© 2019 Yifan Hao

DEEP INTELLIGENCE AS A SERVICE: A REAL-TIME SCHEDULING PERSPECTIVE

BY

YIFAN HAO

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Master of Science in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2019

Urbana, Illinois

Adviser:

Professor Tarek Abdelzaher

ABSTRACT

This thesis presents a new type of cloud service, called *deep intelligence as a service*, that is expected to become increasingly common in the near future to support emerging "smart" embedded applications. This work offers a real-time scheduling model motivated by the special needs of embedded applications that use this service. A simple run-time scheduler is proposed for the server and prove an approximation bound in terms of application-perceived service utility. The service is implemented on representative device hardware and tested with a machine vision application illustrating the advantages of our scheme. The work is motivated by the proliferation of increasingly ubiquitous but resource-constrained embedded devices (often referred to as the *Internet of Things* – *IoT* – devices) and the growing desire to endow them with advanced interaction capabilities, such as voice recognition or machine vision. The trend suggests that machine intelligence will be increasingly offloaded to cloud or edge services that will offer advanced capabilities to the otherwise simple devices. New services will feature farms of complex trainable (or pre-trained) classifiers that client-side applications can send data to in order to gain certain types of advanced functionality, such as face recognition, voice command recognition, gesture recognition, or other. These new services will revolutionize human interaction with their physical environment, but may impose interesting real-time scheduling challenges in order to maintain responsiveness while maximizing service quality. This work includes challenges, designs for an efficient real-time scheduling algorithm for the new machine intelligence service, and evaluation on an implemented prototype.

ACKNOWLEDGMENTS

I would like to thank my advisor Professor Tarek Abdelzaher for his guidance over the past years. My thesis is mainly shaped by his knowledge, and his ideas have inspired me when I was tackling various problems. He has dedicated his efforts to help all of his students to do better research, and I appreciate that a lot.

I would also like to thank Dr. Shuochao Yao for his brilliant mind and help on my thesis work. Working with him has been an illuminating experience for me. The theory part of this work will not be complete without his support. I would like to thank everyone in the research group. Being a comrade and friend with you has been the source of my energy. I would like to thank Jongdeog Lee for helping me get started with my graduate research. May you have a wonderful journey back to Korea. I would like to thank Dr.Prasanna Giridhar, who assigned me a tiny Apollo demo project and I was able to showcase my coding skills to Tarek and prove that I'm a qualified graduate student worth paying 2000 bucks a month. I would like to thank Huajie Shao for his dedications to sharing rumors and made my stay in office a lot happier than it should be. I would like to thank Dongxin Liu and Shengzhong Liu. Being SJTU and UIUC alumni means we share loads of memories. Hope our path will join again in the future. I would like to thank Yiran Zhao, for being my CS424 real-time systems course TA. What I have learned from that course forms the basis of my master thesis. I would like to thank Tianshi Wang, who took over my legacy SocialCube framework and led the development of it. I was able to focus on my thesis research thanks to you. I want to thank Jinyang Li, for chatting with me for random things, working with me on my last course in my life and inviting me to join Illini Danceport (sadly I missed it). I would also want to thank Tai-sheng Cheng, who played surviv.io, Helldiver, Overcook, Basketball, Bowling and other games with me. Skipping classes and snowboarding in Granite Peak for days was truly an unforgettable memory in my life. I promise I'll grab a beer with you on my next trip to Seattle.

I would like to show my appreciation to the managers of Rainbow Garden, Lao Sze Chuan, Burger King, Gyu-Kaku and Szechuan China. You have provided wonderful venues for our "sporadic" group lunches and dinners. I would like to thank Nvidia for having me work as an intern. Without the internship salary, my graduate school life would be miserable and I would not be able to afford Netflix for my leisure time. I would also like to thank VMware for offering me a full-time job I like, without which I would be living in the terror of being jobless and being a Ph.D. student haunting in the greater land of Urbana-Champaign.

I would like to show appreciation to everyone I knew in my graduate school journey. Hanging out with you has endowed me with great comfort. I would like to thank Pearl for sharing the last part of the journey with me. You have been an amazing listener and supporter to me. Let's raise a cat in the Bay Area. Lastly, I would like to thank my parents for their unconditional care for their irresponsible son. Without their support, I will never achieve this much.

TABLE OF CONTENTS

CHAPTER 1 PRELIMINARY . . . 1
1.1 Real Time Systems . . . 1
1.2 Real Time Scheduling Algorithms . . . 1
1.3 Deep Neural Networks . . . 3
1.4 Real Time Schedulers in Linux . . . 4

CHAPTER 2 INTRODUCTION . . . 7
2.1 Challenges . . . 7
2.2 Overview . . . 8
2.3 Thesis Organization . . . 9

CHAPTER 3 MODEL . . . 10
3.1 Algorithms . . . 10
3.2 Framework Architecture . . . 15

CHAPTER 4 IMPLEMENTATION . . . 20

CHAPTER 5 EVALUATION . . . 23
5.1 Confidence Calibration & Dynamic Confidence Updates . . . 23
5.2 Scheduler Overhead . . . 24
5.3 Transmission vs. Computation Bottleneck . . . 26
5.4 Deep Intelligence as a Service . . . 26

CHAPTER 6 RELATED WORKS . . . 31

CHAPTER 7 CONCLUSION . . . 33

REFERENCES . . . 34

CHAPTER 1: PRELIMINARY

1.1 REAL TIME SYSTEMS

The correctness of a real time system depends on not only the functional results of the algorithm, but also the timing as well. Compared with traditional computer systems, which mainly focus on throughput, fairness and mean response time, designing a real time system is inherently more difficult because of the timing constraints.

A plethora of real time systems or applications in have emerged real life, ranging from multimedia contents that need to display frame by frame at particular frequency, to automated driving systems that need to act quickly with various environmental signals to guarantee safety. Based on the consequence of deadline violation, real-time systems may be further divided into two categories:

- *Hard Real Time Systems:* They have strict requirements on the timing of the tasks. Violation of deadline may result in system failures.
- *Soft Real Time Systems:* The timing requirements of soft real time system are relaxed. Violation of deadline may cause degradation of quality of results, but will not cause system failures.

The difficulty of designing a real time system lies in the interplay of multiple real time tasks that have different timing constraints. How much computation time should be allocated to a specific resource and when should the system schedule the task? If the system can not satisfy all the timing constraints, can some of the tasks yield their quota to other more serious tasks?

1.2 REAL TIME SCHEDULING ALGORITHMS

Fortunately, researchers have proposed and implemented various real time scheduling algorithms to address the problem. This section contains a brief survey for some classic scheduling algorithms, which forms the basis for evaluation and comparison. Before presenting a specific algorithm, abstractions for studying real time systems need to be introduced. Real time tasks are typically abstracted as periodic tasks with following parameters:

- Computation Time C: Maximum time needed to process its workload.
- Period P: Time interval between consecutive activations of the task. For a simplistic model, the deadline of a task can be calculated as the arrival time plus the period.

Therefore, the utilization rate for a given task i can be written as:

$$\eta_i = \frac{C_i}{P_i} \tag{1.1}$$

1.2.1 First In First Out

First in first out (FIFO) scheduling algorithm is self-explanatory: each task will run to completion once it is fetched from some task queues. It can be best modeled by a queue data structure. Strictly speaking, this algorithm is not a real time algorithm as it does not consider the deadline of a task, and the decision is made purely based on when (start time) does the task arrive at the system.

1.2.2 Round Robin

Round robin scheduling assigns time slice to each round of a task, and each task is scheduled to run in order. Each task either consumes its time slice or yields to other tasks, then goes back to task queue and wait for next round. This scheduling algorithm forms the basis for modern operating systems scheduling algorithm. For example, Linux kernel implements completely fair queuing, which is a variant of round robin by categorizing processes into different priority levels. Within each level, the kernel performs round robin scheduling. This algorithm suffers from the same problem as FIFO algorithm, which does not take the deadline of tasks into account.

1.2.3 Earliest Deadline First

With earliest deadline first (EDF) scheduler, priority of tasks are based on the absolute deadlines. Whichever task that has the nearest deadline will be assigned the highest priority. To check whether a stream of real time tasks can be scheduled under an EDF scheduler, it only needs to guarantee the sum of the utilization rate for all the tasks is less than 1:

$$\sum_i \frac{C_i}{P_i} < 1 \tag{1.2}$$

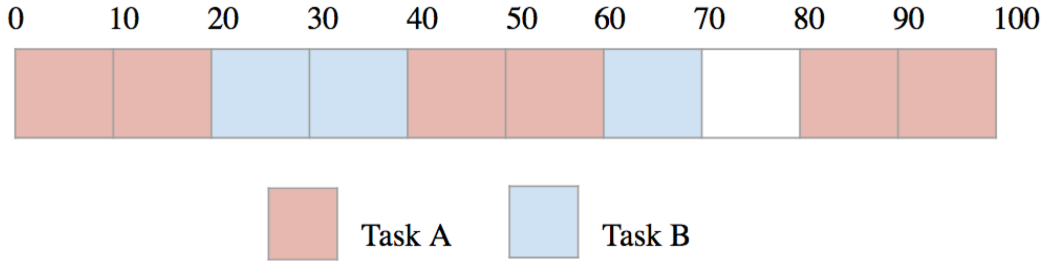


Figure 1.1: In illustration of Rate Monotonic Scheduling, with P(A) <P(B)

1.2.4 Rate Monotonic

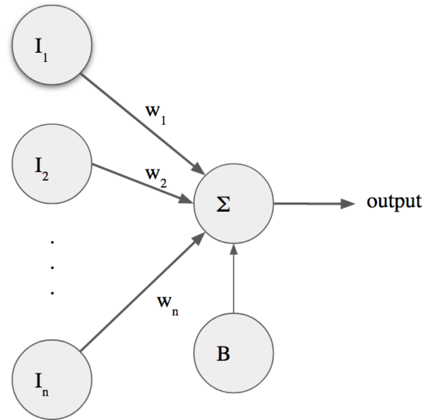
Under rate monotonic scheduler (RMS), priority of tasks are based on the difference between deadline and start time, namely the period P of the task. Whichever task that possesses the least period will be assigned the highest priority. Note that a task with nearer deadline might be assigned a lower priority simply because its period is large. To guarantee tasks are schedulable under RMS, the following inequality needs to be satisfied [1]:

$$\sum_i \frac{C_i}{P_i} < ln2 \tag{1.3}$$

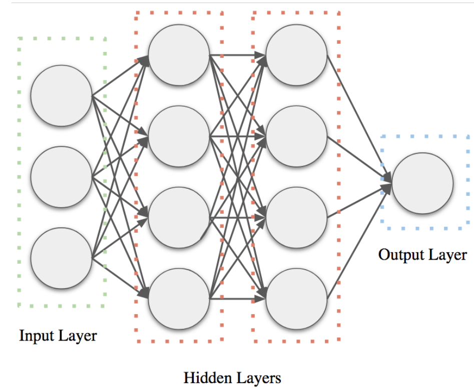
Rate monotonic scheduling is guaranteed to be optimal. If tasks cannot be scheduled with rate monotonic scheduler, the tasks cannot be properly scheduled with any other static priority schedulers.

1.3 DEEP NEURAL NETWORKS

Deep learning is a branch under machine learning methods. Deep learning can be architected as deep neural networks (DNN), convolutional neural networks (CNN), recurrent neural networks (RNN), etc. Deep neural networks are constructed based on a layer by layer data structure, which itself is constructed from multiple conceptual building blocks called neurons. A DNN contains at least an input layer and an output layer, with some hidden layers between them. In essence, a layer calculates a thresholded weighted sum of input set, and its output is fed to next layer or treated as final answer. The weights in each layer are learned in an offline training phase with some training data.



(a) Structure of a neuron



(b) Layer structure of a Deep Neural Network

Figure 1.2: Overview of Deep Neural Network

1.4 REAL TIME SCHEDULERS IN LINUX

Since the body of this work is based on Linux operating systems (OS), a short introduction to Linux scheduler is necessary to complete this work. Linux is a general purpose operating system (GPOS), whereas a real time operating system (RTOS) naturally serves real time tasks and cope with their deadlines. A normal process in Linux uses non-real time scheduling policy. An administrator of a Linux OS can choose to use a real time scheduling policy for a process when deemed necessary. Therefore, Linux distros with real time scheduling policies can be used as a platform for evaluating real time algorithms.

A Linux OS contains the following scheduling policies:

1.4.1 SCHED_IDLE

SCHED_IDLE policy is intended for processes that has the lowest priority. Each processor possesses an idle task created by Linux OS, which is scheduled on the processor when no other processes can be run. The idle process puts the processor to sleep to save energy. The idle task for each processor is scheduled under SCHED_IDLE policy.

1.4.2 SCHED_BATCH

SCHED_BATCH policy is similar to SCHED_NORMAL (explained below). However, SCHED_BATCH is assumed to be assigned to CPU computation bound processes, and thus will be given a slight penalty when making scheduling decisions. It inherently has lower

priority than SCHED_NORMAL policy.

1.4.3 SCHED_NORMAL

SCHED_NORMAL is the de-facto scheduling policy for almost all the processes in a Linux system. It is a non-real time scheduling policy. This scheudling policy is used when the processes do not require special real time mechanisms.

1.4.4 SCHED_FIFO

SCHED_FIFO is one of the real time scheduling policies implemented in Linux kernel. It is an implementation of FIFO algorithm. When a SCHED_FIFO task becomes runnable, it will preempt other SCHED_NORMAL, SCHED_BATCH, SCHED_IDLE processes in the system. A SCHED_FIFO task will only be preempted when either it is blocked by some system services (eg. system calls), it is preempted by a higher priority process, or it explicitly yields to other processes.

1.4.5 SCHED_RR

SCHED_RR is an enhancement to SCHED_FIFO scheduling policy with an extra time quota allocated to each process. It is an implementation of round robin algorithm in Linux kernel. However, the time slices for each SCHED_RR task is not controlled by the administrator, and modifying the mechanism for calculating time quantum will require changes to kernel. The framework presented in this work does not use this scheduling policy due to this reason.

1.4.6 SCHED_DEADLINE

SCHED_DEADLINE is provided as another real time scheduling policy since Linux 3.14. Its implementation combines earliest deadline first algorithm with constant bandwidth server (CBS) [2]. Since Linux 4.13, SCHED_DEADLINE uses Greedy Reclamation of Unused Bandwidth (GRUB) algorithm instead of CBS [3]. When schedule a process with this policy, the administrator needs to provide the system call with the period as well as worst case execution time (WCET). Internally the kernel will compute the utilization rate of the process, as well as the total utilization rate for all the processes running with SCHED_DEADLINE. If the total utilization rate exceeds the bandwidth provided by the system, the system call

will fail and the process will not be scheduled under SCHED_DEADLINE policy. Due to this constraint, the framework presented by this work does not use this scheduling class for implementation.

CHAPTER 2: INTRODUCTION

2.1 CHALLENGES

This work envisions a novel type of IoT services motivated by the proliferation of internetworked embedded sensing devices (the "things") and the desire to endow them with capabilities to perform timely and intelligent interactions with their environment. Examples include speech recognition, vision, and gesture understanding. The disparity between the resource-constrained nature of such devices and the computational needs of the aforementioned interactions suggest that data processing will be increasingly offloaded to external servers. Today, precursors of such offloading include speech recognition for home controllers (e.g., Amazon Echo) and language translation for mobile phones, both done in the cloud. Soft time-constraints arise from the need to ensure an appropriate interaction response time. Future services will conceivably support multiple classes of service that differ in their server-side latency guarantees. Note that, with the increasing popularity of *edge computing*, the external server will likely move closer to the client. A business, such as a shopping mall, for example, might host it's own edge servers to satisfy the needs of its local embedded devices (such as all the surveillance cameras). Hence, in the rest of this work when referring to "cloud", it means either cloud or edge.

Cloud services will offer farms of machine learning algorithms, such as classifiers or predictors. For simplicity, they are referred as *classifiers*. These algorithms can be trained with users' data (or may come pre-trained) to do a myriad of common recognition tasks based on visual inputs, speech, or gestures. Such services will endow mobile and embedded devices with human-like capabilities, thereby revolutionizing our interactions with the physical world. Recently, deep learning has emerged as the state-of-the-art computational intelligence solution for a large spectrum of IoT applications [4]. Besides breakthroughs in processing images and speech using deep learning techniques [5,6], specific neural network structures have been designed to fuse multiple sensing modularities and extract temporal relationships [7]. The increasing number of studies on applying deep learning in the area of cyber-physical systems (CPS) and IoT [7–10], motivate researchers to consider a deep-learning based back-end service; hence, the term *deep* intelligence (as a service). Since timeliness of interactions is important and may be related to quality-of-service agreements of different classes of users, a natural step from a real-time perspective is to investigate the challenges that the design of such services imposes on the underlying scheduler.

Consider a service that offers classifiers (or other machine learning algorithms) trained using deep learning solutions. Such algorithms have a layered structure. External inputs,

such as images or sound clips, are applied to the first layer. The output of one layer is then the input to the next. The total number of layers is called neural network *depth.* More complex inputs need more layers, making the needed depth *data-dependent*. For example, in an image recognition task, simple images (e.g., a picture of an empty blue sky) might need a smaller number of layers to yield a high quality classification result compared to complex cluttered images. Hence, the scheduled depth of neural network processing becomes another scheduling parameter (besides end-to-end deadlines derived from desired interaction latency for the given user class). Since task complexity is data dependent, this parameter is not known *a priori*, making for an interesting problem.

A related challenge lies in the lack of well-defined output *utility metrics* and *utility functions* to serve as a foundation for deciding on the best neural network depth. Indeed, quantification of utility has always been a challenge in most research aiming at optimizing application-perceived quality metrics. In contrast to assumptions made in much prior work (e.g., on imprecise computations), our utility curve (that offers a measure of output quality versus computation time) is in general not available ahead of time, because it depends on individual input (e.g., image) complexity.

2.2 OVERVIEW

In this work, the utility *metric* challenge is solved by proposing *confidence in results* as the utility measure. Confidence in results is independent of what the results are used for, making it a widely applicable metric across a variety of machine learning algorithms and applications. The utility *function* can then be computed using solutions proposed in recent literature that estimate probabilistic confidence in output results of deep learning systems (e.g., confidence in correctness of classifer output) [9]. As shown in this work, the computed confidence can be updated dynamically, as partial processing is done on inputs. Hence, better estimates of "utility" are computed over time as processing occurs, leading to refinement in utility-maximizing schedules. A bound on the efficacy of the resulting schedule at global utility maximization subject to real-time deadline constraints is then computed.

To this end, this work proposes RTDeepIoT, the first real-time scheduling pipeline for deep neural networks. RTDeepIoT consists of three main modules, each of which separates and solves one major challenge.

- *Confidence calibration:* calibrates estimates of confidence in neural network outputs, thus producing an unbiased estimator of output quality.
- *Dynamic confidence curve update:* dynamically refines utility curves to help estimate

needed neural network depth.

- *Scheduler:* determines the depth and sequence of neural network executions that offers an approximation bound on global utility maximization.

The uncertainty calibration module lays the foundation for the whole pipeline by producing an unbiased output utility estimator. The dynamic confidence curve update module refines confidence in outputs progressively, given input data and results of partial processing. The scheduling algorithm determines the order and depth of neural network executions based on the updated confidence curves and task deadlines using submodular maximization. Furthermore, this work includes design for an efficient approximation of the scheduling algorithm to reduce the scheduling overhead. It also introduces implementation of a user space scheduling framework to verify the effectiveness of RTDeepIoT.

2.3 THESIS ORGANIZATION

The rest of this thesis is organized as follows. chapter 3 introduces the technical details of scheduling model with performance analysis. chapter 3 describes the deep intelligence as a service with real-time scheduling. System implementation is described in chapter 4. The evaluation is presented in chapter 5. Related work is presented in chapter 6 and conclude in chapter 7.

CHAPTER 3: MODEL

3.1 ALGORITHMS

This section introduces RTDeepIoT, a real-time scheduling model motivated by the special needs of "smart" embedded/IoT applications that derive their intelligent behavior from offloading measurements to servers that process them using deep neural networks. The model in this section is relatively abstract, and the technical details of applying this model in practice to an actual service prototype is deferred to Section 3.2.

A few complexities are ignored first, such as the manner in which accurate utility values are estimated. Those will be discussed later, when the abstract model is mapped to the actual design of our service.

The discussion below presents the scheduling model on the server that must run deep neural networks on request from client devices when data from these devices arrives at the server. It is assumed that communication with the server is not the bottleneck. Later, the evaluation shows that this is the case (due to the comparatively heavier computational demand of deep neural network processing). Hence, the following part models server-side execution only. Let program $\mathcal{T}_i$ refer to a program with a sequence of computation stages $\{T_i^{(l)}\}$ for $l = 1, \cdots, L_i$, and a relative deadline, d_i. The set $\mathcal{T}_i^{(1:l)}$ is defined as the set of stages $\{T_i^{(1)}, \cdots, T_i^{(l)}\}$. The utility for running all stages in $\mathcal{T}_i^{(1:l)}$ by the deadline is denoted by $\mathcal{U}(\mathcal{T}_i^{(1:l)}) = u_i^{(l)}$. No utility is derived from the execution of stages that miss the deadline.

When executing a stage (typically a layer or more in the neural network), computations include matrix multiplications and element-wise operations. Matrices and vectors in neural networks are usually large enough that the available parallelism exceeds the number of underlying cores by a significant multiplicative factor. Therefore, without loss of schedulability, a stage can be deployed on all cores concurrently, leading to a model where the same stages run in parallel on all cores. Thus, the system is scheduled as a (replicated) uniprocessor. As discussed later, our system uses EDF as the underlying scheduling algorithm, which is optimal for the uniprocessor scheduling model.

Diminishing-return utility curves are considered, a widely encountered case in data processing systems. Recent studies on deep learning have shown that the improvement in result accuracy is indeed diminishing in the depth of the neural network [5]. The formal definition of diminishing-return utility is as follows:

Definition 3.1. *For a utility curve $\mathcal{U}_i(\cdot)$, define the differential utility as $\Delta u_i^{(l)} = \mathcal{U}_i(\mathcal{T}_i^{(1:l)}) - \mathcal{U}_i(\mathcal{T}_i^{(1:l-1)})$, where $\mathcal{U}_i(\mathcal{T}_i^{(1:0)}) = 0$. The property of diminishing returns is that $\Delta u_i^{(l_1)} <=$*

$\Delta u_i^{(l_2)}$, *if* $l_1 > l_2$.

Algorithm 3.1 The greedy submodular maximization algorithm

1: Input: $\mathcal{T}, \mathcal{U}(\cdot), \forall i : d_i$; Output: $\mathcal{S}_G$
2: $\mathcal{S}_G \leftarrow \emptyset, \mathcal{G} \leftarrow \mathcal{T}$
3: **while** $\mathcal{G} \neq 0$ **do**
4: $\quad v* \leftarrow \arg\max_{v \in \mathcal{G}} \mathcal{U}(\mathcal{S}_G \cup v) - \mathcal{U}(\mathcal{S}_G)$
5: $\quad \mathcal{G} \leftarrow \mathcal{G} \backslash \{v*\}$
6: $\quad$ **if** $\mathcal{S}_G \cup \{v*\}$ can be scheduled under deadlines $\{\tau_i\}$ **then**
7: $\quad\quad \mathcal{S} \leftarrow \mathcal{S}_G \cup \{v*\}$
8: $\quad$ **end if**
9: **end while**
10: **return** $\mathcal{S}_G$

In order to ensure the stages of a task are executed in sequence, the set of program stages are abstracted as a multiset, allowing for multiple instances for each of its elements.

Definition 3.2. *For n stage-wise programs, the set of all stages of all programs are denoted as a multiset $\mathcal{T} = \{T_i^{L_i}\}$ with $i = 1, \cdots, n$, where the multiplicity of task stage T_i is L_i. The cardinality of multiset is defined as $|\mathcal{T}| = \sum_{i=1}^{n} L_i$. In addition, set $\{\mathcal{T}_i^l\}$ denotes $\mathcal{T}_i^{(1:l)}$.*

The element T_i in multiset does not explicitly refer to executing a specific computation stage, but to the chance of executing one computation stage in the i-th program with a pre-defined sequence order. If two T_i are selected from the multiset, the i-th program will be scheduled to run with the depth of 2.

The goal of the scheduler is to choose the best depth, h_i, for each program $\mathcal{T}_i$, such that (i) the resulting system is schedulable under the underlying operating system's scheduling policy (the proposed system uses EDF), and (ii) the overall utility (given by the sum of $\sum_{i=1}^{n} u_i^{(h_i)}$) is maximized. This is akin to a knapsack problem with a additional constraint that ensures schedulability.

Let us define $\mathcal{S}$ to be the multiset of all task stages that are chosen for execution by our scheduler (i.e., $\mathcal{S} = \{\mathcal{T}_i^{h_i}\}$). Let us further denote $t_i^{(h_i)}$ as the time period from the arrival time of task $\mathcal{T}_i$ to the time point when all stages in $\mathcal{T}_i$ are completed. the problem can be formulated as:

$$\begin{aligned} \max_{\mathcal{S} \subseteq \mathcal{T}} \quad & U(\mathcal{S}) = \sum_{i=1}^{n} u_i^{(h_i)} \\ s.t. \quad & t_i^{(h_i)} < d_i \end{aligned} \tag{3.1}$$

As shown in Definition 1, the utility curve of deep learning execution stages follow the

diminishing-return property. The overall utility of scheduling set, therefore, diminishes as more scheduling stages are added. It is expressed more formally as Lemma 1:

Lemma 3.1. *The objective set function $U(\mathcal{S})$ is monotone submodular,* i.e.*, for every $\mathcal{S} \subseteq \mathcal{W} \subseteq \mathcal{T}$ and every $v \in \mathcal{T} \backslash \mathcal{W}$, it satisfies that $U(\mathcal{S} \cup v) - U(\mathcal{S}) \geq U(\mathcal{W} \cup v) - U(\mathcal{W})$; and for every $\mathcal{S} \subseteq \mathcal{W}$, it satisfies $U(\mathcal{S}) \leq U(\mathcal{W})$.*

The submodularity of objective set function motivates us to solve the scheduling problem (3.1) with the submodular maximization. In many submodular maximization problems settings, the simple greedy algorithm can be implemented in an efficient way, while the optimal solution is proven to be NP-Hard. Moreover, the submodularity of objective set function can often provide a good performance guarantee depending on the structure of feasible set. Thus, in this subsection, a greedy algorithm for our scheduling problem (3.1) is designed and then analyze its performance accordingly.

The greedy algorithm for our scheduling problem is shown in Algorithm 3.1. The algorithm starts from an empty set $\mathcal{S}_G$. In each step, the algorithm picks a computation stage $v*$ with the maximum differential utility (where utility in our implementation, as shown later, is the estimated confidence in results), and deletes $v*$ from the candidate set $\mathcal{G}$. The later part includes verification for whether the new $\mathcal{S}_G$ set is still feasible and satisfying the deadline constraints, when the picked execution stage $v*$ is added to $\mathcal{S}_G$. The verification can be done easily with the EDF algorithm. If $v*$ passes the verification, it is added to $\mathcal{S}_G$ (Line 4-6). The algorithm keeps the loop until the candidate set $\mathcal{G}$ is empty. Thus the system can schedule $\mathcal{S}_G$ with the simple EDF algorithm.

When the objective set function is monotone submodular, the greedy algorithm can often ensure a guaranteed performance when the set of feasible solutions has a nice structure. p-independence systems [11] are one of these structures:

Definition 3.3. *An independence system is a pair $(\mathcal{T}, \mathcal{I})$ such that $\mathcal{T}$ is a finite set, and $\mathcal{I} \subseteq 2^{\mathcal{T}}$ is a collection of subsets of $\mathcal{T}$ (called the independent sets) satisfying the following two properties: (1) $\emptyset \in \mathcal{I}$ and (2) for each $\mathcal{S}' \subseteq \mathcal{S} \subseteq \mathcal{T}$, if $\mathcal{S} \in \mathcal{I}$ then $\mathcal{S}' \in \mathcal{I}$.*

Given an independence system $(\mathcal{T}, \mathcal{I})$ and a set $\mathcal{S} \subseteq \mathcal{T}$, it is assumed that $B(\mathcal{S})$ denotes the set of maximal independent sets in $\mathcal{S}$. It is an independent set that is not a subset of any other independent set. it can be formally as $B(\mathcal{S}) = \{\mathcal{S}' \subseteq \mathcal{S} : \mathcal{S}' \in \mathcal{I}, \text{and } \nexists e \in \mathcal{S} \backslash \mathcal{S}' \text{ such that } \mathcal{S}' \cup \{e\} \in \mathcal{I}\}$.

Definition 3.4. *An independence system $(\mathcal{T}, \mathcal{I})$ is called a p-independence system if for all $\mathcal{S} \subseteq \mathcal{I}$,*

$$\frac{\max_{\mathcal{S}' \in B(\mathcal{S})} |\mathcal{S}'|}{\min_{\mathcal{S}'' \in B(\mathcal{S})} |\mathcal{S}''|} \leq p, \tag{3.2}$$

where $|\mathcal{S}|$ denotes the cardinality of set $\mathcal{S}$. In addition, 1*-independence system is also called matroid.*

Now it is easily obtained that the feasible solutions of our scheduling problem (3.1) form an independence system. Firstly, the empty set is schedulable, which is a feasible solution to (3.1). Secondly, for a feasible set $\mathcal{S}$, all its subsets $S' \subseteq S$ are also feasible to the problem. Therefore, according to *Definition 2*, the feasible sets of our scheduling problem (3.1) form an independence system.

Here, an illustrative example for the previous two definitions is provided. Assume that a system has two programs $\mathcal{T}_1 = \{T_1^{(1)}\}$ and $\mathcal{T}_2 = \{T_2^{(1)}, T_2^{(2)}\}$. Both of them arrive at the same time with the same deadline, $d_1 = d_2 = 5$s. The execution time of the stage in $\mathcal{T}_1$ is 4s, and the execution time of each stage in $\mathcal{T}_2$ is 2s. In this case, $\mathcal{T} = \{T_1, T_2, T_2\}$, $\mathcal{I} = \{\emptyset, \{T_1\}, \{T_2\}, \{T_2, T_2\}\}$, $\max_{\mathcal{S}' \in B(\mathcal{S})} = \{T_2, T_2\}$, and $\min_{\mathcal{S}' \in B(\mathcal{S})} = \{T_1\}$. Therefore, in this case, $(\mathcal{T}, \mathcal{I})$ is a 2-independence system according to Definition 4.

After identifying the structure of feasible sets, a general guaranteed performance of our greedy scheduling algorithm 3.1 can be provided.

Theorem 3.1. *The optimal solution of the scheduling problem* (3.1) *can be approximated by the greedy algorithm, Algorithm 3.1, with a factor of* $1/(1+p)$ *in the worst case, when the feasible sets form a p-independence system,* i.e., $U(\mathcal{S}_G) \geq \max_{\mathcal{S} \subseteq \mathcal{T}} U(\mathcal{S})/(1+p)$.

The above result assumes that the utility is known excatly. In our system, utility is represented by confidence in results. Note that, due to the fact that stages must be executed in order, at any point in time, one only needs to estimate utility of executing the single next stage. In other words, one needs to estimate confindence in results of a task if one more layer of its neural network is executed. As shown in the next section, this problem is solvable, but only approximately. Assume that the system has an α-approximate oracle for the (next stage of the) utility curve. Hence, the following applies [11]:

Theorem 3.2. *When only an α-approximate oracle for the utility curve is available for* $\alpha < 1$*, the optimal solution of the scheduling problem* (3.1) *can be approximated by Algorithm 3.1, with a factor of* $\alpha/(\alpha+p)$ *in the worst case.*

The above two theorems provide Algorithm 3.1 a worst-case performance guarantee. The guarantee depends on p defined in (3.2). In general, p is usually small for the system that schedules multi-stage deep learning applications. The value is p is decided by the ratio between the maximum and minimum cardinalities of sets that are just meet the schedulability condition. In practice, such ratio is small for two reasons. On one hand, the execution time of different deep learning stages are in the same order of magnitude. On the other hand,

when scheduling deep intelligence services on high-speed cloud servers, the cardinalities of sets that are just schedulable are very large. Therefore, the ratio of two large cardinalities is small or even approximates 1.

It turns out, since the deep intelligence service would typically run the same code (e.g., face recognition) but on different inputs, the worst-case execution time of each layer can be the same across different tasks. Thus, the following assumption holds:

With an addition assumption that the worst-case running times are the same for all executable stages in $\mathcal{T} = \{T_i^{(l)}\}$, *i.e.*, $t(T_i^{(l)}) = w$, where $t(\cdot)$ denotes the worst-case running time, the previous two theorems can lead to the a stronger corollary:

Corollary 3.1. *With the assumption that worst-case running times are the same for all executable stages in* $\mathcal{T}$*, the feasible set of the scheduling problem* (3.1) *forms a matroid,* i.e., 1*-independence system. The optimal solution of* (3.1) *can be approximated by the greedy algorithm, Algorithm 3.1, with a factor of* $1/2$ *in the worst case,* i.e., $U(\mathcal{S}_G) \geq \max_{\mathcal{S} \subseteq \mathcal{T}} U(\mathcal{S})/2$. *When only an* α*-approximate oracle for the utility curve is available for* $\alpha < 1$*, the optimal solution of* (3.1) *can be approximated by Algorithm 3.1, with a factor of* $\alpha/(\alpha+1)$ *in the worst case.*

Algorithm 3.2 The top-k greedy submodular maximization algorithm

1: Input: $\mathcal{T}, \mathcal{U}(\cdot), \forall i : d_i$; Output: $\mathcal{S}_G$
2: $\mathcal{S}_G \leftarrow \emptyset, \mathcal{G} \leftarrow \mathcal{T}$
3: **while** $|\mathcal{S}_G| < k$ **do**
4: $\quad v* \leftarrow \arg\max_{v \in \mathcal{G}} \mathcal{U}(\mathcal{S}_G \cup v) - \mathcal{U}(\mathcal{S}_G)$
5: $\quad \mathcal{G} \leftarrow \mathcal{G} \backslash \{v*\}$
6: $\quad$ **if** $\mathcal{S}_G \cup \{v*\}$ can be scheduled under deadlines $\{\tau_i\}$ **then**
7: $\quad\quad \mathcal{S} \leftarrow \mathcal{S}_G \cup \{v*\}$
8: $\quad$ **end if**
9: **end while**
10: **return** $\mathcal{S}_G$

During the scheduling process, due to the data-dependent complexities of deep learning tasks, the utility curves may have to be updated constantly. Therefore, the old scheduling sequence is deprecated with by a new sequence through running Algorithm 3.1 with updated information. This property of constant updating causes some issues. On one hand, the system utilizes only the beginning part of the scheduling sequence. On the other hand, constant running the greedy algorithm over the whole candidate set causes a high overhead. Therefore, Algorithm 3.1 is further approximated by limiting the size of scheduling set. As shown in Algorithm 3.2, the ending condition of while loop is changed into $|\mathcal{S}_G| < k$ (Line 2). Therefore, A scheduling set $\mathcal{S}_G$ with size k can be obtained through Algorithm 3.2, and then the system can schedule $\mathcal{S}_G$ with simple EDF algorithm.

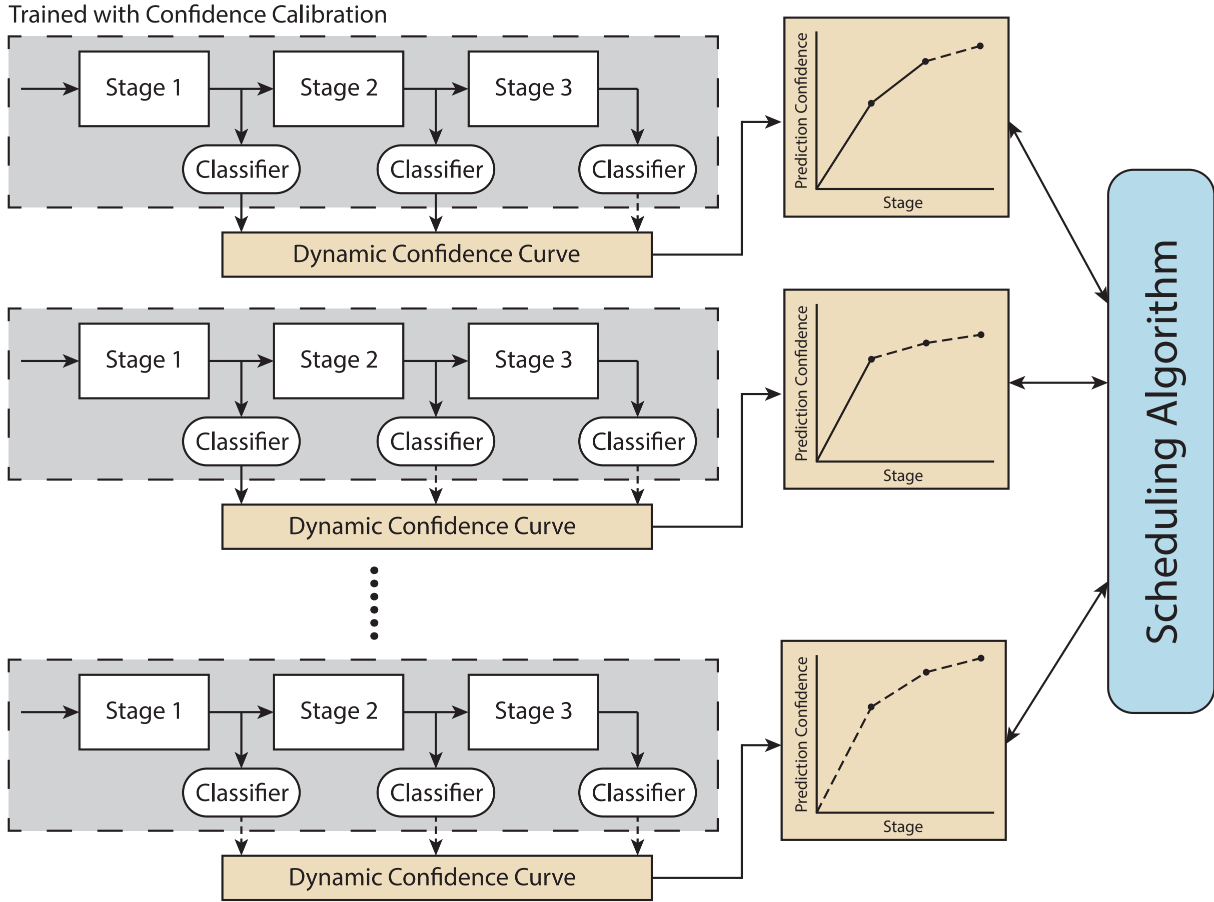


Figure 3.1: The overview of Deep Intelligence as a Service.

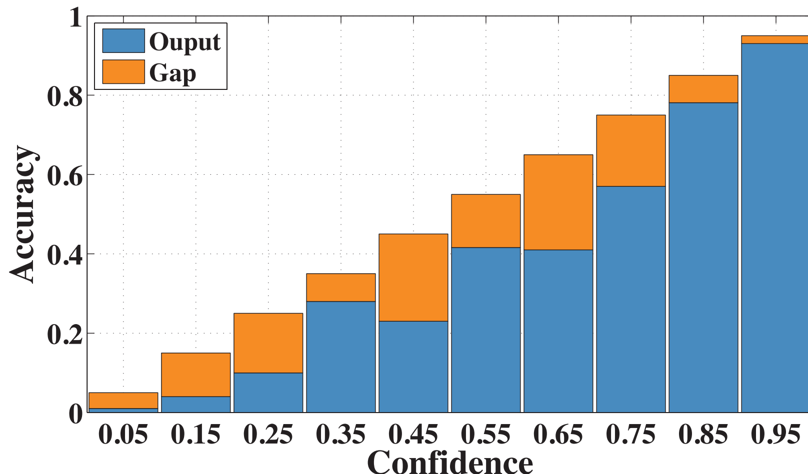
3.2 FRAMEWORK ARCHITECTURE

This section includes overview our service architecture, where trained classifiers perform on-demand processing on client data. For example, an airport might send images from security cameras across the terminal for processing to detect unattended baggage (and report it to a human operator). The architecture is shown in Figure 3.1.

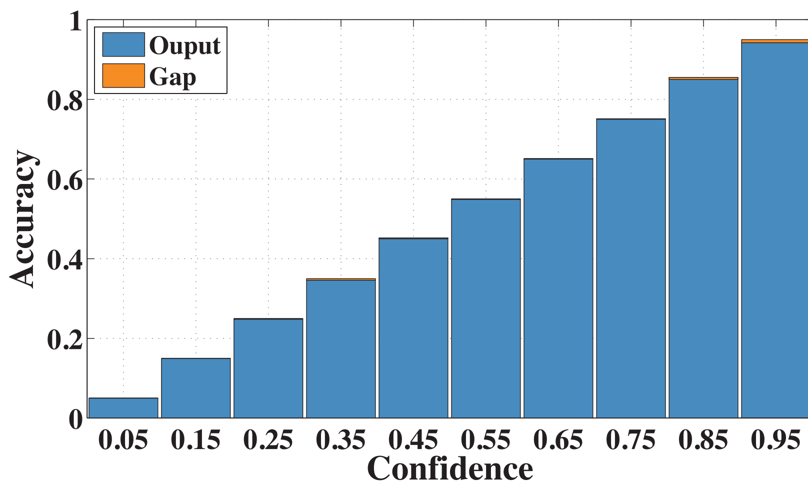
As mentioned earlier, neural network processing is separated into multiple layers. These layers can be grouped into a small number of stages (in general of multiple layers each). At the end of each stage, a thin softmax function layer can be attached to compute a classification at selected internal layers. Two challenges ensue:

- *Confidence calibration:* calibrate the confidence of neural network classification at intermediate layers.
- *Dynamic confidence curve updates:* construct a utility curve by dynamically refining the confidence in outputs over time.

The technical details of these two modules are discussed in the following subsections.



(a) Without confidence calibration



(b) With the entropy-based calibration.

Figure 3.2: The reliability diagrams of ResNet on CIFAR-10.

3.2.1 Utility Metric: Confidence Calibration

For a classification problem, the output of a neural network classifier is a vector of probabilities, where the largest probability is called the classification confidence. Ideally, a well-calibrated classification confidence should be equal to the likelihood of classification correctness, which is an unbiased estimator of classification accuracy. Unfortunately, most deep learning systems are not well-calibrated. With the growing capability and advances in deep learning, although classification accuracy has greatly improved, the classification confidence is not as accurate [12].

The calibration of confidence can be visually represented by the reliability diagram [13]. As shown in Figure 3.2, the diagram plots expected classification accuracy as a function of confidence. If the neural network is perfect, then the diagram should plot the identity

function. Any deviation from a perfect diagonal represents miscalibration.

In order to represent the degree of miscalibration with a scalar that summarizes statistics of calibration, Expected Calibration Error (ECE) [14] is introduced. First, classification results are grouped into M bins with equal-width $1/M$. $\mathcal{S}_m$ is denoted as the set of samples whose classification confidence falls into the interval $((m-1)/M, m/M]$. Then, the average accuracy of $\mathcal{S}_m$ can be defined as:

$$acc(\mathcal{S}_m) = \frac{1}{|\mathcal{S}_m|} \sum_{\mathbf{S}_i \in \mathcal{S}_m} \mathbb{1}(\hat{y}_i = y_i), \tag{3.3}$$

where $\hat{y}_i$ and y_i are the predicted and true label of sample $\mathbf{S}_i$. Next, the average confidence of $\mathcal{S}_m$ can be defined as:

$$conf(\mathcal{S}_m) = \frac{1}{|\mathcal{S}_m|} \sum_{\mathbf{S}_i \in \mathcal{S}_m} p_i, \tag{3.4}$$

where p_i is the classification confidence of sample $\mathbf{S}_i$. The ECE metric is defined as the weighted average of the difference between average accuracy and confidence in M bins.

$$ECE = \sum_{m=1}^{M} \frac{|\mathcal{S}_m|}{m} \Big| acc(\mathcal{S}_m) - conf(\mathcal{S}_m) \Big|. \tag{3.5}$$

Accurate confidence estimation has drawn growing attention in recent studies [9, 15, 16]. However, existing efforts either focus mainly on regression problems [9, 15, 16] or tend to underestimate or overestimate the confidence [15, 16].

The behaviour of confidence underestimation and overestimation can be described by the formulation of average accuracy (3.3) and confidence (3.4). $\mathcal{S}$ is denoted as the set of all samples. When $acc(\mathcal{S}) < conf(\mathcal{S})$, the neural network tends to underestimate the classification results. When $acc(\mathcal{S}) > conf(\mathcal{S})$, the neural network tends to overestimate. The target is to make $acc(\mathcal{S}) \approx conf(\mathcal{S})$ and $ECE \to 0$, making the confidence in neural network results be an unbiased estimator of classification accuracy (*i.e.*, the utility metric).

A straightforward approach is to adjust the average confidence $conf(\mathcal{S})$ to the value of average accuracy $acc(\mathcal{S})$ with fine-tuning on a validation dataset. A natural metric to control the classification confidence is entropy, $H(\mathbf{p}_i)$, where $\mathbf{p}_i$ is the vector of confidences over all targeted classes. Therefore, a simple entropy-based regularization method for confidence calibration with fine-tuning is introduced. The loss function of the fine-tuning process can be reformulated as:

$$\mathcal{L} = CE(\mathbf{p}_i, \mathbf{y}_i) + \alpha \cdot H(\mathbf{p}_i), \tag{3.6}$$

where $CE(\cdot, \cdot)$ is the cross entropy; $\mathbf{y}_i$ is label of sample i in one-hot representation; and α is

the hyper-parameter for the entropy regularization. Tuning the value of α is simple. When the confidence underestimate the accuracy, it is assumed that $\alpha < 0$ and vice-versa.

Our confidence calibration method is simple but works well in practice. Detailed evaluation is shown in Section 5.1.

3.2.2 Utility Curve: Dynamic Confidence Updates

The idea of dynamic confidence updates is to gradually refine confidence during the execution process. At the beginning, predicted confidence in results is the same for all tasks, and is based on overall statistics computed from training data. However, as tasks computes results at intermediate stages, each task is able to update its own confidence in computed results. It can then update confidence in results of future (subsequent) stages. This is achieved using regression models that relate computed confidence in results of the executed stage(s) to predicted confidence in results of future stages.

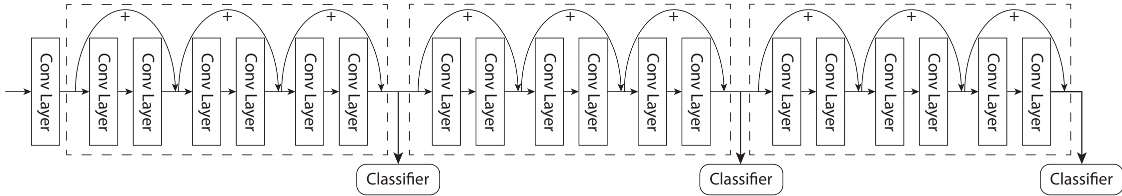


Figure 3.3: The illustration of three-stage ResNet.

Specifically, the Gaussian process regression model [17] is chosen. This choice is based on two reasons. First, Gaussian process is the state-of-the-art regression model. Second, Gaussian processes produce a Gaussian distribution as the output, from which the system can easily compute the mean value or desired confidence intervals. The framework presented in this work simply selects the expected mean value, because RTDeepIoT focuses on maximizing the classification accuracy.

For a three-stage neural network, as shown in Figure 3.1, three Gaussian process regression models are trained, $\hat{p}_i^{(2)} = \mathcal{GP}_{1_2}(p_i^{(1)})$, $\hat{p}_i^{(3)} = \mathcal{GP}_{1_3}(p_i^{(1)})$, and $\hat{p}_i^{(3)} = \mathcal{GP}_{2_3}(p_i^{(2)})$, where $p_i^{(l)}$ denotes the classification confidence of sample i at neural network stage l. These regression models are learned from the confidence curves of training data.

However, Gaussian process is notorious for its long inference time, which is unacceptable for a runtime predictor. Fortunately, the inputs of these gaussian models are bounded, *i.e.*, $p_i^{(l)} \in [0, 1]$. Therefore, these complex Gaussian process regression models can be approximated with simple piece-wise linear functions with two steps:

1. profiling the Gaussian process regression model with a set of input confidences, $\{0, 1/M, \cdots, 1\}$.

2. connecting these profiling points with a piece-wise linear function.

Thus, these computationally efficient piece-wise linear functions can be used during the runtime for updating the dynamic confidence curve. In practice, a good approximation can be achieved by choosing $M = 10$.

Detailed evaluation is shown in Section 5.1.

CHAPTER 4: IMPLEMENTATION

This work includes implementation of a user space scheduling framework to verify the effectiveness of RTDeepIoT. Implementing the scheduler in *user space* solves two key concerns. First, it makes it compatible with popular operating systems, such as Linux, which facilitates deployment at scale. Second, it enables us to integrate the scheduler with widely deployed deep learning libraries. Specifically, the system is integrated with TensorFlow [18].

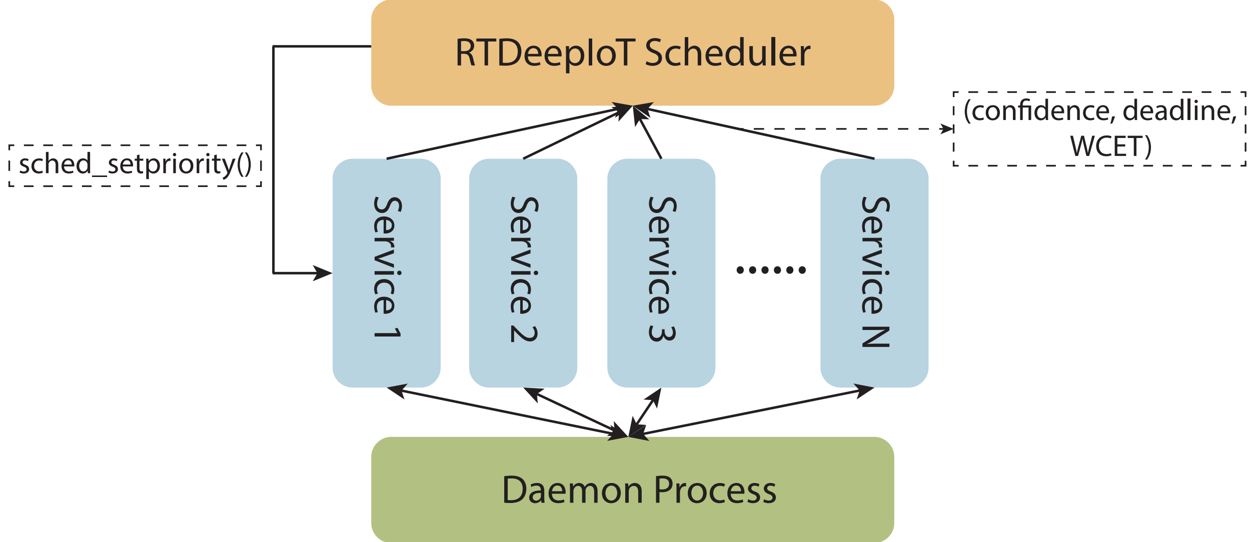


Figure 4.1: Structure of scheduling framework implementation.

To implement classifiers for our proof-of-concept prototype, an image recognition service is chosen based on a state-of-the-art convolutional neural network (CNN) structure; namely, residual neural networks (ResNet). As shown in Figure 3.3, compared to traditional CNNs, ResNets add extra shortcut connections between convolutional layers. The whole ResNet is divided into three stages. Except for the bottom convolutional layer on the left side, each stage consists of six convolutional layers with three residual shortcut connections. At the end of each stage, a simple softmax classifier is appended, using the end-of-stage aggregated features for classification. The whole network is trained on the CIFAR-10 dataset 50000 training images.

Figure 4.1 demonstrates the structure of our framework. The RTDeepIoT scheduler spawns a pool of worker processes, and each worker process runs an instance of Tesnfor-Flow and classifies a total of 10000 images from CIFAR-10 dataset. These processes wait on input images to arrive. Each image represents a task and is submitted to the system with a deadline by which it is to be classified. The deadline is inherited from the client's class of service. When an input image arrives, it is assigned to a process in the pool. The

process runs the aforementioned deep neural network on the new input. The execution of the process features an explicit separation into stages. A stage might contain multiple layers. When finished, each stage will output a tuple in the form *(predicted value, confidence)*. *Predicted value* is the classification result from the current stage, specifying the most likely classification. *Confidence* describes the likelihood that this classification is correct. For example, a picture can be classified as a cat, dog, or cow, with probabilities 0.6, 0.3, and 0.1, respectively. The classification result is then ("cat", 0.6). The *confidence* in classification will then be sent to our scheduler through a named pipe in Linux. Other parameters that are given to the scheduler include the deadline from the original image classification request and the worst case execution time (WCET) of the task, known from previous profiling of the system.

A daemon process monitors the elapsed time for each classification task. If the elapsed time for a task exceeds the deadline, the daemon process will send a signal to stop the current computation. The process is returned to the pool and is made available to handle new requests. The scheduler and the daemon process run at the highest priority.

The frameworks, worker process programs as well as the deadline notification daemons all run under SCHED_FIFO scheduling class. Since SCHED_FIFO has inherent higher priority than SCHED_NORMAL scheduling class, implementing the frameworks with SCHED_FIFO can eliminate interference from other processes managed by the system, and thus evaluating the framework under almost all hardware resources. Although Linux kernel has throttling feature to guarantee at least 5% computation time to non-real-time scheduling classes (SCHED_NORMAL and other lower priority classes), it is equivalent to running the framework with a less potent hardware. Besides, using SCHED_FIFO instead of other real-time scheduling classes such as SCHED_RR and SCHED_DEADLINE gives us more flexibility. The framework is able to emulate more scheduling policies without the constraints imposed by the kernel by explicitly yield in the worker processes.

Below shows the work-flow for a single classification process:

1. The process picks a classification task. Namely, it gets an image to classify.

2. It notifies the daemon process of its deadline, and is assigned a priority accordingly.

3. When a stage is finished, the process sends the updated confidence value in results of subsequent stages to the scheduler.

4. The scheduler will update its estimate of utility of future stages and recompute the set of stages to execute using Algorithm 3.2.

5. If the process finishes all the stages of the current classification task, it goes back to the pool and waits for new assignments.
6. If the process cannot finish by the deadline, it will be interrupted by the daemon process, and forced to return to the pool.

Note that, since our greedy algorithm tends to choose stages with the maximum incremental utility for future execution, tasks with lower initial classification confidence values tend to be selected for another execution stage. This has the side-effect of attaining better fairness as well.

CHAPTER 5: EVALUATION

To verify the effectiveness of our proposed scheduling algorithm, the scheduler is tested with several processes running the aforementioned residual neural network. Each process classifies images from the CIFAR-10 dataset. The dataset consists of 60000 images of 10 classes. Images arrive in a randomly shuffled order. The workstation that runs the scheduler and the classification processes has 8 Intel i7-4770 CPUs, with 32 GB memory. The evaluation is performed under Ubuntu 16.04 with kernel version 4.13. The residual neural network is implemented on TensorFlow 1.4.0.

The following subsections evaluates our RTDeepIoT real-time scheduling pipeline from different perspectives, including classification confidence, scheduler overhead, pipeline bottlenecks, and overall classification accuracy under different workloads.

5.1 CONFIDENCE CALIBRATION & DYNAMIC CONFIDENCE UPDATES

The evaluation of classification confidence includes two parts: confidence calibration and dynamic confidence updates.

This section includes evaluation for the quality of confidence calibration. The three-stage ResNet structure shown in Figure 3.3 is trained on the CIFAR-10 dataset with following calibration method:

1. RTDeepIoT: the entropy-based confidence calibration method introduced in (3.6).

2. RDeepSense: the state-of-the-art confidence calibration method with dropout operations [9].

3. Uncalibrated: the original neural network without confidence calibration method.

An illustration of reliability diagrams is shown in Figure 3.2. Compared to the uncalibrated result, the RTDeepIoT method has greatly reduced miscalibration error between the estimated confidence and actual classification accuracy. A quantitative analysis with the ECE metric, defined in Equation (3.5), is shown in Table 5.1. RTDeepIoT achieves the smallest ECE among all three stages, even compared to the state-of-the-art RDeepSense method. The evaluation results show that our proposed simple entropy-based confidence calibration method can provide a good estimation of classification accuracy, making it possible for the RTDeepIoT scheduling pipeline to utilize the calibrated classification confidence as the utility metric.

Table 5.1: The ECE of confidence calibration methods with three-stage ResNet on CIFAR-10 dataset .

	Uncalibrated	RDeepSense	RTDeepIoT
Stage 1	0.134	0.058	**0.010**
Stage 2	0.146	0.046	**0.012**
Stage 3	0.123	0.054	**0.008**

Table 5.2: The Mean Absolute Error (MAE) and coefficient of determination (R^2) of dynamic confidence curve prediction for three-stage ResNet on CIFAR-10 dataset .

	$\mathcal{GP}_{1_2}$	$\mathcal{GP}_{1_3}$	$\mathcal{GP}_{2_3}$
MAE	0.124	0.108	0.072
R^2	0.57	0.43	0.78

Next, the quality of our dynamic confidence updates predicted for three-stage ResNet is evaluated, which contains three regression models for predicting future-stage classification confidence values, *i.e.*, $\hat{p}_i^{(2)} = \mathcal{GP}_{1_2}(p_i^{(1)})$, $\hat{p}_i^{(3)} = \mathcal{GP}_{1_3}(p_i^{(1)})$, and $\hat{p}_i^{(3)} = \mathcal{GP}_{2_3}(p_i^{(2)})$. The evaluation results on Mean Absolute Error (MAE) and coefficient of determination (R^2) are shown in Table 5.2. Overall, the method provides a decent confidence prediction result. As the number of finished stages increases, the dynamic prediction improves. Although a certain degree of error remains, it can still provide the scheduling algorithm good estimates of the relative confidence gains obtained among different deep learning services. The following sections shows that the dynamic confidence update method provides better overall classification accuracy than simple heuristics.

5.2 SCHEDULER OVERHEAD

This subsection evaluates the overhead of our framework. Since the framework contains a user space scheduler and a daemon process running at highest priority, the framework needs to make sure these two modules do not impose significant overhead. Comparison of the total time needed to finish a specific number of worker processes with and without our scheduler can be used to derive the runtime consumption. The tasks are a stream of images to be classified with variable deadlines. Each experiment runs twice and the data point is derived by taking the average.

Much to our surprise, running RTDeepIoT would cut down the total time needed to finish the classification. As shown in Table 5.3, RTDeepIoT scheduling framework needs less

time to classify 10000 images under the constraint of variable deadlines. A straightforward explanation is that RTDeepIoT is able to throw away some unnecessary computation, with which the tasks would not improve its accuracy and would more likely to miss the deadline. However, it shoud be noted that RTDeepIoT still cost some computation power in the server. The reason for the total runtime to be less in RTDeepIoT case is that the computation thrown away by our framework offsets the extra scheduling overhead needed by RTDeepIoT. Implementing our schedule in user space on top of the TensorFlow framework, therefore, does not impose extra overhead on the scheduling tasks. While more efficient implementation of the scheduler are possible, the current approach of implementation is choosen because of its compatibility with tools already used in the machine learning community, which significantly increases the likelihood of us making impact in that community using our results. Namely, our solutions uses Linux unmodified and uses TensorFlow (the standard library for machine intelligence applications) unmodified as well. On top of those, our scheduler inherits

Table 5.3: The averaged overhead of RTDeepIoT scheduler and daemon process for 10000 images with different number of worker processes.

	16 procs	20 procs	24 procs
with Scheduler (ms)	8866	9617	10261
without Scheduler (ms)	8993	10661	11177
Difference (ms)	127	1044	916

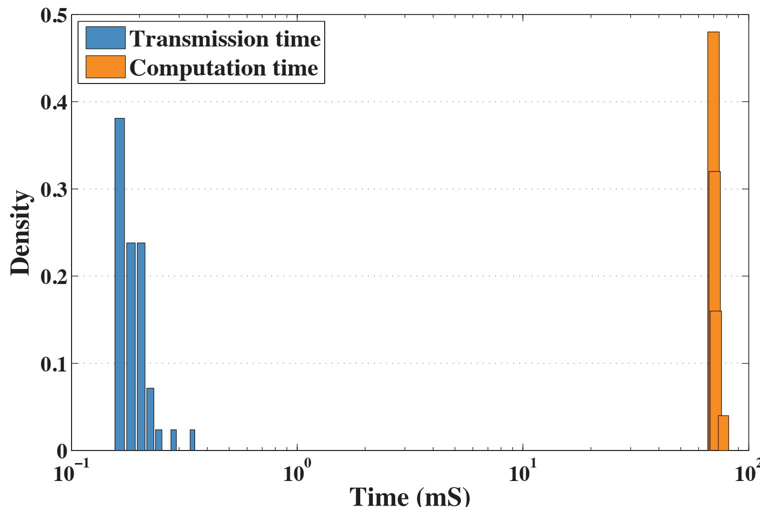


Figure 5.1: Histogram of transmission and computation time.

TensorFlow overheads, which is the price paid for compatibility/portability. As shown later,

despite this overhead, a higher total utility is achieved because the scheduler is able to allocate more judiciously the resources remaining after overhead is paid.

5.3 TRANSMISSION VS. COMPUTATION BOTTLENECK

The deep-intelligence-as-a-serive scheduler presented in this work makes sense only if the CPU is indeed the bottleneck resource. If the bottleneck lies in the communication network, then the rate at which new pictures (or data) arrive for classification will be slow enough for the CPU to never be overloaded. Thus, all classification tasks will always run to completion and this work is not needed.

This subsection compares empirical measurements of transmission and computation delays from our prototype of deep-learning based image recognition services. A local desktop constantly transmits images from CIFAR-10 dataset to a remote workstation through a secure copy protocol. The trained ResNet, as shown in Figure 3.3, takes the received images from the local desktop as input and runs the whole three stages to classify the image content. measurements of the distributions of transmission and computation delays are collected and Figure 5.1 on a log scale along the time axis is plotted. There is a clear separation between transmission and computation time distributions. Therefore, computation time is orders of magnitude higher and is indeed the bottleneck in the system justifying our scheduler design.

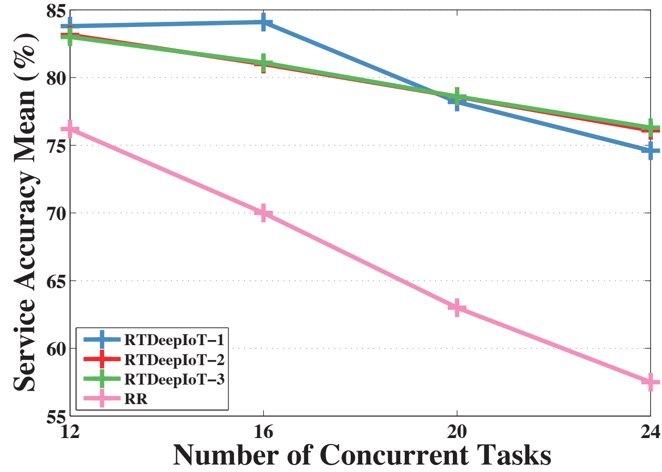
5.4 DEEP INTELLIGENCE AS A SERVICE

This subsection includes evaluation of the deep learning based image recognition services on the workstation. In order to illustrate the effectiveness of our RTDeepIoT pipeline, the back-end scheduler is based on the following algorithms:

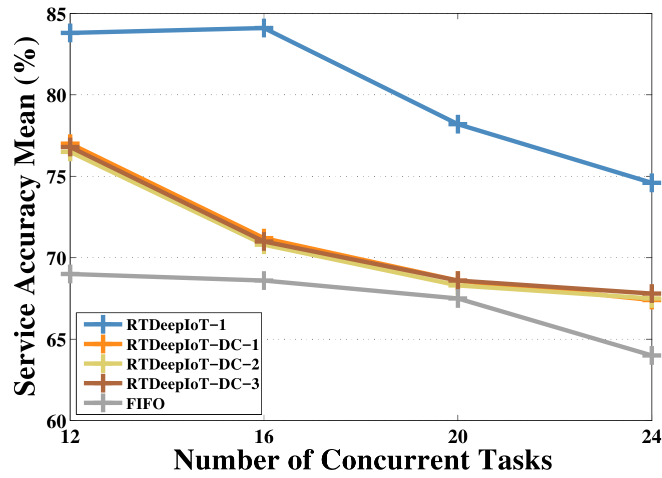
1. RTDeepIoT-k: this is our proposed scheduling pipeline, where k denotes the size of scheduling set choosen in Algorithm 3.2. During the whole experiments, k is selected to be $\{1, 2, 3\}$.

2. RTDeepIoT-DC-k: this is a variant of our scheduling pipeline. Instead of using dynamic confidence updates, the confidence is assumed to increases with the same slope. Therefore, the confidence gain of the current stage is used as the gain of future stages. The value set for k is the same as RTDeepIoT-k.

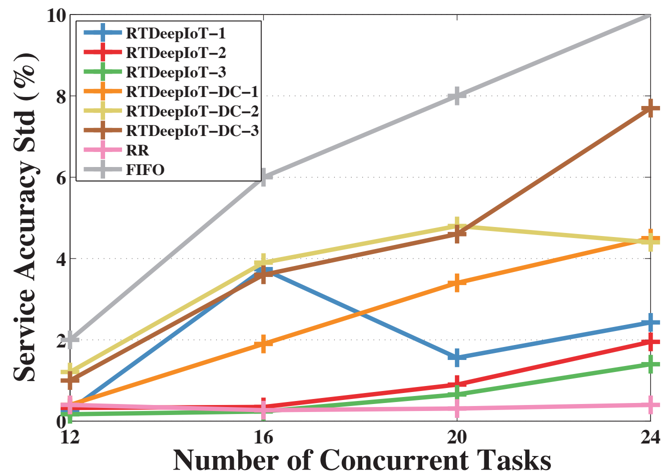
3. RR: this is a stage-level round-robin scheduling algorithm. The scheduler will select a stage to run among all the deep learning services in a round-robin manner.



(a) The mean of service classification accuracy over RTDeepIoT-k and RR.

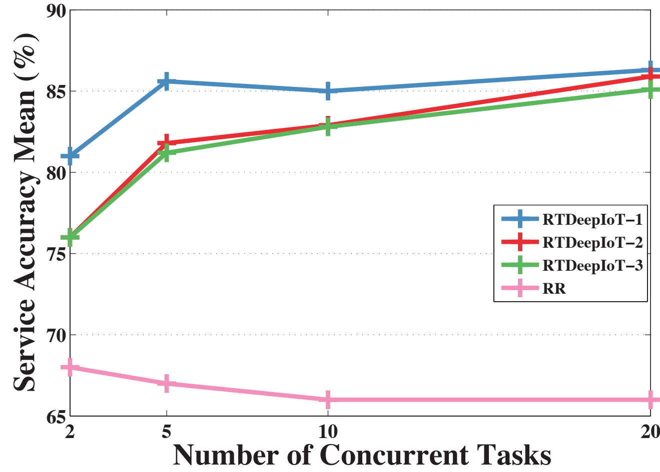


(b) The mean of service classification accuracy over RTDeepIoT-1, RTDeepIoT-DC-k, and FIFO.

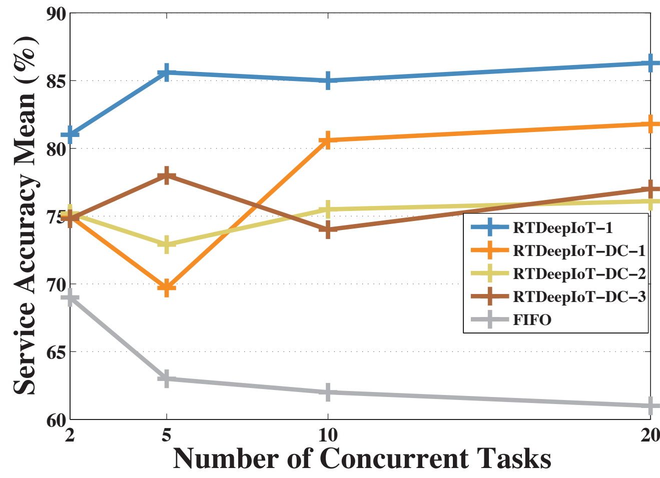


(c) The standard deviation of service classification accuracy.

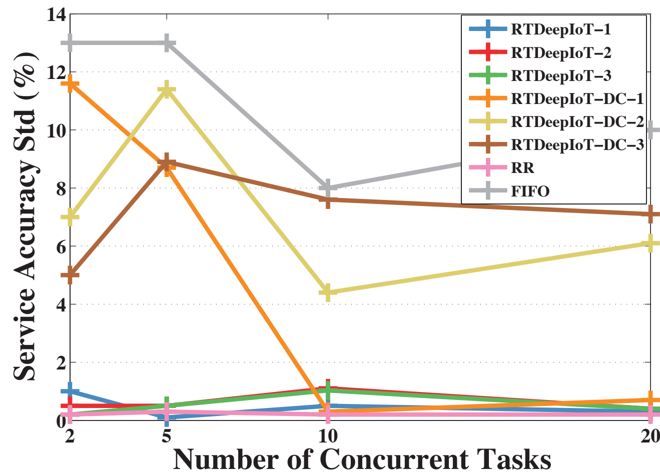
Figure 5.2: The intensity test for scheduling algorithms with ResNet on CIFAR-10.



(a) The mean of service classification accuracy over RTDeepIoT-k and RR.



(b) The mean of service classification accuracy over RTDeepIoT-1, RTDeepIoT-DC-k, and FIFO.



(c) The standard deviation of service classification accuracy.

Figure 5.3: The scalability test for scheduling algorithms with ResNet on CIFAR-10.

4. FIFO: this is a FIFO scheduling algorithm, where the scheduler runs the deep learning service on images in a first come first served manner, and runs all stages to the end.

In this evaluation, two kinds of service workloads are considered. In the first one, the deadlines of deep learning services are kept within a range, and increase the number of concurrent deep learning tasks. In the second one, the range of service deadline and the number of concurrent deep learning tasks are increased proportionally. Each deep learning task will log its classification results over stages with timestamps. The final classification result is the last result before its deadline.

In the first set of experiments, the deadlines are set between $0.8s$ and 1.2s randomly, while increasing the number of concurrent deep learning tasks from 12 to 24 with a step of 4. Prior to the experiments, the time needed for different stages are collected and the average total time for finishing all the stages take roughly 0.12s. The deadline range chosen can illustrate the scenario where the computation power can only satisfy the deadlines for part of the tasks but not all. This scenario can show the efficacy of the algorithms. The mean value and the standard deviation of classification accuracy over concurrent tasks for the scheduling algorithms are presented in Figure 5.2. From Figure 5.2c, the scheduling algorithms fall into two categories. The first type of algorithms, including RTDeepIoT-k and RR, tend to balance the classification accuracy over multiple tasks under the intensive workload. The second type of algorithms, including RTDeepIoT-DC-k and FIFO, tend to maximize the classification accuracy of a small set of tasks under the intensive workload, creating some imbalance.

RTDeepIoT-k is constantly be the best scheduling algorithm under all cases. However, an interesting observation that appears in both RTDeepIoT-k and RTDeepIoT-DC-k is that increasing the size of the scheduling set k may hurt the overall classification accuracy, which is a bit counterintuitive. Such phenomenon is caused by two reasons. On one hand, although *Lemma 3* and *Proposition 1* can provide us the worst case guarantee with Algorithm 3.1 even when the system only has an approximately correct utility curve, Algorithm 3.1 may not be the optimal choice for a normal case, which is our experiment setting. On the other hand, Algorithm 3.1 assumes the underlying scheduling policy to be EDF, which is not the best choice under the intensive workload. However, as intensity of workload is increased, RTDeepIoT-3 starts to achieve the best performance among all.

The two baseline scheduling algorithms, RR and FIFO, show relatively bad classification accuracy under a mild-intensive workload. This means that, by informing the scheduler with the classification confidence, better classification accuracy can be achieved through utilizing the heterogeneous input data complexity. When comparing RTDeepIoT-k to RTDeepIoT-DC-k, RTDeepIoT-k consistently achieves better performance with a large margin. There-

fore, the dynamic confidence update and the greedy submodular maximization scheduler helps to maximize the overall classification quality give the limited system resources and deadline constraints.

In the second set of experiments, the deadline range and the number of concurrent tasks are proportionally increased from $0.12 - 0.16$s and 2 to $0.30 - 0.40$s and 5, $0.6 - 0.8$s and 10, as well as $1.2 - 1.6$s and 20. The mean value and the standard deviation of classification accuracy over concurrent tasks for the scheduling algorithms are illustrated in Figure 5.3. The standard deviation of classification accuracy, shown in Figure 5.3c, still see divergence between two types of algorithms. However, when increasing the scale of the workload, some algorithms that do well for a small number of tasks will tend to do worse, such as DeepIoT-DC-1. Our proposed scheduling algorithm 3.2 can balance the computation over services, even with a very biased utility curve.

RTDeepIoT-1 is the best-performing scheduling algorithm. However, its accuracy gain tends to be minimized when the scale of the workload increases, which is consistent with our previous set of experiments. In this experiment, increasing the size of the scheduling set k may still hurt the overall classification accuracy. However, the performance degradation is diminished as the scale of the workload is increased.

When using algorithm 3.2 as backend, *i.e.*, RTDeepIoT-k and RTDeepIoT-DC-k, the overall classification accuracy among deep learning services increases as the workload is scald up. Therefore, our proposed scheduling model, RTDeepIoT, benefits from the scaled workload, which fits its envisioned use in future deep intelligence services.

In addition, general baseline algorithms, RR and FIFO, show performance degradation as the workload is scaled. Therefore, the scalability of these algorithms is poor, while our RTDeepIoT does better as the workload is scaled. This again illustrates the importance of taking neural network depth as a new dimension for scheduling.

CHAPTER 6: RELATED WORKS

Recently the intersection of machine learning and real time systems has sparkled research efforts on effective scheduling on learning tasks. [19], [20], [21] all trades off accuracy of learning tasks with other metrics such as schedulability or power consumption. These works are similar to our work in the sense that this work aims to build systems to effectively schedule tasks under the constraints of deadlines. However, their approaches require inspection on inner structures of each layer in the neural network and perform a per layer optimization. Our work is orthogonal to their efforts because our framework can drop in any learning platforms and does not need to explore natures of each layer. It is possible to combine both our framework as well as their system to achieve better results.

The essence of the scheduling problem lies in determining the number of processing stages (and hence neural network layers) that are sufficient to obtain an accurate output (e.g., a good classification). A body of scheduling algorithms that comes close to ours are those that support approximate computing. The general concept pervades many areas of computer science, spanning circuit design [22], architecture [23], energy efficient computing [24], machine learning [25], algorithm design [26] [27]. Our work resembles approximate computing in that our system aims to provide better quality of service (QoS) in the context of offering machine intelligence as the service paradigm, where our system intentionally discard some stages of the neural network.

A prime example of approximate computing in the real time research community is the literature on imprecise computations. The work trades off result quality versus computation time [28–32]. Our work resembles imprecise computations in that our system uses intermediate results from a prematurely terminated real-time process. The work assumes processes to be monotone, and propose an indicator for the quality of the imprecise results. More recently [30], imprecise computation models were proposed where the scheduler decides on the execution of an optional section of processes by taking deadlines and required QoS into consideration. However, our work focus on the stage-wise computation with sequential dependency and dynamic utility function.

The QoS optimization and management have also been heavily addressed in real-time literature. Rajkumar et al. presented an analytical model for QoS management in systems with multiple constraints [33,34]. Lee et al. extended the QoS management analysis with the discrete and non-concave utility functions [35]. Abdelzaher et al. proposed a real-time QoS negotiation model for maximizing system utility with guaranteed performance [36]. Curescu et al. presented a QoS optimization scheme for mobile networks [37]. Koliver et al. designed a fuzzy-control approach for QoS adaptation [38]. However, all of the previous studies assume

a known utility function beforehand. In this work, our framework consider a case where the exact utility function is not known as a priori but is rather revealed approximately as the computation proceeds.

In addition, our scheduler has been integrated with Tensor Flow - a library for deep learning systems. This makes it the first real-time scheduler to be implemented in the context of a mainstream deep learning software framework.

CHAPTER 7: CONCLUSION

This work presented a novel service model, suitable for smart IoT applications where simple devices with sensing capabilities offload their “machine intelligence” to the cloud or to an edge server. The work focuses on deep learning as the state of the art enabler of machine intelligence. A key observation was that trained deep neural networks can be partitioned into stages with results available at different degrees of fidelity after each stage. The number of stages of processing that an input item needs (e.g., for purposes of detection, prediction, or classification) depends on the data. This key insight was used to build a service, where the scheduler determines the best number of stages needed to process each input. As successive processing stages were completed, this number would be refined. Evaluation shows that the resulting schedules improve the average quality of results, essentially by allocating computing resources where they engender the best improvement in result accuracy. The service is currently being extended to other deep learning libraries (besides machine vision) to offer rich support for deep intelligence as a (real-time) service.

REFERENCES

[1] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, vol. 20, pp. 46–61, 1973.

[2] L. Abeni and G. C. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *RTSS*, 1998.

[3] G. Lipari and S. K. Baruah, "Greedy reclamation of unused bandwidth in constant-bandwidth servers," in *ECRTS*, 2000.

[4] S. Yao, Y. Zhao, A. Zhang, S. Hu, H. Shao, C. Zhang, S. Lu, and T. Abdelzaher, "Deep learning for the internet of things," *Computer*, vol. 51, no. 5, pp. 32–41, May 2018. [Online]. Available: doi.ieeecomputersociety.org/10.1109/MC.2018.2381131

[5] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 770–778.

[6] G. Hinton, L. Deng, D. Yu, G. E. Dahl, A.-r. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath et al., "Deep neural networks for acoustic modeling in speech recognition: The shared views of four research groups," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 82–97, 2012.

[7] S. Yao, S. Hu, Y. Zhao, A. Zhang, and T. Abdelzaher, "Deepsense: a unified deep learning framework for time-series mobile sensing data processing," in *Proceedings of the 26th International Conference on World Wide Web*. International World Wide Web Conferences Steering Committee, 2017.

[8] S. Yao, Y. Zhao, A. Zhang, L. Su, and T. Abdelzaher, "Deepiot: Compressing deep neural network structures for sensing systems with a compressor-critic framework," in *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*. ACM, 2017.

[9] S. Yao, Y. Zhao, H. Shao, A. Zhang, C. Zhang, S. Li, and T. Abdelzaher, "Rdeepsense: Reliable deep mobile computing models with uncertainty estimations," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 1, no. 4, p. 173, 2018.

[10] N. D. Lane, P. Georgiev, and L. Qendro, "Deepear: robust smartphone audio sensing in unconstrained acoustic environments using deep learning," in *Proceedings of the 2015 ACM International Joint Conference on Pervasive and Ubiquitous Computing*. ACM, 2015, pp. 283–294.

[11] G. Calinescu, C. Chekuri, M. Pál, and J. Vondrák, "Maximizing a monotone submodular function subject to a matroid constraint," *SIAM Journal on Computing*, vol. 40, no. 6, pp. 1740–1766, 2011.

[12] C. Guo, G. Pleiss, Y. Sun, and K. Q. Weinberger, "On calibration of modern neural networks," *arXiv preprint arXiv:1706.04599*, 2017.

[13] M. H. DeGroot and S. E. Fienberg, "The comparison and evaluation of forecasters," *The statistician*, pp. 12–22, 1983.

[14] M. P. Naeini, G. F. Cooper, and M. Hauskrecht, "Obtaining well calibrated probabilities using bayesian binning." in *AAAI*, 2015, pp. 2901–2907.

[15] Y. Gal and Z. Ghahramani, "Dropout as a bayesian approximation: Representing model uncertainty in deep learning," in *international conference on machine learning*, 2016, pp. 1050–1059.

[16] B. Lakshminarayanan, A. Pritzel, and C. Blundell, "Simple and scalable predictive uncertainty estimation using deep ensembles," *arXiv preprint arXiv:1612.01474*, 2016.

[17] C. E. Rasmussen, "Gaussian processes in machine learning," in *Advanced lectures on machine learning*. Springer, 2004, pp. 63–71.

[18] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard et al., "Tensorflow: A system for large-scale machine learning." in *OSDI*, vol. 16, 2016, pp. 265–283.

[19] S. Bateni and C. Liu, "Apnet: Approximation-aware real-time neural network," *2018 IEEE Real-Time Systems Symposium (RTSS)*, pp. 67–79, 2018.

[20] S. Bateni, H. Zhou, Y. Zhu, and C. Liu, "Predjoule: A timing-predictable energy optimization framework for deep neural networks," *2018 IEEE Real-Time Systems Symposium (RTSS)*, pp. 107–118, 2018.

[21] H. Zhou, S. Bateni, and C. Liu, "S3dnn: Supervised streaming and scheduling for gpu-accelerated real-time dnn workloads," *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 190–201, 2018.

[22] V. Gupta, D. Mohapatra, S. P. Park, A. Raghunathan, and K. Roy, "Impact: Imprecise adders for low-power approximate computing," *IEEE/ACM International Symposium on Low Power Electronics and Design*, pp. 409–414, 2011.

[23] S. Venkataramani, V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan, "Quality programmable vector processors for approximate computing," *2013 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 1–12, 2013.

[24] J. Han and M. Orshansky, "Approximate computing: An emerging paradigm for energy-efficient design," *2013 18th IEEE European Test Symposium (ETS)*, pp. 1–6, 2013.

[25] D. J. Rezende, S. Mohamed, and D. Wierstra, "Stochastic backpropagation and approximate inference in deep generative models," in *ICML*, 2014.

[26] M. Muja and D. G. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration," in *VISAPP*, 2009.

[27] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," *2006 47th Annual IEEE Symposium on Foundations of Computer Science (FOCS'06)*, pp. 459–468, 2006.

[28] J.-Y. Chung and K.-J. Lin, "Scheduling periodic jobs using imprecise results," 1987.

[29] J. W.-S. Liu, K.-J. Lin, W.-K. Shih, A. C. shi Yu, J.-Y. Chung, and W. Zhao, "Algorithms for scheduling imprecise computations," *Computer*, vol. 24, pp. 58–68, 1991.

[30] J.-M. Chen, W.-C. Lu, W.-K. Shih, and M.-C. Tang, "Imprecise computations with deferred optional tasks," *J. Inf. Sci. Eng.*, vol. 25, pp. 185–200, 2009.

[31] M. Amirijoo, J. Hansson, and S. H. Son, "Specification and management of qos in real-time databases supporting imprecise computations," *IEEE Transactions on Computers*, vol. 55, pp. 304–319, 2006.

[32] W. Feng and J. W. S. Liu, "An extended imprecise computation model for time-constrained speech processing and generation," 1993.

[33] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek, "A resource allocation model for qos management," in *Real-Time Systems Symposium, 1997. Proceedings., The 18th IEEE*. IEEE, 1997, pp. 298–307.

[34] R. Rajkumar, C. Lee, J. P. Lehoczky, and D. P. Siewiorek, "Practical solutions for qos-based resource allocation problems," in *Real-Time Systems Symposium, 1998. Proceedings. The 19th IEEE*. IEEE, 1998, pp. 296–306.

[35] C. Lee, J. Lehoezky, R. Rajkumar, and D. Siewiorek, "On quality of service optimization with discrete qos options," in *Real-Time Technology and Applications Symposium, 1999. Proceedings of the Fifth IEEE*. IEEE, 1999, pp. 276–286.

[36] T. Atdelzater, E. M. Atkins, and K. G. Shin, "Qos negotiation in real-time systems and its application to automated flight control," *IEEE Transactions on Computers*, vol. 49, no. 11, pp. 1170–1183, 2000.

[37] C. Curescu and S. Nadjm-Tehrani, "Time-aware utility-based qos optimization," in *Real-Time Systems, 2003. Proceedings. 15th Euromicro Conference on*. IEEE, 2003, pp. 83–92.

[38] C. Koliver, K. Nahrstedt, J.-M. Farines, J. da Silva Fraga, and S. A. Sandri, "Specification, mapping and control for qos adaptation," *Real-Time Systems*, vol. 23, no. 1-2, pp. 143–174, 2002.