counting Exit 2
IoT Device
Edge Server
Cloud Server
Pedestrian Counting Exit 1
Pedestrian Counting Exit 3
Accident Detection Exit 1
Accident Detection Exit 3

(a)





(b) (c)

Figure 1.3: Sample DNNs in our T2C system demonstrate the following T2C features: (a) multi-task and early exits, (b) hitchhiking, and (c) reconfiguration.

1.2 Limitations

We assume that:

- The containers of each DNN are already downloaded to computing devices.
- DNN information such as the number of tasks, the number of exit points, the number of layers, and the topology are given.
- The layers of the DNNs are chain topologies. For those DAG topology DNNs, the layers can be grouped into units so that the unit-wise layers are chain topologies [34].

1.3 Thesis Organizations

We first introduce our work in Chapter 1. We then discuss the related techniques of IoT analytics deployment in Chapter 2. We summarize the related studies of our proposed features and DNN deployment in Chapter 3. Next, we show the design of our proposed system in Chapter 4. In Chapters 5 and 6, we formulate and present our proposed deployment planning algorithm and dynamic reconfiguration algorithm, respectively. We show our prototype testbed implementation in Chapter 7 and evaluate the performance of our system in Chapter 8. Last, we conclude our work and give some future directions in Chapter 9.

Chapter 2

Background

In this chapter, we introduce the concept of IoT, deep neural networks, and the development of computation offloading techniques from cloud to edge. Last, we discuss the thing-to-cloud continuum.

2.1 Internet-of-Things

The concept of Internet-of-Things (IoT) was proposed by Kevin Ashton in 1999 [7] for radio-frequency identification (RFID). They try to let the computers manage objects in the real world. Nowadays, physical objects, including sensors, appliances, vehicles, etc., are often connected to networks for data exchange. It is estimated that over 75% of devices will be IoT by 2030 [24]. The data from the objects are analyzed and inter-exchanged for intelligent human lives. Sample applications include:

- **Smart home**. Sensors such as temperature, humidity, and light sensors are deployed in the houses for environment monitoring. These sensors are further integrated with actuators and smart home assistants like Amazon Alexa and Google Assistant, which allow humans to change the indoor environment, such as light and temperature with voices. Appliances such as smart refrigerators can monitor the inventory with the built-in surveillance camera and remind you to buy food. Moreover, house security can be improved by smart devices such as doorbells with surveillance cameras.
- **Transportation**. Cars, trains, buses, and other vehicles are integrated with sensors and actuators to help the drivers know the conditions of the vehicles. For example, the power, oil quantity, and tire pressure, to name a few. Some sensors, such as cameras and LiDAR are installed outside the vehicles to detect obstacles or monitor the angles that drivers can not see. Surveillance cameras can also be deployed at the train station, bus stop, and intersections to monitor traffic conditions.

- **Healthcare**. Smart watches or smart bands are getting popular nowadays. Multiple sensors, such as accelerometer, heart rate monitor, and oximetry sensor are installed in the watch to track the users' daily physical conditions. It can also detect user's fallsand call the ambulance automatically, which is useful for elder care. In hospitals, surveillance cameras can also be deployed to monitor the patients' situations and medical inventory, and RFID can be used to track the specimen.

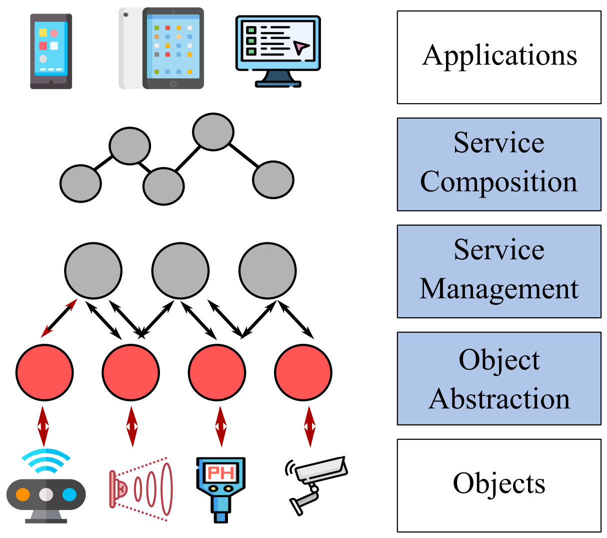


Figure 2.1: SOA-based architecture for the IoT middleware [8].

The abovementioned applications can not be done by objects themselves. In fact, there are some middleware layers between objects and applications to make the objects and their data usable for humans, as shown in Fig. 2.1. The services upon the object abstraction layer are also called *IoT analytics*, which analyzes the raw data given by the objects and provides information for applications. IoT analytics can be implemented by statistic analysis, artificial intelligence, and so on.

2.2 Deep Neural Networks

Deep neural network, or deep learning, is one of the most popular techniques in artificial intelligence [54]. The concept of the neural network is proposed in the 1940s, using mathematical functions to simulate the computation of the neurons in the brains. Multiple neurons form a layer, and a neural network has an input layer, which passes the computed values to the middle layers, and an output layer, which gives the results to the users. The middle layers are also called hidden layers. Deep neural network is referred to the neural networks that obtain more than one hidden layer. The number of layers of DNNs can be more than a thousand. The neural networks are trained to determine the weights between layers, and after training, the weights are applied to the computation for inference.

The first applications of neural networks is the LeNet, which performs handwriting digit recognition [32]. After that, DNNs are applied in the domain of speech recognition [17], image recognition [29], and so on. Nowadays, DNNs are largely used in different domains for different purposes. For example, videos and images are analyzed to detect objects or backgrounds, which can be further used for object classification and background elimination [11,53]; voices and texts are analyzed for translation [52] or natural language processing (NLP) [62], so that computers can translate different languages and speak the sentences. Moreover, DNNs can be applied in the medical domains to help humans detect diseases [18].

2.3 Cloud Computing

With the growing demands on computation, cloud computing is proposed. The National Institute of Standards and Technology defined cloud computing as a model to provide shared, on-demand computing resources such as networks, servers, storage, and so on [40]. Essential characteristics include: (i) on-demand self-service, users can require computing resources without human interaction with the service provider, (ii) broad network access, the computing resources can be accessed remotely via client platforms, (iii) resource pooling, the computing resources are pooled for multi-tenant usage and dynamically assigned to the users, (iv) rapid elasticity, resources can be assigned or released on demand, and (v) measured service, resource usage is monitored and reported to both users and provider. Hence, with cloud computing, users can access fruitful computing resources remotely on a client platform, which provides the demanded resources with scalability and transparency.

The service models of cloud computing can be categorized into: (i) Software as a Service (Saas), (ii) Platform as a Service (PaaS), and (iii) Infrastructure as a Service (IaaS). SaaS provides applications running on the cloud, which multiple client devices can access. For instance, Google Workspace, Dropbox, and Cisco WebEx are SaaS that is highly used in our daily lives. Users have little control to these applications, however, users do not need to install, manage, or maintain the applications alone. PaaS allows users to deploy applications using the given framework. Users create their software applications upon the framework without consideration of infrastructure. Example includes AWS Elastic Beanstalk, Windows Azure, and OpenShift. Last, IaaS allows the users fully controlling the infrastructure. Compared to PaaS, the users of IaaS need to manage the operating system, middleware, data, and other aspects by themselves. With the "pay-as-you-go" model, users select their own needed hardware (CPU, GPU, RAM, storage, and so on) and build their applications on their chosen operating system. For example, Amazon Web Services (AWS) is one of the most popular IaaS in the world.

2.4 Edge Computing

With the growth of IoT, data generated by the IoT devices is estimated to exceed the amount of traffic the Internet can hold. The resulting traffic congestion makes the transmission latency from IoT devices to the cloud unacceptable. Hence, sending all these data to the remote cloud for analysis is inefficient. To cope with this issue, edge computing is proposed to perform the computation at the edge of the network, for example, a gateway between the IoT devices, i.e., the data sources, and the cloud is considered as an edge server that can hold the computation at the edge. Fig. 2.2 shows the differences between cloud computing and edge computing.

Shi et al. [50] analyzed that edge computing is needed for the following reasons:

- **Low response times are needed.** The bandwidth of the network is a bottleneck of response time with the growth of the data amount, applications such as automatic driving can not stand for high response time.
- **The number of data producers is increased.** The electrical devices will become part of IoT, which will generate tremendous amount of sensor data at the edge of the network. Moreover, taking photos or recording videos on mobile phones and share on the social media becomes human's daily lives. The pre-processing of these multimedia before uploading to the cloud is needed to prevent the traffic congestion.
- **Data is private.** Nowadays, humans focus more on their privacy, and the data generated at the network's edge is usually privacy-sensitive. Processing the data at the edge provides more security than sending the raw data to the cloud since the sensitive information is not sent to the public server. [48, 51].

They also argued that some features should be supported for a reliable system at the edge of the network, including:

- *Differentiation.* Services should be given different priorities. Urgent services, such as fall and car accident detection should be executed first.
- *Extensibility.* When a new service is needed, the system should be flexible enough so that the previous services are not impacted.
- *Isolation.* The crash of one service should not impact the whole system.
- *Reliability.* The system should be able to maintain the components in the system and network topology, and it should take action when a node failure is detected.

Last, for the workload allocation between edge and cloud, multiple metrics that should be considered: (i) latency, the trade-off between computation and transmission latency should be considered, (ii) bandwidth, despite the consideration of transmission time, users' data plan may also be the limitation when deciding to upload the data to the cloud, (iii) energy, the energy consumption of running the computation or sending the data should be considered, and (iv) cost, using the cloud resources or leveraging cache resources at edge add additional costs.

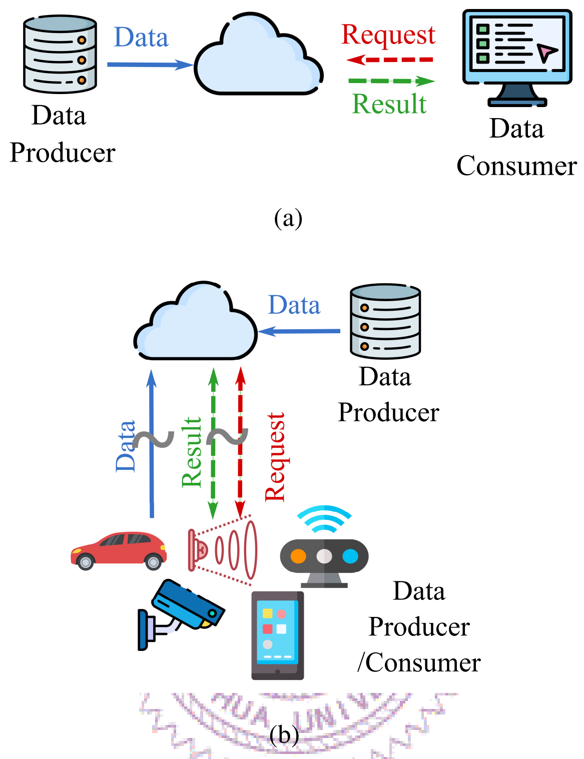


Figure 2.2: Paradigm of (a) cloud computing and (b) edge computing [50].

2.5 Thing-to-Cloud Continuum

IoT devices are getting more computation powerful with the advance of technologies. It is also possible to put the computation on the IoT device itself to further reduce the transmission of the data for lower transmission latency. Furthermore, people are focusing more on their data privacy. Sending the raw data to a public and shared server results in privacy concerns. With the rice of putting computation on IoT devices, a continuum among IoT devices, edge servers, and cloud servers are formed, as illustrated in Fig. 1.2. To orchestrate the computation among the continuum, several techniques can be applied. For example, containers [41] are used to create an isolated environment to prevent the library conflicts on different computing devices. Kubernetes [2] is originally proposed to manage and deploy the containers among the distributed cloud clusters [9], but it can also be applied in edge computing [20, 25, 57] and edge-cloud cooperation scenarios [21].

Furthermore, a lightweight version of Kubernetes [1] is built for resource-constraint IoT devices, making it more suitable for the thing-to-cloud continuum.

Chapter 3

Related Work

In this chapter, we first present the studies related to our proposed features: multi-task, early exit, and reconfiguration. Last, we survey the studies of computation deployment in the thing-to-cloud continuum.

3.1 Multi-task Networks

Reusing intermediate data from common prefix layers across several similar IoT analytics is called multi-task learning [15]. Studies related to shared layers can be roughly grouped into two classes: (i) hard and (ii) soft parameter sharing [47]. The former class shares the identical DNN layers among multiple IoT analytics. In contrast, the latter class shares the parameters across different neural network layers of different DNNs. Hard parameter sharing reduces the possibility of overfitting. Soft parameter sharing provides more flexibility for each tasks but may not scale well. Most multi-task studies focused on training, only a couple of them [16, 27, 35] studied the DNN inference with shared prefix layers. For instance, Jiang et al. [27] trained several DNNs for multiple video analytics with different numbers of shared layers to trade accuracy and complexity. They adaptively adopted different models during inference depending on available resources. Ma et al. [35] searched for common shared subgraphs in DNNS and stored these subgraphs in memory. They dynamically adjusted the subgraphs in memory to meet the demands of individual IoT analytics. Chao et al. [16] chose a few popular networks, and replaced the classifier (fully-connected layers) and some convolution layers with the target tasks. The number of convolution layers replaced depended on the size of the target task dataset. They trained the task-specific models with transfer learning [45]. Although multi-task increases the efficiency, deploying a multi-task model in the thing-to-cloud continuum has not been thoroughly studied.

3.2 Early Exit

DNNs with multiple exits have been proposed, which enable IoT analytics to trade accuracy levels for lower resource demands. Teerapittayanon et. al [55] originally proposed the early exit concept to infer with prefix layers as long as the accuracy reaches the target. The layers where inference can be stopped are referred to as exit points. They then created an exit point for each computing device, and sent the intermediate data to the next computing device only if the current accuracy was insufficient [56]. Li et al. [33] dynamically computed the deployment plans that specify partition and exit points between an IoT device and an edge server. They implemented a prototype system on a Raspberry Pi and an Intel PC. They showed that more layers are preferred when network bandwidth is increased, which leads to higher accuracy. Laskaridis et al. [31] trained DNNs with exit points. They partitioned the DNNs and guaranteed that the local device has at least an exit point so that the request could still be served even when the link is down. Torres et al. [58] introduced a framework based on Kafka-ML with fault-tolerance and efficient communication layers to manage DNN models. They combined the advantage of early exit and edge-cloud architecture to reduce the latency.

3.3 Dynamic Reconfiguration

Dynamic reconfiguration, or *repartition*, has been studied to cope with estimation errors of QoS levels. Yang et al. [61] considered the dynamic mobile cloud environments. They proposed reconfiguring mobile apps during cloud offloading to maximize the execution speed. They predicted the network status by users' mobility and designed an algorithm to make the reconfiguration decision for the predictable duration. Their evaluation showed that reconfiguration reduced the application execution time by at least 35%. McNamee et al. [39] considered the reconfiguration of DNN under diverse resource conditions. They investigated if reconfiguration is useful in edge computing area. They run eight pre-trained DNNs under different conditions such as CPU/memory load and network rate. The results showed that network condition has larger impacts on the performance and reconfiguration can provide better performance. Majeed et al. [36] considered reducing resulted service downtime when reconfiguring DNNs between edge and cloud. Instead of the "Pause and Resume" method, they decided to deploy a secondary pipeline and switch to the new pipeline after the deployment was finished. Their proposed method reduced the service downtime from 6 seconds to less than one millisecond. However, twice the amount of resources is needed.

Existing work has focused only on singular aspects between multi-task, early exit, and dynamic reconfiguration. To the best of our knowledge, T2C is the first system to be capable of performing all of them.

3.4 DNN Deployment

Although the deployment of DNNs for inference has been proposed [19,38], prior studies often deployed individual DNNs in their entirety. In particular, Fang et al. [19] proposed to prune models into compact versions, referred to as seeds, which can be converted back to original models. When conducting the inference, the model that best fits the current available resources is selected. Mathur et al. [38] classified layers into: (i) convolution and (ii) fully-connected ones. They then scheduled these two layer classes in FIFO and greedy manner, respectively, which resulted in reduced inference time without accuracy drop by utilizing more resources. Deploying entire DNNs on IoT devices or edge servers may overload the devices/servers, potentially turning the users away.

With the improved capabilities of IoT devices, approaches to split DNN models and distribute them across IoT devices, edge servers, and cloud servers have gained popularity through the use of container architectures. One way is to divide the input raw sensor data [37,63]. Each device/server hosts a complete DNN. By reducing the input data size, the workload on each device/server is reduced.

A more popular way is to divide the DNNs by layers. Kang et al. [28] proposed to partition DNNs for mobile devices and cloud servers in the unit of layers. Because each device/server now handles a smaller number of layers (and neurons), the workload on each device/server is reduced. Li et al. [34] considered diver DNN topologies: for chain topology, they partitioned models by layers; for DAG topology, they partitioned models in a grouped-layers unit. Hu et al. [26] considered DNNs in graph representations. For light workloads, they proposed to convert the problem to a weighted min-cut problem and solve it; for heavy workloads, they reduced the problem from the most balanced minimum st-vertex cut problem and proved it is NP-hard. Besides cutting by layers, some approaches used units smaller than a layer. Hsu et al. [25] presented a container-based solution for slicing TensorFlow-based [6] DNNs in the unit of an operator, the smallest computing unit of a model. The authors consider visual analytics, such as surveillance and traffic monitoring, at the edge devices as the driving usage scenario. Mohammed et al. [42] employed matching theory from economics to partition DNNs into multiple subgraphs. They then adaptively offload these subgraphs to individual devices/servers. They employed a python based simulator for evaluations, showing that they achieved up to 4.2 times faster performance than the state-of-the-art.

Distributed inference of conventional DNNs offers all-or-nothing IoT analytics, as only one exit (output) is supported, which could lead to abandoned IoT analytics when the available resources of the IoT infrastructure are limited. Note that these papers only considered the deployment plan between thing-edge or edge-cloud, the topology of thing-edge-cloud had not been fully studied yet.

3.5 IoT Analytics Deployment

Bonomi et al. [12] propose the concept of fog computing, which extends the cloud computing to end devices. The authors focus on the advantages of fog computing. Moving resources closer to IoT devices makes fog computing suitable for IoT deployment [22, 23, 59]. Cardellini et al. [14] implement a QoS-aware scheduler for stream processing systems using fog computing. They place the applications near to the data sources and subscribers and show that it outperforms the centralized scheduler. Lakshmanan et al. [30] surveyed the operator placement problem for streaming tasks, concluding the characteristics of operator placement and compare to the existing works. They drew a decision tree for future researchers to find the related operator placement algorithms. In recent years, data streaming processing (DSP) applications have been transferred from cloud to edge. Thus, the placement problem starts to take the heterogeneity of computing and network resources into account. Hong et al. [22] consider the diverse QoS levels and heterogeneous devices to make the optimal placement decisions in a fog computing platform. Cardellini et al. [13] propose a general formulation of the optimal DSP placement (ODP) and further show several DSP placement heuristics to compare the performance of different ways under different configurations [43].

Our proposed system deploys multi-task DNN over the thing-to-cloud continuum and we enable the early exits to trade the accuracy and latency considering request requirements. While most of existing studies focus on one-time requests, we consider long-term, recurring requests from multi-tenant in a smart environment when making the deployment plans. We also look into the performance of our plans and reconfigure them at runtime, which is not been thoroughly considered before.

Chapter 4

System Overview

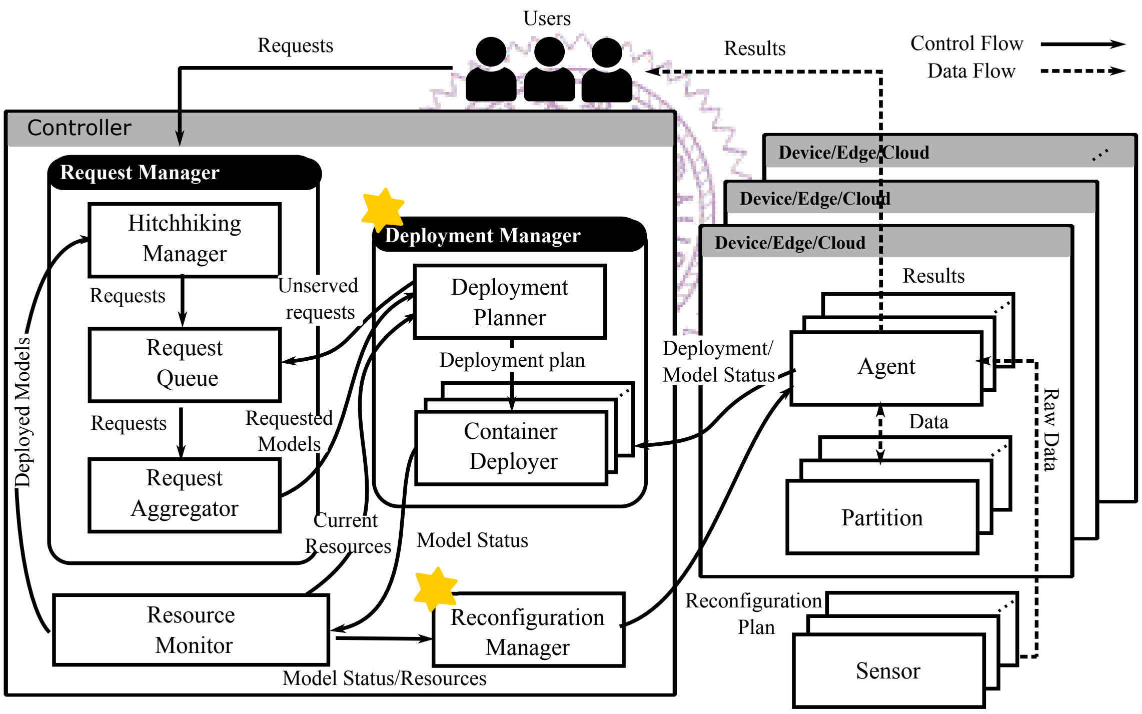


Figure 4.1: The component diagram of our proposed T2C system.

In this chapter, we present the design and workflow of our proposed T2C system. Fig. 4.1 presents the design of our T2C system. The gray outer boxes represent different machines, namely the controller and computing devices, which can be IoT devices, edge servers, and cloud servers. The controller manages the system and the computing devices execute the DNN models. Users send requests to the controller for recurring IoT analytics results, which are delivered by computing devices.

4.1 Components

The controller makes the key decisions on the deployment of the DNNs based on user requests with the following components:

- *Request manager* takes the requests from users. These incoming requests first go through the *hitchhiking manager*, which checks the already-deployed models to see if any of them can satisfy the requirements of the requests. If found, users are pointed to the results of these models, and the requests are considered handled. Other requests are pushed into the *request queue*. The *request aggregator* periodically aggregates the requested tasks in the request queue into as few multi-task DNN models as possible. The interval of each aggregation is referred to as the *aggregation period*, which is a system parameter. In the aggregation process, the aggregator takes the strictest requirements from the requests of the same task to be the requirements of that task.
- *Deployment manager* computes and executes the deployment decisions. Within it, the *deployment planner* takes the aggregated models from the request aggregator and computes the decisions under the available resource levels. The *container deployers* then deploy the DNN models following the deployment plans. Each container deployer is responsible for monitoring the model it deployed. Note that some DNN models may fail to be deployed due to resource limitations, and their corresponding requests are pushed back to the request queue for future deployment. Those requests that are considered unserviceable with the current resource status are also pushed back to the request queue.
- *Resource monitor* keeps track of the computation/network resources of the system and the status of already-deployed models reported by the container deployers.
- *Reconfiguration manager* monitors the QoS levels of the deployed models. It is triggered once every *reconfiguration period*, and decides whether a model needs to be reconfigured to handle changes in the system and environment dynamics.

Each *computing device* runs the subsets of layers belonging to the deployed DNN models, i.e., *partitions*. Each partition is managed by an *agent*. The agent facilitates internal communications, including:

- Receive data from sensors.
- Send/receive intermediate data from other partitions.
- Report DNN status to container deployers.
- Listen to the commands from reconfiguration manager and reconfigure the partition.

Furthermore, the agent sends IoT analytics results to users via message exchange services.

4.2 Workflow

When a user request comes, it first goes through the hitchhiking manager to check if any matched deployed models exist. If yes, the request is served. Otherwise, it is pushed into the request queue and then aggregated to a model by the request aggregator. The deployment planner takes the aggregated models from the aggregator and gets the current resource status from the resource monitor. It makes the deployment plan and lets the container deployer deploy the model to the computing devices. Those unserved requests are pushed back to the request queue. At run time, the agent sends the results to the users and reports the status of the model to the container deployer. The container deployer logs the model status to the resource monitor. The reconfiguration manager then monitors the QoS of the model by the logs in the resource monitor and makes the reconfiguration plan with the current resource status if a QoS drop is detected. When the model is finished, the agent informs the container deployer and terminates the corresponding partition. The container deployer then lets the resource monitor know that the model is terminated.

With these software components, we divide the deployment decision-making process into two phases: *planning* and *operation*. In the planning phase, we solve a *deployment planning* problem, which generates a deployment plan given resource levels. In the operation phase, we solve a *dynamic reconfiguration* problem, which checks the QoS of the already-deployed DNN models and reconfigures the deployment plans if needed. These two problems are solved by the deployment manager and reconfiguration manager, respectively. We present these two problems and algorithms in the next two chapters.

Chapter 5

Planning Phase: Deployment Planning

In this chapter, we formulate the deployment planning problem and present our system models for latency prediction. We summarize frequently used symbols in Table. 5.1. Last, we present our deployment planning algorithm.

Table 5.1: Frequently Used Symbols

Symbol	**Description**
M	Number of models
T_m	Number of tasks of model m
L_m	Number of layers of model m
R_t	Number of requests of task t
δ_r	Latency requirement of request r
a_r	Accuracy requirement of request r
$\mathbf{p}_m$	Partition points of model m
$\mathbf{x}_m$	Exit points of model m
h_t	Throughput of task t
$\mathbf{Y}_m$	Layer computing latency table of model m
$\hat{\mathbf{Y}}_m$	Exit layer computing latency table of model m
$\mathbf{C}$	Computing power
$\mathbf{B}$	Network bandwidth
$\mathbf{N}$	Background traffic throughput
$\boldsymbol{\tau}$	Scaling factors

5.1 Notations

There are M DNN models in each deployment planning problem. Each model $m \in M$, contains T_m tasks and L_m layers in total. Each task $t \in [1, T_m]$, comes with $L_{m,t}$ layers and $X_{m,t}$ exits. We notice that the indices of *model layers* L_m and those of *task layers* $L_{m,t}$ are different. We define a function $\mu(l_m) = l_{m,t}$ to map a model layer index l_m into its corresponding task layer index $l_{m,t}$, and $\mu'(l_{m,t}) = l_m$ as its inverse function. For notation convenience, $\mu(l_m)$ returns $L_m + 1$ if layer l_m is irrelevant to task t. We use R_t to denote the number of requests asking for task t. For a specific request r, where $r \in [1, R_t]$, we write its accuracy requirement as a_r and its latency requirement as δ_r.

Several types of resources in thing-to-cloud continuum need to be carefully allocated. We assume that each IoT device is connected to a single edge server, which is then connected to a single cloud server. This is reasonable because IoT devices are resource-limited and thus each of them only connects to a single gateway, serving as its edge server. For computing power, we let $\mathbf{C} = \langle c, c', c'' \rangle$, where $c, c', c'' \in \mathbb{N}$ denote the numbers of free CPU cores on the IoT device, edge server, and cloud server, respectively. Although we only consider three computing devices, sensors are often connected to IoT devices with short-range wireless links such as Bluetooth and Zigbee. Therefore, for networking resources, we let $\mathbf{B} = \langle b, b', b'' \rangle$ be the network bandwidth of the sensor-device, device-edge, and edge-cloud links. $\mathbf{B}$ represents the link capacity, which could be partially consumed by background traffic due to, e.g., already-deployed DNN models. Therefore, we also define $\mathbf{N} = \langle n, n', n'' \rangle$ as the background traffic throughput, which is measured at the run-time.

The decision variables of our problem are composed of *partition points* and *exit points* of each m, where $m \in [1, M]$. Partition points $\mathbf{p}_m = \langle p_m, p'_m \rangle$, where $p_m, p'_m \in [1, L_m]$, specifies the boundary layer indices between: (i) the IoT device and edge server and (ii) the edge and cloud servers. In typical cases, layers of task t with task layer indices $l \in (0,\ \mu(p_m)]$ are deployed on the IoT device, $l \in (\mu(p_m),\ \mu(p'_m)]$ are deployed on the edge server, and $l \in (\mu(p'_m),\ L_{m,t}]$ are deployed on the cloud server.

To keep track of resource consumption, we introduce auxiliary boolean variables $\langle \hat{c}_{m,t}, \hat{c}'_{m,t}, \hat{c}''_{m,t} \rangle$ for task t and $\langle \hat{c}_m, \hat{c}'_m, \hat{c}''_m \rangle$ for model m to indicate whether the IoT device, edge server, and cloud server participate in task t and model m, respectively. We let $\hat{c}_{m,t} = 1$ iff the IoT device helps execute task t. It is not hard to see $\hat{c}_m = max_{t \in [1,T_m]}(\hat{c}_{m,t})$. We define $\hat{c}'_{m,t}$, $\hat{c}'_m$, $\hat{c}''_{m,t}$, and $\hat{c}''_m$ in a similar way. These auxiliary variables are functions of $\mathbf{p}_m$. There are seven cases of $\langle p_m, p'_m \rangle$, which affect the definition of auxiliary variables. For the sake of brevity, we give two examples: (i) if $0 < \mu(p_m) < \mu(p'_m) < L_m$, t uses all three computing devices, i.e., $\hat{c}_{m,t} = \hat{c}'_{m,t} = \hat{c}''_{m,t} = 1$, and (ii) if

$\mu(p_m) = 0,\ \mu(p'_m) = L_{m,t}$, all layers of t run on edge server, i.e., $\hat{c}_{m,t} = \hat{c}''_{m,t} = 0$ and $\hat{c}_{m,t} = 1$.

Last, for exit points, we let $x_{m,t} \in [1, X_{m,t}]$ be the exit point index of task t. We use $\epsilon(x_{m,t})$ to denote the last task layer index of task t at exit point $x_{m,t}$. We collectively write all $\mathbf{p_m}$ as $\mathbf{p}$, all $x_{m,t}$ as $\mathbf{x_m}$, and all $\mathbf{x_m}$ as $\mathbf{x}$ for the ease of presentation.

5.2 System Models

We construct the latency and accuracy models to estimate the QoS levels achieved by any given solution of a deployment planning problem. Considering model m, we denote the expected latency of task t as $\overline{\delta}_{m,t}$ and the expected accuracy as $\overline{a}_{m,t}$. We first write the latency of t as a function of $\mathbf{p}_m$ and $x_{m,t}$, i.e., $\overline{\delta}_{m,t} = \overline{\delta}(\mathbf{p}_m, x_{m,t}) = \overline{\delta}_c(\mathbf{p}_m, x_{m,t}) + \overline{\delta}_s(\mathbf{p}_m, x_{m,t})$, where $\overline{\delta}_c(\cdot)$ and $\overline{\delta}_s(\cdot)$ represent the computing and transmission latency components, respectively. Like prior studies [28, 33], we empirically build $\overline{\delta}_{m,t}$ functions. Particularly, we construct lookup tables through performance profiling, although other modeling techniques, like regression analysis and machine learning techniques can also be adopted.

Upon profiling, the computing latency table $\langle Y_{m,t,l}, Y'_{m,t,l}, Y''_{m,t,l}\rangle \in \mathbf{Y_m}$ estimates the computing latency of executing layer l on IoT device, edge server, and cloud server. Note that, the fully-connected layers of task t are identical at all exit points. Hence, instead of profiling them individually for multiple times, we aggregate the fully-connected layers of t at any exit point of m as an *exit layer* $\hat{l}_{m,t}$. We then model $\hat{l}_{m,t}$ using another lookup table $\langle \hat{Y}_{m,t}, \hat{Y}'_{m,t}\hat{Y}''_{m,t}\rangle \in \hat{\mathbf{Y}}_{m,t}$ for the computing latency of the exit layers.

To better adapt to dynamic environments, we employ *scaling factors* $\boldsymbol{\tau}$ to accommodate the ratio between real (from live profiling) and estimated (from current model) QoS levels. Take the computing latency as an example, we calculate the scaling factors $\boldsymbol{\tau}_c = \langle \tau_c, \tau'_c, \tau''_c\rangle$ in a sliding window and multiply the estimates from $\mathbf{Y}_m$ and $\hat{\mathbf{Y}}_{m,t}$ by the proper τ_c for the final, corrected, estimations. With the above models, we write $\overline{\delta}_c(\mathbf{p}_m, x_{m,t})$ as:

$$\sum_{l=1}^{\substack{min(p_{m,t},\\ L_{m,t,x})}} \tau_c Y_{m,\mu'(l)} + \sum_{\substack{l=min(p_{m,t},\\ L_{m,t,x})+1}}^{\substack{min(p'_{m,t},\\ L_{m,t,x})}} \tau'_c Y'_{m,\mu'(l)} + \sum_{\substack{l=min(p'_{m,t},\\ L_{m,t,x})+1}}^{L_{m,t,x}+1} \tau''_c Y''_{m,\mu'(l)} +$$

$$\tau_c \hat{Y}_{m,t} \mathbf{1}_{\substack{p_{m,t}<\\ L_{m,t,x}}} + \tau'_c \hat{Y}'_{m,t} \mathbf{1}_{\substack{L_{m,t,x}\leq\\ p_{m,t}}} \mathbf{1}_{\substack{p'_{m,t}<\\ L_{m,t,x}}} + \tau''_c \hat{Y}''_{m,t} \mathbf{1}_{\substack{L_{m,t,x}\leq\\ p'_{m,t}}}\ , \tag{5.1}$$

where we sum up the latency before exit points, and use the indicator function $\mathbf{1}_{\text{expr}}$ to add that of the exit layer. In this equation, we write $p_{m,t} = \mu(p_m),\ p'_{m,t} = \mu(p'_m)$, and $L_{m,t,x} = \epsilon(x_{m,t})$ for brevity.

The transmission latency can be written as a function of bandwidth b from $\mathbf{B}$, throughput n from $\mathbf{N}$, and size of layer l of task t belonging to model m, denoted as $Z_{m,t,l}$. In particular, we profile Z into a lookup table. We then write the transmission latency on sensor-device link as:

$$s_{m,t,l} = \frac{Z_{m,t,l}}{\alpha(b-n)}, \tag{5.2}$$

where α is a decimal number between 0 and 1, providing a cushion for possible surges of background traffic. The transmission latency of device-edge $s'_{m,t,l}$ and edge-cloud $s''_{m,t,l}$ can be defined similarly. As a corner case, we write the transmission latency of raw sensor data as $\langle s_{m,t,0}, s'_{m,t,0}, s''_{m,t,0} \rangle$ based on $Z_{m,t,0}$. We also apply scaling factors $\boldsymbol{\tau}_s = \langle \tau_s, \tau'_s, \tau''_s \rangle$ on transmission latency. We collectively write $\boldsymbol{\tau}_c$ and $\boldsymbol{\tau}_s$ as $\boldsymbol{\tau}$. With the above definitions, we write $\overline{\delta}_s(\mathbf{p}_m, x_{m,t})$ as:

$$\begin{aligned}
&[\tau_s s_{m,t,0} + (1-c_m) max(c'_m, c''_m)\tau'_s s'_{m,t,0} + \\
&(1-c_m)(1-c'_m)c''_m \tau''_s s''_{m,t,0}] + \\
\mathbf{1}_{p_{m,t}<L_{m,t,x}}&[max(c'_m, c''_m)\tau'_s s'_{m,t,p_{m,t}} + c''_m \tau''_s s^{b''}_{m,t,p_{m,t}}] + \\
\mathbf{1}_{p'_m<L_{m,t,x}}&[max(c'_m, c''_m)\tau'_s s^{b'}{}_{m,t,p'_m} + c''_m \tau''_s s^{b''}{}_{m,t,p'_m}].
\end{aligned} \tag{5.3}$$

Here, we use auxiliary variables $\langle c_m, c'_m, c''_m \rangle$ to determine if a link is used. As an instance, the link device-edge is used iff edge server or cloud server is used, i.e., $max(c'_m, c''_m) = 1$. The first term in Eq. (5.3) accounts for the transmission latency of raw data. The second and third terms account for the transmission latency between partitions, where indicator functions determine if this latency should be included in task t, as t may exit earlier. Last, we calculate the expected latency by $\overline{\delta}_{m,t} = \overline{\delta}_c(\mathbf{p}_m, x_{m,t}) + \overline{\delta}_s(\mathbf{p}_m, x_{m,t})$.

Finally, for the expected accuracy, we define a function $\overline{a}(x_{m,t})$, which gives the average accuracy at exit point $x_{m,t}$. The values are also profiled in an offline process, and compiled into a lookup table. The expected accuracy of task t at exit $x_{m,t}$ is written as $\overline{a}_{m,t} = \overline{a}(x_{m,t})$.

5.3 Formulation

Our goal is to maximize the number of requests satisfying both the latency and accuracy requirements. For each model m, where $m \in [1, M]$, we define *throughput* h_t, where $t \in [1, T_m]$, to account for the number of satisfied requests of task t. h_t can be defined as:

$$h_t = \textstyle\sum_{r=1}^{R_t} \mathbf{1}_{\delta_r \geq \overline{\delta}_t \text{ and } a_r \leq \overline{a}_r}. \tag{5.4}$$

Next, we write the deployment planning problem as:

$$\underset{\mathbf{p},\mathbf{x}}{maximize} \sum_{m=1}^{M} \sum_{t=1}^{T_m} h_t \tag{5.5}$$

$$\begin{aligned} s.t.: \sum_{m=1}^{M} \hat{c}_m &\leq c; \\ \sum_{m=1}^{M} \hat{c}'_m &\leq c'; \\ \sum_{m=1}^{M} \hat{c}''_m &\leq c''. \end{aligned} \tag{5.6}$$

Eq. (5.5) maximizes the number of satisfied requests and Eq. (5.6) ensures sufficient computing power. Note that, we do not explicitly limit the network resources, because they are captured by $\overline{\delta}_t$ and then by h_t.

5.4 Our Proposed Algorithm

We design an efficient algorithm to solve the deployment planning problem in Fig. 5.1. We first sort all M models in the descending order of a model weight:

$$g_m = \sum_{t=1}^{T_m} R_t, \tag{5.7}$$

which represents the number of requests that can potentially served [1]. We then iterate through each model m, and select the most conservative $\mathbf{x}_m$ so that all tasks' accuracy requirements are met. We enumerate all feasible $\mathbf{p}_m$ for the task with the largest number of layers, but without exceeding the available computing powers into a set $\mathbf{P}$. Within $\mathbf{P}$, we pick $\hat{\mathbf{p}}_m$ that leads to the largest number of served requests for model m. Here, we break ties by shorter expected latency. We then update the consumed computing powers and move on to the next iteration. We stop once no computing power is available or all the models are served. Our algorithm has a polynomial time complexity of $O(ML^2)$, where $L = \max_{m=1}^{M} L_m$. L^2 comes from the enumeration in line 14, which dominates the for-loops in lines 5 and 7.

[1]Different definitions of g_m can be adopted in diverse scenarios, as our algorithm does not rely on any property of g_m.

```
Input: $M$, $\mathbf{C}$, $\mathbf{B}$, and $\mathbf{N}$
Output: $\mathbf{p}$ and $\mathbf{x}$.
1:  sort $M$ models with weights $g_m$ in the desc. order
2:  for $m \in [1, M]$ do
3:      do
4:          Calculate $\langle s_{m,t,l}, s'_{m,t,l}, s''_{m,t,l} \rangle,\ l \in [0, L_m]$
5:          for $t \in [1, T_m]$ do
6:              $a_{m,t} = max(a_r),\ \forall r \in [1, R_t]$
7:              for $x_{m,t} \in [1, X_{m,t}]$ do
8:                  if $\overline{a}(x_{m,t}) \geq a_{m,t}$ then
9:                      Put $x_{m,t}$ into $\mathbf{x}_m$
10:                     Break
11:                 end if
12:             end for
13:         end for
14:         Enumerate all feasible $\mathbf{p}_m$ into $\mathbf{P}$
15:         Let $\hat{\mathbf{p}}_m$ be the $\mathbf{p}_m \in \mathbf{P}$ with the largest $\sum_{t=1}^{T_m} h_t$
16:         Add $\hat{\mathbf{p}}_m$ into $\mathbf{p}$
17:         Calculate $\langle \hat{c}_m, \hat{c}'_m, \hat{c}''_m \rangle$ of $\hat{\mathbf{p}}_m$
18:         $c = c - \hat{c}_m;\ c' = c' - \hat{c}'_m;\ c'' = c'' - \hat{c}''_m$
19:     while $c + c' + c'' > 0$
20: end for
21: Return $\mathbf{p}$ and $\mathbf{x}$
```

Figure 5.1: Our proposed deployment planning algorithm.

Chapter 6

Operation Phase: Dynamic Reconfiguration

In this chapter, we formulate the dynamic reconfiguration problem and present our proposed dynamic reconfiguration algorithm.

6.1 Problem

The dynamic reconfiguration problem is working for a deployed model m that is considered having a drop on QoS levels at run time. A deployed model m contains T_m tasks, each task $t \in [1, T_m]$ serves R_t requests. Each request $r \in [1, R_t]$ has a latency requirement δ_r and accuracy requirement a_r. When the runtime latency and accuracy can not meet the requested latencies or accuracies, we consider changing the decision of partition points $\mathbf{p}_m$ and exit points $\mathbf{x}_m$ to maximize the throughput $\sum_{t=1}^{Tm} h_t$ given the assigned computing devices $\langle \hat{c}_m, \hat{c}'_m, \hat{c}''_m \rangle$ from the planning phase and runtime network resources $\mathbf{B}$ and $\mathbf{N}$.

We let $\langle \overline{c}_m, \overline{c}'_m, \overline{c}''_m \rangle$ to be the auxiliary boolean variables of the new decision of $\mathbf{p}_m$. To estimate the throughput of the new decision, we applied the system models presented in Sec. 5.2 and applied Eq. (5.4) to calculate the throughput. Hence, we write the dynamic reconfiguration problem as:

$$\underset{\mathbf{p}_m, \mathbf{x}_m}{maximize} \sum_{t=1}^{T_m} h_t \tag{6.1}$$

$$\begin{aligned} s.t.: \sum_{m=1}^{M} \overline{c}_m &\leq \hat{c}_m; \\ \sum_{m=1}^{M} \overline{c}'_m &\leq \hat{c}'_m; \\ \sum_{m=1}^{M} \overline{c}''_m &\leq \hat{c}''_m. \end{aligned} \tag{6.2}$$

Eq. (6.1) maximizes the number of satisfied requests and Eq. (6.2) ensures we only use the assigned computing power.

6.2 Our Proposed Algorithm

We used optimal algorithm to solve the dynamic reconfiguration problem since the algorithm only considers one model and it is only executed when the model has a drop on QoS levels. We consider a model as having a QoS drop if any of its requests are not satisfied. Different definitions of QoS drop can be adopted in diverse scenarios.

Fig. 6.1 gives the dynamic reconfiguration algorithm. We first enumerate all set of exit points $\mathbf{x}_m$ into a set $\mathbf{X}$, then for each $\hat{\mathbf{x}}_m \in \mathbf{X}$, we enumerate all feasible partition points $\mathbf{p}_m$ with the largest number of layers and the assigned computing powers. We then select $\hat{\mathbf{p}}_m$ with the shortest expected latency, i.e., the one that can potentially satisfy most latency requirements with current exit points $\hat{\mathbf{x}}_m$. By now, we have a pair of exit points and partition points candidates that has an expected latencies $\overline{\delta}_{m,t} = \overline{\delta}_c(\mathbf{p}_m, x_{m,t}) + \overline{\delta}_s(\mathbf{p}_m, x_{m,t})$ and expected accuracy $\overline{a}_{m,t} = \overline{a}(x_{m,t})$ for all task $t \in [1, T_m]$. Then the expected throughput $\overline{h}_m = \sum_{t=1}^{T_m} h_t$ can be calculated with Eq. (5.4). Finally, after searching all possible exit points and the corresponding partition points, we adopt the $\hat{\mathbf{x}}_m$ and $\hat{\mathbf{p}}_m$ that lead to the highest throughput.

In this algorithm, we consider exit points that may not satisfy the accuracy requirements of all requests, which is different from the deployment planning algorithm. We are doing this because the drop on QoS levels is indicating that the current environments can not satisfy the latency requirements with the given accuracy. Our algorithm in Fig. 6.1 has a time complexity of $O(X^T L^2)$, where $X = \max_{t=1}^{T_m} X_{m,t}$, $T = \max_{m=1}^{M} T_m$, and $L = \max_{m=1}^{M} L_m$. We consider the complexity acceptable since T is usually a small number.

```
Input: $m$, $\langle c_m, c'_m, c''_m \rangle$, $\mathbf{B}$, and $\mathbf{N}$.
Output: $\mathbf{p}_m$ and $\mathbf{x}_m$.
 1: Let $h_m = 0$
 2: Let $\mathbf{X}$ be the set of all possible $\mathbf{x}_m$
 3: for $\hat{\mathbf{x}}_m \in \mathbf{X}$ do
 4:     Let $\mathbf{P}$ be the set of all feasible $\mathbf{p}_m$ with $\hat{\mathbf{x}}_m$
 5:     Let $\hat{\mathbf{p}}_m$ be the partition points in $\mathbf{P}$ with the shortest expected latency
 6:     Calculate expected throughput $\overline{h}_m$ with $\hat{\mathbf{x}}_m$ and $\hat{\mathbf{p}}_m$
 7:     if $\overline{h}_m > h_m$ then
 8:         $\mathbf{x}_m \leftarrow \hat{\mathbf{x}}_m$
 9:         $\mathbf{p}_m \leftarrow \hat{\mathbf{p}}_m$
10:         $h_m \leftarrow \overline{h}_m$
11:     end if
12: end for
```

Figure 6.1: Our proposed dynamic reconfiguration algorithm.

Chapter 7

Implementations

This chapter introduces the prototype testbed used for our T2C system. We then show the implementation of DNN partitions upon Kubernetes. Last, we offer the multi-task models we trained for our system.

7.1 Testbed

We have implemented our proposed T2C system and algorithms. Our implementation is built upon multiple open-source projects, including Docker [41], Kubernetes [2], and ZeroMQ [4]. The detail is given below.

- **Docker** [41]. We pack the DNNs into containers to avoid library conflicts. Moreover, containers can take arguments at initialization time for higher flexibility. We assume all the devices have downloaded and cached the required container images.
- **Kubernetes** [2]. Kubernetes clusters help us to deploy containers and monitor their resource consumption. It can also automatically recover or restart the containers under exceptions.
- **ZeroMQ** [4]. ZeroMQ (ZMQ) exchanges data among the DNN models and agents. With ZMQ's socket-based Request-Reply pattern, each agent on a computing device sends the intermediate (or raw) data to the next agent using a Request, and waits for an acknowledgement in a Reply. The socket-based pattern enables faster data transfer.

We deploy our implementation on a Kubernetes cluster with three computing devices. In particular, we adopt: (i) an Upboard [5] with an Intel Atom CPU @1.44 GHz and 4 GB RAM as the IoT device, (ii) an NUC with an Intel i3 CPU @1.7 GHz and 16 GB RAM as the edge server, and (iii) a laptop with Intel i7 CPU @2.3 GHz and 32 GB RAM as the cloud server. Fig. 7.1 shows our IoT device and edge server. Moreover, we have a PC with Intel i7 CPU @3.6 GHz and 16 GB RAM serving as the controller.

Fig. 7.2 shows the network topology of our testbed. The devices are connected with Ethernet cables. Specifically, the control plane allows the controller to communicate with computing devices and the data plane exchanges data among DNNs. The resulting testbed serves as a proof-of-concept, and is used for evaluations.

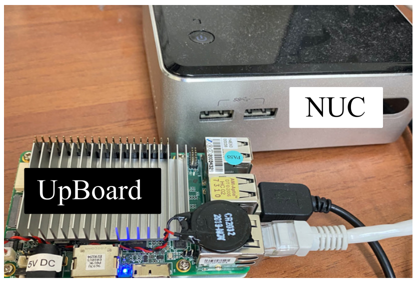


Figure 7.1: A photo of our testbed.

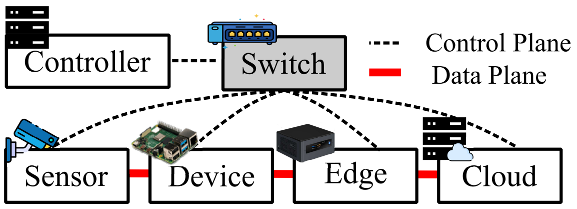


Figure 7.2: Network topology of our testbed.

7.2 Kubernetes and Model Deployment

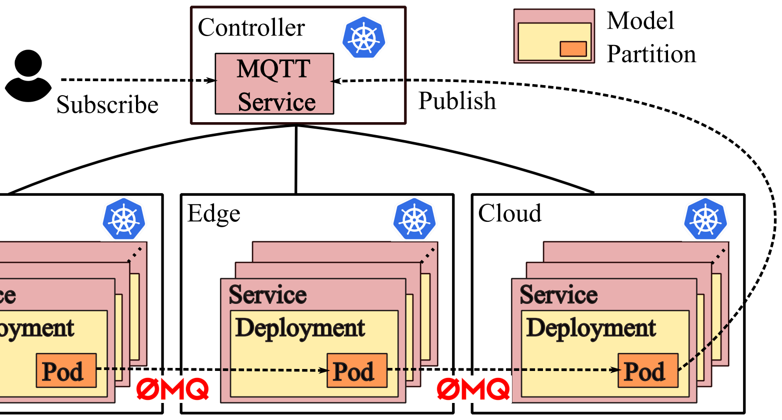


Figure 7.3: The implementation of our Kubernetes-based testbed.

Fig. 7.3 illustrates our implementation of our Kubernetes-based testbed. We used three components of Kubernetes for each model partition: (i) pod, (ii) deployment, and (iii) service. A pod holds one or more docker images in the Kubernetes. We pass the arguments in the config file of the pod to execute the assigned layers in the partition. Deployment holds the pod, configuring the pod's behavior. In our implementation, we leverage deployment to: (i) set the CPU limits of each pod to 1000 m, so that each deployed partition can only consume a CPU core, (ii) determine where to run the pods, and (iii) restart the pods if any of the exceptions occurs. Finally, the service describes how to connect the pods. Here, we assign a cluster-internal IP to each pod for the data exchange. Service also provides DNS service with the name of the pod, this helps us find the target pod IP for communication.

Next, we edit the network configuration services on each computing device, restricting that the communication between the computing devices can only go through the specific network interface. Hence, we can separate the data plane and control plane illustrated in Fig. 7.2. With this setting, we monitor the throughput of each network interface with Linux ifstat. We set the monitor period to 5 s and report the values to the runtime system.

To send the data between partitions, we use the assigned cluster IP (the DNS name) with ZeroMQ and record the timestamp when sending out the message to determine the transmission latency. To cope with the timestamp synchronization issues, we run Linux Precision Time Protocol (PTP) on each computing device. After finishing the inference, the results are published to the users who subscribe to the model. Here, we use the pub-sub service provided by MQTT [3].

7.3 Multi-task Models

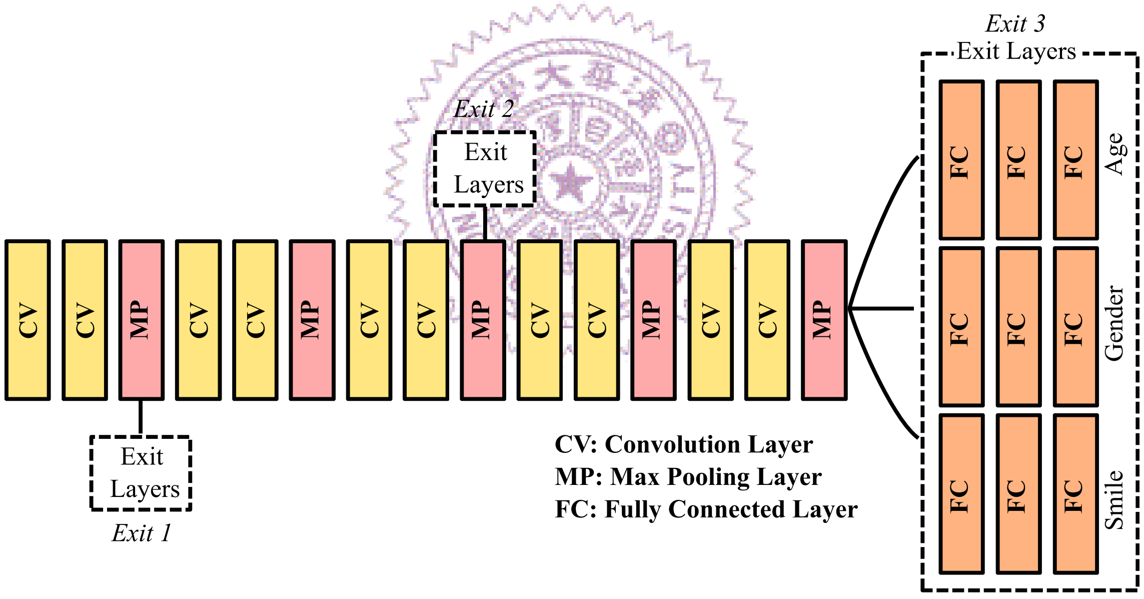


Figure 7.4: The layer structure of a sample multi-task age-smile-gender classification model.

Table 7.1: Average Computing Latency of Layers

Layer Type	Avg. Computing Latency	Norm. to Convolution
Convolution	7.24 ms	100%
Max Pooling	1.40 ms	19.33%
Fully Connected	1.24 ms	17.11%

We train the Age-Smile-Gender multi-task DNNs based on Viet and Bao [60] on top of PyTorch [46], which enables the early exit features with BranchyNet structure [55], as illustrated in Fig. 7.4. To build the lookup tables, we execute the inference 100 times and get the average computing time of each layer reported by Pytorch profiler. We then record the output data size of each layer after the compression with zlib. The results show that different types of layers incur different complexity. For example, our profiling reveals that max-pooling and fully-connected layers consume 19.33% and 17.11% running time compared to convolution layers in our multi-task networks, as reported in Table 7.1. To exercise the trade-offs between accuracy and complexity, we develop five variants of the model by varying the numbers of the convolution, max-pooling, and fully-connected layers in the DNN variants.

Chapter 8

Evaluations

In this chapter, we introduce the setup, metrics, and baselines of the evaluation. We then present the results of our T2C algorithms in both the planning and operation phase. Last, we show the implications of system parameters.

Table 8.1: Considered Network Conditions

Links	**Simulate**	**Bandwidth**	**Latency**
Sensor-Device	Bluetooth	1 Mbps	1 ms
Device-Edge	WiFi	24 Mbps	3 ms
Edge-Cloud	Broadband	40 Mbps	5 ms

Table 8.2: Compared Algorithms

Feature	**NEU**	**Edgent**	**T2C$_{M}$**	**T2C$_{MH}$**	**T2C$_{MHE}$**	**T2C$_{MHER}$**
Multi-task			✓	✓	✓	✓
Early Exit		✓			✓	✓
Hitchhiking				✓	✓	✓
Reconfiguration						✓

8.1 Setup

We employ Linux Traffic Control (tc) to emulate different network conditions on individual data plane links. More specifically, we assume that the sensor-device link is over Bluetooth with 1 Mbps bandwidth and 1 ms delay, the device-edge link is over WiFi with 24 Mbps bandwidth and 3 ms delay, and the edge-cloud link is over broadband access with 40 Mbps and 5 ms delay. Following the specifications of the computing devices, we set the computing powers of the IoT device, edge server, and cloud server to 3, 3, and 7, respectively. We set the aggregation period to 10 s and 20 s in the experiment of the planning phase and operation phase, respectively. We set the value of α for estimating transmission latency to 0.9. We also configure the reconfiguration period to 45 s, sliding window of $\boldsymbol{\tau}$ to 1 m, and the experiment duration for each run to 5 m.

For the workload, we generate user requests based on a public 311 call trace [44] following Poisson arrival processes. We take arrival rates from the top most frequently requested events and randomly assign them to our tasks. To exercise T2C under different workload, we amplify the arrival rates by $\{1X, 20X, 40X, 60X, 80X\}$ in the planning phase and $\{40X, 60X, 80X, 100X, 120X\}$ in the operation phase when conducting the experiments. For sensor data, we employ real surveillance videos collected from our smart campus testbed on the NTHU campus.

We have implemented our deployment planning and dynamic reconfiguration algorithms. To understand the merits of individual T2C features, we create four variants with different features: $T2C_{M}$, $T2C_{MH}$, $T2C_{MHE}$, and $T2C_{MHER}$, as summarized in Table 8.2. For comparison, we also implemented two baseline algorithms: Neurosurgeon (NEU) and Edgent. NEU [28] partitioned the DNN models between the mobile device and cloud server at the boundaries of layers. They selected the partition point for either the shortest latency or lowest mobile energy consumption. Edgent [33] adopted models with early exits and dynamically selected the exits and partition points between the IoT device and edge server. They sorted the exit points with the expected accuracy in the descending order and searched if there was any partition point for a given exit that satisfied the latency requirement. If not, they considered the next exit point with lower accuracy until the partition point was found or no other exit was available. Note that while the original Edgent algorithm only considers latency requirements, we augment Edgent always to select the exits meeting the accuracy requirements for a fair comparison.

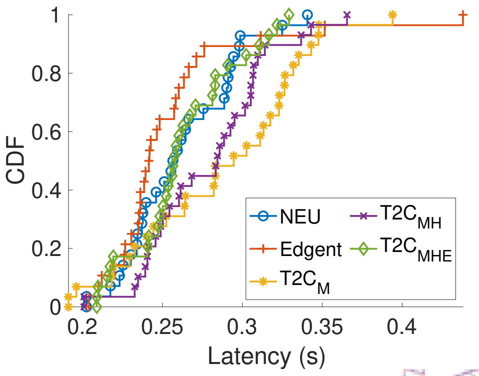
We measure the following metrics:

- *Throughput*. The number of served requests.
- *Satisfied Ratio*. The fraction of requests meeting both requirements over served requests.
- *Latency*. The inference time of IoT analytics.
- *Accuracy*. The accuracy of IoT analytics.
- *Queuing time*. The waiting time of each user request in the request queue.
- *CPU utilization*. The fraction of algorithm running time on our Intel i7 PC @3.6 GHz.
- *Deployment time*. Time difference between deployment plan generation and container launch.

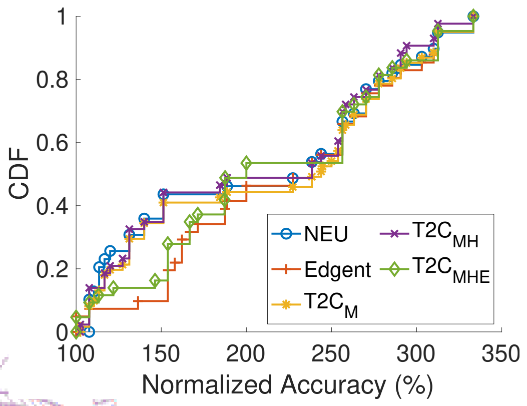
- *Data plane throughput*. The bandwidth consumption among DNN partitions.
- *Control overhead*. The bandwidth consumption of control messages.

We repeat each experiment five times.

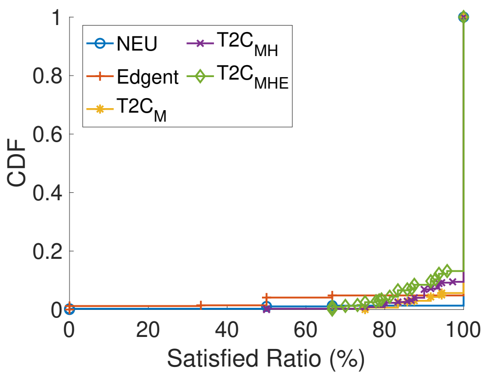
8.2 Planning Phase Results



(a)



(b)

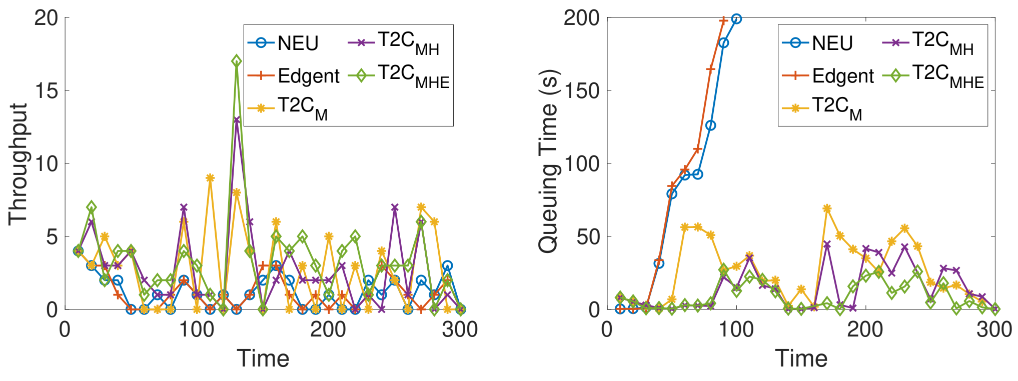


(c)

Figure 8.1: CDF from a sample run with default settings: (a) latency, (b) normalized accuracy, and (c) satisfied ratio.

Sample results under default setting. We report results from a sample run with 40X arrival rate[1]. For each run, we record the average performance once every 10 s. We plot latency, normalized accuracy, and satisfied ratio results into Cumulative Distribution Function (CDF) in Fig. 8.1. Fig. 8.1(a) shows that algorithms with early exits, i.e., Edgent and $T2C_{MHE}$, achieve much lower latency than other algorithms. Fig. 8.1(b) reveals that our T2C algorithms always achieve $\geq 100\%$ normalized accuracy w.r.t. the requirements. Fig. 8.1(c) shows that over 90% satisfied ratio is $\geq 90\%$. We plot throughput and queuing

[1]We could not use 1X because it only incurs four user requests throughout each experiment run.



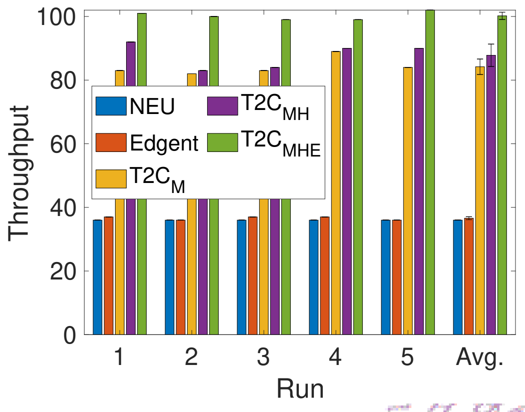
(a) (b)

Figure 8.2: Results from a sample run with default settings: (a) throughput and (b) queuing time.

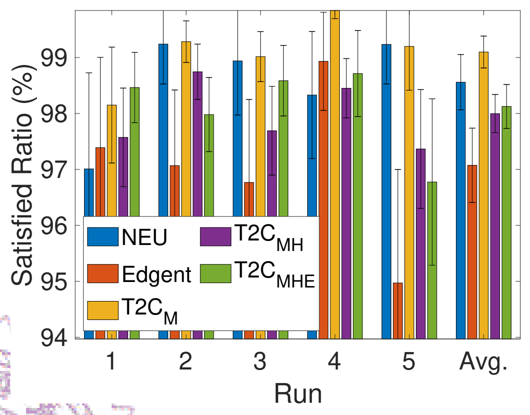
time results in Fig. 8.2. Fig. 8.2(a) shows that with multi-task, the number of served requests can have bursts among the time since we can serve many requests at one time. Fig. 8.2(b) reveals that at the start of the system, T2C needs to wait a bit longer, i.e., the aggregation period. However, as the resources are limited, NEU and Edgent use up resources easily and thus the queuing time of their requests grows rapidly. We can also see that hitchhiking obtains more requests with low queuing time. To gain a high-level view of the performance of our T2C algorithms, we report the average results across five runs with 95% confidence intervals in the rest of this section.

Multi-task and hitchhiking improve throughput without sacrificing satisfied ratio and latency. Fig. 8.3 gives the overall results from the default settings. When comparing their performance, we separate these algorithms into two groups: with and without the early exit feature. This is because early exits significantly reduce the latency as shown in Fig. 8.1(a). Fig. 8.3(a) reveals that T2C_{M} and T2C_{MH} achieve 2.34 and 2.44 times of throughput compared to NEU. Similarly, T2C_{MHE} delivers 2.74 times of throughput compared to Edgent. This figure demonstrates that multi-task largely increases and hitchhiking further boosts the number of served requests. Fig. 8.3(b) shows that our algorithms satisfy $\geq$ 97% of user requests, even when the throughput is high. Such high satisfied ratios can be attributed to the minimum ($\leq$ 20 ms) latency increase, compared to single-task NEU and Edgent, which is reported in Fig. 8.3(c). Last, Fig. 8.3(d) shows that all algorithms achieve at least 2 times of the accuracy requirement.

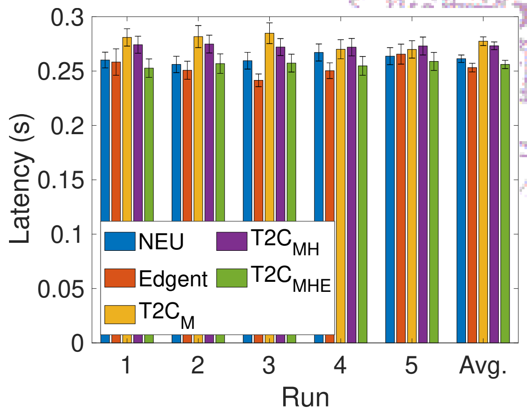
Early exit achieves lower latency while satisfying accuracy requirements. Fig. 8.3(c) shows that both Edgent and T2C_{MHE} achieve 3.85% lower latency compared to NEU and 7.68% lower latency compared to T2C_{M}. Fig. 8.3(d) depicts that such latency reductions do not negatively affect the normalized accuracy w.r.t. to requirement. This indicates that



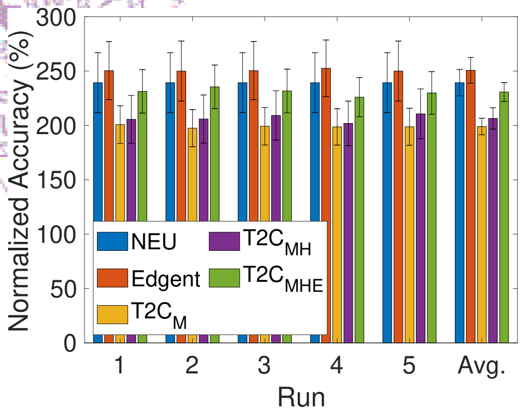
(a)



(b)

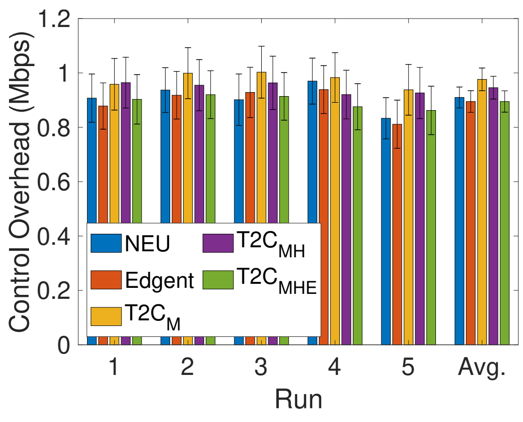


(c)

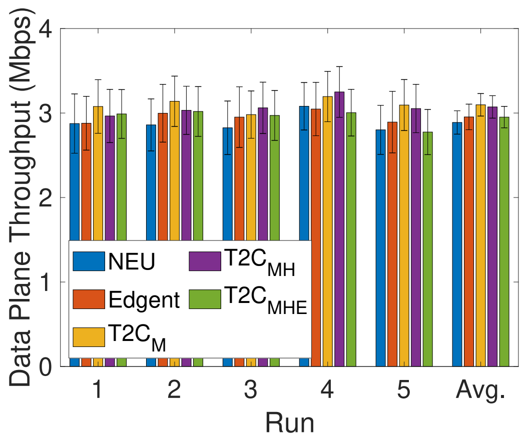


(d)

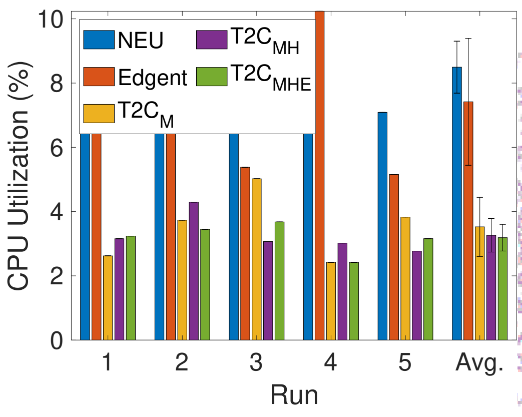
Figure 8.3: Overall results from the default settings: (a) throughput, (b) satisfied ratio, (c) latency, and (d) normalized accuracy.



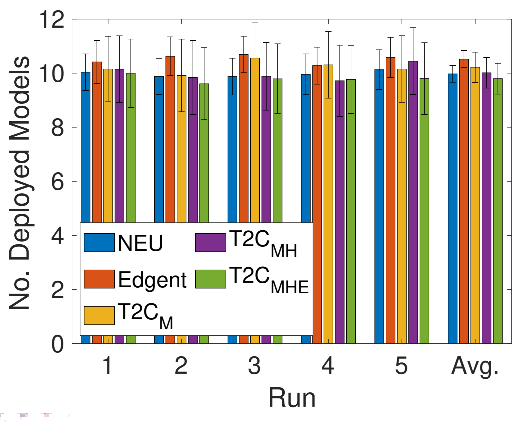
(a)



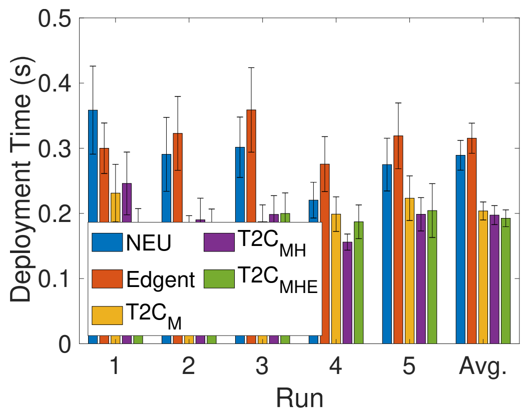
(b)



(c)



(d)



(e)

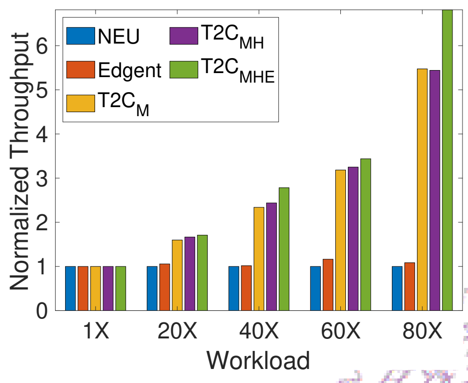
Figure 8.4: Overall overhead from the default settings: (a) control overhead, (b) data plane throughput, (c) CPU utilization, (d) number of deployed models, and (e) deployment time.

early exits can reduce the latency, and thus serve more user requests without violating their accuracy requirements.

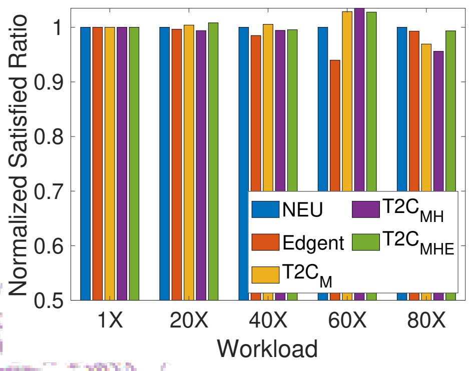
Our system incurs acceptable overhead. Fig. 8.4 reports the overall overhead of different algorithms under the default settings. Fig. 8.4(a) shows that our T2C algorithms cause comparable control overhead compared to the baseline algorithms. In fact, all algorithms introduce around 1 Mbps control overhead at the controller, which is manageable for modern LAN/WAN technologies. Fig. 8.4(b) reports that our T2C algorithms have similar data plane throughput compared to NEU and Edgent. In fact, multi-task without early exit incurs a bit higher throughput, but the difference is smaller than 0.3 Mbps. Fig. 8.4(c) reports the controller CPU utilization due to the deployment planning algorithm. We observe that T2C$_{\text{MHE}}$ consumes merely 37.86% and 42.86% of CPU utilization compared to NEU and Edgent. This can be attributed to the request aggregation of our T2C system, which reduces the number of deployment decisions needed to make. In addition, the aggregation also reduces the number of deployed models, as illustrated in Fig. 8.4(d) [2]. Fewer models further lead to an overhead reduction, including the average deployment time reported in Fig. 8.4(e) which shows that our algorithms reduce the container deployment time by ∼40%, compared to baselines.

Our system scales well under heavy workload. Figs. 8.5 and 8.6 reports the performance and overhead of algorithms under diverse workload. In these two figures, we normalized the performance results to NEU at the same workload for fair comparison. Fig 8.5(a) reveals that our T2C algorithms gain larger improvement when the workload is increased: T2C$_{\text{MH}}$ and T2C$_{\text{MHE}}$ achieve at most 5.4X and 6.8X of throughput compared to the baseline NEU. Fig. 8.5(b) reveals that the satisfied ratio has a higher difference at a higher workload. Our T2C$_{\text{MHE}}$ obtains the comparable satisfied ratio at the highest workload compared to the baseline NEU. Fig. 8.5(c) reports that T2C$_{\text{MHE}}$ has the lowest latency at different workloads. The latency of T2C$_{\text{MHE}}$ is as low as 14.54% of that of NEU at the heaviest workload. Fig. 8.5(d) demonstrates that the queuing time of our T2C algorithms is as low as 12.21% of that of NEU. This can be attributed to several factors: (i) using a model to serve more user requests cuts the waiting time for computing resources, (ii) applying hitchhiking allows some requests to be served right away, and (iii) both aggregation and hitchhiking help avoid duplicated deployment of the same model. We next report the computing, networking, and deployment overhead in Figs 8.6(a), 8.6(b), and 8.6(c), which show that T2C algorithms do not incur significantly more overhead, especially when the workload is heavier. More specifically, Fig. 8.6(c) reveals that the performance of deploy time improves as the workload gains higher, which achieves at most 59.8% lower than that of NEU at the heaviest workload.

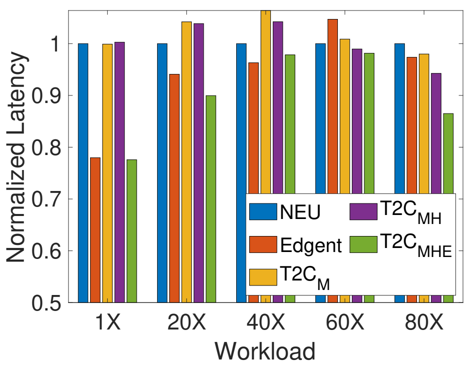
[2]Average number of deployed models is measured once every 10 s.



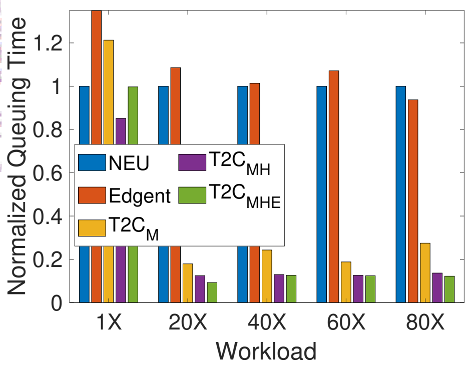
(a)



(b)

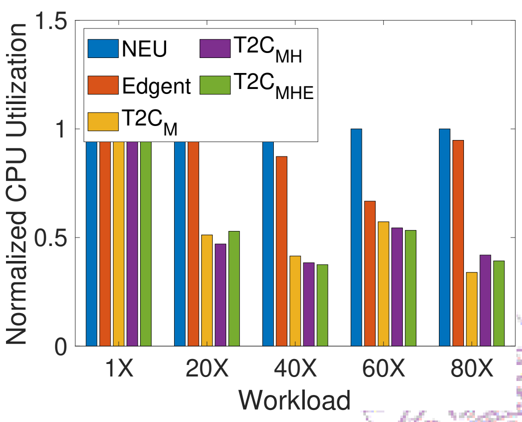


(c)

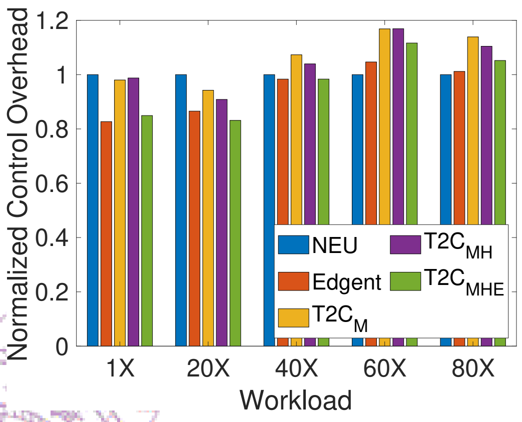


(d)

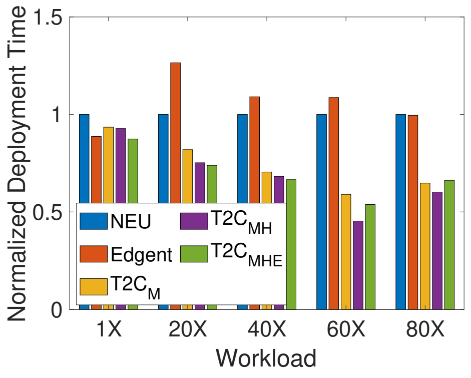
Figure 8.5: Overall results under different workloads normalized to NEU: (a) throughput, (b) satisfied ratio, (c) latency, and (d) queuing time.



(a)



(b)



(c)

Figure 8.6: Overall overhead under different workloads normalized to NEU: (a) CPU utilization, (b) control overhead, and (c) deploy time.

Summary. In the planning phase, we observe that (i) multi-task and hitchhiking improve throughput by at most 5.4X, (ii) early exits reduce latency without violating accuracy requirements, and (iii) our T2C system leads to a higher performance boost under a higher workload, e.g., 6.8X throughput and 12.21% queuing time. To sum up, T2C$_{\text{MHE}}$ significantly outperforms NEU, Edgent, and other algorithms. Hence, in the evaluation of the operation phase, we no longer consider T2C$_{\text{M}}$ and T2C$_{\text{MH}}$.

8.3 Operation Phase Results

Note that the dynamic reconfiguration kicks in only when the QoS levels drop. Under resource-scarce conditions, our goal is to retain the satisfied ratio as much as possible.

T2C$_{\text{MHER}}$ obtains the highest throughput and satisfied ratio. As dynamic reconfiguration algorithms are triggered more often when the workload is higher, we adopt a heavier workload here, with 80X as the default workload. Moreover, we throttle the network bandwidth by half at 20 s into each run to emulate network congestion. We plot the overall results with (T2C$_{\text{MHER}}$) and without (T2C$_{\text{MHE}}$) dynamic reconfiguration in Fig. 8.7. Figs. 8.7(a) and 8.7(b) show that T2C$_{\text{MHER}}$ achieves higher throughput and satisfied ratio. The higher satisfied ratio can be attributed to the network congestion that results in unexpected higher latency. With dynamic reconfiguration, T2C$_{\text{MHER}}$ decides to change the partition points to reduce the transmission latency: the network throughput in the data plane is reduced by 10.28% as shown in Fig. 8.7(d). Because of this, T2C$_{\text{MHER}}$ obtains 7.67% lower latency as shown in Fig. 8.7(c). Indeed, Fig. 8.7(b) shows that T2C$_{\text{MHER}}$ results in a 5.59% higher satisfied ratio, compared to T2C$_{\text{MHE}}$. With the benefits in terms of latency and satisfied ratio, the queuing time of T2C$_{\text{MHER}}$ is slightly increased by 2 seconds due to the overhead of reconfiguration, which is considered acceptable since we still achieve higher throughput.

T2C$_{\text{MHER}}$ incurs acceptable overhead even under heavier workload. Fig. 8.7(d) shows that T2C$_{\text{MHER}}$ incurs lower data plane throughput. We plot other overhead in Fig. 8.8. Fig. 8.8(a) reports the CPU utilization, which shows that T2C$_{\text{MHER}}$ has a CPU utilization of $\leq$ 5%, which is only a few ms for each invocation. Fig. 8.8(b) gives the control overhead, where In addition, T2C$_{\text{MHER}}$ consumes extra bandwidth due to message exchanges between the reconfiguration manager and agents. Nonetheless, the resulting control overhead is $<$ 1.5 Mbps, which is manageable in modern networks. Fig. 8.8(c) shows that the Deployment time is slightly increased since the reconfiguration restarts the containers, but the average difference is $<$ 20 ms. In summary, T2C$_{\text{MHER}}$ incurs acceptable overhead under the default workload. Next, we plot the normalized performance of T2C$_{\text{MHER}}$ w.r.t. T2C$_{\text{MHE}}$ in Figs. 8.9 and 8.10. Fig. 8.9(a) shows that T2C$_{\text{MHER}}$ always

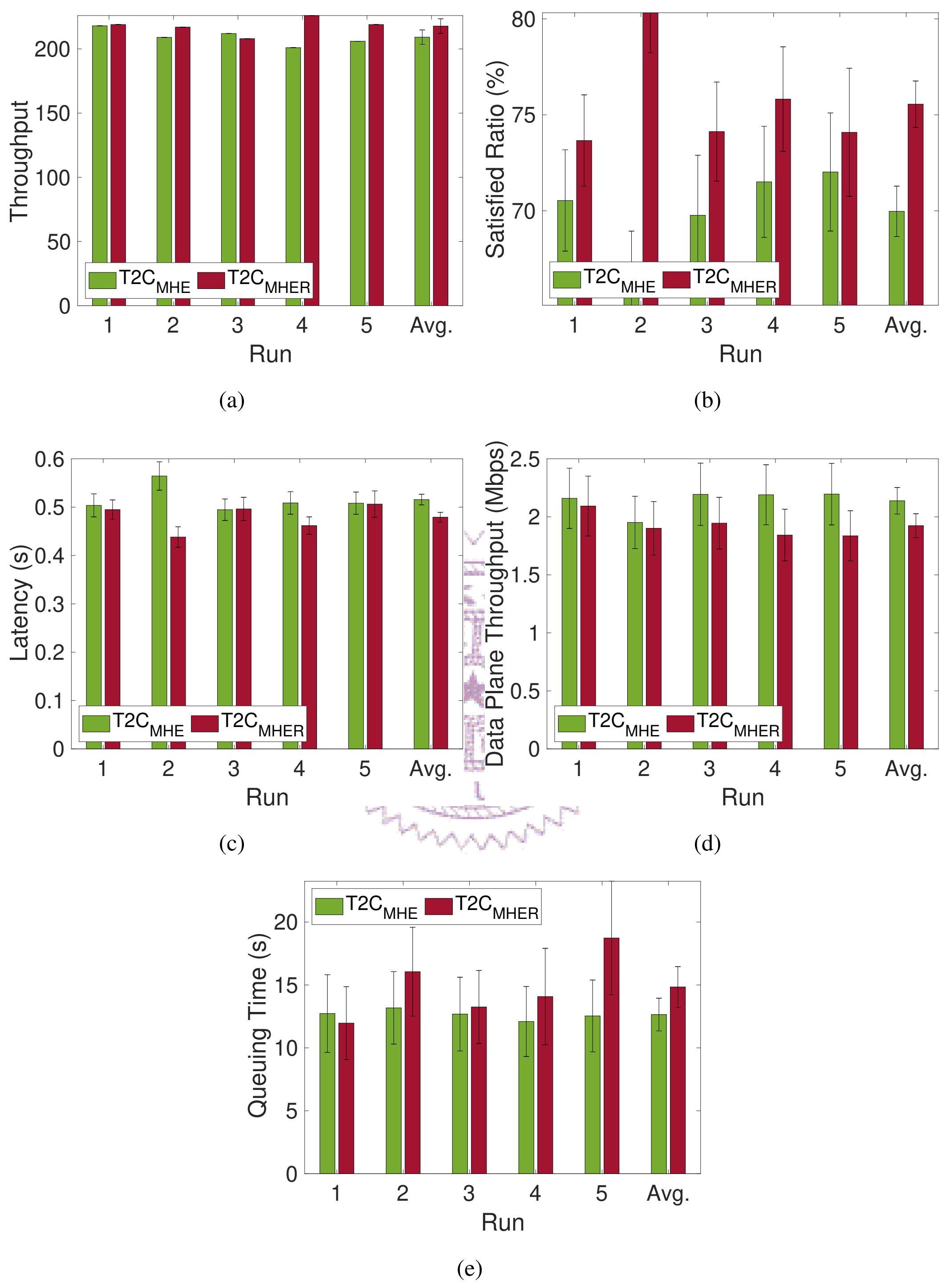
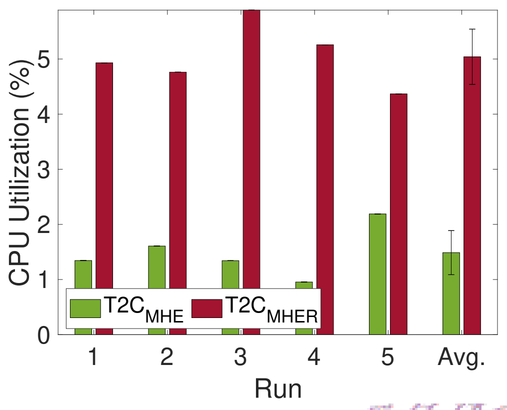
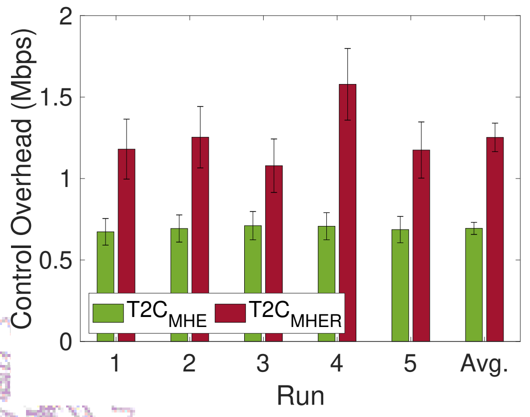


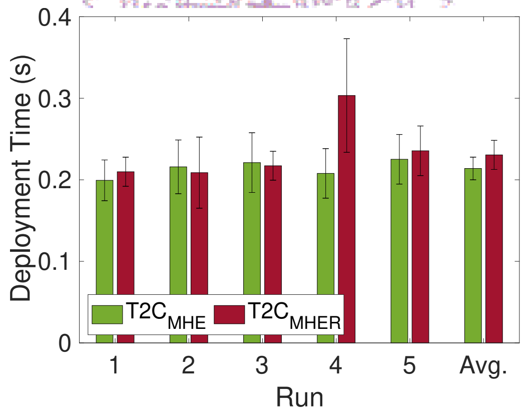
Figure 8.7: Overall results from T2C with and without dynamic reconfiguration: (a) throughput, (b) satisfied ratio, (c) latency, (d) data plane throughput, and (e) queuing time.



(a)

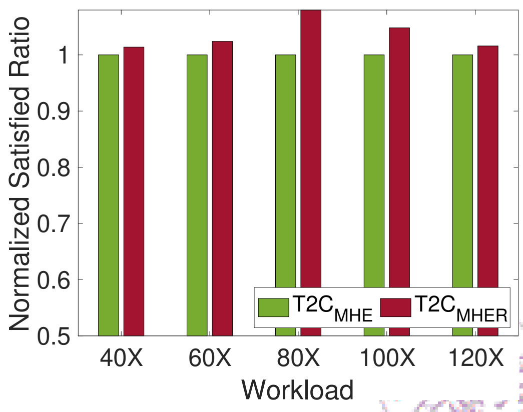


(b)

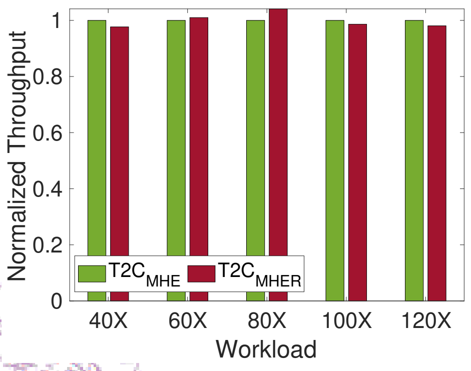


(c)

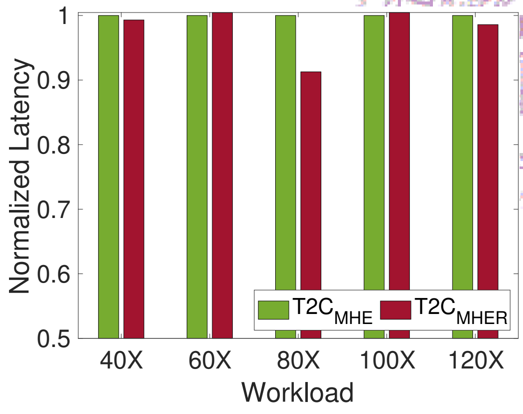
Figure 8.8: Overall overhead from T2C with and without dynamic reconfiguration: (a) CPU utilization, (b) control overhead, and (c) deploy time.



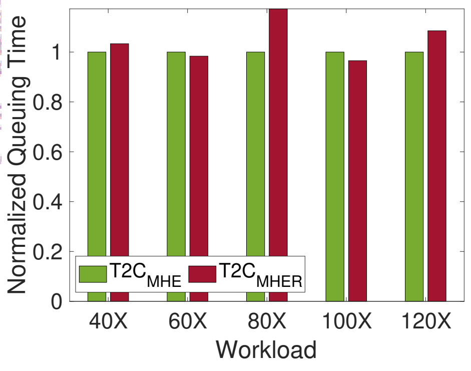
(a)



(b)

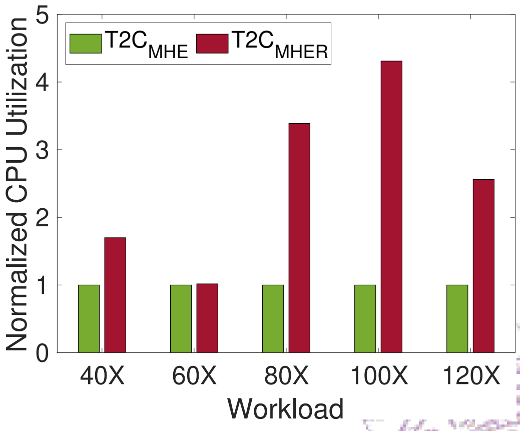


(c)

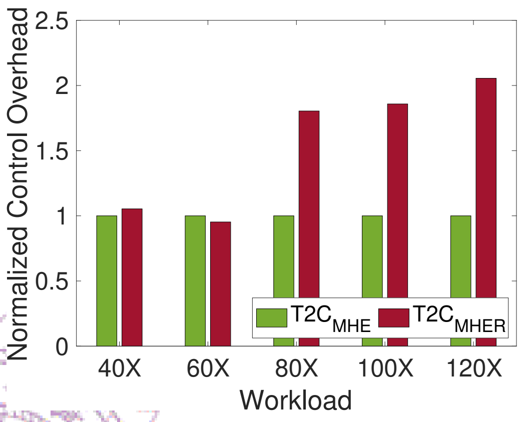


(d)

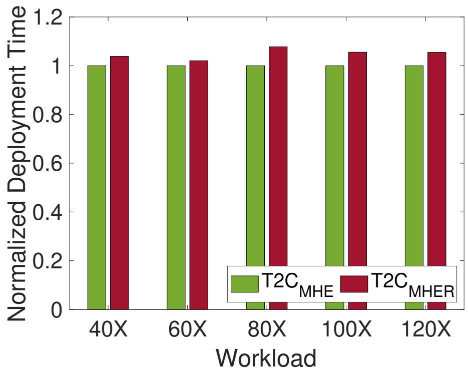
Figure 8.9: Overall results under different workloads normalized to $T2C_{MHE}$: (a) satisfied ratio, (b) throughput, (c) latency, and (d) queuing time.



(a)



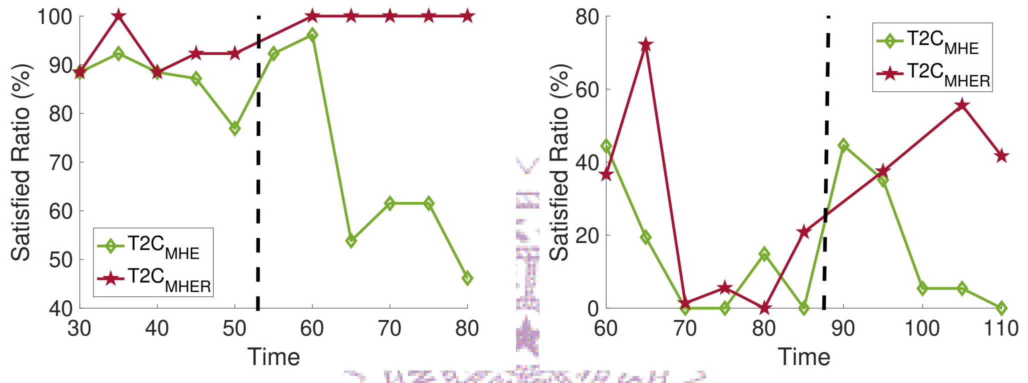
(b)



(c)

Figure 8.10: Overall overhead under different workloads normalized to T2C$_{\text{MHE}}$: (a) CPU utilization, (b) control overhead, and (c) deploy time.

achieves better satisfied ratio under different workloads. Figs. 8.9(b) and 8.9(c) show that T2C$_{\text{MHER}}$ obtains comparable throughput and latency to T2C$_{\text{MHE}}$. Because the purpose of reconfiguration is to increase the QoS at runtime, latency and throughput may not always benefit from it. Fig. 8.9(d) reveals that the queuing time of T2C$_{\text{MHER}}$ does not get worse as the workload increase. Fig. 8.10 presents the normalized CPU utilization time, control overhead, and deployment time. While up to 4.31X of CPU utilization and 2.05X of control overhead, a closer look indicates that the absolute CPU utilization and control overhead of T2C$_{\text{MHER}}$ are merely $< 6\%$ and < 1.5 Mbps, respectively. The deployment time of T2C$_{\text{MHER}}$ is only 1.07X, which has a < 17 ms difference compared to T2C$_{\text{MHE}}$. This shows that T2C$_{\text{MHER}}$ scales well to a heavier workload.



(a) (b)

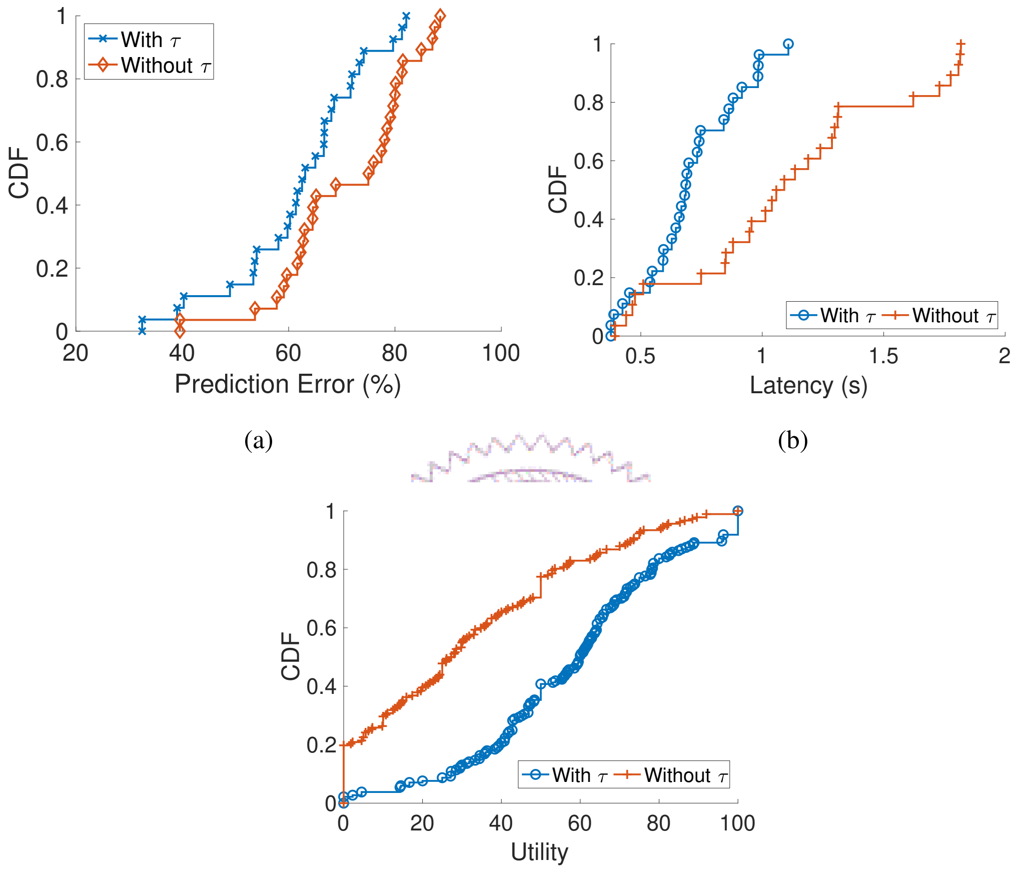
Figure 8.11: Satisfied ratio with and w/o reconfiguration of a sample model under: (a) light and (b) heavy workload.

Reconfiguration increases the satisfied ratio. To better observe the impact of dynamic reconfiguration, we zoom into the performance of a single model. Fig. 8.11 gives the satisfied ratio under different workloads. The vertical dashed lines annotate the time instance of a reconfiguration event. Fig. 8.11(a) is from a light workload with only 4 deployed models. We observe that the satisfied ratio is increased by 10% after the reconfiguration. Without reconfiguration, the satisfied ratio drops quickly, as high as a 54% performance gap between T2C$_{\text{MHE}}$ and T2C$_{\text{MHER}}$ is observed. Fig. 8.11(b) is from a heavy workload with 13 deployed models. We see the curves are not stable because the network environments are highly dynamic due to the behavior of other deployed models. Nevertheless, the satisfied ratio is increased by at most 35% after the reconfiguration. T2C$_{\text{MHER}}$ satisfies at most 50% more user requests compared to T2C$_{\text{MHE}}$.

Summary. We conclude that in the operation phase, our reconfiguration manager successfully detects the drop in QoS levels. By triggering the dynamic reconfiguration algorithm, we can increase the satisfied ratio under such brutal conditions. That is, T2C$_{\text{MHER}}$

achieves a higher satisfied ratio compared to T2C$_{\text{MHE}}$ without incurring excessive overhead even under the heavier workload.

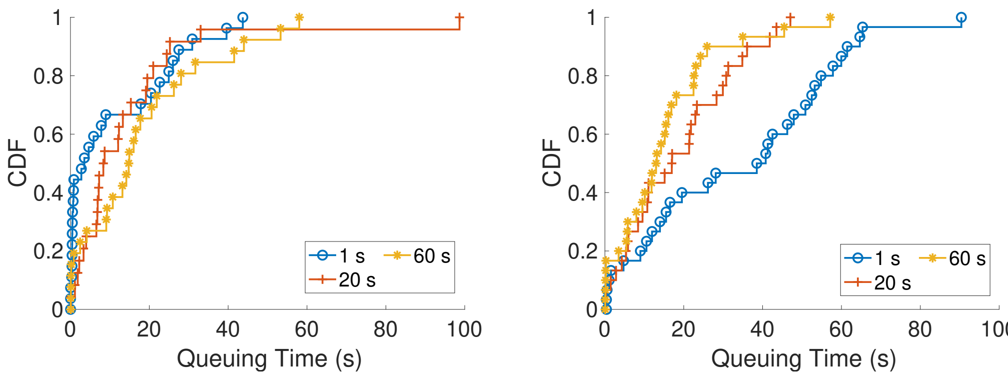
8.4 Implications of System Parameters



(a) (b) (c)

Figure 8.12: Impact of scaling factors under dynamic environment at 80X arrival rate: (a) prediction error, (b) satisfied ratio, and (c) latency.

Impact of scaling factors. We run a set of experiments in a dynamic network environment without scaling factors for comparison. We report the normalized latency of the runtime and expected prediction error in Fig. 8.12(a). The results show that the prediction error is reduced by 12.55%. The reduction of prediction error can further impact the performance of the system. We report the CDF of satisfied ratio and latency. Fig. 8.12(b) shows that with the scaling factor, latency is 1.56X lower than the without one, and hence the satisfied ratio is 26.27% higher, as reported in Fig. 8.12(c). Because scaling factors capture the runtime situations, i.e., the congestion on network, thus it determines which latency requirements can be satisfied more accurately, preventing the drop of satisfied ratio.



(a) (b)

Figure 8.13: Queuing time with different aggregation period under: (a) light and (b) heavy workload.

Impact of aggregation period. We run a set of experiments with different aggregation periods and report the CDF of queuing time under different workloads in Fig. 8.13. Obviously, the aggregation period directly impacts the queuing time since requests should wait in the queue before the next aggregation. In Fig. 8.13(a), the queuing time of 60 s aggregation period is 1.8X of the 1 s one. However, when the workload increase, the queuing time of 1 s becomes 2.3X of the 60 s one as reported in Fig. 8.13(b), due to the resource limitation. A higher aggregation period considers more requests when making the deployment plan and hence leverages the multi-task structure of our DNN. The number of duplicated models is thus reduced, saving more computing power for queuing or future requests.

8.5 Discussion and Recommendations

We observed that $T2C_{MHE}$ works well in the static environment with high throughput, high satisfied ratio, low latency, and low queuing time under different workloads. When the environment is dynamic, $T2C_{MHER}$ can reduce the latency and increase the satisfied ratio by reconfiguring the deployment at runtime. However, this comes with higher queuing time and overhead, such as CPU utilization and control overhead. Hence, we suggest using $T2C_{MHE}$ in a static or slightly dynamic environment and using $T2C_{MHER}$ when the environment, especially the network condition, is highly dynamic. Perhaps, an adaptation mechanism can be developed to automatically enable the reconfiguration feature based on historical data, which remains one of our future tasks.

Chapter 9

Conclusion

In this thesis, we proposed a multi-tenant system T2C for deploying DNNs in thing-to-cloud continuum. We proposed to leverage (i) multi-task, (ii) hitchhiking, (iii) early exit, and (iv) reconfiguration to serve as many user requests as possible with the awareness of runtime QoS levels. We proposed checking the deployed models to see if any of them can serve the coming requests. We then aggregated the queuing requests to their corresponding multi-task DNN models. We monitored the status of the deployed models and reconfigured the models with an QoS drop at runtime. We divided the DNN deployment decision-making problem into two phases: planning and operation. In the planning phase, we decided the deployment plan based on the current resource status, while in operation phase, we checked the status of deployed models and decided whether to conduct reconfiguration. We proposed several deployment planning and dynamic reconfiguration algorithms to generate deployment and reconfiguration plans for multi-task DNNs across IoT devices, edge servers, and cloud servers. We implemented our proposed T2C system in a prototype testbed with a controller, IoT device, edge server, and cloud server. We then extensively evaluated our proposed system and algorithms. The results show that:

- Multi-task and hitchhiking highly improve throughput.
- Early exits reduce the latency without violating accuracy requirements.
- Reconfiguration improves satisfied ratio at runtime.
- Our system scales well under heavy workload.

More specifically, T2C_{MHE} serves 6.8X more requests at low latency and T2C_{MHER} improves up to 35% satisfied ratio. Since T2C_{MHER} comes with higher overhead, we suggest using T2C_{MHE} in the static environments and use T2C_{MHER} if the environment is highly dynamic. Note that the prototype testbed is used for demonstration. Our system is not limited to Kubernetes-based environment. It can fit larger-scale systems as long as it provides the ability to send arguments to configure the deployment plan of DNNs.

Nowadays, IoT sensors are deployed everywhere for human intelligent life. Analyzing the generated sensor raw data with low latency is essential for many applications, such as accident detection. However, powerful computing devices are shared by multiple users, meaning a system to consume these requests and deploy IoT analytics is needed. Our system can serve DNN-based IoT analytics requests with high QoS and provide high throughput under resource constraints. Even in the resource-limited countryside, our system can work well, providing the users with lower queuing time and the best-effort QoS. Moreover, in crowded intersections in the cities, surveillance cameras are used for accident detection, traffic jam detection, traffic flow analysis, and traffic violation detection, to name a few. These requests are executed all day, or at least for those crowded hours, for the safety of road users and traffic control. With our system considering long-duration deployments, it is able to hold all these analysis requests and reconfigure the deployment under the dynamic environment. Hence, the QoS of these IoT analytics is improved with low response time. We believe our system can increase the life quality of humans.

9.1 Future Work

Last, we provide some possible extensions of our T2C system.

- **Consider the cost of reconfiguration.** Although reconfiguration improves the performance at runtime, restarting docker images results in service downtime. One way to avoid the service downtime is to terminate the old deployment after the new deployment is up [36]. However, this takes twice the resources during the switching, which may be impossible for a resource-limited environment. Hence, considering the cost of reconfiguration, i.e., the trade-off between service downtime and performance improvement can be included in the decision-making process of dynamic reconfiguration.
- **Include multiple computing devices in each infrastructure layer.** We only consider one computing device in one infrastructure layer in the thing-to-cloud continuum. However, it is possible to put various computing devices in a layer. While Kubernetes can label multiple computing devices with the same tag, we can keep track of the total amount of resources of each layer and let the Kubernetes scheduler to select the computing devices to use.
- **Predict workload and adaptively adjust aggregation period.** In this thesis, we use a certain aggregation period throughout the experiments. When waiting for the aggregation, requests are waiting in the request queue and may result in higher queuing time under a low workload. However, when the workload increases, aggregation helps to reduce the duplicated models and hence reduces the queuing time. We report the queuing time of different aggregation periods with light and heavy

workloads in Fig. 8.13, showing that a smaller aggregation period works better under a light workload and vice versa. Therefore, it is possible to predict the future workload and adjust the aggregation period with the consideration of the current resource status.

- **Assign priority to requests.** Each request may have different degrees of emergency levels. Assigning higher priorities to more urgent requests allows the system to deploy corresponding models sooner. Moreover, it is possible to temporarily increase the priority of those requests that have been waiting in the queue. By doing so, we may avoid request starvation.

Bibliography

[1] 2022 K3s project authors. Lightweight Kubernetes, 2022. https://k3s.io/.

[2] 2022 Kubernetes authors. Kubernetes, 2022. http://kubernetes.io/.

[3] 2022 MQTT.org. MQTT, 2022. https://mqtt.org/.

[4] 2022 ZeroMQ authors. ZeroMQ, 2022. https://zeromq.org/.

[5] AAEON Technology Inc. Upboard, 2022. https://up-board.org/up/specifications/.

[6] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[7] K. Ashton et al. That 'Internet of Things' thing. *RFID Journal*, 22(7):97–114, 2009.

[8] L. Atzori, A. Iera, and G. Morabito. The Internet of Things: A survey. *Computer Networks*, 54(15):2787–2805, May 2010.

[9] D. Bernstein. Containers and cloud: From LXC to docker to Kubernetes. *IEEE Cloud Computing*, 1(3):81–84, September 2014.

[10] L. Bittencourt, R. Immich, R. Sakellariou, N. Fonseca, E. Madeira, M. Curado, L. Villas, L. DaSilva, C. Lee, and O. Rana. The Internet of Things, fog and cloud continuum: Integration and challenges. *Elservier Internet of Things*, 3:134–155, October 2018.

[11] A. Bochkovskiy, C. Wang, and H. Liao. YOLOv4: Optimal speed and accuracy of object detection. *arXiv preprint arXiv:2004.10934*, 2020.

[12] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli. Fog computing and its role in the internet of things. In *Proc. of the International Workshop on Mobile Cloud Computing (MCC'12)*, pages 13–16, Helsinki, Finland, August 2012.

[13] V. Cardellini, V. Grassi, F. Presti, and M. Nardelli. Optimal operator placement for distributed stream processing applications. In *Proc. of the ACM International Conference on Distributed and Event-based Systems (DEBS'16)*, pages 69–80, Irvine, California, June 2016.

[14] V. Cardellini, V. Grassi, L. Presti, and M. Nardelli. On QoS-aware scheduling of data stream applications over fog computing infrastructures. In *Proc. of IEEE Symposium on Computers and Communication (ISCC'15)*, pages 271–276, Larnaca, Cyprus, July 2015.

[15] R. Caruana. Multitask learning. *Springer Machine Learning*, 28(1):41–75, July 1997.

[16] M. Chao, R. Stoleru, L. Jin, S. Yao, M. Maurice, and R. Blalock. AMVP: Adaptive CNN-based multitask video processing on mobile stream processing platforms. In *Proc. of IEEE/ACM Symposium on Edge Computing (SEC'20)*, pages 96–109, Virtual, November 2020.

[17] L. Deng, J. Li, J. Huang, K. Yao, D. Yu, F. Seide, M. Seltzer, G. Zweig, X. He, J. Williams, et al. Recent advances in deep learning for speech research at Microsoft. In *Proc. of IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP'13)*, pages 8604–8608, Vancouver, Canada, May 2013.

[18] A. Esteva, B. Kuprel, R. Novoa, J. Ko, S. Swetter, H. Blau, and S. Thrun. Dermatologist-level classification of skin cancer with deep neural networks. *Nature*, 542(7639):115–118, Januray 2017.

[19] B. Fang, X. Zeng, and M. Zhang. NestDNN: Resource-aware multi-tenant on-device deep learning for continuous mobile vision. In *Proc. of the Annual International Conference on Mobile Computing and Networking (MobiCom'18)*, pages 115–127, New Delhi, India, October 2018.

[20] T. Goethals, F. Turck, and B. Volckaert. FLEDGE: Kubernetes compatible container orchestration on low-resource edge devices. In *Proc. of International Conference on Internet of Vehicles (IOV'19)*, pages 174–189, Kaohsiung, Taiwan, November 2019.

[21] Y. Han, S. Shen, X. Wang, S. Wang, and V. Leung. Tailored learning-based scheduling for Kubernetes-oriented edge-cloud system. In *Proc. of IEEE Conference on*

Computer Communications (INFOCOM'21), pages 1–10, Vancouver, Canada, May 2021.

[22] H. Hong, P. Tsai, A. Cheng, M. S. Uddin, N. Venkatasubramanian, and C. Hsu. Supporting internet-of-things analytics in a fog computing platform. In *Proc. of IEEE International Conference on Cloud Computing Technology and Science (CloudCom'17)*, pages 138–145, Hong Kong, China, December 2017.

[23] H. Hong, P. Tsai, and C. Hsu. Dynamic module deployment in a fog computing platform. In *Proc. of Asia-Pacific Network Operations and Management Symposium (APNOMS'16)*, pages 1–6, Kanazawa, Japan, October 2016.

[24] J. Howarth. 80+ amazing IoT statistics (2022-2030), 2022. https://explodingtopics.com/blog/iot-stats.

[25] K. Hsu, K. Bhardwaj, and A. Gavrilovska. Couper: DNN model slicing for visual analytics containers at the edge. In *Proc. of the ACM/IEEE Symposium on Edge Computing (SEC'19)*, pages 179–194, Arlington, Virginia, November 2019.

[26] C. Hu, W. Bao, D. Wang, and F. Liu. Dynamic adaptive DNN surgery for inference acceleration on the edge. In *Proc. of IEEE International Conference on Computer Communications (INFOCOM'19)*, pages 1423–1431, Paris, France, April 2019.

[27] A. Jiang, D. Wong, C. Canel, L. Tang, I. Misra, M. Kaminsky, M. Kozuch, P. Pillai, D. Andersen, and G. Ganger. Mainstream: Dynamic stem-sharing for multi-tenant video processing. In *Proc. of Internatioal USENIX Annual Technical Conference (ATC'18)*, pages 29–42, Boston, Massachusetts, July 2018.

[28] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang. Neurosurgeon: Collaborative intelligence between the cloud and mobile edge. *ACM SIGARCH Computer Architecture News*, 45(1):615–629, April 2017.

[29] A. Krizhevsky, I. Sutskever, and G. Hinton. Imagenet classification with deep convolutional neural networks. *Communications of the ACM*, 60(6):84–90, June 2017.

[30] G. Lakshmanan, Y. Li, and R. Strom. Placement strategies for Internet-scale data stream systems. *IEEE Internet Computing*, 12(6):50–60, November 2008.

[31] S. Laskaridis, S. Venieris, M. Almeida, I. Leontiadis, and N. Lane. Spinn: synergistic progressive inference of neural networks over device and cloud. In *Proc. of the Annual International Conference on Mobile Computing and Networking (MobiCom'20)*, pages 1–15, London, UK, April 2020.

[32] Y. Le Cun, L. Jackel, B. Boser, J. Denker, H. Graf, I. Guyon, D. Henderson, R. Howard, and W. Hubbard. Handwritten digit recognition: Applications of neural network chips and automatic learning. *IEEE Communications Magazine*, 27(11):41–46, November 1989.

[33] E. Li, L. Zeng, Z. Zhou, and X. Chen. Edge AI: On-demand accelerating deep neural network inference via edge computing. *IEEE Transactions on Wireless Communications*, 19(1):447–457, October 2019.

[34] H. Li, C. Hu, J. Jiang, Z. Wang, Y. Wen, and W. Zhu. JALAD: Joint accuracy-and latency-aware deep structure decoupling for edge-cloud execution. In *Proc. of IEEE International Conference on Parallel and Distributed Systems (ICPADS'18)*, pages 671–678, Singapore, Singapore, December 2018.

[35] Y. Ma, J. Wu, and C. Long. DasNet: Dynamic adaptive structure for accelerating multi-task convolutional neural network. In *Proc. of International Conference on Neural Information Processing (ICONIP'19)*, pages 138–150, Sydney, Australia, December 2019.

[36] A. Majeed, P. Kilpatrick, I. Spence, and B. Varghese. NEUKONFIG: Reducing edge service downtime when repartitioning DNNs. In *Proc. of IEEE International Conference on Cloud Engineering (IC2E'21)*, pages 118–125, San Francisco, California, October 2021.

[37] J. Mao, X. Chen, K. Nixon, C. Krieger, and Y. Chen. MoDNN: Local distributed mobile computing system for deep neural network. In *Proc. of IEEE Design, Automation & Test in Europe Conference & Exhibition (DATE'17)*, pages 1396–1401, Lausanne, Switzerland, March 2017.

[38] A. Mathur, N. Lane, S. Bhattacharya, A. Boran, C. Forlivesi, and F. Kawsar. Deepeye: Resource efficient local execution of multiple deep vision models using wearable commodity hardware. In *Proc. of the Annual International Conference on Mobile Systems, Applications, and Services (MobiSys'17)*, pages 68–81, Niagara Falls, New York, June 2017.

[39] F. McNamee, S. Dustdar, P. Kilpatrick, W. Shi, I. Spence, and B. Varghese. The case for adaptive deep neural networks in edge computing. In *Proc. of IEEE International Conference on Cloud Computing (CLOUD'21)*, pages 43–52, Chicago, Illinois, September 2021.

[40] P. Mell, T. Grance, et al. The NIST definition of cloud computing. September 2011. https://csrc.nist.gov/publications/detail/sp/800-145/final.

[41] D. Merkel et al. Docker: lightweight Linux containers for consistent development and deployment. *Linux Journal*, 239, March 2014.

[42] T. Mohammed, C. Joe-Wong, R. Babbar, and M. Francesco. Distributed inference acceleration with adaptive DNN partitioning and offloading. In *Proc. of IEEE Conference on Computer Communications (INFOCOM'20)*, pages 854–863, Toronto, Canada, July 2020.

[43] M. Nardelli, V. Cardellini, V. Grassi, and F. Presti. Efficient operator placement for distributed data stream processing applications. *IEEE Transactions on Parallel and Distributed Systems*, 30(8):1753–1767, January 2019.

[44] PABLOMONLEON. 311 service requests NYC, 2022. https://www.kaggle.com/datasets/pablomonleon/311-service-requests-nyc/.

[45] S. Pan and Q. Yang. A survey on transfer learning. *IEEE Transactions on Knowledge and Data Engineering*, 22(10):1345–1359, October 2009.

[46] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in Neural Information Processing Systems*, 32, 2019.

[47] S. Ruder. An overview of multi-task learning in deep neural networks. *arXiv preprint arXiv:1706.05098*, June 2017.

[48] M. Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, January 2017.

[49] Sergey Bondarev. Top 4 enterprise IoT trends of 2022 and beyond, 2022. https://www.voltactivedata.com/blog/2022/06/top-4-enterprise-iot-trends-of-2022/.

[50] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, June 2016.

[51] W. Shi and S. Dustdar. The promise of edge computing. *Computer*, 49(5):78–81, May 2016.

[52] S. Singh, A. Kumar, H. Darbari, L. Singh, A. Rastogi, and S. Jain. Machine translation using deep learning: An overview. In *Proc. of International Conference on Computer, Communications and Electronics (Comptelix'17)*, pages 162–167, Jaipur, India, July 2017.

[53] R. Socher, B. Huval, B. Bath, C. Manning, and A. Ng. Convolutional-recursive deep learning for 3D object classification. *Advances in Neural Information Processing Systems*, 25, 2012.

[54] V. Sze, Y. Chen, T. Yang, and J. Emer. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE*, 105(12):2295–2329, November 2017.

[55] S. Teerapittayanon, B. McDanel, and H. Kung. Branchynet: Fast inference via early exiting from deep neural networks. In *Proc. of IEEE International Conference on Pattern Recognition (ICPR'16)*, pages 2464–2469, Cancun, Mexico, December 2016.

[56] S. Teerapittayanon, B. McDanel, and H. Kung. Distributed deep neural networks over the cloud, the edge and end devices. In *Proc. of IEEE International Conference on Distributed Computing Systems (ICDCS'17)*, pages 328–339, Atlanta, Georgia, July 2017.

[57] L. Toka, G. Dobreff, B. Fodor, and B. Sonkoly. Machine learning-based scaling management for kubernetes edge clusters. *IEEE Transactions on Network and Service Management*, 18(1):958–972, January 2021.

[58] D. Torres, C. Martín, B. Rubio, and M. Díaz. An open source framework based on Kafka-ML for distributed DNN inference over the cloud-to-things continuum. *Journal of Systems Architecture*, 118:102214, September 2021.

[59] P. Tsai, H. Hong, A. Cheng, and C. Hsu. Distributed analytics in fog computing platforms using Tensorflow and Kubernetes. In *Proc. of Asia-Pacific Network Operations and Management Symposium (APNOMS'17)*, pages 145–150, Seoul, Korea, September 2017.

[60] S. Viet and C. Bao. Effective deep multi-source multi-task learning frameworks for smile detection, emotion recognition and gender classification. *Informatica*, 42(3), September 2018.

[61] L. Yang, J. Cao, S. Tang, D. Han, and N. Suri. Run time application repartitioning in dynamic mobile cloud environments. *IEEE Transactions on Cloud Computing*, 4(3):336–348, September 2014.

[62] W. Yin, K. Kann, M. Yu, and H. Schütze. Comparative study of CNN and RNN for natural language processing. *arXiv preprint arXiv:1702.01923*, 2017.

[63] Z. Zhao, K. Barijough, and A. Gerstlauer. DeepThings: Distributed adaptive deep learning inference on resource-constrained IoT edge clusters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2348–2359, November 2018.