DEEP LEARNING FOR THE INTERNET OF THINGS

BY

SHUOCHAO YAO

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2018

Urbana, Illinois

Doctoral Committee:

Professor Tarek Abdelzaher, Chair
Professor Jiawei Han
Professor Jian Peng
Professor Nicholas Lane, University of Oxford

ABSTRACT

The proliferation of IoT devices heralds the emergence of intelligent embedded ecosystems that can collectively learn and that interact with humans in a human-like fashion. Recent advances in deep learning revolutionized related fields, such as vision and speech recognition, but the existing techniques remain far from efficient for resource-constrained embedded systems. This dissertation pioneers a broad research agenda on *Deep Learning for IoT*. By bridging state-of-the-art IoT and deep learning concepts, I hope to enable a future sensor-rich world that is smarter, more dependable, and more friendly, drawing on foundations borrowed from areas as diverse as sensing, embedded systems, machine learning, data mining, and real-time computing.

Collectively, this dissertation addresses five research questions related to architecture, performance, predictability and implementation. First, are current deep neural networks fundamentally well-suited for learning from time-series data collected from physical processes, characteristic to IoT applications? If not, what architectural solutions and foundational building blocks are needed? Second, how to reduce the resource consumption of deep learning models such that they can be efficiently deployed on IoT devices or edge servers? Third, how to minimize the human cost of employing deep learning (namely, the cost of data labeling in IoT applications)? Fourth, how to predict uncertainty in deep learning outputs? Finally, how to design deep learning services that meet responsiveness and quality needed for IoT systems?

This dissertation elaborates on these core problems and their emerging solutions to help lay a foundation for building IoT systems enriched with effective, efficient, and reliable deep learning models.

To my parents and Ailing, for their love and support.

ACKNOWLEDGMENTS

It has been my distinctive privilege and honor to have had the opportunity to work with my advisor, Professor Tarek F. Abdelzaher, through my Ph.D. career. His vision and passion for research have always inspired me. Without his invaluable guidance and generous support, it would not have been possible at all for me to come this far.

I also owe my gratitude to Professor Jiawei Han, Professor Nicholas Lane, and Professor Jian Peng, with whom I have collaborated on various projects, coauthored research papers, and/or from whom I have gotten research inspiration, throughout the years. I am incredibly honored to be able to have them on my Ph.D. committee, and feel the utmost gratitude for all their help and support.

I am also fortunate enough to have studied and worked in highly interdisciplinary and collaborative environments, with exceptionally talented mentors and colleagues from across multiple universities and research labs. I would like to take this opportunity to express my deepest appreciation to Dr. Shaohan Hu, Dr. Shen Li, Dr. Shiguang Wang, Dr. Aston Zhang, Dr. Chao Zhang, Mr. Hongwei Wang, Prof. Lu Su, Dr. Md Tanvir Al Amin, Dr. Lance Kaplan, Prof. Xinbing Wang, Prof. Aylin Yener, Prof. Haiming Jin and Mr. Wenjun Jiang, for the countless insightful discussions, their generous help, and the many exciting and fruitful collaborations.

Being part of the Cyber-Physical Computing Group at UIUC has also given me the opportunity to meet great people throughout the years. We have been close colleagues as well as good friends. Here I would like to give a shout out to Yiran Zhao, Huajie Shao, Shengzhong Liu, Dongxin Liu, Yifan Hao, Tianshi Wang, Jinyang Li, Jiahao Wu, Tai-Sheng Cheng, Yu Shi, Tuo Yu, Bo Chen, Jongdeog Lee, Prasanna Giridhar for all the fun moments and hardworking days that we have shared together.

I would also like to thank the U.S. Army Research Labs, Defense Advanced Research Projects Agency, and National Science Foundation for their financial support and assistance.

Lastly, I would like to thank my parents Junhua Yao and Jianhui Wang, my fiancée Ailing Piao, as well as my entire family, for their unconditional love and support throughout my journey. To them I owe too much, and to them I dedicate this dissertation, humbly.

TABLE OF CONTENTS

CHAPTER 1: INTRODUCTION

The proliferation of internetworked mobile and embedded devices leads to visions of the Internet of Things (IoT), giving rise to a sensor-rich world where physical things in our everyday environment are increasingly enriched with computing, sensing, and communication capabilities. Such capabilities promise to revolutionize the interactions between humans and physical objects.

Indeed, significant research efforts have been spent towards building smarter and more user-friendly applications on mobile and embedded devices and sensors. At the same time, recent advances in deep learning have greatly changed the way that computing devices process human-centric content such as images, video, speech and audio. Applying deep neural networks to IoT devices could thus bring about a generation of applications capable of performing complex sensing and recognition tasks to support a new realm of interactions between humans and their physical surroundings [1]. I discuss five key research questions towards the architecture, performance, predictability and implementation of such novel interactions between humans and (deep-)learning-enabled physical things, namely: First, are current deep neural networks fundamentally well-suited for learning from time-series data collected from physical processes, characteristic to IoT applications? If not, what architectural solutions and foundational building blocks are needed? Second, how to reduce the resource consumption of deep learning models such that they can be efficiently deployed on IoT devices or edge servers? Third, how to minimize the human cost of employing deep learning (namely, the cost of data labeling in IoT applications)? Fourth, how to predict uncertainty in deep learning outputs? Finally, how to design deep learning services that meet responsiveness and quality needed for IoT systems?

To elaborate on the above challenges, first, observe that IoT applications often depend on collaboration among multiple sensors, which requires designing novel neural network for multisensor data fusion collected from physical processes. The design of neural network incorporates two main parts: 1) neural network structure: arranging deep learning components to form an architecture that controls the data learning flow, and 2) fundamental neural network building block: decoding the underlying physical process within input sensing data that facilitates the learning process. Therefore, the structure should be able to model complex interactions among multiple sensory inputs over time; the component should be able to effectively encode features of sensory inputs with underlying physical processes that are pertinent to desired recognition and other tasks. We propose general deep learning architecture and component for this purpose, called DeepSense [2] and STFNet respectively. DeepSense is a unified yet customizable deep learning network structure, customized specifically for learning from multisensor time-series data. DeepSense exploits the nature of multi-sensor measurements, and designs a local-global sensor fusion component that is able to model complex interactions among multiple sensory inputs over time. It demonstrates that certain combinations of deep neural network topologies are particularly well-suited for learning from sensor data, outperforming traditional modeling techniques by a large margin. I also introduced new fundamental neural network building blocks for IoT systems. Conventional neural networks extract features well-suited for *external perceptual* tasks. In contrast, the internal *physical processes* underlying sensor measurements in IoT systems have properties (such as physical inertia, characteristics of wireless signal propagation, and signal resonance) that depend more on signal *frequency*, motivating feature extraction in the *frequency domain*. It is no coincidence that much of classical signal processing literature works by transforming time-series data to the frequency domain first. To help capture signatures of internal physical processes the way a brain captures their externally perceived properties, I developed a novel neural network structure, called Short-Time Fourier Neural Networks (STFNets), that operates directly in the frequency domain. I demonstrated that it is not enough to simply covert sensing signals into the frequency domain, and then apply conventional neural networks. Instead, STFNets make two key additional improvements. First, STFNets leverage and preserve frequency domain semantics that encode time and frequency information. Second, STFNets offer novel multi-resolution processing to circumvent the uncertainty principle in time-frequency analysis. Extensive experiments demonstrated that STFNets bring significant improvements for diverse sensing modalities, including motion sensor, WiFi, ultrasound, and visible light.

Second, IoT devices are usually low-end systems with limited computational, energy, and memory resources. One key impediment in deploying deep neural networks on IoT devices therefore lies in the high resource demand of trained deep neural network models. While existing neural network compression algorithms can effectively reduce the number of model parameters, not all of these models lead to matrix representations that can be efficiently implemented on commodity IoT devices. My study DeepIoT proposes a particularly effective deep learning compression algorithm that can directly compress the structures of commonly used deep neural networks [3]. DeepIoT compresses neural network structures into smaller dense matrices (instead of large sparse matrices) by finding the minimum number of non-redundant hidden elements, while keeping the performance of IoT applications almost the same. Importantly, it does so using an approach that obtains a global view of parameter redundancies with reinforcement learning, which is shown to produce superior compression. My recent study FastDeepIoT further reveals that changing neural network size (or FLOPs) does not proportionally affect the neural network execution time [4]. Rather, extreme runtime nonlinearities exist over the network configuration space. FastDeepIoT automatically uncovers the non-linear relation between neural network structure and execution time, then exploits that understanding to find network configurations that further significantly improve the trade-off between execution time and accuracy on IoT devices.

Third, although IoT devices can generate a great amount of data, labeling data for learning purposes is time-consuming. One must teach sensing devices to recognize objects and concepts without the benefit of (many) examples, where ground truth values for such objects and concepts are given. My work on SenseGAN describes a semi-supervised deep learning framework that can leverage abundant unlabelled sensing data thereby minimizing the need for labelling effort [5]. SenseGAN designs an adversarial game among three components; a classifier, generator, and discriminator that can mutually boost their performance, which helps the classifier learn to predict correct labels from unlabelled data. SenseGAN effectively handles multimodal sensing inputs and easily stabilizes the adversarial training process, which helps improve the performance of the classifier. SenseGAN demonstrates the possibility of learning with limited labeled (and mostly unlabeled) samples, while approaching the performance of learning from fully labeled data, for future almost self-learning IoT applications.

Fourth, other than effectiveness and efficiency, reliability assurances are also important in cyber-physical and IoT applications. The need for offering such assurances calls for well-calibrated estimation of uncertainty associated with learning results. My work RDeepSense proposes the first deep learning model that provides well-calibrated uncertainty estimations for resource-constrained IoT device [6, 7]. RDeepSense estimates uncertainty by adopting a tunable proper scoring rule as the training criterion and dropout as the implicit Bayesian approximation, which theoretically proves its correctness. To reduce the computational complexity, RDeepSense employs efficient dropout and predictive distribution estimation for inference operations.

Finally, besides exploiting the efficiency of neural network models, we also explore the design of the execution environment of deep learning services. With the growing desire to endow everyday IoT devices with advanced interaction capabilities, a increasingly large proportion of machine intelligences will be offloaded to cloud or edge servers. As a complement to system-efficient deep neural network, my work RTDeepIoT proposes the first real-time scheduling pipeline for deep neural networks, maintaining the responsiveness of deep learning services on cloud or edge servers while maximizing service quality. For the same deep learning service with different input samples, the complexities for achieving the same-quality inference are different. RTDeepIoT exploits such data-dependent variance by setting neural network execution depth as another scheduling parameter. RTDeepIoT proposes a run-time scheduler for the server accordingly 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.

I elaborate on these core problems and their emerging solutions to help lay a foundation for building IoT systems enriched with effective, efficient, and reliable deep learning models.

CHAPTER 2: RESEARCH OVERVIEW

In this section, we provide an overview of the challenges and our solutions in the path of building the effective, efficient, reliable, and label-efficient deep learning models in IoT systems.

2.1 DEEP LEARNING FOR SENSOR-RICH IOT SYSTEMS

A key research challenge towards the realization of learning-enabled IoT systems lies in the design of deep neural network structures and basic building blocks that can effectively estimate outputs of interest from noisy time-series multi-sensor measurements.

Despite the large variety of embedded and mobile computing tasks in IoT contexts, one can generally categorize them into two common subtypes: estimation tasks and classification tasks, depending on whether prediction results are continuous or categorical, respectively. The question therefore becomes whether or not a general neural network architecture exists that can effectively learn the structure of models needed for estimation and classification tasks from sensor data. Such a general deep learning neural network architecture would, in principle, overcome disadvantages of today's approaches that are based on analytical model simplifications or the use of hand-crafted engineered features.

Traditionally, for estimation-oriented problems, such as tracking and localization, sensor inputs are processed based on the physical models of the phenomena involved. Sensors generate measurements of physical quantities such as acceleration and angular velocity. From these measurements, other physical quantities are derived (such as displacement through double integration of acceleration over time). However, measurements of commodity sensors are noisy. The noise in measurements is nonlinear and may be correlated over time, which makes it hard to model. It is therefore challenging to separate signal from noise, leading to estimation errors and bias.

For classification-oriented problems, such as activity and context recognition, a typical approach is to compute appropriate features derived from raw sensor data. These hand-crafted features are then fed into a classifier for training. Designing good hand-crafted features can be time consuming; it requires extensive experiments to generalize well to diverse settings such as different sensor noise patterns and heterogeneous user behaviors.

A general deep learning framework can effectively address both of the aforementioned challenges by automatically adapting the learned neural network to complex correlated noise patterns, while at the same time converging on the extraction of maximally robust signal features that are most suited for the task at hand. Our recent framework, called DeepSense, demonstrates a case for feasibility of such a general solution. DeepSense integrates convolutional neural networks (CNN) and recurrent neural networks (RNN). Sensory inputs are aligned and divided into time intervals for processing time-series data. For each interval, DeepSense first applies an individual CNN to each sensor, encoding relevant local features within the sensor's data stream. Then, a (global) CNN is applied on the respective outputs to model interactions among multiple sensors for effective sensor fusion. Next, an RNN is applied to extract temporal patterns. At last, either an affine transformation or a softmax output is used, depending on whether we want to model an estimation or a classification task.

This architecture solves the general problem of learning multi-sensor fusion tasks for purposes of estimation or classification from time-series data. For estimation-oriented problems, DeepSense learns the physical system and noise models to yield outputs from noisy sensor data directly. The neural network acts as an approximate transfer function. For classification-oriented problems, the neural network acts as an automatic feature extractor encoding local, global, and temporal information. As a unified model, DeepSense can be easily customized for a specific IoT application. The application designer needs only to decide on the number of sensory inputs, input/output dimensions, and the training objective function.

Motivated by the needs of IoT applications, this dissertation also presents a principled way of designing deep neural networks that learn (from IoT sensing signals) features inspired by the fundamental properties of the underlying domain of measurements; namely, properties of physical signals. We refer by IoT applications to those where sensors measure some physical quantities, generating (possibly complex and multi-dimentional) time-series data, typically reflecting some underlying physical process. The human brain (whose wiring inspires the structure of conventional neural networks) extracts features well-suited for *external perceptual* tasks, which explains the great success of such networks at those tasks. In contrast, the internal *physical processes* underlying sensor measurements in IoT systems have properties (such as physical intertia, characteristics of wireless signal propagation, and signal resonance) that depend more on signal *frequency*, motivating feature extraction in the *frequency domain*. It is no coincidence that much of classical signal processing literature works by transforming time-series data to the frequency domain first. To help capture signatures of internal physical processes the way a brain captures their externally perceived properties, we develop a new neural network block designed specifically for learning in the frequency domain.

The design of neural network structures greatly influences efficiency of signal modelling and ease of extraction of hidden patterns. Convolutional neural networks (CNNs) for image recognition, for example, align perfectly with biological studies of the visual cortex [8] and with domain knowledge in digital image processing [9]. We thus ask a fundamental question: what structures are well-suited for the domain of physical sensor measurements, which we henceforth call the domain of IoT?

Previous research on customizing deep learning models to the needs of IoT applications [1, 2, 10] mainly focused on designing neural network structures that integrate conventional deep learning components, such as convolutional and recurrent layers, to extract spatial and temporal properties of inputs. On the other hand, since the physics of measured phenomena are best expressed in the frequency domain, decades of research on signal processing developed powerful techniques for time-frequency analysis of signals, including motion sensor signals [11, 12], radio frequency signals [13, 14], acoustic signals [15, 16], and visible light signals [17]. A popular transform that maps time-series measurements to the frequency domain is the Short-Time Fourier Transform (STFT). We, therefore, propose a new neural network model, namely, Short-Time Fourier Neural Networks (STFNets) that operate directly in the frequency domain.

One potential approach for learning in the frequency domain might simply be to convert sensing signals into the frequency domain first, and then apply conventional neural network components, possibly extending them to support operations on complex-numbers so they can represent frequency-domain quantities [18]. These approaches miss two key opportunities for improvement, described below, that we take advantage of in this work. As a result, our work leads to more accurate results, as shown in the evaluation section. The two reasons that account for our improvements are as follows.

First, different from traditional neural networks, where the internal representations constitute features with no physical meaning, the internal representations in STFNet leverage frequency domain semantics that encode time and frequency information. All operations and learnable parameters we propose are explicitly made compatible with the basic properties of spectral data, and align corresponding frequency and time components. In our design, we categorize spectral manipulations into three main types: filtering, convolution, and pooling. Filtering refers to the general spectral filtering and global template matching operation; convolution refers to the local motif detection including shift detection and local template detection; and pooling refers to dimension reduction over the frequency domain. We then design the spectral-compatible parameters and operating rules for these three manipulation categories respectively, which have shown superior performance in evaluations compared to the application of *conventional* neural networks in the domain of complex-numbers.

Second, transforming signals to the frequency domain is governed by the *uncertainty prin-*

ciple [19]. The transformed representation cannot achieve both a high frequency resolution and a high time resolution at the same time. In STFT, the time-frequency resolution is controlled by the length of the sliding window (the length of the part of the time-series being converted at a time). With a longer window, we can obtain a finer-grained frequency representation. However, we then cannot achieve a time resolution smaller than the window size. The uncertainty principle causes a dilemma in traditional time-frequency analysis. One often needs to guess the best time-frequency resolution using trial and error. In STFNet, we circumvent this dilemma by simultaneously computing multiple STFTs with different time-frequency resolutions. The representations with different time-frequency resolutions are then mutually enhanced in a data-driven manner. The network then automatically learns the best resolution or resolutions, where the most useful features are present. STFNet defines a formal way to extract features from multiple time-frequency transformations with the same set of spectral-compatible operations and parameters, which greatly reduces model complexity while improving accuracy.

Broadly speaking, the main contributions of STFNet to the general research landscape of deep learning and IoT are twofold:

1. STFNet presents a principled way of designing neural networks that reveal the key properties of physical processes underlying the sensing signals from the time-frequency perspective.

2. STFNet unveils the benefit of incorporating domain-specific analytic modelling and transformation techniques into the neural network design.

2.2 DEEP LEARNING FOR RESOURCE-CONSTRAINED IOT SYSTEMS

Resource constraints of IoT devices remain an important impediment towards deploying deep learning models. A key question is therefore whether or not it is possible to compress deep neural networks, such as those described in the previous section, to a point where they fit comfortably on low-end embedded devives, enabling real-time “intelligent” interactions with their environment. Can a unified approach compress commonly used deep learning structures, including fully-connected, convolutional, and recurrent neural networks, as well as their combinations? To what degree does the resulting compression reduce energy, execution time, and memory needs in practice?

We prosed such a compression framework, called DeepIoT, that compresses commonly used deep neural network structures for sensing applications through deciding the minimum number of elements in each layer. Previous illuminating studies on neural network compression sparsify large dense parameter matrices into large sparse matrices [20]. In contrast, DeepIoT minimizes the number of elements in each layer, which results in converting parameters into a set of small dense matrices. A small dense matrix does not require additional storage for element indices and is efficiently optimized for processing. DeepIoT greatly reduces the effort of designing efficient neural structures for sensing applications by deciding the number of elements in each layer in a manner informed by the topology of the neural network.

DeepIoT borrows the idea of dropping hidden elements from a widely-used deep learning regularization method called *dropout*. The dropout operation gives each hidden element a dropout probability. During the dropout process, hidden elements can be pruned according to their dropout probabilities. A "thinned" network structure can thus be generated. The challenge is to set these dropout probabilities in an informed manner to generate the optimal slim network structure that preserves the accuracy of sensing applications while maximally reducing their resource consumption. An important purpose of DeepIoT is thus to find the optimal dropout probability for each hidden element in the neural network.

To obtain the optimal dropout probabilities for nodes in the neural network, DeepIoT exploits the network parameters themselves. From the perspective of model compression, an element that is more redundant should have a higher probability to be dropped. A contribution of DeepIoT lies in exploiting a novel compressor neural network to solve this problem. It takes model parameters of each layer as input, learns parameter redundancies, and generates the dropout probabilities accordingly. The compressor neural network is optimized jointly with the original neural network to be compressed in an iterative manner that tries to minimize the loss function of the original IoT application.

DeepIoT greatly reduces the size of model parameters, and speeds up the execution time by getting rid of the inefficient sparse matrix multiplication. However, a formal way to explore the neural network structure design and underlying system efficiency is still unclear. Most manually designed time-efficient neural network structures for mobile devices use parameter size or FLOPs (floating point operations) as the indicator of model execution time [21–23]. Even the official TensorFlow website recommends to use the total number of floating number operations (FLOPs) of neural networks "to make rule-of-thumb estimates of how fast they will run on different devices".[1] However, in practice, counting the number of neural network parameters and the total FLOPs does not lead to good estimates of execution time because the relation between these predictors and execution time is not proportional. We

[1] https://www.tensorflow.org/versions/r1.5/mobile/optimizing

therefore design FastDeepIoT [4], showing how a better understanding of the non-linear relation between neural network structure and performance can further improve execution time and energy consumption without impacting accuracy.

2.3 DEEP LEARNING FOR LABEL-LIMITED IOT SYSTEMS

Labelling data is always time-consuming. This laborious process has become one key factor that hinders researchers and engineers from applying neural networks to sensing and recognition tasks on IoT devices. IoT applications with a large amount of sensing data therefore call for a *semi*-supervised deep learning framework to solve the challenge of limited labelled data.

In attacking this problem, we propose SenseGAN, a semi-supervised deep learning framework for IoT applications [5]. One core feature of SenseGAN is its capability to leverage unlabelled data for training deep learning networks. SenseGAN can run on resource-constrained IoT devices without additional time or energy consumption compared with its supervised counterpart after training on workstations. Specifically, we adopt the idea of enabling a discriminator to differentiate the joint data/label distributions between the real data/label samples and the partially generated data/label samples made by either the generator or the classifier. Such design can easily decouple the functionalities of discriminator and classifier into two separate neural networks. For an IoT application, users can design their own neural network structure for classification and replace the classifier in the SenseGAN framework with users' own design for the purpose of semi-supervised learning. The adversarial game among the discriminator, generator, and classifier mutually enhances the performance of all automatically.

In order to stabilize the semi-supervised learning process of SenseGAN, on one hand, we use the Wasserstein metric, also called Earth mover's distance, as the objective function of the discriminator instead of the Jensen-Shannon divergence that may cause numerical instability [24]. On the other hand, we use the Gumbel-Softmax function for categorical representations to eliminate the non-differentiable problem of traditional categorical variables [25]. These two changes greatly improve training stability as well as the final predictive performance. In addition, we also design the specific neural network structures for the discriminator and generator that are suitable for multimodal sensor inputs from IoT applications.

2.4 DEEP LEARNING FOR RELIABLE IOT SYSTEMS

The next problem concerns the reliability of deep learning models. In particular, how to offer principled uncertainty estimates that can faithfully reflect the correctness of model predictions? Principled uncertainty estimation is critical when deep learning is used to support IoT application that require quantified reliability assurances.

Recent work focused on two related challenges. First, how to develop methods that provide accurate uncertainty estimates in prediction results obtained from deep learning models? Second, how to develop resource-efficient solutions for the uncertainty estimation problem, such that they can be implemented on resource-limited IoT devices?

In this section, we introduce a simple, well-calibrated, and efficient uncertainty estimation algorithm for a multilayer perceptron (MLP), called RDeepSense. RDeepSense enables uncertainty estimation with theoretically proven error bounds for IoT applications.

There are only two steps in computing uncertainty for an arbitrary fully-connected neural network. First, insert dropout operations to each fully-connected layer. Second, adopt a proper scoring rule as the loss function and emit a distribution estimate instead of a point estimate at the output layer.

Intuitively speaking, the dropout operations convert a traditional (deterministic) neural network with parameters into a random Bayesian neural network model with random variables, which equates a neural network to a statistical model. Proper scoring rules (based on the loss function) then measure the accuracy of probabilistic predictions.

The loss function has a large effect on the final results. Taking a regression problem as an example, using the mean square error as the loss function tends to underestimate the uncertainties. This is so because the training process is focused on predicting an accurate mean value without concerning itself with the variance. At the same time, using negative log-likelihood as the loss function tends to overestimate the uncertainties. The reason is that, during the early phase of training a neural network with log-likelihood loss, it is relatively hard to generate an accurate estimate of the mean. Increasing the value of estimated variance can consistently decrease the negative log-likelihood loss with a high probability. Therefore, the predicted uncertainty tends to favor a larger variance that overestimates the true uncertainty.

RDeepSense applies a tuneable function, based on a weighted sum of negative log-likelihood and mean square error, as the loss function. The underestimation effect of mean square error and the overestimation effect of negative log-likelihood are thus balanced by tuning the weighted sum. RDeepSense was shown to generate well-calibrated uncertainty estimates.

Regarding resource efficiency, since RDeepSense emits a distribution estimate instead of a point estimate at the output layer, it can do the uncertainty estimation in a single run. Compared with sampling-based and ensemble-based methods that require running a model k times for k samples, RDeepSense results in much reduced execution time and energy consumption.

2.5 DEEP LEARNING SERVICES ON EDGE/CLOUD

We envision 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 dissertation when referring to "cloud", we mean either cloud or edge.

The intelligent cloud services that offer farms of machine learning algorithms, such as classifiers or predictors. For simplicity, we shall henceforth call them *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 [1]. Besides breakthroughs in processing images and speech using deep learning techniques [26, 27], specific neural network structures have been designed to fuse multiple sensing modularities and extract temporal relationships [2]. The increasing number of studies on applying deep learning in the area of cyber-physical systems (CPS) and IoT [2, 3, 6], motivate us 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.

In this dissertation, we solve the utility *metric* challenge 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) [6]. As we show in this chapter, 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. We then compute a bound on the efficacy of the resulting schedule at global utility maximization subject to real-time deadline constraints.

To this end, we propose 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, we design an efficient approximation of the scheduling algorithm to reduce the scheduling overhead. We implement a user space scheduling framework to verify the effectiveness of RTDeepIoT.

CHAPTER 3: DEEP LEARNING FOR SENSOR-RICH IOT SYSTEMS

In this section, we will first introduce the technical details of the DeepSense framework, a general deep learning architecture for multi-sensor fusion problem. Next, we present the design of STFNet, a novel building block of neural networks for IoT sensing signals.

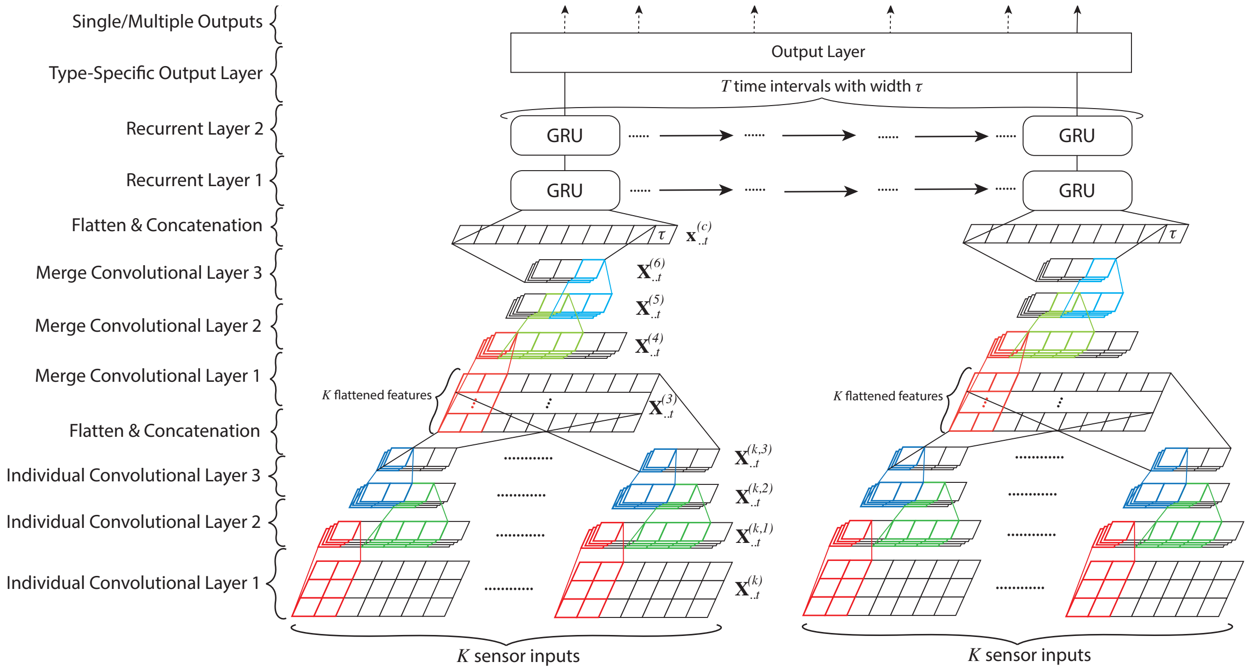


Figure 3.1: Main architecture of the DeepSense framework.

3.1 THE DESIGN OF DEEPSENSE FRAMEWORK

We further separate our description of DeepSense framework into three parts. The first two parts, convolutional layers and recurrent layers, are the main building blocks for DeepSense, which are the same for all applications. The third part, the output layer, is the specific layer for two different types of applications; regression-oriented and classification-oriented.

For the rest of this dissertation, all vectors are denoted by bold lower-case letters (e.g., $\mathbf{x}$ and $\mathbf{y}$), while matrices and tensors are represented by bold upper-case letters (e.g., $\mathbf{X}$ and $\mathbf{Y}$). For a vector $\mathbf{x}$, the j^{th} element is denoted by $\mathbf{x}_{[j]}$. For a tensor $\mathbf{X}$, the t^{th} matrix along the third axis is denoted by $\mathbf{X}_{\cdot\cdot t}$, and other slicing denotations are defined similarly. We use calligraphic letters to denote sets (e.g., $\mathcal{X}$ and $\mathcal{Y}$). For any set $\mathcal{X}$, $|\mathcal{X}|$ denotes the cardinality of $\mathcal{X}$.

For a particular application, we assume that there are K different types of input sensors $\mathcal{S} = \{S_k\}$, $k \in \{1, \cdots, K\}$. Take a sensor S_k as an example. It generates a series of measurements over time. The measurements can be represented by a $d^{(k)} \times n^{(k)}$ matrix $\mathbf{V}$ for measured values and $n^{(k)}$-dimensional vector $\mathbf{u}$ for time stamps, where $d^{(k)}$ is the dimension for each measurement (e.g., measurements along x, y, and z axes for motion sensors) and $n^{(k)}$ is the number of measurements. We split the input measurements $\mathbf{V}$ and $\mathbf{u}$ along time (i.e., columns for $\mathbf{V}$) to generate a series of non-overlapping time intervals with width τ, $\mathcal{W} = \{(\mathbf{V}_t^{(k)}, \mathbf{u}_t^{(k)})\}$, where $|\mathcal{W}| = T$. Note that, τ can be different for different intervals, but here we assume a fixed time interval width for succinctness. We then apply Fourier transform to each element in $\mathcal{W}$, because the frequency domain contains better local frequency patterns that are independent of how time-series data is organized in the time domain. We stack these outputs into a $d^{(k)} \times 2f \times T$ tensor $\mathbf{X}^{(k)}$, where f is the dimension of frequency domain containing f magnitude and phase pairs. The set of resulting tensors for each sensor, $\mathcal{X} = \{\mathbf{X}^{(k)}\}$, is the input of DeepSense.

As shown in Fig. 3.1, DeepSense has three major components; the convolutional layers, the recurrent layers, and the output layer, stacked from bottom to top. In the following subsections, we detail these components, respectively.

3.1.1 Convolutional Layers

The convolutional layers can be further separated into two parts: an individual convolutional subnet for each input sensor tensor $\mathbf{X}^{(k)}$, and a single merge convolutional subnet for the output of K individual convolutional subnets' outputs.

Since the structures of individual convolutional subnet for different sensors are the same, we focus on one individual convolutional subnet with input tensor $\mathbf{X}^{(k)}$. Recall that $\mathbf{X}^{(k)} \in \mathbb{R}^{d^{(k)} \times 2f \times T}$, where $d^{(k)}$ is the sensor measurement dimension, f is the dimension of frequency domain, and T is the number of time intervals. For each time interval t, the matrix $\mathbf{X}_{\cdot\cdot t}^{(k)}$ will be fed into a CNN architecture (with three layers). There are two kinds of features/relationships embedded in $\mathbf{X}_{\cdot\cdot t}^{(k)}$ we want to extract. The relationships within the frequency domain and across sensor measurement dimension. The frequency domain usually contains lots of local patterns in some neighbouring frequencies. And the interaction among sensor measurement usually including all dimensions. Therefore, we first apply 2d filters with shape $(d^{(k)}, cov1)$ to $\mathbf{X}_{\cdot\cdot t}^{(k)}$ to learn interaction among sensor measurement dimensions and local patterns in frequency domain, with the output $\mathbf{X}_{\cdot\cdot t}^{(k,1)}$. Then we apply 1d filters with shape $(1, cov2)$ and $(1, cov3)$ hierarchically to learn high-level relationships, $\mathbf{X}_{\cdot\cdot t}^{(k,2)}$ and $\mathbf{X}_{\cdot\cdot t}^{(k,3)}$.

Then we flatten matrix $\mathbf{X}_{\cdot\cdot t}^{(k,3)}$ into vector $\mathbf{x}_{\cdot\cdot t}^{(k,3)}$ and concat all K vectors $\{\mathbf{x}_{\cdot\cdot t}^{(k,3)}\}$ into a K-row matrix $\mathbf{X}_{\cdot\cdot t}^{(3)}$, which is the input of the merge convolutional subnet. The architecture of the merge convolutional subnet is similar as the individual convolutional subnet. We first apply 2d filters with shape $(K, cov4)$ to learn the interactions among all K sensors, with output $\mathbf{X}_{\cdot\cdot t}^{(4)}$, and then apply 1d filters with shape $(1, cov5)$ and $(1, cov6)$ hierarchically to learn high-level relationships, $\mathbf{X}_{\cdot\cdot t}^{(5)}$ and $\mathbf{X}_{\cdot\cdot t}^{(6)}$.

For each convolutional layer, DeepSense learns 64 filters, and uses ReLU as the activation function. In addition, batch normalization [28] is applied at each layer to reduce internal covariate shift. We do not use residual net structures [29], because we want to simplify the network architecture for mobile applications. Then we flatten the final output $\mathbf{X}_{\cdot\cdot t}^{(6)}$ into vector $\mathbf{x}_{\cdot\cdot t}^{(f)}$; concatenate $\mathbf{x}_{\cdot\cdot t}^{(f)}$ and time interval width, $[\tau]$, together into $\mathbf{x}_{t}^{(c)}$ as inputs of recurrent layers.

3.1.2 Recurrent Layers

Recurrent neural networks are powerful architectures that can approximate function and learn meaningful features for sequences. Original RNNs fall short of learning long-term dependencies. Two extended models are Long Short-Term Memory (LSTM) [30] and Gated Recurrent Unit (GRU) [31]. In this design, we choose GRU, because GRUs show similar performance as LSTMs on various tasks [31], while having a more concise expression, which reduces network complexity for mobile applications.

DeepSense chooses a stacked GRU structure (with two layers). Compared with standard (single-layer) GRUs, stacked GRUs are a more efficient way to increase model capacity [32]. Compared to bidirectional GRUs [33], which contain two time flows from start to end and from end to start, stacked GRUs can run incrementally, when there is a new time interval, resulting in faster processing of stream data. In contrast, we cannot run bidirectional GRUs until data from all time intervals are ready, which is infeasible for applications such as tracking. We apply dropout to the connections between GRU layers [34] for regularization and apply recurrent batch normalization [35] to reduce internal covariate shift among time steps. Inputs $\{\mathbf{x}_{t}^{(c)}\}$ for $t = 1, \cdots, T$ from previous convolutional layers are fed into stacked GRU and generate outputs $\{\mathbf{x}_{t}^{(r)}\}$ for $t = 1, \cdots, T$ as inputs of the final output layer.

3.1.3 Output Layer

The output of recurrent layer is a series of vectors $\{\mathbf{x}_{t}^{(r)}\}$ for $t = 1, \cdots, T$. For the regression-oriented task, since the value of each element in vector $\mathbf{x}_{t}^{(r)}$ is within ±1, $\mathbf{x}_{t}^{(r)}$

encodes the output physical quantities at the end of time interval t. In the output layer, we want to learn a dictionary $\mathbf{W}_{out}$ with a bias term $\mathbf{b}_{out}$ to decode $\mathbf{x}_t^{(r)}$ into $\hat{\mathbf{y}}_t$, such that $\hat{\mathbf{y}}_t = \mathbf{W}_{out} \cdot \mathbf{x}_t^{(r)} + \mathbf{b}_{out}$. Therefore, the output layer is a fully connected layer on the top of each interval with sharing parameter $\mathbf{W}_{out}$ and $\mathbf{b}_{out}$.

For the classification task, $\mathbf{x}_t^{(r)}$ is the feature vector at time interval t. The output layer first needs to compose $\{\mathbf{x}_t^{(r)}\}$ into a fixed-length feature vector for further processing. Averaging features over time is one choice. More sophisticated methods can also be applied to generate the final feature, such as the attention model [36], which has illustrated its effectiveness in various learning tasks recently. The attention model can be viewed as weighted averaging of features over time, but the weights are learnt by neural networks through context. In this design, we still use averaging features over time to generate the final feature, $\mathbf{x}^{(r)} = (\sum_{t=1}^{T} \mathbf{x}_t^{(r)})/T$. Then we feed $\mathbf{x}^{(r)}$ into a softmax layer to generate the predicted category probability $\hat{\mathbf{y}}$.

3.1.4 Task-Specific Customization

In this section, we first describe how to trivially customize the DeepSense framework to different mobile sensing and computing tasks. Next, we instantiate the solution with three specific tasks used in our evaluation.

General Customization Process

In general, we need to customize a few parameters of the main architecture of DeepSense, shown in Section 3.1, for specific mobile sensing and computing tasks. Our general DeepSense customization process is as follows:

1. Identify the number of sensor inputs, K. Pre-process the sensor inputs into a set of tensors $\mathcal{X} = \{\mathbf{X}^{(k)}\}$ as input.

2. Identify the type of the task. Whether the application is regression or classification-oriented. Select one of the two types of output layer according to the type of task.

3. Design a customized cost function or choose the default cost function (namely, mean square error for regression-oriented tasks and cross-entropy error for classification-oriented tasks).

Therefore, if opt for the default DeepSense configuration, we need only to set the number of inputs, K, preprocess the input sensor measurements, and identify the type of task (i.e., regression-oriented versus classification-oriented).

The pre-processing is simple, as stated at the beginning of Section 3.1. We just need to align and chunk the sensor measurements, and apply Fourier transform to each sensor chunk. For each sensor, we stack these frequency domain outputs into $d^{(k)} \times 2f \times T$ tensor $\mathbf{X}^{(k)}$, where $d^{(k)}$ is the sensor measurement dimension, f is the frequency domain dimension, and T is the number of time intervals.

To identify the number of sensor inputs K, we usually set K to be the number of different sensing modalities available. If there exist two or more sensors of the same modality (e.g., two accelerometers or three microphones), we just treat them as one multi-dimensional sensor and set its measurement dimension accordingly.

For the cost function, we can design our own cost function other than the default one. We denote our DeepSense model as function $\mathcal{F}(\cdot)$, and a single training sample pair as $(\mathcal{X}, \mathbf{y})$. We can express the cost function as:

$$\mathcal{L} = \ell(\mathcal{F}(\mathcal{X}), \mathbf{y}) + \sum_j \lambda_j P_j \tag{3.1}$$

where $\ell(\cdot)$ is the loss function, P_j is the penalty or regularization function, and λ_j controls the importance of the penalty or regularization term.

Customize Mobile Sensing Tasks

In this section, we provide three instances of customizing DeepSense for specific IoT applications used in our evaluation.

Car tracking with motion sensors (CarTrack): In this task, we apply accelerator, gyroscope, and magnetometer to track the trajectory of a car without initial speed. Therefore, according to our general customization process, carTrack is a regression-oriented problem with $K = 3$ (i.e. accelerometer, gyroscope, and magnetometer). Instead of applying default mean square error loss function, we design our own cost function according to Equation (3.1).

During the training step, the ground-truth 2D displacement of car in each time interval, $\mathbf{y}$, is obtained by GPS signal, where $\mathbf{y}_{[t]}$ denotes the 2D displacement in time interval t. Yet a problem is that GPS signal also contains noise. Training the DeepSense model to recover the displacement obtained from by GPS signal will generate sub-optimal results. We apply Kalman filter to covert displacement $\mathbf{y}_{[t]}$ into a 2D Gaussian distribution $\mathbf{Y}_{[t]}(\cdot)$ with mean value $\mathbf{y}^{(t)}$ in time interval t. Therefore, we use negative log likelihood as loss function $\ell(\cdot)$

with additional penalty terms:

$$\mathcal{L} = -\log\big(\mathbf{Y}_{[t]}\big(\mathcal{F}(\mathcal{X})_{[t]}\big)\big) + \sum_{t=1}^{T} \lambda \cdot \max\big(0, \cos(\theta) - S_c\big(\mathcal{F}(\mathcal{X})_{[t]}, \mathbf{y}^{(t)}\big)\big) \tag{3.2}$$

where $S_c(\cdot, \cdot)$ denotes the cosine similarity, the first term is the negative log likelihood loss function, and the second term is a penalty term controlled by parameter λ. If the angle between our predicted displacement $\mathcal{F}(\mathcal{X})_{[t]}$ and $\mathbf{y}^{(t)}$ is larger than a pre-defined margin $\theta \in [0, \pi)$, the cost function will get a penalty. We introduce the penalty, because we find that predicting a correct direction is more important during the experiment, as described in Section 3.2.4.

Heterogeneous human activity recognition (HHAR): In this task, we perform leave-one-user-out cross-validation on human activity recognition task with accelerometer and gyroscope measurements. Therefore, according to our general customization process, HHAR is a classification-oriented problem with $K = 2$ (accelerometer and gyroscope). We use the default cross-entropy cost function as the training objective.

$$\mathcal{L} = H(\mathbf{y}, \mathcal{F}(\mathcal{X})) \tag{3.3}$$

where $H(\cdot, \cdot)$ is the cross entropy for two distributions.

User identification with motion analysis (UserID): In this task, we perform user identification with biometric motion analysis. We classify users' identity according to accelerometer and gyroscope measurements. Similarly, according to our general customization process, UserID is a classification-oriented problem with $K = 2$ (accelerometer and gyroscope). Similarly as above, we use the default cross-entropy cost function as the training objective.

3.2 THE EVALUATION OF DEEPSENSE

In this section, we evaluate DeepSense on three mobile computing tasks. We first introduce the experimental setup for each, including datasets and baseline algorithms. We then evaluate the three tasks based on accuracy, energy, and latency. We use the abbreviations, CarTrack (Car Tracking with Motion Sensors), HHAR (Heterogeneous Human Activity Recognition), and UserID (User Identification with Motion Analysis) for the three tasks.

3.2.1 Data Collection and Datasets

For the CarTrack task, we collect 17,500 phone-miles worth of driving data. Namely, we collect around 500 driving hours in total using three cars fitted with 20 mobile phones in the Urbana-Champaign area. Mobile devices include Nexus 5, Nexus 4, Galaxy Nexus, and Nexus S. Each mobile device collects measures of accelerometer, gyroscope, magnetometer, and GPS. GPS measurements are collected roughly every second. Collection rates of other sensors are set to their highest frequency. After obtaining the raw sensor measurements, we first segment them into data samples. Each data sample is a zero-speed to zero-speed journey, where the start and termination are detected when there are at least three consecutive zero GPS speed readings. Each data sample is then separated into time intervals according to the GPS measurements. Hence, every GPS measurement is an indicator of the end of a time interval. In addition, each data sample contains one additional time interval with zero speed at the beginning. Furthermore, for each time interval, GPS latitude and longitude are converted into map coordinates, where the origin of coordinates is the position at the first time interval. Fourier transform is applied to each sensor measurement in each time interval to obtain the frequency response of the three sensing axes. The frequency responses of the accelerator, gyroscope, and magnetometer at each time interval are then composed into the tensors as DeepSense inputs. At last, for evaluation purposes, we apply a Kalman filter to coordinates obtained by the GPS signal, and generate the displacement distribution of each time interval. The results serve as ground truth for training.

For both the HHAR and UserID tasks, we use the dataset collected by Allan et al. [11]. This dataset contains readings from two motion sensors (accelerometer and gyroscope). Readings were recorded when users executed activities scripted in no specific order, while carrying smartwatches and smartphones. The dataset contains 9 users, 6 activities (biking, sitting, standing, walking, climbStair-up, and climbStair-down), and 6 types of mobile devices. For both tasks, accelerometer and gyroscope measurements are model inputs. However, for HHAR, activities are used as labels, and for UserID, users' unique IDs are used as labels. We segment raw measurements into 5-second samples. For DeepSense, each sample is further divided into time intervals of length τ, as shown in Figure 3.1. We take $\tau = 0.25$ s. Then we calculate the frequency response of sensors for each time interval, and compose results from different time intervals into tensors as inputs.

3.2.2 Evaluation Platforms

Our evaluation experiments are conducted on two platforms: Nexus 5 with Qualcomm Snapdragon 800 SoC [37] and Intel Edison Compute Module [38]. We train DeepSense on Desktop with GPU. And trained DeepSense models are run solely on mobile with CPU: quad core 2.3 GHz Krait 400 CPU on Nexus 5 and dual-core 500 MHz Atom processor on Intel Edison. Here, we do not exploit the additional computation power of mobile GPU and DSP units [39].

3.2.3 Baseline Algorithms

We evaluate our DeepSense model and compare it with other competitive algorithms in three tasks. There are three global baselines, which are the variants of DeepSense model by removing one design component in the architecture. The other baselines are specifically designed for each single task.

DS-singleGRU: This model replaces the 2-layer stacked GRU with a single-layer GRU with larger dimension, while keeping the number of parameters. This baseline algorithm is used to verify the efficiency of increasing model capacity by staked recurrent layer.

DS-noIndvConv: In this mode, there are no individual convolutional subnets for each sensor input. Instead, we concatenate the input tensors along the first axis (i.e., the input measurement dimension). Then, for each time interval, we have a single matrix as the input to the merge convolutional subnet directly.

DS-noMergeConv: In this variant, there are no merge convolutional subnets at each time interval. Instead, we flatten the output of each individual convolutional subnet and concatenate them into a single vector as the input of the recurrent layers.

CarTrack Baseline:

- **GPS:** This is a baseline measurement that is specific to the CarTrack problem. It can be viewed as the ground truth for the task, as we do not have other means of more accurately acquiring cars' locations. In the following experiments, we use the GPS module in Qualcomm Snapdragon 800 SoC.
- **Sensor-fusion:** This is a sensor fusion based algorithm. It combines gyroscope and accelerometer measurements to obtain the pure acceleration without gravity. It uses accelerometer, gyroscope, and magnetometer to obtain absolute rotation calibration. Android phones have proprietary solutions for these two functions [40]. The algorithm then applies double integration on pure acceleration with absolute rotation calibration to obtain the displacement.

- **eNav (w/o GPS):** eNav is a map-aided car tracking algorithm [41]. This algorithm constrains the car movement path according to a digital map, and computes moving distance along the path using double integration of acceleration derived using principal component analysis that removes gravity. The original eNav uses GPS when it believes that dead-reckoning error is high. For fairness, we modified eNav to disable GPS.

HHAR Baselines:

- **HAR-RF:** This algorithm [11] selects all popular time-domain and frequency domain features from [42] and ECDF features from [43], and uses random forest as classifier.
- **HAR-SVM:** Feature selection of this model is same as the HAR-RF model. But this model uses support vector machine as classifier [11].
- **HRA-RBM:** This model is based on stacked restricted Boltzmann machines with frequency domain representations as inputs [44].
- **HRA-MultiRBM:** For each sensor input, the model processes it with a single stacked restricted Boltzmann machine. Then it uses another stacked restricted Boltzmann machine to merge the results for activity recognition [45].

UserID Baselines:

- **GaitID:** This model extracts the gait template and identifies user through template matching with support vector machine [46].
- **IDNet:** This model first extracts the gait template, and extracts template features with convolutional neural networks. Then this model identifies user through support vector machine and integrates multiple verifications with Wald's probability ratio test [47].

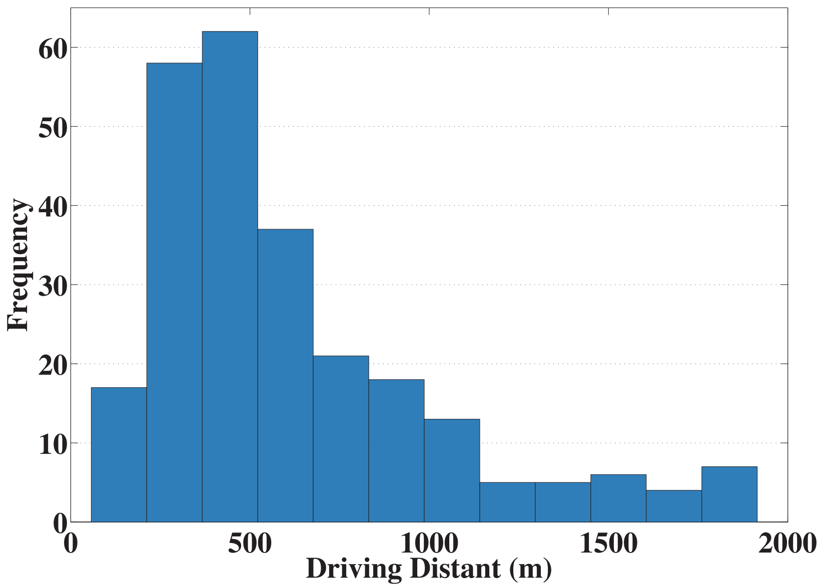


Figure 3.2: Histogram of Driving Distance.

3.2.4 Car Tracking with Motion Sensors

We use 253 zero-speed to zero-speed car driving examples to evaluate the CarTrack task. The histogram of evaluation data driving distance is illustrated in Fig. 3.2.

During the whole evaluation, we regard filtered GPS signal as ground truth. CarTrack is a regression problem. Therefore, we first evaluate all algorithms with mean absolute error (MAE) between predicted and true final displacements with 95% confidence interval except for the eNav (w/o GPS) algorithm, which is a map-aided algorithm without tracking real trajectories. The results about mean absolute errors are illustrated in the second column of Table 3.1.

Compared with senior-fusion algorithm, DeepSense reduces the tracking error by an order of magnitude, which is mainly attributed to its capability to learn the composition of noise model and physical laws. Then, we compare our DeepSense model with three variants as mentioned before. The results show the effectiveness of each designing component of our DeepSense model. The individual and merge convolutional subnets learn the interaction within and among sensor measurements respectively. The stacked recurrent structure increases the capacity of model more efficiently. Removing any component will cause performance degradation.

DeepSense model achieves $40.43 \pm 5.24m$ mean absolute error. This is almost equivalent to half of traditional city blocks ($80m \times 80m$), which means that, with the aid of map and the assumption that car is driving on roads, DeepSense model has a high probability to provide accurate trajectory tracking. Therefore, we propose a naive map-aided track method here. For each segment of original tracking trajectory, we assign them to the most probable road segment on map (i.e., the nearest road segment on map). We then compare the resulted trajectory with ground truth. If all the trajectory segments are the same as the ground truth, we regard it as a successful tracking trajectory. Finally, we compute the percentage of successful tracking trajectories as accuracy. eNav (w/o GPS) is a map-aided algorithm, so we directly compare the trajectory segments. Sensor-fusion algorithm generates tracking errors that are comparable to driving distances, so we exclude it from the comparison. We show the accuracy of map-aided versions of algorithms in the third column of Table 3.1. DeepSense outperforms eNav (w/o GPS) with a large margin, because eNav (w/o GPS) intrinsically depends on occasional on-demand GPS samples to correct tracking error.

We next examine how tracking performance is affected by driving distances. We first sort all evaluation samples according to driving distance. Then we separate them into 10 groups with 200m step size. Finally, we compute mean absolute error and accuracy of map-aided track for DeepSense algorithm separately for each group. We illustrate the results in Fig. 3.3.

Table 3.1: CarTrack Task Accuracy

	MAE (meter)	Map-Aided Accuracy
DeepSense	**40.43 ± 5.24**	**93.8%**
DS-SingleGRU	44.97 ± 5.80	90.2%
DS-noIndvConv	52.15 ± 6.24	88.3%
DS-noMergeConv	53.06 ± 6.59	87.5%
Sensor-fusion	606.59 ± 56.57	
eNav (w/o GPS)		6.7%

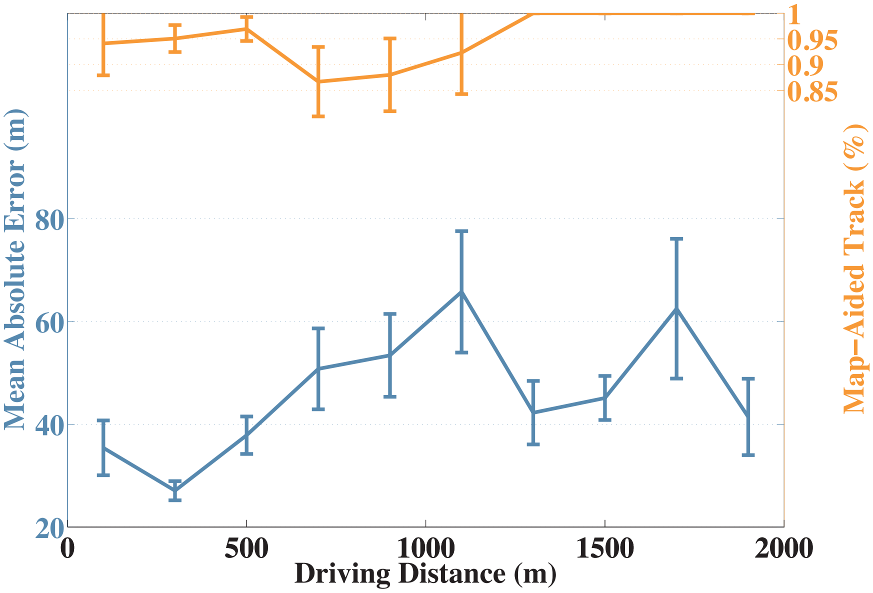


Figure 3.3: Performance over driving distance.

For the mean absolute error metric, driving longer distance generally results in large error, *but the error does not accumulate linearly over distance.* There are mainly two reasons for this phenomenon. On one hand, we observe that the error of our predicted trajectory usually occurs during the beginning of the driving, where uncertainty in predicting driving direction is the major cause. This is also the motivation that we add the penalty term for cost function in Section 3.1.4. On the other hand, longer-driving cases in our testing samples are more stable, because we extract the trajectory from zero-speed to zero-speed. For the map-aided track, longer driving distances even yields slightly better accuracy. This is because long-distance trajectory usually contains long trajectory segments, which can help to find the ground truth on the map.

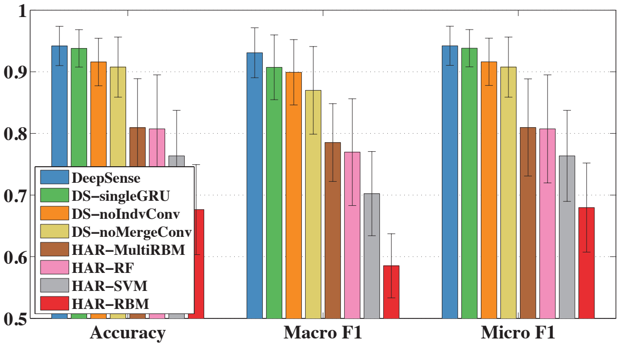


Figure 3.4: Performance metrics of HHAR task.

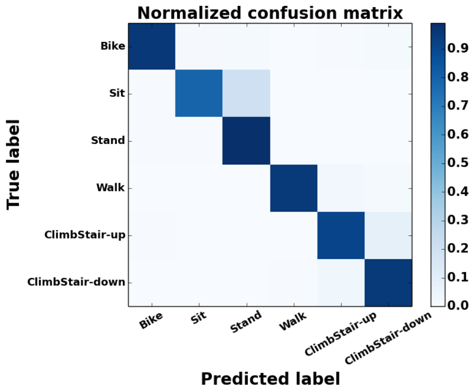


Figure 3.5: Confusion matrix of HHAR task.

3.2.5 Heterogeneous Human Activity Recognition

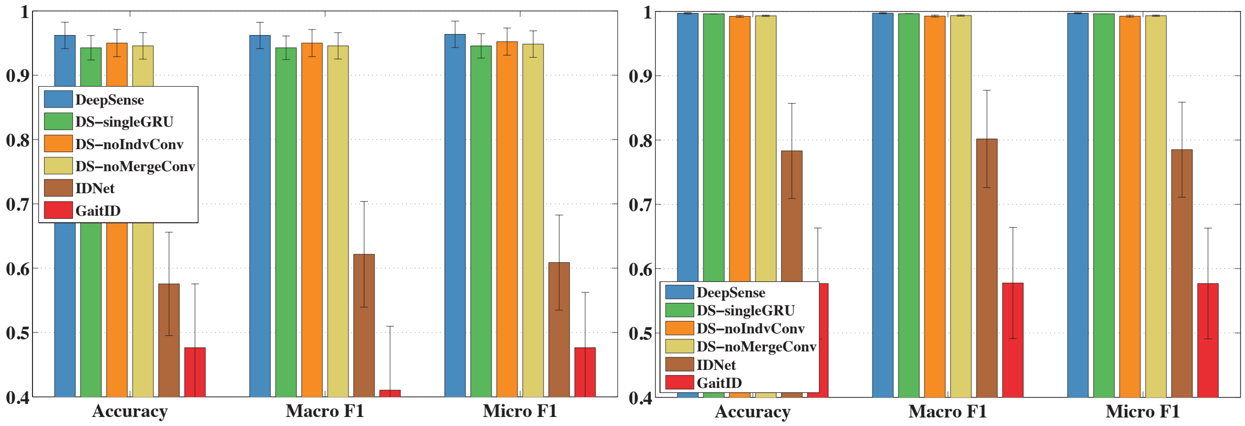
For HHAR task, we perform leave-one-user-out evaluation (i.e., leaving the whole data from one user as testing data) on datasets consisting of 9 users, which are labelled from a to i. We illustrate the result of evaluations according to three metrics: accuracy, macro F_1 score, and micro F_1 score with 95% confidence interval in Fig. 3.4.

The DeepSense based algorithms (including DeepSense and three variants) outperform other baseline algorithms with a large margin (i.e., at least 10%). Compared with two hand-crafted feature based algorithms HAR-RF and HAR-SVM, DeepSense model can automatically extract more robust features, which generalize better to the user who does not appear in the training set. Compared with a deep model, such as HAR-RBM and HAR-MultiRBM, DeepSense model exploit local structures within sensor measurements, dependency along time, and relationships among multiple sensors to generate better and more robust features from data. Compared with three variants, DeepSense still achieves the best performance (accuracy: 0.942 ± 0.032, macro F_1: 0.931 ± 0.041, and micro F_1: 0.942 ± 0.032). This reinforces the effectiveness of our design components in DeepSense model.

Then we illustrate the confusion matrix of best-performing DeepSense model in Fig. 3.5. Predicting *Sit* as *Stand* is the largest error. It is hard to classify these two, because two activities should have similar motion sensor measurements by nature, especially when we have no prior information about testing users. In addition, the algorithm has a minor error about misclassification between *ClimbStair-up* and *ClimbStair-down*.

3.2.6 User Identification with Motion Analysis

This task focuses on user identification with biometric motion analysis. We evaluate all algorithms with 10-fold cross validation. We illustrate the result of evaluations according to



(a) 5 time intervals: 1.25s

(b) 20 time intervals: 5s

Figure 3.6: Performance metrics of UserID task.

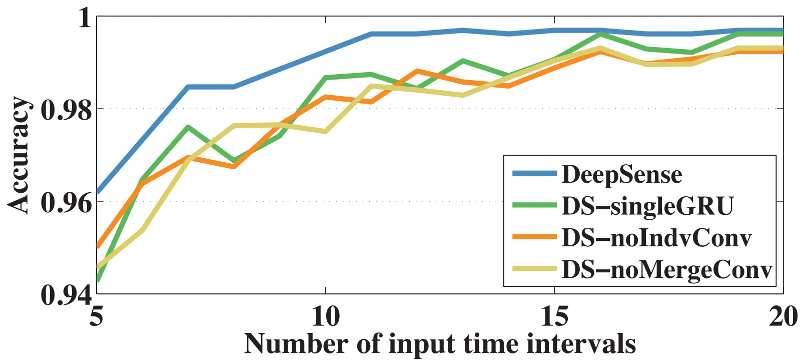


Figure 3.7: Accuracy over input measurement length of UserID task.

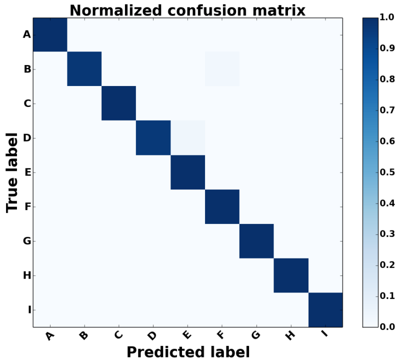


Figure 3.8: Confusion matrix of UserID task.

three metrics: accuracy, macro F_1 score, and micro F_1 score with 95% confidence interval in Fig. 3.6. Specifically, Fig. 3.6a shows the results when algorithms observe 1.25 seconds of evaluation data, Fig. 3.6b shows the results when algorithms observe 5 seconds of evaluation data.

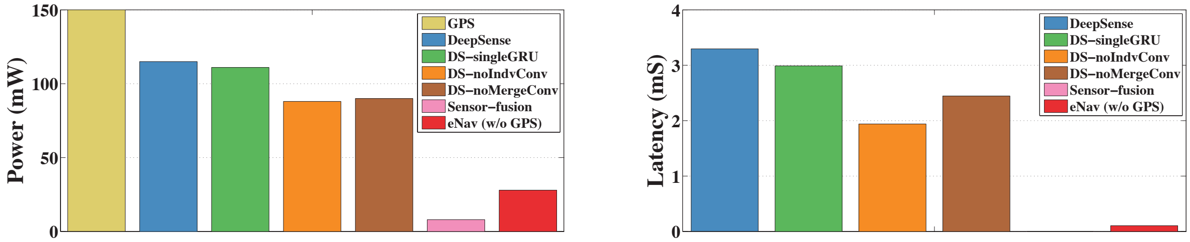
DeepSense and three variants outperform other baseline algorithms with a large margin again (i.e. at least 20%). Compared with the template extraction and matching method, GaitID, DeepSense model can automatically extract distinct features from data, which fit well to not only walking but also all other kinds of activities. Compared with method that first extracts templates and then apply neural network to learn features, IDNet, DeepSense solves the whole task in the end-to-end fashion. We eliminate the manually processing part and exploit local, global, and temporal relationships through our architecture, which results better performance. In this task, although the performance of different variants is similar when observing data with 5 seconds, DeepSense still achieves the best performance (accuracy: 0.997 ± 0.001, macro F_1: 0.997 ± 0.001, and micro F_1: 0.997 ± 0.001).

We further compare DeepSense with three variants by changing the number of evaluation time intervals from 5 to 20, which corresponds to around 1 to 5 seconds. We compute the accuracy for each case. The results illustrated in Fig. 3.7 suggest that DeepSense performs better than all the other variants with a relatively large margin when algorithms observe sensing data with shorter time. This indicates the effectiveness of design components in DeepSense.

Then we illustrate the confusion matrix of best-performing DeepSense model when observing sensing data with 5 seconds in Fig. 3.8. It shows that the algorithm gives a pretty good result. On average, only about two misclassifications appear during each testing.

3.2.7 Latency and Energy

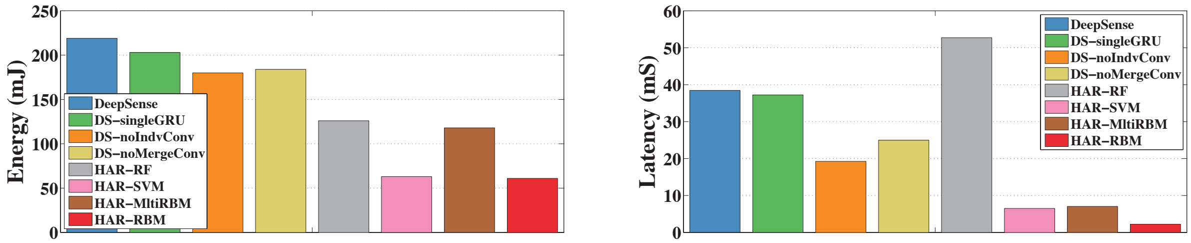
Final, we examine the computation latency and energy consumption of DeepSense—stereotypical deep learning models are traditionally power hungry and time consuming—we illustrate, through our careful measurements in all three example application scenarios, the feasibility of directly implementing and deploying DeepSense on mobile devices without any additional optimization.



(a) Power

(b) Latency

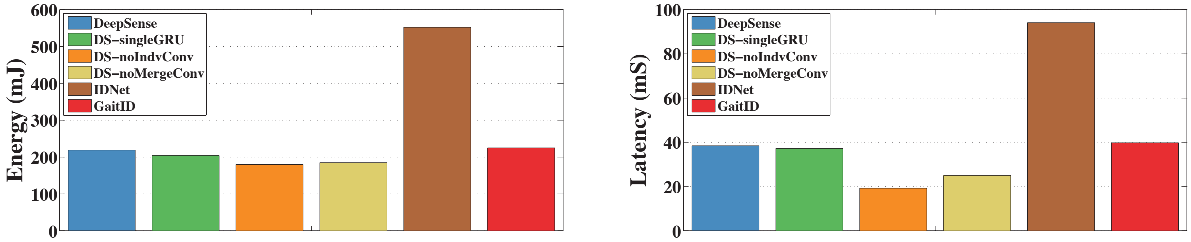
Figure 3.9: Power and Latency of carTrack solutions on Nexus 5



(a) Energy

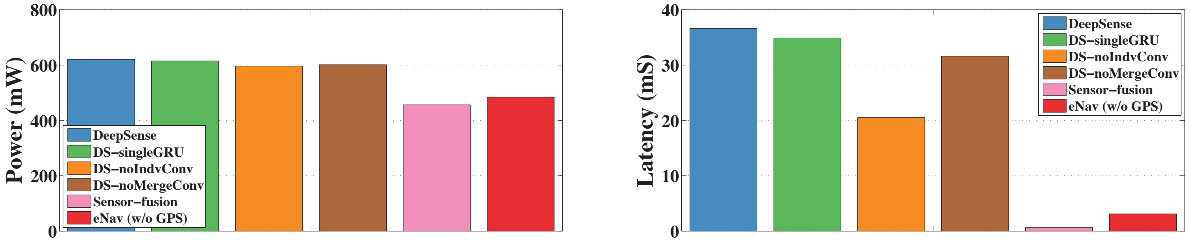
(b) Latency

Figure 3.10: Energy and Latency of HHAR solutions on Nexus 5



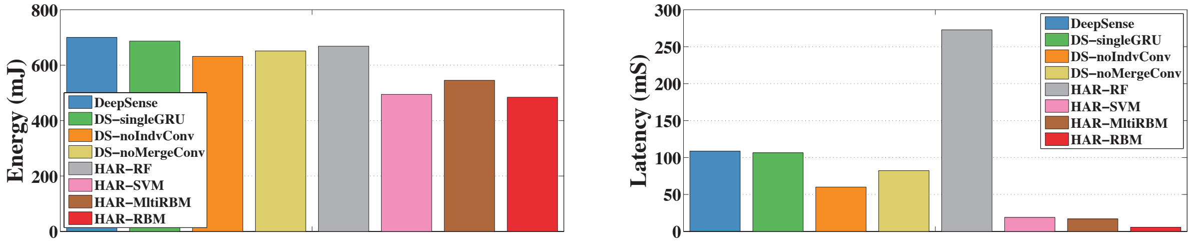
(a) Energy (b) Latency

Figure 3.11: Energy and Latency of UserID solutions on Nexus 5



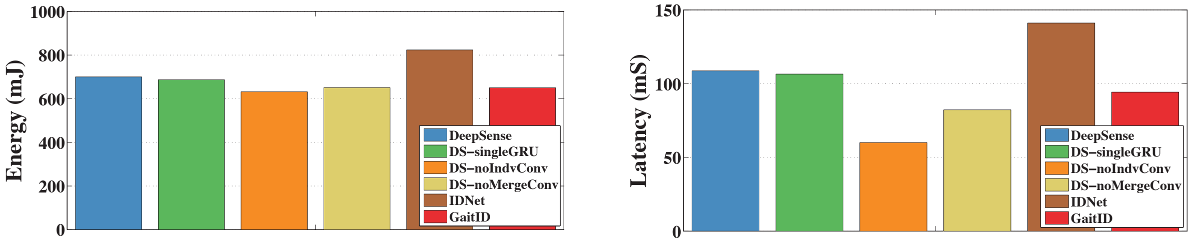
(a) Power (b) Latency

Figure 3.12: Power and Latency of carTrack solutions on Edison



(a) Energy (b) Latency

Figure 3.13: Energy and Latency of HHAR solutions on Edison



(a) Energy (b) Latency

Figure 3.14: Energy and Latency of UserID solutions on Edison

Experiments measure the whole process on smart devices including reading the raw sensor inputs and are conducted on two kinds of devices: Nexus 5 and Intel Edison. The energy consumption of applications on Nexus 5 is measured by PowerTutor [48], while the energy consumption of Intel Edison is measured by an external power monitor. The evaluations of energy and latency on Nexus 5 are shown in Fig. 3.9, 3.10, and 3.11, and Intel Edison Fig. 3.12, 3.13, and 3.14. Since algorithms for carTrack are designed to report position every second, we show the power consumption in Fig. 3.9a and 3.12a. Other two tasks are not periodical tasks by nature. Therefore, we show the per-inference energy consumption in Fig. 3.10a, 3.13a, 3.11a, and 3.14a. For experiments on Intel Edison, notice that we measured total energy consumption, containing 419mW idle-mode power consumption.

For the carTrack task, all DeepSense based models consume a bit less energy compared with 1-Hz GPS samplings on Nexus 5. The running times are measured in the order of microsecond on both platforms, which meets the requirement of per-second measurement.

For the HHAR task, all DeepSense based models take moderate energy and low latency to obtain one classification prediction on two platforms. An interesting observation is that HHAR-RF, a random forest model, has a relatively longer latency. This is due to the fact that random forest is an ensemble method, which involves combining a bag of individual decision tree classifiers.

For the UserID task, except for the IDNet baseline, all other algorithms show similar running time and energy consumption on two platforms. IDNet contains both a multi-stage pre-processing process and a relative large CNN, which takes longer time and more energy to compute in total.

3.3 THE DESIGN OF STFNET

We introduce the technical details of STFNets in this section. We separate the technical descriptions into six parts. In the first two subsections, we provide some background followed by a high-level overview of STFNet components, including (i) hologram interleaving, (ii) STFNet-filtering, (iii) STFNet-convolution, and (iv) STFNet-pooling. In the remaining four subsections, we describe the technical details of each of these components, respectively.

3.3.1 Background and STFNet Overview

IoT devices sample the physical environment generating time-series data. Discrete Fourier Transform (DFT) is a mathematical tool that converts n samples over time (with a sampling rate of f_s) into a n components in frequency (with a frequency step of f_s/n). The more samples are selected, the finer the component resolution is in frequency. We can always transform the whole sequence of data with DFT, achieving a high frequency resolution. However, we then lose information on signal evolution over time, or the time resolution. In order to solve this problem, Short-Time Fourier Transform (STFT) divides a longer time

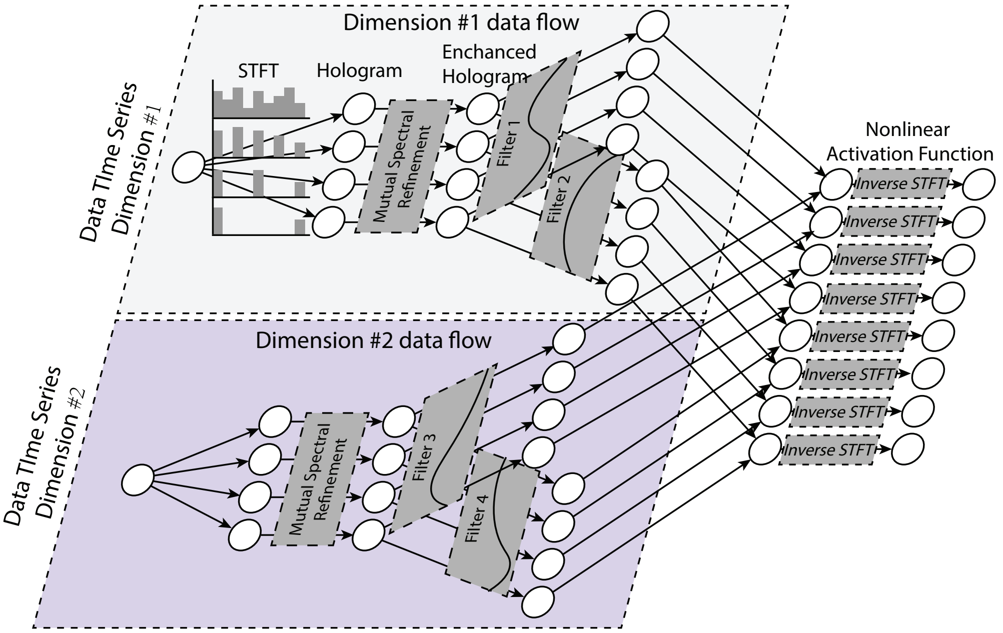


Figure 3.15: Data Flow within a Block of STFNet.

signal into shorter segments of equal length and computes DTF separately on each shorter segment. By losing a certain degree of frequency resolution, STFT helps us regain the time resolution to some extent. In choosing n, there arises a fundamental trade-off between the attainable time and frequency resolution, which is called the *uncertainty principle* [19]. For the purposes of learning to pedict a given output, the optimal trade-off point depends on the time and frequency granularity of the features that best determine the outputs we want to reproduce. The goal of STFNets is thus to learn frequency domain features that predict the output, while at the same time learn the best resolution trade-off point in which the relevant features exist.

The building component of an STFNet is an *STFNet block*, shown in Figure 3.16. An STFNet block is the layer-equivalent in our neural network. The larger network would normally be composed by stacking such layers. Within each block, STFNet circumvents the uncertainty principle by computing multiple STFT representations with different time-frequency resolutions. Collectively, these representations constitute what we call the *time-frequency hologram*. And we call an individual time-frequency signal representation, a hologram representation. They are then used to mutually enhance each other by filling-in missing frequency components in each.

Candidate frequency-domain features are then extracted from these enhanced representations via general spectral manipulations that come in two flavors; filtering and convolution. They represent global and local feature extraction operations, respectively. The filtering and convolution kernels are learnable, making each STFNet layer a building block for spectral manipulation and learnable frequency domain feature extraction. In addition, we also design a new mechanism, called pooling, for frequency domain dimensionality reduction in STFNets. Combinations of features extracted using the above manipulations then pass through activation functions and an inverse STFT transform to produce (filtered) outputs in the time domain. Stacking STFNet blocks has the effect of producing progressively sharper (i.e., higher order) filters to shape the frequency domain signal representation into more relevant and more fine-tuned features.

Figure 3.15 gives an example of an SFTNet block that accepts as input a two-dimensional time-series signal (e.g., 2D accelerometers data). Each dimension is then transformed to the frequency domain at four different resolutions using STFT, generating four different internal nodes, each representing the signal in the frequency domain at a different time-frequency resolution. Collectively, the four representations constitute the hologram. In the next step, mutual enhancements are done improving all representations. Each representation then undergoes a variety of alternative spectral manipulations (called "filters" in the figure). Two filters are shown in the figure for each dimension. The parameters of these filters are the weights multiplied by the frequency components of the filter input; a different weight per component. These parameters are what the network learns. Note that, a filter does not change the time-frequency resolution of the corresponding input. Filter outputs of the same time-frequency resolution are then combined additively across all dimensions and passed through a non-linear activation function (as in a conventional convolutional neural network). An inverse STFT brings each such combined output back to the time domain, where it becomes an input to the next STFNet block. (Alternatively, the inverse STFT can be applied after dimension combination and before the activation function.) Hence, each output time-series is produced by applying spectral manipulation and fusion to one particular time-frequency resolution of all input time-series. Once converted to the time domain, however, the output time-series can be resampled in the next block at different time-frequency resolutions again. The goal of STFNet is to learn the weighting of different frequency components within each filter in each block such that features are produced that best predict final network outputs.

3.3.2 STFNet Block Fundamentals

In this subsection, we introduce the formulation of our design elements wihin each STFNet block.

We denote the input to the STFNet block as $\mathbf{X} \in \mathbb{R}^{T \times D}$, where we divide the input D-dimension time-series into windows of size T samples. We call T the signal length and D

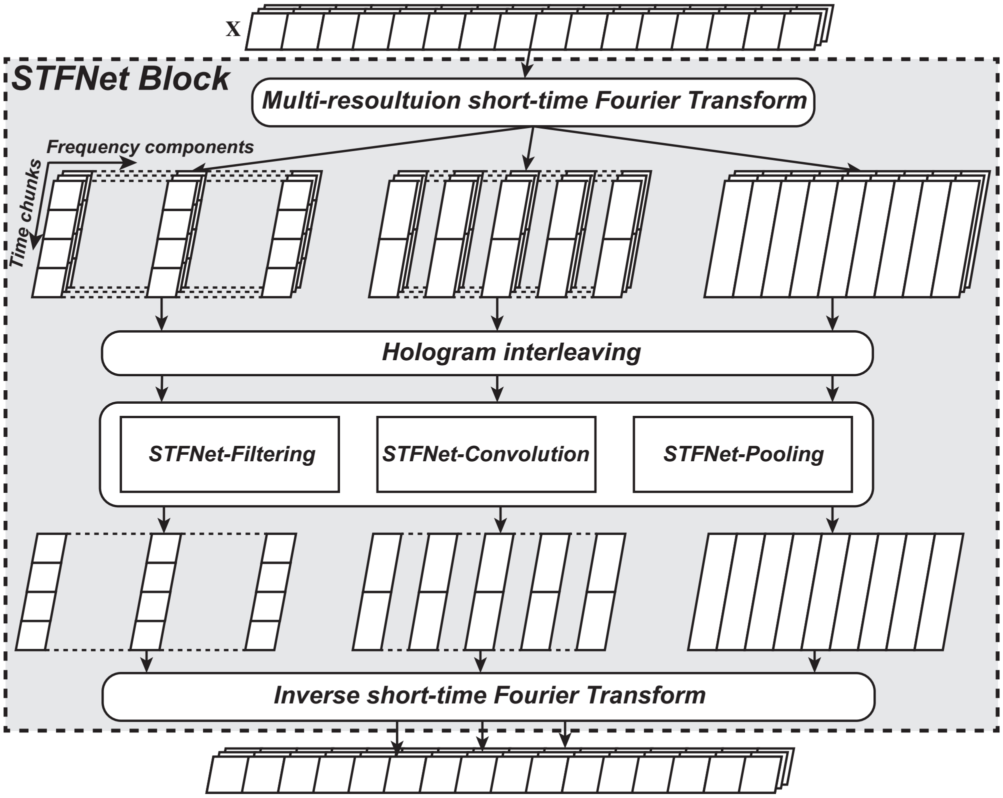


Figure 3.16: The overview design of STFNet block.

the signal dimension. Since we concentrate on sensing signals, we assume that all the raw and internal-manipulated sensing signals are real-valued in time domain.

As shown in Figure 3.16, the input signal $\mathbf{X}$ first goes through a multi-resolution short-time Fourier transform (Multi_STFT), which is a compound traditional short-time Fourier transform (STFT), to provide a time-frequency hologram of the signal. STFT breaks the original signal up into chunks with a sliding window, where sliding window $\mathbf{W}(t)$ with width τ only has non-zero values for $1 \leq t \leq \tau$. Then each chunk is Discrete-Fourier transformed,

$$\mathbf{STFT}^{(\tau,s)}(\mathbf{X})_{[m,k,d]} = \sum_{t=1}^{T} \mathbf{X}_{[t,d]} \cdot \mathbf{W}(t - s \cdot m) \cdot e^{-j\frac{2\pi k}{\tau}(t - s\cdot m)}, \tag{3.4}$$

where $\mathbf{STFT}^{(\tau,s)}(\mathbf{X}) \in \mathbb{C}^{M\times K\times D}$ denotes the short-time Fourier transform with width τ and sliding step s. M denotes the number of time chunks. K denotes the number of frequency components. Since input signal $\mathbf{X}$ is real-valued, its discrete Fourier transform is conjugate symmetric. Therefore, we only need the $\lfloor \tau/2 \rfloor + 1$ frequency components to represent the signal, *i.e.*, $K = \lfloor \tau/2 \rfloor + 1$. In this dissertation, we focus on sliding chunks with rectangular window and no overlaps to simplify the formulation, *i.e.*, $s = \tau$ and $M = T/\tau$. We therefore denote of short-time Fourier transform as $\mathbf{STFT}^{(\tau)}(\mathbf{X})$.

The Multi_STFT operation is composed of multiple short-time Fourier transform with different window widths $\mathcal{T} = \{\tau_i\}$. The window width, τ_i, determines the time-frequency resolution of STFT. Larger τ_i provides better frequency resolution, while smaller τ_i provides better time resolution. In this dissertation, we set the window widths to be powers of 2, *i.e.*, $\tau_i = 2^{p_i}\ \forall p_i \in \mathbb{Z}_0^+$, to simplify the design later. We can thus formulate Multi_STFT as:

$$\textbf{Multi\_STFT}^{(\mathcal{T})}\{\mathbf{X}\} = \left\{\textbf{STFT}^{(\tau_i)}(\mathbf{X})\right\} \text{ for } 2^{p_i} \in \mathcal{T}. \tag{3.5}$$

Next, according to Figure 3.16, the multi-resolution representations go into the hologram interleaving component, which enables the representations to compensate and balance their time-frequency resolutions with each other. The technical details of the hologram interleaving component are introduced in Section 3.3.3.

The STFNet block then manipulates multiple hologram representations with the same set of spectral-compatible operation(s), including STFNet-filtering, STFNet-convolution, and STFNet-pooling. We will formulate these operations in Section 3.3.4, 3.3.5, and 3.3.6, respectively.

Finally, the STFNet block converts the manipulated frequency representations back into the time domain with the inverse short-time Fourier transform. The resulting representations from different views of the hologram are weighted and merged as the input "signal" for the next block. Since we merge the output representations from different views of the hologram, we reduce the output feature dimension of STFNet-filtering and convolution operations by the factor of $1/|\mathcal{T}|$ to prevent the dimension explosion.

3.3.3 STFNet Hologram Interleaving

In this subsection, we introduce the formulation of hologram interleaving. Due to the Fourier uncertainty principle, the representations in time-frequency hologram either have high time resolution or high frequency resolution. The hologram interleaving tires to use representations with high time resolution to instruct the representations with low time resolution to highlight the important components over time. This is done by two steps:

1. Revealing the mathematical relationship of aligned time-frequency components among different representations in the time-frequency hologram.

2. Updating the original relationship in a data-driven manner through neural-network attention components.

We start from the definition of time-frequency hologram, generated by Multi_STFT defined in (3.5). Note that, the window width set $\mathcal{T}$ is defined as $\{2^{p_i}\}$, $\forall p_i \in \mathbf{Z}_0^+$. Without loss of

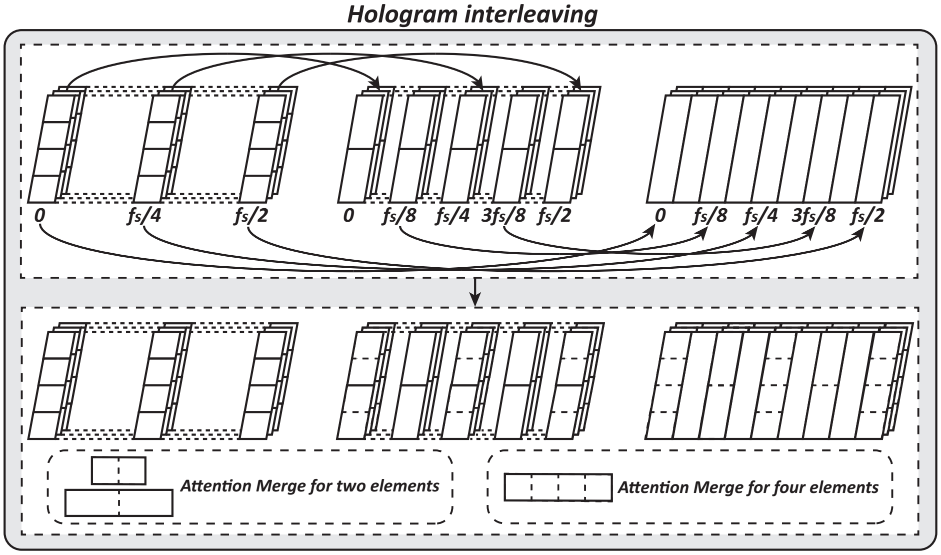


Figure 3.17: The design of hologram interleaving.

generality, an illustration of multi-resolution short-time Fourier transformed representations with input signal having length 16 and signal dimension 3 as well as $\mathcal{T} = \{4, 8, 16\}$ are illustrated in Figure 3.17.

In order to find out the relationship of aligned time-frequency components, we start with the frequency-component dimension. Since different representations only change the window width τ_i of STFT but not the sampling frequency f_s of input signal, these frequency components represent frequencies from 0 to $f_s/2$ (Nyquist frequency) with step f_s/τ_i. Then we can first obtain the relationship of frequency ranging steps among different representations,

$$\forall p_i > p_j \ , \ \frac{f_s/\tau_j}{f_s/\tau_i} = 2^{p_i - p_j} \in \mathbf{Z}_0^+. \tag{3.6}$$

Therefore, a low frequency-resolution representation (with window width 2^{p_j}) can find their frequency-equivalent counterparts for every $2^{p_i - p_j}$ frequency components in a high frequency-resolution representation (with window width 2^{p_i}). The upper part of Figure 3.17 provides a simple illustration of such relationship. In the following analysis, we will use the original index k and corresponding frequency $k \cdot f_s/\tau_i$ interchangeably to recall the frequency component from the time-frequency hologram $\mathbf{STFT}^{(\tau)}(\mathbf{X})_{[m,k,d]}$.

Next, we analyze the relationship over the time-chunk dimension, when two representations have frequency-equivalent components. Note that time chunks in $\mathbf{STFT}^{(\tau)}(\mathbf{X})$ are generated by sliding rectangular window without overlap. Based on (3.4), for representations having window widths $\tau_i = 2^{p_i}$ and $\tau_j = 2^{p_j}$ $(p_i > p_j)$,

$$
\begin{aligned}
\mathbf{STFT}^{(\tau_i)}(\mathbf{X})_{[m,2^{p_i-p_j}k,d]} &= \sum_{t=2^{p_i}m+1}^{2^{p_i}(m+1)} \mathbf{X}_{[t,d]} \cdot e^{-j\frac{2\pi 2^{p_i-p_j}k}{2^{p_i}}(t-m\cdot 2^{p_i})}, \\
&= \sum_{m_j=2^{p_i-p_j}m}^{2^{p_i-p_j}(m+1)-1} \sum_{t=m_j+1}^{2^{p_j}(m_j+1)} \mathbf{X}_{[t,d]} \cdot e^{-j\frac{2\pi k}{2^{p_j}}(t-m\cdot 2^{p_j})}, \\
&= \sum_{m_j=2^{p_i-p_j}m}^{2^{p_i-p_j}(m+1)-1} \mathbf{STFT}^{(\tau_j)}(\mathbf{X})_{[m_j,k,d]}.
\end{aligned}
\tag{3.7}
$$

Therefore, given the equivalent frequency component, a time component in low time-resolution representation (with window width 2^{p_i}) is the sum of $2^{p_i-p_j}$ aligned time components of the high time-resolution representation (with window width 2^{p_j}). As a toy example in Figure 3.17, the first row of the middle tensor is equal to the sum of first two rows of the left tensor for frequencies 0, $f_s/4$, and $f_s/2$. The row of the right tensor is equal to the sum of four rows of the left tensor for frequencies 0, $f_s/4$, and $f_s/2$. The row of the right tensor is equal to the sum of two rows of the middle tensor for frequencies $f_s/8$ and $3f_s/8$, etc.

According to the analysis above, the high frequency-resolution representations lose their fine-grained time resolutions at certain frequencies by summing the corresponding frequency components up over a range of time. However, the high time-resolution representations preserve these information.

The idea of hologram interleaving is to replace the sum operation in high frequency-resolution representation with a weighted merge operation to highlight the important information over time. For a certain frequency component, the weight of merging is learnt through the most fine-grained information preserved in the time-frequency hologram. In this dissertation, we implement the weighted merge operation as a simple attention module. For a merging input $\mathbf{z} \in \mathbb{C}^{S\times 1}$, where S is the number of elements to be merged, the merge operation is formulated as:

$$
\begin{aligned}
\mathbf{a} &= \text{softmax}(|\mathbf{W}_m \mathbf{z}|), \\
y &= S \times \mathbf{a}^{\intercal}\mathbf{z},
\end{aligned}
\tag{3.8}
$$

where $|\cdot|$ is the piece-wise magnitude operation for complex-number vector; and $\mathbf{W}_m \in \mathbb{C}^{S\times S}$ is the learnable weight matrix. Notice that the final merged result is rescaled by the factor S to imitate the "sum" property of Fourier transform.

3.3.4 STFNet-Filtering Operation

Starting from this subsection, we will introduce our three spectral-compatible operations in STFNet. In each subsection, the introduction includes two main parts: 1) the basic

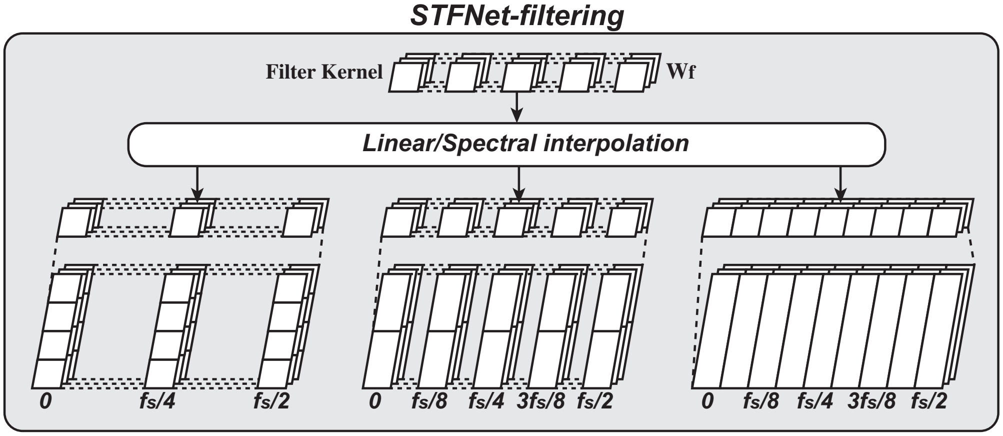


Figure 3.18: The STFNet-filtering operation.

formulation of proposed spectral-compatible operation, and 2) extending a single operation to multi-resolution data.

Spectral filtering is a widely-used operation in time-frequency analysis. The STFNet-filtering operation replaces the traditional manually designed spectral filter with a learnable weight that can update during the training process. Although the spectral filtering is equivalent to the time-domain convolution according to convolution theorem [1], the filtering operation helps to handle the multi-resolution time-frequency analysis, and facilitates the parameterization and modelling. We denote the input tensor as $\mathbf{X} \in \mathbf{C}^{M \times K \times D}$, where M is the number of time chunk, K frequency component number, and D input feature dimension. The STFNet-filtering operation is formulated as:

$$\mathbf{Y}_{[m,k,\cdot]} = \mathbf{X}_{[m,k,\cdot]} \mathbf{W}_{f[k,\cdot,\cdot]}, \tag{3.9}$$

where $\mathbf{W}_f \in \mathbb{C}^{K \times D \times O}$ is the learnable weight matrix, O the output feature dimension, and $\mathbf{Y} \in \mathbb{C}^{M \times K \times O}$ the output representation.

The function of STFNet-filtering operation is providing a set of learnable global frequency template matchings over the time. However, it is not straightforward to extend the matching operation to the representations with different time-frequency resolutions. Although we can create multiple $\mathbf{W}_f$ with different frequency resolutions K, it can introduce unnecessary complexity and redundancy.

STFNet-filtering solves this problem by interpolating the frequency components in weight matrix. As we mentioned in Section 3.3.3, data in hologram with different frequency resolutions have the same frequency range (from 0 to $f_s/2$) but different frequency steps (f_s/τ). Therefore, STFNet-filtering operation only has one weight matrix $\mathbf{W}_f$ with $K = \lfloor \tau/2 \rfloor + 1$ frequency components. When the operation input has $K' = \lfloor \tau'/2 \rfloor + 1$ frequency compo-

[1] `https://en.wikipedia.org/wiki/Convolution_theorem`

nents with $K' < K$, we can subsample the frequency components in $\mathbf{W}_f$. When $K' > K$, we interpolate the frequency components of $\mathbf{W}_f$. STFNet provides two kind of interpolation methods: 1) linear interpolation and 2) spectral interpolation.

The linear interpolation generates the missing frequency components in extended weight matrix $\mathbf{W}'_f \in \mathbb{C}^{K' \times D \times O}$ from the two neighbouring frequency components in $\mathbf{W}_f$:

$$\begin{aligned} & k_l = \left\lfloor k' \frac{\tau}{\tau'} \right\rfloor \quad k_r = k_l + 1, \\ & \mathbf{W}'_{f[k',\cdot,\cdot]} = \mathbf{W}_{f[k_l,\cdot,\cdot]} \Big(k_r - k' \frac{\tau}{\tau'} \Big) + \mathbf{W}_{f[k_r,\cdot,\cdot]} \Big(k' \frac{\tau}{\tau'} - k_l \Big). \end{aligned} \tag{3.10}$$

The spectral interpolation utilizes the relationship between discrete-time Fourier transform (DTFT) and discrete Fourier transform (DFT). For a time-limited signal (with length τ), DTFT regards it as a infinite-length data with zeros outside the time-limited range, while DFT regards it as a τ-periodic data. As a result, DTFT generates a continuous function over the frequency domain, while DFT generates a discrete function. Therefore, DFT can be regarded as a sampling of DTFT with step f_s/τ. In order to increase the frequency resolution of $\mathbf{W}_f$, we can increase the sampling step from f_s/τ to f_s/τ', which is called spectral interpolation. Spectral interpolation can be done through zero padding in the time domain [19],

$$\mathbf{W}'_{f[\cdot,d,o]} = \mathbf{DFT}\Big(\mathbf{ZeroPad}_{\tau' - \tau} \mathbf{IDFT}\big(\mathbf{W}_{f[\cdot,d,o]} \big) \Big), \tag{3.11}$$

where $\mathbf{ZeroPad}_t$ denotes padding t zeros at the end of sequence, and $\mathbf{IDFT}(\cdot)$ denotes the inverse discrete Fourier transform. Please note that, if we pad infinite zeros to the IDFT result, then DFT turns into DTFT. An simple illustration of STFNet-filtering operation is shown in Figure 3.18.

3.3.5 STFNet-Convolution Operation

In this subsection, we introduce our design of STFNet-convolution operation. Other than filtering operation that handles global pattern matching, we still need the convolution operation to deal with local motifs in the frequency domain. We denote the input tensor as $\mathbf{X} \in \mathbf{C}^{M \times K \times D}$, where M is the number of time chunk, K number of frequency component, and D input feature dimension. The convolution operation involves two steps: 1) padding the input data, and 2) convolving with kernel weight matrix $\mathbf{W}_c \in \mathbb{C}^{1 \times S \times D \times O}$, where S is the kernel size along the frequency axis and O is still the output feature dimension.

Without the padding step, the output of convolution operation will shrink the number of frequency components, which may break the underlying structure and information in

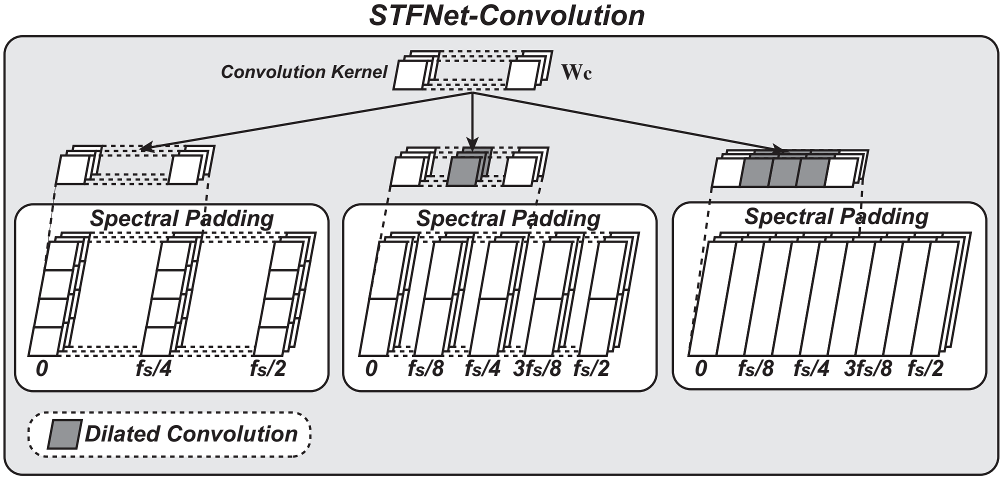


Figure 3.19: The STFNet-convolution operation with dilated configuration.

the frequency domain. Therefore, we need to pad extra "frequency component" to keep the shape of output tensor unchanged compared to that of the input data. In the deep learning research, padding zeros is a common practice. Zero padding is reasonable for inputs such as images and signal in the time domain, meaning no additional information in the padding range. However, padding zero-valued frequency component introduces additional information in the frequency domain.

Therefore, STFNet-convolution operation proposes the spectral padding for time-frequency analysis. According to the definition of DFT, transformed data is periodic within the frequency domain. In addition, if the original signal is real-valued, then the transformed data is conjugate symmetric within each period. Previously, we cut the number of frequency components of a τ-length signal to $K = \lfloor \tau/2 \rfloor + 1$ for reducing the redundancy. In the spectral padding, we add these frequency components back according to the rule

$$\mathbf{X}_{[\cdot,\tau-k,\cdot]} = \mathbf{X}_{[\cdot,-k,\cdot]} = \mathbf{X}^{*}_{[\cdot,k,\cdot]}, \tag{3.12}$$

where $\mathbf{X}^{*}$ denotes complex conjugation. In addition, the number of padding before and after the input tensor is same as the previous padding techniques.

Then we can define the basic convolution operation in STFNet

$$\mathbf{Y} = \mathbf{SpectralPad}(\mathbf{X}) \circledast \mathbf{W}_c, \tag{3.13}$$

where $\mathbf{SpectralPad}(\cdot)$ denotes our spectral padding operation, and $\circledast$ denotes the convolution operation.

Next, we discuss the way to share the kernel weight matrix $\mathbf{W}_c$ with multi-resolution data. Other than interpolating the kernel weight matrix as shown in (3.10) and (3.11), we propose

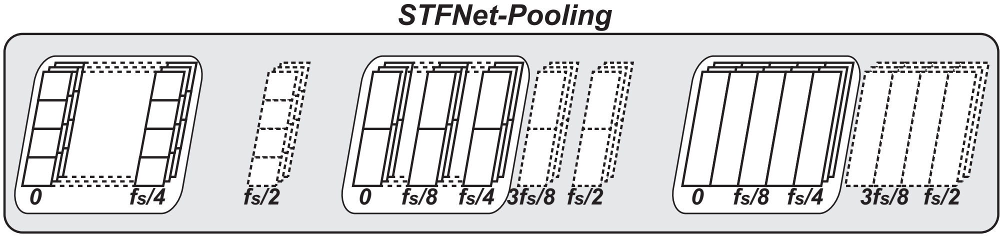


Figure 3.20: The low-pass STFNet-pooling operation.

another solution for the STFNet-convolution operation. The convolution operation concerns more about the pattern of relative positions on the frequency domain. Therefore, instead of providing additional kernel details on fine-grained frequency resolution, we can just ensure that the convolution kernel is applied with the same frequency spacing on representations with different frequency resolutions. Such idea can be implemented with the dilated convolution [49]. If $\mathbf{W}_c$ is applied to a input tensor with $K = \lfloor \tau/2 \rfloor + 1$ frequency components, for a input tensor with $K' = \lfloor \tau'/2 \rfloor + 1$ frequency components ($\tau' > \tau$), the dilated rate r is set to $\tau'/\tau - 1$. An simple illustration of STFNet-convolution with dilated configuration is shown in Figure 3.19.

3.3.6 STFNet-Pooling Operation

In order to provide a dimension reduction method for sensing series within STNet, we introduce the STFNet-pooling operation. STFNet-pooling truncates the spectral information over time with a pre-defined frequency pattern. As a widely-used processing technique, filtering zeroes unwanted frequency components in the signal. Various filtering techniques have been designed, including low-pass filtering, high-pass filtering, and band-pass filtering, which serve as templates for our STFNet-pooling. Instead of zeroing unwanted frequency components, STFNet-pooling removes unwanted components and then concatenates the left pieces. For applications with domain knowledge about signal-to-noise ratio over the frequency domain, specific pooling strategy can be designed. In this dissertation, we focus on low-pass STFNet-pooling as an illustrative example.

To extend the STFNet-pooling operation to multiple resolutions and preserving spectral information, we make sure that all representations have the same cut-off frequency according to their own frequency resolutions. A simple example of low-pass STFNet-pooling operation is shown in Figure 3.20. We can see that our three tensors are truncated according to the same cut-off frequency, $f_s/4$.

3.4 THE EVALUATION OF STFNET

In this section, we evaluate the STFNet with diverse sensing modularities. We focus on the device-based and device-free human activity recognitions with motion sensors (accelerometer and gyroscope), WiFi, ultrasound, and visible light. We first introduce the experimental setting, including data collection and baseline algorithms. Next, we show the performance metrics of leave-one-user-out evaluation of human activity recognition with different modularities. Finally, we analyze the effectiveness of STFNet through several ablation studies.

3.4.1 Experimental Settings

In this subsection, we first introduce detailed information of the dataset we used or collected for each evaluation task. Then we specify the way to test the performance of evaluation task.

Motion Sensor: In this experiment, we recognize human activity with motion sensors on smart devices. We use the dataset collected by Allan et al. [11]. This dataset contains readings from two motion sensors (accelerometer and gyroscope). Readings were recorded when users executed activities scripted in no specific order, while carrying smartwatches and smartphones. The dataset contains 9 volunteers, 6 activities (biking, sitting, standing, walking, climbStair-up, and climbStair-down). The raw data of two sensor readings with sampling rate that approximates to 100Hz. We align two sensor readings, linear interpolate two readings by 100Hz, and segment them into non-overlapping data samples with time interval 5.12s. Therefore, each data sample is a 512×6 matrix, where both accelerometer and gyroscope have readings on x, y, and z axis.

WiFi: In this experiment, we make use of Channel State Information (CSI) to analyze human activities. CSI refers to the known channel properties of a communication link, which can be affected by the presence of humans and their activities. We employ 11 volunteers (including both men and women) as the subjects and collect CSI data from 6 different rooms in two different buildings. In particular, we build a WiFi infrastructure, which includes a transmitter (a wireless router) and two receivers. We choose to use the Intel Wireless Link 5300 NIC to collect the CSI data, and the transmission rate is set to 200 packets per second. We use the tool to report CSI values of 30 OFDM subcarriers [50]. The experiment contains 6 activities (wiping the whiteboard, walking, moving a suitcase, rotating the chair. sitting, as well as standing up and sitting down). We linearly interpolate the CSI data with a uniform sampling period, and down-sample the measurements into 100Hz. Then we segment the down-sampled CSI data into non-overlapping data samples with time interval 5.12s. Therefore, each data sample is a 512×30 matrix, where each CSI measurement has readings from 30 subcarriers.

Ultrasound: In this experiment, we conduct human activity recognition based on ultrasound. We employ 12 volunteers (including both men and women) as the subjects to conduct the 6 different activities (wiping the whiteboard, walking, moving a suitcase, rotating the chair, sitting, as well as standing up and sitting down). The activity data are collected from 6 different rooms in two different buildings. The transmitter is an iPad on which an ultrasound generator app is installed, and it can emit an ultrasound signal of approximately 19 KHz. The receiver is a smartphone and we use the installed recorder app to collect the sound waves. We demodulate the received signal with carrier frequency 19KHz, and down-sample the measurement into 100Hz. Then we segment the down-sampled ultrasound data into non-overlapping data samples with time interval 5.12s. Therefore, each sample is a 512×1 matrix.

Visible light: In this experiment, we capture the human activity in the visible light system. We build an optical system using photoresistors to capture the in-air body gesture, which can detect the illuminance change (lux) caused by the body interaction. Specifically, we employ the cadmiumsulfide (CdS) cells, which are basically resistors that change their resistive value in ohms depending on the amount of light which is shining onto the squiggly face. In the experiment, there are three light conditions (natural mode, warm mode, and cool mode) and 4 hand gestures (drawing an anticlockwise circle, drawing a clockwise circle, drawing a cross, and shaking hand side to side). We employ 6 volunteers (including both men and women) as the subjects and each of them performs 20 trials of every gesture under a given lighting condition. We linearly interpolate and down-sample the measurements into 25Hz. Then we segment the data into non-overlapping data samples with time interval 5.12s. Therefore, each sample is a 128×6 matrix, where each measurement contains readings from 6 CdS cells.

Testing: In the whole evaluation, to illustrate the generalization ability of STFNet and other baseline models, we perform leave-one-user-out cross validation for every task. For each time, we choose the data from one user as testing data with the left as training data. We then compare the performance of models according to their accuracy and F1 score with 95% confidence interval.

Table 3.2: Illustration of models with two sensor inputs.

STFNet-Filter/Conv		DeepSense/ComplexNet	
Sensor Data 1	Sensor Data 2	Chunked Sensor Data 1	Chunked Sensor Data 2
STFNet1-1	STFNet1-2	Conv Layer1-1	Conv Layer1-2
STFNet2-1	STFNet2-2	Conv Layer2-1	Conv Layer2-2
STFNet3-1	STFNet3-2	Conv Layer3-1	Conv Layer3-2
STFNet-pooling		Max pooling	
STFNet4		Conv Layer4	
STFNet5		Conv Layer5	
STFNet6		Conv Layer6	
Averaging		GRU	
Softmax		Softmax	

3.4.2 Models in Comparison

In order to evaluate, when compared to conventional deep learning components (*i.e.*, convolutional and recurrent layers), whether our proposed STFNet component is better at decoding information and extracting features from sensing inputs, we substitute components in the state-of-the-art neural network structure for IoT applications with STFNet. In the whole evaluation, we choose DeepSense as the state-of-the-art structure, which has shown signifiant improvements on various sensing tasks [2]. The illustration of structures of five comparing models with two sensor inputs are shown in Table 3.2. Detailed information of our comparing models are listed as follows,

1. ***STFNet-Filter:*** This model integrates the proposed STFNet component and the DeepSense structure. Within the STFNet component, we use the STFNet-filtering operation designed in Section 3.3.4. The intuition of DeepSense structure is to first perform local processing within each sensor and then perform global sensor fusion over multiple sensors. In this model, we replace all convolutional layers used in local/global sensor data processing with our time-frequency analyzing component, STFNet. Since our model has already incorporated time-domain analysis within the STFNet component through multi-resolution processing, we replace the Gated Recurrent Units (GRU) with simple feature averaging time at last.

2. ***STFNet-Conv:*** This model is almost the same as the STFNet-Filter, except that we use the STFNet-convolution operation designed in Section 3.3.5.

3. ***DeepSense-Freq:*** This model is the original DeepSense [2]. It divides the input sensing data into chunks, and processes each chunk with DFT. It treats the real and

imagery parts of discrete Fourier transformed time chunks as the additional feature dimensions. This is the state-of-the-art deep learning model for sensing data modelling and IoT applications.

4. ***DeepSense-Time:*** This model is almost the same as the DeepSense-Freq, except that it directly takes the chunked raw sensing data without DFT as input.

5. ***ComplexNet:*** This model is a complex-value neural network [18] that can operate on complex-value inputs. Instead of using simple CNN and RNN structure as originally proposed [18], we cheat in their favor by using the DeepSense structure, which improves the performance in all tasks. The network inputs are chunked sensing data with DFT.

3.4.3 Effectiveness

In this section, we discuss about the effectiveness of our proposed STFNet based on extensive experiments and diverse sensing modalities, compared with other state-of-the-art deep learning models.

As we mentioned in Section 3.4.1, all models are evaluated through leave-one-user-out cross validation with accuracy and F1 score accompanied by the 95% confidence interval. STFNet-based models (STFNet-Filter and STFNet-Conv) take a sliding window set for multi-resolution short-time Fourier transform. We choose the set to be $\{16, 32, 64, 128\}$ for activity recognition with motion sensors, WiFi, and ultrasound; and choose set to be $\{8, 16, 32, 64\}$ for activity recognition with visible light. DeepSense-based models (DeepSense-Freq and DeepSense-Time) need a sliding window for chunking input signals. In the evaluation, we cheat in their favor by choosing the best-performing window size from $\{8, 16, 32, 64, 128\}$ according to the accuracy metric. In addition, we consistently configure STFNet-filtering operation with linear interpolation, and STFNet-convolution operation with spectral padding. We will show further evaluations on multi-resolution operations and the effects of diverse operation settings in Section 3.4.4.

Motion Sensors

For device-based activity recognition with motion sensors, there are 9 users. The accuracy and F1 score with the 95% confidence interval for leave-one-user-out cross validation are illustrated in Figure 3.21. STFNet based models, *i.e.*, STFNet-Filter and STFNet-Conv, outperform all other baseline models with a large margin. The confidence interval lower bound of STNet-Filter and STFNet-Conv is even better than the confidence interval

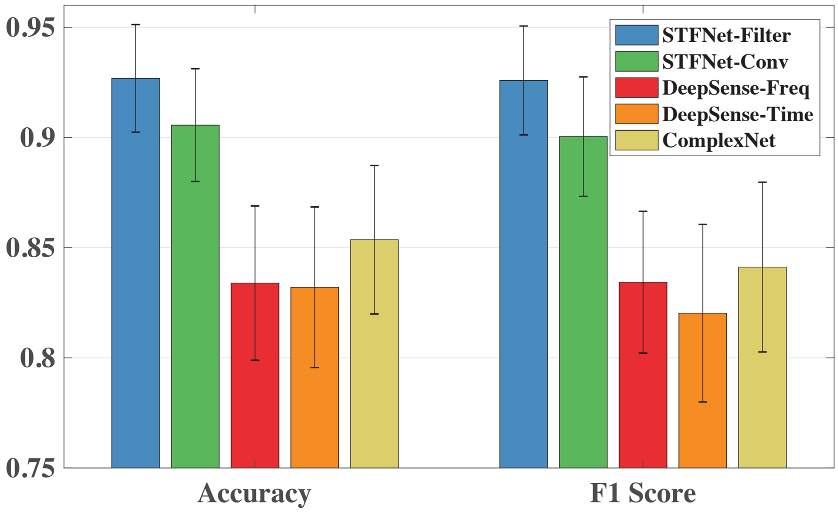


Figure 3.21: The accuracy and F1 score with 95% confidence interval for motion sensors.

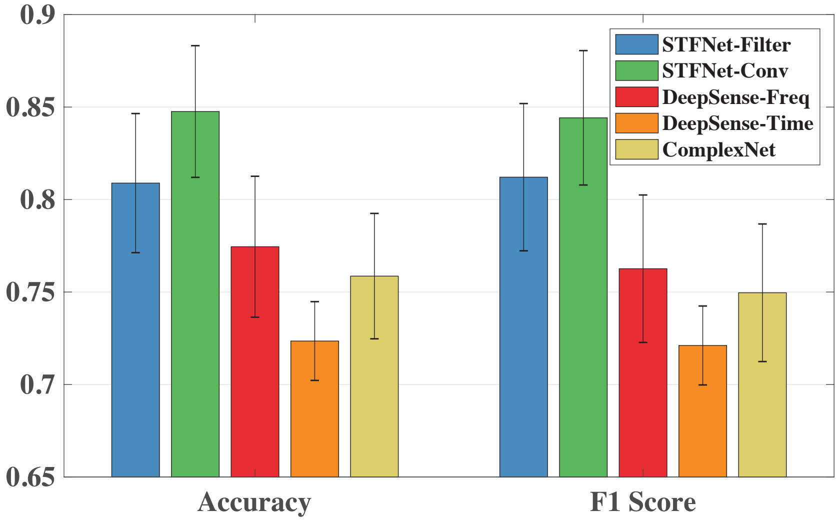


Figure 3.22: The accuracy and F1 score with 95% confidence interval for WiFi.

upper bound of DeepSense-Freq and DeepSense-Time. STFNet-Filter performs better than STFNet-Conv in this experiment, indicating that different activities have distinct global profiling patterns with motion sensor readings in the frequency domain, even among different users. STFNet-Filter is able to learn the accurate global frequency profiling, which makes it the top-performance model in this task. In addition, compared to ComplexNet, STFNet based models show clear improvements. Therefore, using just complex-value neural network for sensing signal is far from enough. The multi-resolution processing and operations that are spectral-compatible are all crucial designs.

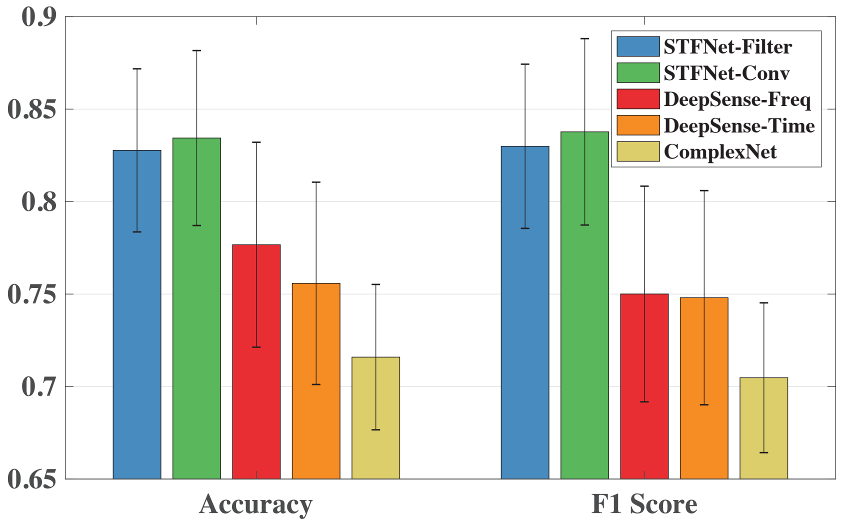


Figure 3.23: The accuracy and F1 score with 95% confidence interval for Ultrasound.

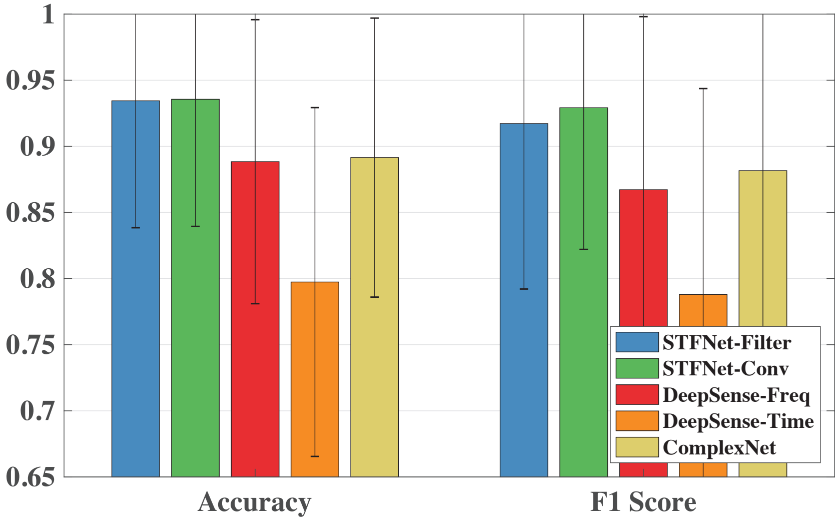


Figure 3.24: The accuracy and F1 score with 95% confidence interval for Visible light.

WiFi

For device-free activity recognition with WiFi signal, there are 11 users. The accuracy and F1 score with the 95% confidence interval for leave-one-user-out cross validation are illustrated in Figure 3.22. STFNet based models still outperform all others with a clear margin, illustrating the effectiveness of principled design of STFNet from time-frequency perspective. DeepSense-Freq outperforms DeepSense-Time in this experiment, which means that even having time-frequency transformation as pre-processing can help. The complex-value network, ComplexNet, performs worse than its real-value counterpart, DeepSense-Freq. This indicates that blindly processing time-frequency representations without preserving their physical meanings can even hurt the final performance. STNet-Conv performs better than STNet-Filter in the WiFi experiment, indicating that local shiftings in the frequency domain are more representative for diverse activities profiled with WiFi CSI.

Ultrasound

There are 12 users in device-free activity recognition with ultrasound experiment. The accuracy and F1 score with the 95% confidence interval for leave-one-user-out cross validation are illustrated in Figure 3.23. STFNet based models still significantly outperforms all other baselines. An interesting observation is that ComplexNet performs even worse than both DeepSense-Freq and DeepSene-Time, which again validates the importance of designing neural networks for sensing signal with multi-resolution processing as well as preserving the time and frequency information.

Visible Light

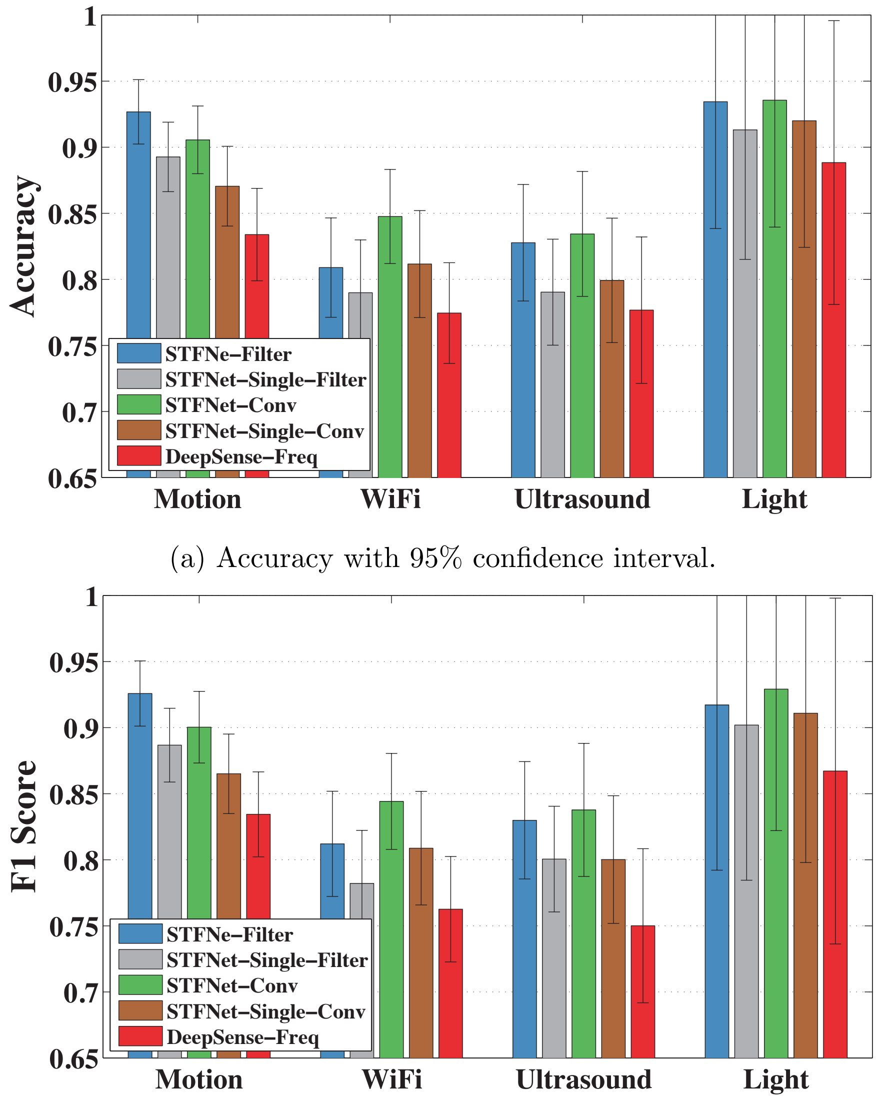
There are 6 users in the experiment of device-free activity recognition with visible light. The accuracy and F1 score with the 95% confidence interval are illustrated in Figure 3.24. Except for the DeepSense-Time, all other models can can achieve an accuracy of approximately 90% or higher. STFNet based models still do the best. There is no significant difference between STFNet-Filter and STFNet-Conv, which indicates that measured visible light readings have quite clean representations in the frequency domain.

3.4.4 Ablation Studies

In the previous section, we illustrate the performance of STFNet compared to other state-of-the-art baselines. In this section, we focus mainly on the STFNet design. We conduct several ablation studies by deleting one designing feature from STFNet at a time.

Multi-Resolution v.s. Single-Resolution

First, we validate the effectiveness of our design of multi-resolution processing in STFNet block. As shown in Figure 3.16, this includes multi-resolution STFT, hologram interleaving, and weights sharing techniques in STFNet-Filtering and STFNet-Convolution operations. In this experiment, we add two more baseline models, STFNet-Single-Filter and STFNet-Single-Conv, generated by deleting the multi-resolution processing in STFNet-Filter and STFNet-Conv respectively. These two models pick the best-performing window size

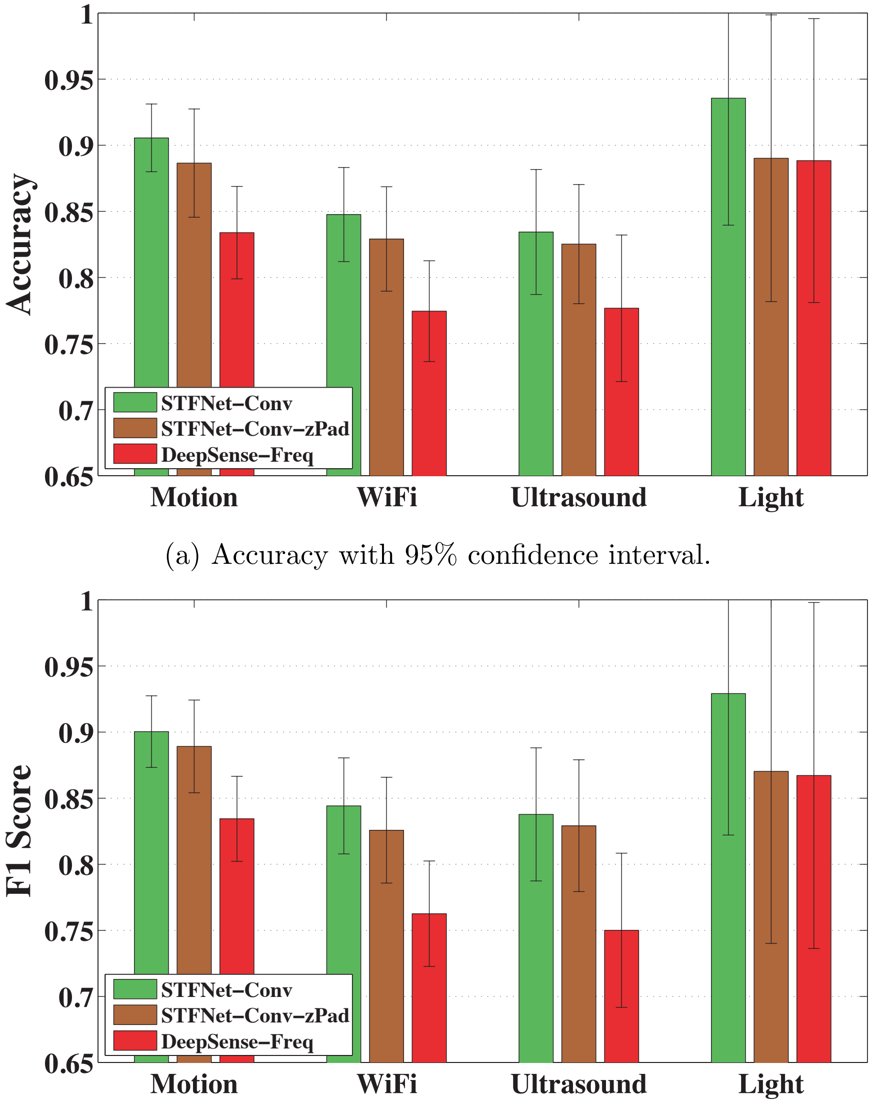


(a) Accuracy with 95% confidence interval.

(b) F1 score with 95% confidence interval.

Figure 3.25: Multi-Resolution v.s. Single-Resolution

from $\{8, 16, 32, 64, 128\}$ according to the accuracy metric. The results for all four tasks are illustrated in Figure 3.25, where DeepSense-Freq severs as a decent performance low-bound. The design of multi-resolution processing significantly impacts the performance of STFNet. STFNet-Single-Filter and STFNet-Single-Conv show clear performance degradation compared to their multi-resolution counterparts. In addition, STFNet-Single-Filter and STFNet-Single-Conv still consistently outperform DeepSense-Freq with a clear margin. This is because our other designed operations, including STFNet-Filtering, STFNet-Convolution, STFNet-Pooling still facilitate the learning in time-frequency domain.



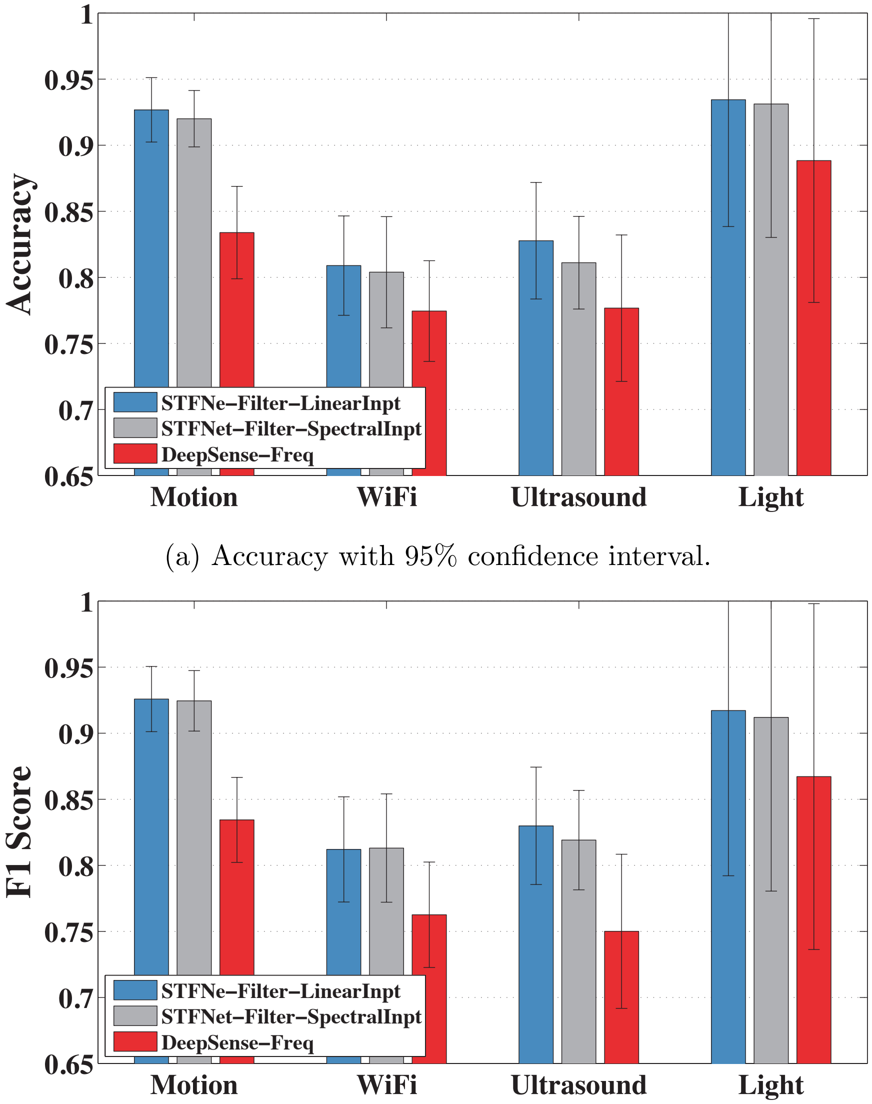
(a) Accuracy with 95% confidence interval.

(b) F1 score with 95% confidence interval.

Figure 3.26: Spectral Padding v.s. Zero Padding

Spectral Padding v.s. Zero Padding

Next, we validate our design of spectral padding in the STFNet-Convolution operation as shown in Figure 3.19. In this experiment, we add a new baseline algorithm, STFNet-Conv-zPad, by replacing spectral padding with traditional zero padding in the STFNet-Conv. The accuracy and F1 score of all four tasks are shown in Figure 3.26. Here, DeepSense-Freq is still treated as a performance low-bound. By comparing STFNet-Conv-zPad and STFNet-Conv, we can see that spectral padding consistently helps improving the model performance. In most cases, the improvement is limited. However, in the case of visible light, spectral padding significantly improves both accuracy and F1 score. Therefore, designing neural



(a) Accuracy with 95% confidence interval.

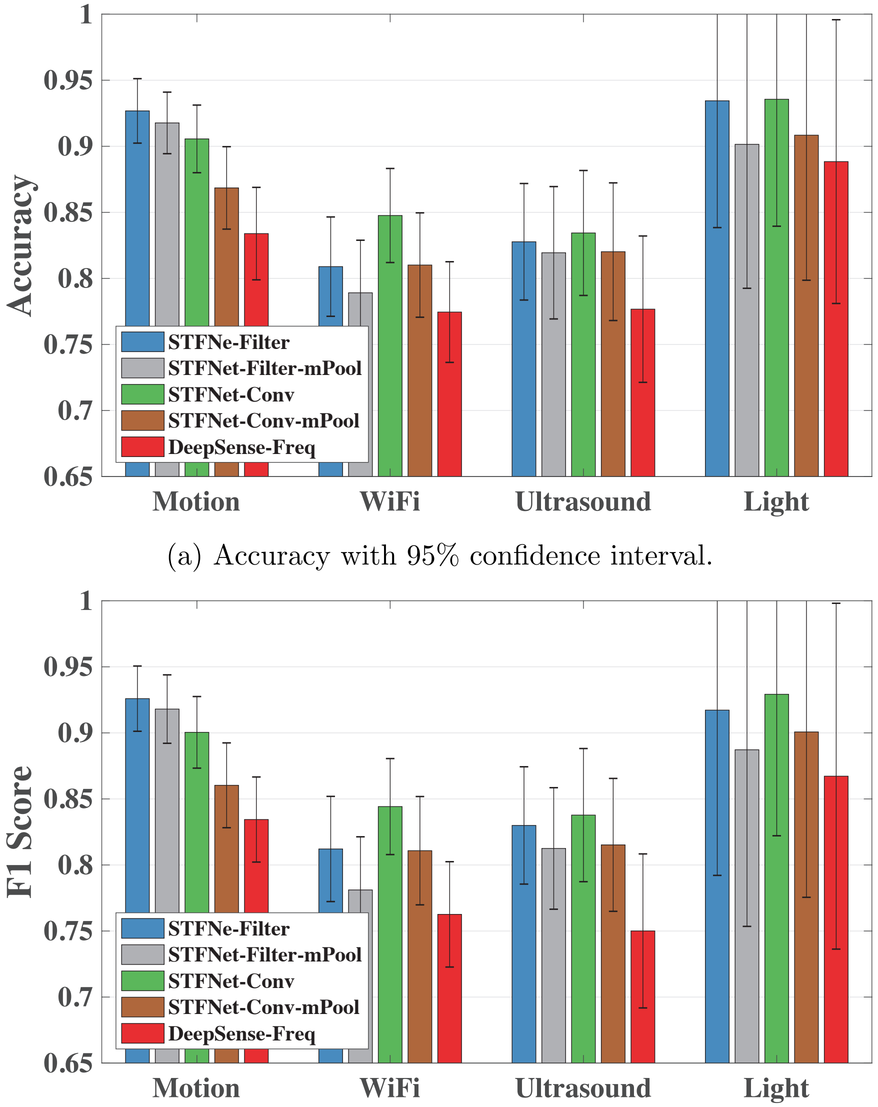
(b) F1 score with 95% confidence interval.

Figure 3.27: Linear Interpolation v.s. Spectral Interpolation

network by preserving the time-frequency semantics of sensing signal is an important rule to follow.

Linear Interpolation v.s. Spectral Interpolation

Then, we compare our two designs of weight interpolation method in the STFNet-Filtering operation, linear interpolation and spectral interpolation, as shown in Figure 3.18. The STFNet-Filter defined in Section 3.4.2 uses linear interpolation, so we rename it as STFNet-Filter-LinearInpt in this experiment. We add a new baseline model called STFNet-Filter-



(a) Accuracy with 95% confidence interval.

(b) F1 score with 95% confidence interval.

Figure 3.28: STFNet Pooling v.s. Mean/Max Pooling

SpectralInpt by using spectral interpolation instead of linear interpolation in STFNet-Filter. The results of all four tasks are illustrated in Figure 3.27. In general, the performance of two design choices are almost the same. At most of time, linear interpolation performs slightly better. In addition, since the implementation of linear interpolation is easier, we recommend using it to improve the time efficiency.

STFNet Pooling v.s. Mean/Max Pooling

Finally, we validate our design of STFNet-Pooling (low-pass deisgn) as shown in Figure 3.20. In this experiment, we add two new baseline algorithms, STFNet-Filter-mPad and STFNet-Conv-mPad, by replacing STFNet-Pooling in STFNet-Filter and STFNet-Conv with traditional max/mean pooling in the time domain (through choosing the one has better accuracy). The results are illustrated in Figure 3.28. In all settings, STFNet-Pooling shows better performance. In some cases, the improvement are significant. We believe that STFNet-Pooling can achieve even better performance if given the detailed signal-to-noise ratio over the frequency domain for each specific sensor. Then we can employ other pooling strategies instead of the low-pass design.

CHAPTER 4: DEEP LEARNING FOR RESOURCE-CONSTRAINED IOT SYSTEMS

In this section, we first introduce the technical details of the DeeoIoT framework, a neural network structure compression framework for resource-constrained IoT systems. Then, we present the design of FastDeepIoT, a framework for understanding and minimizing neural network execution time on mobile and embedded devices.

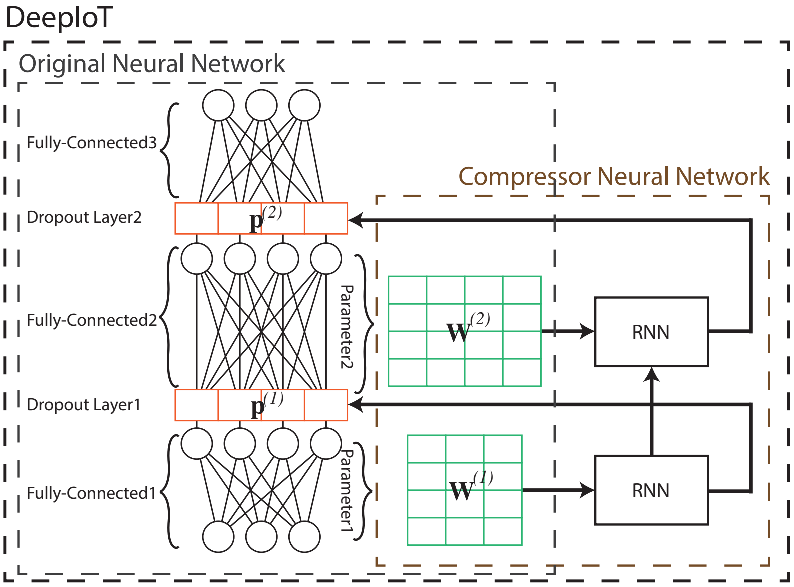


Figure 4.1: Overall DeepIoT system framework. Orange boxes represent dropout operations. Green boxes represent parameters of the original neural network.

4.1 THE DESIGN OF DEEPIOT FRAMEWORK

Without loss of generality, before introducing the technical details, we first use an example of compressing a 3-layer fully-connected neural network structure to illustrate the overall pipeline of DeepIoT. The detailed illustration is shown in Figure 4.1. The basic steps of compressing neural network structures for sensing applications with DeepIoT can be summarized as follows.

1. Insert operations that randomly zeroing out hidden elements with probabilities $\mathbf{p}^{(l)}$ called dropout (red boxes in Figure 4.1) into internal layers of the original neural network. The internal layers exclude input layers and output layers that have the fixed dimension for a sensing application. This step will be detailed in Section 4.1.1.

2. Construct the compressor neural network. It takes the weight matrices $\mathbf{W}^{(l)}$ (green boxes in Figure 4.1) from the layers to be compressed in the original neural network as inputs, learns and shares the parameter redundancies among different layers, and generates optimal dropout probabilities $\mathbf{p}^{(l)}$, which is then fed back to the dropout operations in the original neural network. This step will be detailed in Section 4.1.2.
3. Iteratively optimize the compressor neural network and the original neural network with the compressor-critic framework. The compressor neural network is optimized to produce better dropout probabilities that can generate a more efficient network structure for the original neural network. The original neural network is optimized to achieve a better performance with the more efficient structure for a sensing application. This step will be detailed in Section 4.1.3.

4.1.1 Dropout Operations in the Original Neural Network

Dropout is commonly used as a regularization method that prevents feature co-adapting and model overfitting. The term "dropout" refers to dropping out units (hidden and visible) in a neural network. Since DeepIoT is a structure compression framework, we focus mainly on dropping out hidden units. The definitions of hidden units are distinct in different types of neural networks, and we will describe them in detail. The basic idea is that we regard neural networks with dropout operations as bayesian neural networks with Bernoulli variational distributions [51–53].

For the fully-connected neural networks, the fully-connected operation with dropout can be formulated as

$$\begin{aligned}
\mathbf{z}^{(l)}_{[j]} &\sim \text{Bernoulli}(\mathbf{p}^{(l)}_{[j]}), \\
\tilde{\mathbf{W}}^{(l)} &= \mathbf{W}^{(l)}\text{diag}\big(\mathbf{z}^{(l)}\big), \\
\mathbf{Y}^{(l)} &= \mathbf{X}^{(l)}\tilde{\mathbf{W}}^{(l)} + \mathbf{b}^{(l)}, \\
\mathbf{X}^{(l+1)} &= f\big(\mathbf{Y}^{(l)}\big).
\end{aligned} \tag{4.1}$$

Refer to (4.1). The notation $l = 1, \cdots, L$ is the layer number in the fully-connected neural network. For any layer l, the weight matrix is denoted as $\mathbf{W}^{(l)} \in \mathbb{R}^{d^{(l-1)} \times d^{(l)}}$; the bias vector is denoted as $\mathbf{b}^{(l)} \in \mathbb{R}^{d^{(l)}}$; and the input is denoted as $\mathbf{X}^{(l)} \in \mathbb{R}^{1 \times d^{(l-1)}}$. In addition, $f(\cdot)$ is a nonlinear activation function.

As shown in (4.1), each hidden unit is controlled by a Bernoulli random variable. In the original dropout method, the success probabilities of $\mathbf{p}^{(l)}_{[j]}$ can be set to the same constant p for all hidden units [53], but DeepIoT uses the Bernoulli random variable with individual success probabilities for different hidden units in order to compress the neural network structure in a finer granularity.

For the convolutional neural networks, the basic fully-connected operation is replaced by the convolution operation [51]. However, the convolution can be reformulated as a linear operation as shown in (4.1). For any layer l, we denote $\mathcal{K}^{(l)} = \left\{\mathbf{K}_k^{(l)}\right\}$ for $k = 1, \cdots, c^{(l)}$ as the set of convolutional neural network (CNN)'s kernels, where $\mathbf{K}_k^{(l)} \in \mathbb{R}^{h^{(l)} \times w^{(l)} \times c^{(l-1)}}$ is the kernel of CNN with height $h^{(l)}$, width $w^{(l)}$, and channel $c^{(l-1)}$. The input tensor of layer l is denoted as $\hat{\mathbf{X}}^{(l)} \in \mathbb{R}^{\hat{h}^{(l-1)} \times \hat{w}^{(l-1)} \times c^{(l-1)}}$ with height $\hat{h}^{(l-1)}$, width $\hat{w}^{(l-1)}$, and channel $c^{(l-1)}$.

Next, we convert convolving the kernels with the input into performing matrix product. We extract $h^{(l)} \times w^{(l)} \times c^{(l-1)}$ dimensional patches from the input $\hat{\mathbf{X}}^{(l)}$ with stride s and vectorize them. Collect these vectorized n patches to be the rows of our new input representation $\mathbf{X}^{(l)} \in \mathbb{R}^{n \times (h^{(l)} w^{(l)} c^{(l-1)})}$. The vectorized kernels form the columns of the weight matrix $\mathbf{W}^{(l)} \in \mathbb{R}^{(h^{(l)} w^{(l)} c^{(l-1)}) \times c^{(l)}}$.

With this transformation, dropout operations can be applied to convolutional neural networks according to (4.1). The composition of pooling and activation functions can be regarded as the nonlinear function $f(\cdot)$ in (4.1). Instead of dropping out hidden elements in each layer, we drop out convolutional kernels in each layer. From the perspective of structure compression, DeepIoT tries to prune the number of kernels used in the convolutional neural networks.

For the recurrent neural network, we take a multi-layer Long Short Term Memory network (LSTM) as an example. The LSTM operation with dropout can be formulated as

$$\begin{aligned}
\mathbf{z}_{[j]}^{(l)} &\sim \text{Bernoulli}(\mathbf{p}_{[j]}^{(l)}), \\
\begin{pmatrix} \mathbf{i} \\ \mathbf{f} \\ \mathbf{o} \\ \mathbf{g} \end{pmatrix} &= \begin{pmatrix} \text{sigm} \\ \text{sigm} \\ \text{sigm} \\ \text{tanh} \end{pmatrix} \mathbf{W}^{(l)} \begin{pmatrix} \mathbf{h}_t^{(l-1)} \odot \mathbf{z}^{(l-1)} \\ \mathbf{h}_{t-1}^{(l)} \odot \mathbf{z}^{(l)} \end{pmatrix}, \\
\mathbf{c}_t^{(l)} &= \mathbf{f} \odot \mathbf{c}_{t-1}^{(l)} + \mathbf{i} \odot \mathbf{g}, \\
\mathbf{h}_t^{(l)} &= \mathbf{o} \odot \tanh\bigl(\mathbf{c}_t^{(l)}\bigr).
\end{aligned} \tag{4.2}$$

The notation $l = 1, \cdots, L$ is the layer number and $t = 1, \cdots, T$ is the step number in the recurrent neural network. Element-wise multiplication is denoted by $\odot$. Operators *sigm* and *tanh* denote sigmoid function and hyperbolic tangent respectively. The vector $\mathbf{h}_t^{(l)} \in \mathbb{R}^{n^{(l)}}$ is the output of step t at layer l. The vector $\mathbf{h}_t^{(0)} = \mathbf{x}_t$ is the input for the whole neural network at step t. The matrix $\mathbf{W}^{(l)} \in \mathbb{R}^{4n^{(l)} \times (n^{(l-1)} + n^{(l)})}$ is the weight matrix at layer l. We let $\mathbf{p}_{[j]}^{(0)} = 1$, since DeepIoT only drops hidden elements.

As shown in (4.2), DeepIoT uses the same vector of Bernoulli random variables $\mathbf{z}^{(l)}$ to control the dropping operations among different time steps in each layer, while individual Bernoulli random variables are used for different steps in the original LSTM dropout [34]. From the perspective of structure compression, DeepIoT tries to prune the number of hidden dimensions used in LSTM blocks. The dropout operation of other recurrent neural network architectures, such as Gated Recurrent Unit (GRU), can be designed similarly.

4.1.2 Compressor Neural Network

Now we introduce the architecture of the compressor neural network. A hidden element in the original neural network that is connected to redundant model parameters should have a higher probability to be dropped. Therefore we design the compressor neural network to take the weights of an original neural network $\{\mathbf{W}^{(l)}\}$ as inputs, learn the redundancies among these weights, and generate dropout probabilities $\{\mathbf{p}^{(l)}\}$ for hidden elements that can be eventually used to compress the original neural network structure.

A straightforward solution is to train an individual fully-connected neural network for each layer in the original neural network. However, since there are interconnections among weight redundancies in different layers, DeepIoT uses a variant LSTM as the structure of compressor to share and use the parameter redundancy information among different layers.

According to the description in Section 4.1.1, the weight in layer l of fully-connected, convolutional, or recurrent neural network can all be represented as a single matrix $\mathbf{W}^{(l)} \in \mathbb{R}^{d_f^{(l)} \times d_{\text{drop}}^{(l)}}$, where $d_{\text{drop}}^{(l)}$ denotes the dimension that dropout operation is applied and $d_f^{(l)}$ denotes the dimension of features within each dropout element. Here, we need to notice that the weight matrix of LSTM at layer l can be reshaped as $\mathbf{W}^{(l)} \in \mathbb{R}^{4 \cdot (n^{(l-1)} + n^{(l)}) \times n^{(l)}}$, where $d_{\text{drop}}^{(l)} = n^{(l)}$ and $d_f^{(l)} = 4 \cdot (n^{(l-1)} + n^{(l)})$. Hence, we take weights from the original network layer by layer, $\mathcal{W} = \left\{\mathbf{W}^{(l)}\right\}$ with $l = 1, \cdots, L$, as the input of the compressor neural network. Instead of using a vanilla LSTM as the structure of compressor, we apply a variant l-step LSTM model shown as

$$\begin{aligned}
\begin{pmatrix}\mathbf{v}_i^\intercal\\ \mathbf{v}_f^\intercal\\ \mathbf{v}_o^\intercal\\ \mathbf{v}_g^\intercal\end{pmatrix} &= \mathbf{W}_c^{(l)}\mathbf{W}^{(l)}\mathbf{W}_i^{(l)}, \qquad \begin{pmatrix}\mathbf{u}_i\\ \mathbf{u}_f\\ \mathbf{u}_o\\ \mathbf{u}_g\end{pmatrix} = \mathbf{W}_h\mathbf{h}_{l-1},\\
\begin{pmatrix}\mathbf{i}\\ \mathbf{f}\\ \mathbf{o}\\ \mathbf{g}\end{pmatrix} &= \begin{pmatrix}\mathrm{sigm}\\ \mathrm{sigm}\\ \mathrm{sigm}\\ \tanh\end{pmatrix}\left(\begin{pmatrix}\mathbf{v}_i\\ \mathbf{v}_f\\ \mathbf{v}_o\\ \mathbf{v}_g\end{pmatrix} + \begin{pmatrix}\mathbf{u}_i\\ \mathbf{u}_f\\ \mathbf{u}_o\\ \mathbf{u}_g\end{pmatrix}\right),\\
\mathbf{c}_l &= \mathbf{f}\odot\mathbf{c}_{l-1} + \mathbf{i}\odot\mathbf{g},\\
\mathbf{h}_l &= \mathbf{o}\odot\tanh\big(\mathbf{c}_l\big),\\
\mathbf{p}^{(l)} &= \mathbf{p}_t = \mathrm{sigm}\big(\mathbf{W}_o^{(l)}\mathbf{h}_l\big),\\
\mathbf{z}_{[j]}^{(l)} &\sim \mathrm{Bernoulli}(\mathbf{p}_{[j]}^{(l)}).
\end{aligned} \tag{4.3}$$

Refer to (4.3), we denote d_c as the dimension of the variant LSTM hidden state. Then $\mathbf{W}^{(l)} \in \mathbb{R}^{d_f^{(l)}\times d_{\mathrm{drop}}^{(l)}}$, $\mathbf{W}_c^{(l)} \in \mathbb{R}^{4\times d_f^{(l)}}$, $\mathbf{W}_i^{(l)} \in \mathbb{R}^{d_{\mathrm{drop}}^{(l)}\times d_c}$, $\mathbf{W}_h \in \mathbb{R}^{4d_c\times d_c}$, and $\mathbf{W}_o^{(l)} \in \mathbb{R}^{d_{\mathrm{drop}}^{(l)}\times d_c}$. The set of training parameters of the compressor neural network is denoted as ϕ, where $\phi = \big\{\mathbf{W}_c^{(l)}, \mathbf{W}_i^{(l)}, \mathbf{W}_h, \mathbf{W}_o^{(l)}\big\}$. The matrix $\mathbf{W}^{(l)}$ is the input matrix for step l in the compressor neural network, which is also the l^{th} layer's parameters of the original neural network in (4.1) or (4.2).

Compared with the vanilla LSTM that requires vectorizing the original weight matrix as inputs, the variant LSTM model preserves the structure of original weight matrix and uses less learning parameters to extract the redundancy information among the dropout elements. In addition, $\mathbf{W}_c^{(l)}$ and $\mathbf{W}_i^{(l)}$ convert original weight matrix $\mathbf{W}^{(l)}$ with different sizes into fixed-size representations. The binary vector $\mathbf{z}^{(l)}$ is the dropout mask and probability $\mathbf{p}^{(l)}$ is the dropout probabilities for the l^{th} layer in the original neural network used in (4.1) and (4.2), which is also the stochastic dropout policy learnt through observing the weight redundancies of the original neural network.

4.1.3 Compressor-Critic Framework

In Section 4.1.1 and Section 4.1.2, we have introduced customized dropout operations applied on the original neural networks that need to be compressed and the structure of compressor neural network used to learn dropout probabilities based on parameter redundancies. In this subsection, we will discuss the detail of compressor-critic compressing process. It optimizes the original neural network and the compressor neural network in an iterative manner and enables the compressor neural network to gradually compress the original neural network with soft deletion.

We denote the original neural network as $F_{\mathcal{W}}(\mathbf{x}|\mathbf{z})$, and we call it critic. It takes $\mathbf{x}$ as inputs and generates predictions based on binary dropout masks $\mathbf{z}$ and model parameters $\mathcal{W}$ that refer to a set of weights $\mathcal{W} = \{\mathbf{W}^{(l)}\}$. We assume that $F_{\mathcal{W}}(\mathbf{x}|\mathbf{z})$ is a pre-trained model. We denote the compressor neural network by $\mathbf{z} \sim \mu_\phi(\mathcal{W})$. It takes the weights of the critic as inputs and generates the probability distribution of the mask vector $\mathbf{z}$ based on its own parameters ϕ. In order to optimize the compressor to drop out hidden elements in the critic, DeepIoT follows the objective function

$$\begin{aligned}\mathcal{L} &= \mathbb{E}_{\mathbf{z}\sim\mu_\phi}\big[L\big(\mathbf{y}, F_{\mathcal{W}}(\mathbf{x}|\mathbf{z})\big)\big] \\ &= \sum_{\mathbf{z}\sim\{0,1\}^{|\mathbf{z}|}} \mu_\phi(\mathbf{W}) \cdot L\big(\mathbf{y}, F_{\mathcal{W}}(\mathbf{x}|\mathbf{z})\big),\end{aligned} \tag{4.4}$$

where $L(\cdot,\cdot)$ is the objective function of the critic. The objective function can be interpreted as the expected loss of the original neural network over the dropout probabilities generated by the compressor.

DeepIoT optimizes the compressor and critic in an iterative manner. It reduces the expected loss as defined in (4.4) by applying the gradient descent method on compressor and critic iteratively. However, since there are discrete sampling operations, *i.e.*, dropout operations, within the computational graph, backpropagation is not directly applicable. Therefore we apply an unbiased likelihood-ratio estimator to calculate the gradient over ϕ [54, 55]:

$$\begin{aligned}\nabla_\phi \mathcal{L} &= \sum_{\mathbf{z}} \nabla_\phi \mu_\phi(\mathcal{W}) \cdot L\big(\mathbf{y}, F_{\mathcal{W}}(\mathbf{x}|\mathbf{z})\big) \\ &= \sum_{\mathbf{z}} \mu_\phi(\mathcal{W}) \nabla_\phi \log \mu_\phi(\mathcal{W}) \cdot L\big(\mathbf{y}, F_{\mathcal{W}}(\mathbf{x}|\mathbf{z})\big) \\ &= \mathbb{E}_{\mathbf{z}\sim\mu_\phi}\big[\nabla_\phi \log \mu_\phi(\mathcal{W}) \cdot L\big(\mathbf{y}, F_{\mathcal{W}}(\mathbf{x}|\mathbf{z})\big)\big].\end{aligned} \tag{4.5}$$

Therefore an unbiased estimator for (4.5) can be

$$\widehat{\nabla_\phi \mathcal{L}} = \nabla_\phi \log \mu_\phi(\mathcal{W}) \cdot L\big(\mathbf{y}, F_{\mathcal{W}}(\mathbf{x}|\mathbf{z})\big) \quad \mathbf{z} \sim \mu_\phi. \tag{4.6}$$

The gradient over $\mathbf{W}^{(l)} \in \mathcal{W}$ is

$$\begin{aligned}\nabla_{\mathbf{W}^{(l)}} \mathcal{L} &= \sum_{\mathbf{z}} \mu_\phi(\mathcal{W}) \cdot \nabla_{\mathbf{W}^{(l)}} L\big(\mathbf{y}, F_{\mathcal{W}}(\mathbf{x}|\mathbf{z})\big) \\ &= \mathbb{E}_{\mathbf{z}\sim\mu_\phi}\big[\nabla_{\mathbf{W}^{(l)}} L\big(\mathbf{y}, F_{\mathcal{W}}(\mathbf{x}|\mathbf{z})\big)\big].\end{aligned} \tag{4.7}$$

Similarly, an unbiased estimator for (4.7) can be

$$\widehat{\nabla_{\mathbf{W}^{(l)}} \mathcal{L}} = \nabla_{\mathbf{W}^{(l)}} L\big(\mathbf{y}, F_{\mathcal{W}}(\mathbf{x}|\mathbf{z})\big) \quad \mathbf{z} \sim \mu_\phi. \tag{4.8}$$

Now we provide more details of $\widehat{\nabla_\phi \mathcal{L}}$ in (4.6). Although the estimator (4.6) is an unbiased estimator, it tends to have a higher variance. A higher variance of estimator can make the convergence slower. Therefore, variance reduction techniques are typically required to make the optimization feasible in practice [56, 57].

One variance reduction technique is to subtract a constant c from learning signal $L\big(\mathbf{y}, F_{\mathcal{W}}(\mathbf{x}|\mathbf{z})\big)$ in (4.5), which still keeps the expectation of the gradient unchanged [56]. Therefore, we keep track of the moving average of the learning signal $L\big(\mathbf{y}, F_{\mathcal{W}}(\mathbf{x}|\mathbf{z})\big)$ denoted by c, and subtract c from the gradient estimator (4.6).

The other variance reduction technique is keeping track of the moving average of the signal variance v, and divides the learning signal by $\max(1, \sqrt{v})$ [57].

Combing the aforementioned two variance reduction techniques, the final estimator (4.6) for gradient over ϕ becomes

$$\widehat{\nabla_\phi \mathcal{L}} = \nabla_\phi \log \mu_\phi(\mathcal{W}) \cdot \frac{L\big(\mathbf{y}, F_{\mathcal{W}}(\mathbf{x}|\mathbf{z})\big) - c}{\max(1, \sqrt{v})} \quad \mathbf{z} \sim \mu_\phi, \tag{4.9}$$

where c and v are the moving average of mean and the moving average of variance of learning signal $L\big(\mathbf{y}, F_{\mathcal{W}}(\mathbf{x}|\mathbf{z})\big)$ respectively.

After introducing the basic optimization process in DeepIoT, now we are ready to deliver the details of the compressing process. Compared with previous compressing algorithms that gradually delete weights without rehabilitation [58], DeepIoT applies “soft” deletion by gradually suppressing the dropout probabilities of hidden elements with a decay factor $\gamma \in (0, 1)$. During the experiments in Section 4.2, we set γ as the default value 0.5. Since it is impossible to make the optimal compression decisions from the beginning, suppressing the dropout probabilities instead of deleting the hidden elements directly can provide the “deleted” hidden elements changes to recover. This less aggressive compression process reduces the potential risk of irretrievable network damage and learning inefficiency.

During the compressing process, DeepIoT gradually increases the threshold of dropout probability τ from 0 with step Δ. The hidden elements with dropout probability, $\mathbf{p}^{(l)}_{[j]}$ that is less than the threshold τ will be given decay on dropout probability, *i.e.*, $\hat{\mathbf{p}}^{(l)}_{[j]} \leftarrow \gamma \cdot \mathbf{p}^{(l)}_{[j]}$. Therefore, the operation in compressor (4.3) can be updated as

$$\mathbf{z}^{(l)}_{[j]} \sim \text{Bernoulli}\Big(\mathbf{p}^{(l)}_{[j]} \cdot \gamma^{\mathbb{1}_{\mathbf{p}^{(l)}_{[j]} \leq \tau}}\Big), \tag{4.10}$$

where $\mathbb{1}$ is the indicator function; $\gamma \in (0, 1)$ is the decay factor; and $\tau \in [0, 1)$ is the threshold. Since the operation of suppressing dropout probability with the pre-defined decay factor γ is differentiable, we can still optimize the original and the compressor neural network through (4.8) and (4.9). The compression process will stop when the percentage of left number of parameters in $F_{\mathcal{W}}(\mathbf{x}|\mathbf{z})$ is smaller than a user-defined value $\alpha \in (0, 1)$.

Algorithm 4.1. Compressor-predictor compressing process

```
1: **Input:** pre-trained predictor $F_{\mathcal{W}}(\mathbf{x}|\mathbf{z})$
2: **Initialize:** compressor $\mu_{\phi}(\mathcal{W})$ with parameter $\phi$, moving average $c$, moving average of variance $v$
3: **while** $\mu_{\phi}(\mathcal{W})$ is not convergent **do**
4:     $\mathbf{z} \sim \mu_{\phi}(\mathcal{W})$
5:     $c \leftarrow \text{movingAvg}\big(L\big(\mathbf{y}, F_{\mathcal{W}}(\mathbf{x}|\mathbf{z})\big)\big)$
6:     $v \leftarrow \text{movingVar}\big(L\big(\mathbf{y}, F_{\mathcal{W}}(\mathbf{x}|\mathbf{z})\big)\big)$
7:     $\phi \leftarrow \phi - \beta \cdot \nabla_{\phi} \log \mu_{\phi}(\mathcal{W}) \cdot \big(L\big(\mathbf{y}, F_{\mathcal{W}}(\mathbf{x}|\mathbf{z})\big) - c\big)/\max(1, \sqrt{v})$
8: **end while**
9: $\tau = 0$
10: **while** the percentage of left number of parameters in $F_{\mathcal{W}}(\mathbf{x}|\mathbf{z})$ is larger than $\alpha$ **do**
11:     $\mathbf{z} \sim \mu_{\phi}(\mathcal{W})$
12:     $c \leftarrow \text{movingAvg}\big(L\big(\mathbf{y}, F_{\mathcal{W}}(\mathbf{x}|\mathbf{z})\big)\big)$
13:     $v \leftarrow \text{movingVar}\big(L\big(\mathbf{y}, F_{\mathcal{W}}(\mathbf{x}|\mathbf{z})\big)\big)$
14:     $\phi \leftarrow \phi - \beta \cdot \nabla_{\phi} \log \mu_{\phi}(\mathcal{W}) \cdot \big(L\big(\mathbf{y}, F_{\mathcal{W}}(\mathbf{x}|\mathbf{z})\big) - c\big)/\max(1, \sqrt{v})$
15:     $\mathcal{W} \leftarrow \mathcal{W} - \beta \cdot \nabla_{\mathcal{W}} L\big(\mathbf{y}, F_{\mathcal{W}}(\mathbf{x}|\mathbf{z})\big)$
16:     update threshold $\tau$: $\tau \leftarrow \tau + \Delta$ for every $T$ rounds
17: **end while**
18: $\hat{\mathbf{z}}^{(l)}_{[j]} = \mathbb{1}\mathbf{p}^{(l)}_{[j]} > \tau$
19: **while** $F_{\mathcal{W}}(\mathbf{x}|\hat{\mathbf{z}})$ is not convergent **do**
20:     $\mathcal{W} \leftarrow \mathcal{W} - \beta \cdot \nabla_{\mathcal{W}} L\big(\mathbf{y}, F_{\mathcal{W}}(\mathbf{x}|\hat{\mathbf{z}})\big)$
21: **end while**
```

After the compression, DeepIoT fine-tunes the compressed model $F_{\mathcal{W}}(\mathbf{x}|\hat{\mathbf{z}})$, with a fixed mask $\hat{\mathbf{z}}$, which is decided by the previous threshold τ. Therefore the mask generation step in (4.10) will be updated as

$$\hat{\mathbf{z}}^{(l)}_{[j]} = \mathbb{1}\mathbf{p}^{(l)}_{[j]} > \tau. \tag{4.11}$$

We summarize the compressor-critic compressing process of DeepIoT in Algorithm 4.1.

The algorithm consists of three parts. In the first part (Line 3 to Line 8), DeepIoT freezes the critic $F_{\mathcal{W}}(\mathbf{x}|\mathbf{z})$ and initializes the compressor $\mu_{\phi}(\mathcal{W})$ according to (4.9). In the second part (Line 9 to Line 17), DeepIoT optimizes the critic and compressor jointly with the gradients calculated by (4.8) and (4.9). At the same time, DeepIoT gradually compresses the predictor by suppressing dropout probabilities according to (4.10). In the final part (Line 18 to Line 21), DeepIoT fine-tunes the critic with the gradient calculated by (4.8) and a deterministic dropout mask is generated according to (4.11). After these three phases, DeepIoT generates a binary dropout mask $\hat{\mathbf{z}}$ and the fine-tuning parameters of the critic $\mathcal{W}$. With these two results, we can easily obtain the compressed model of the original neural network.

4.2 THE EVALUATION OF DEEPIOT

In this section, we evaluate DeepIoT through three representative sensing tasks. The first set is motivated by the prospect of enabling future smarter embedded "things" (physical objects) to interact with humans using user-friendly modalities such as visual cues, handwritten text, and speech commands, while the second evaluates human-centric context sensing, such as human activity recognition and user identification. In the following subsections, we first describe the comparison baselines that are current state of the art deep neural network compression techniques. We then present the first set of experiments that demonstrate accuracy and resource demands observed if IoT-style smart objects interacted with users via natural human-centric modalities thanks to deep neural networks compressed, for the resource-constrained hardware, with the help of our DeepIoT framework. Finally, we present the second set of experiments that demonstrate accurancy and resource demands when applying DeepIoT to compress deep neural networks trained for human-centric context sensing applications.

4.2.1 Evaluation Platforms

Our hardware is based on Intel Edison computing platform [38]. The Intel Edison computing platform is powered by the Intel Atom SoC dual-core CPU at 500 MHz and is equipped with 1GB memory and 4GB flash storage. For fairness, all neural network models are run solely on CPU during experiments.

All the original neural networks for all sensing applications are trained on the workstation with NVIDIA GeForce GTX Titan X. For all baseline algorithms mentioned in Section 4.2.2, the compressing processes are also conducted on the workstation. The compressed models are exported and loaded into the flash storage on Intel Edison for experiments.

We installed the Ubilinux operation system on Intel Edison computing platform [59]. Far fairness, all compressed deep learning models are run through Theano [60] with only CPU device on Intel Edison. The matrix multiplication operations and sparse matrix multiplication operations are optimized by BLAS and Sparse BLAS respectively during the implementation. No additional run-time optimization is applied for any compressed model and in all experiments.

4.2.2 Baseline Algorithms

We compare DeepIoT with other three baseline algorithms:

1. **DyNS:** This is a magnitude-based network pruning algorithm [20]. The algorithm prunes weights in convolutional kernels and fully-connected layer based on the magnitude. It retrains the network connections after each pruning step and has the ability

to recover the pruned weights. For convolutional and fully-connected layers, DyNS searches the optimal thresholds separately.

2. **SparseSep:** This is a sparse-coding and factorization based algorithm [61]. The algorithm simplifies the fully-connected layer by finding the optimal code-book and code based on a sparse coding technique. For the convolutional layer, the algorithm compresses the model with matrix factorization methods. We greedily search for the optimal code-book and factorizaiton number from the bottom to the top layer.
3. **DyNS-Ext:** The previous two algorithms mainly focus on compressing convolutional and fully-connected layers. Therefore we further enhance and extend the magnitude-based method used in DyNS to recurrent layers and call this algorithm DyNS-Ext. Just like DeepIoT, DyNS-Ext can be applied to all commonly used deep network modules, including fully-connected layers, convolutional layers, and recurrent layers. If the network structure does not contain recurrent layers, we apply DyNS instead of DyNS-Ext.

For magnitude-based pruning algorithms, DyNS and DyNS-Ext, hidden elements with zero input connections or zero output connections will be pruned to further compress the network structure. In addition, all models use 32-bit floats without any quantization.

Handwritten digits recognition with LeNet5

The first human interaction modality is recognizing handwritten text. In this experiment, we consider a meaningful subset of that; namely recognizing handwritten digits from visual inputs. An example application that uses this capability might be a smart wallet equipped with a camera and a tip calculator. We use MNIST[1] as our training and testing dataset. The MNIST is a dataset of handwritten digits that is commonly used for training various image processing systems. It has a training set of 60000 examples, and a test set of 10000 examples.

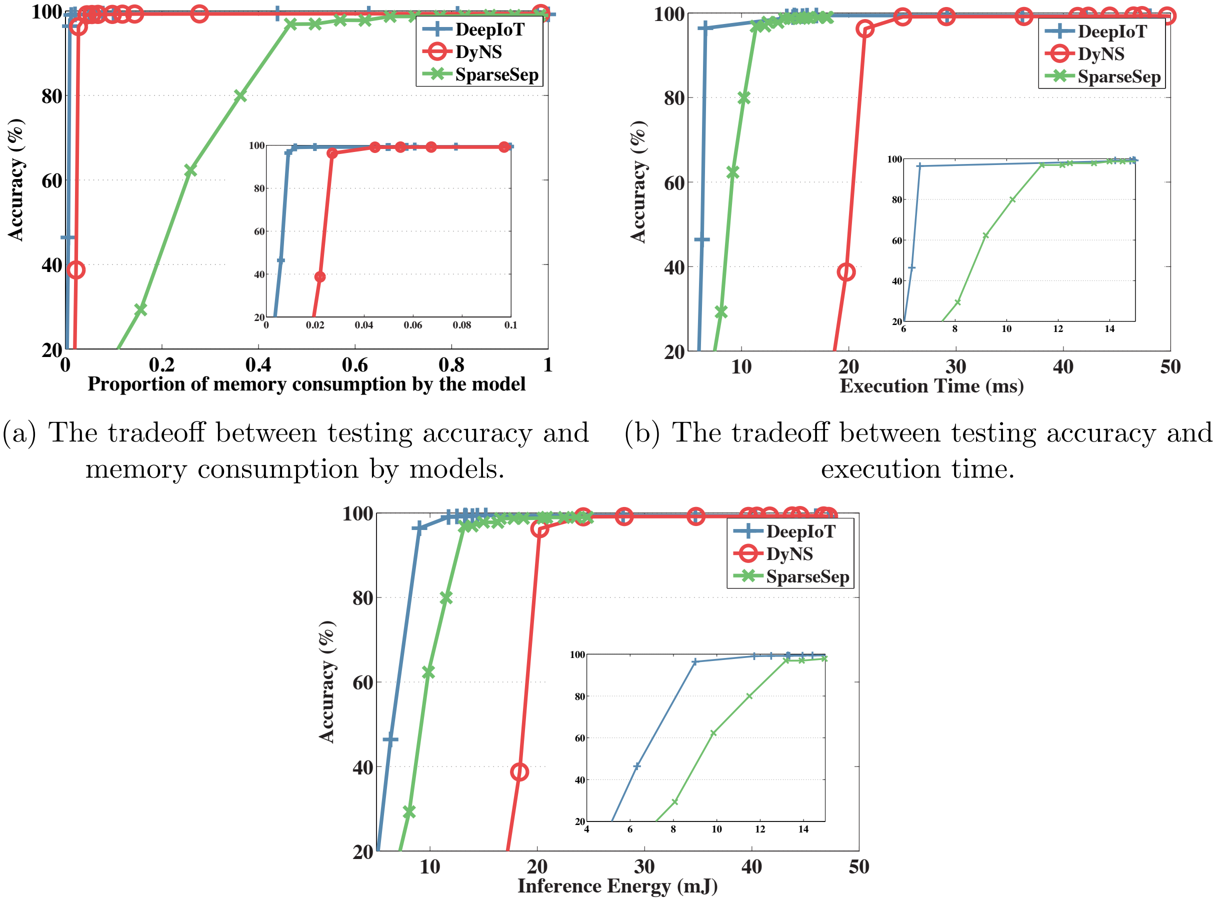
We test our algorithms and baselines on the LeNet-5 neural network model. The corresponding network structure is shown in Table 4.1. Notice that we omit all the polling layers in Table 4.1 for simplicity, because they do not contain training parameters.

The first column of Table 4.1 represents the network structure of LeNet-5, where “convX” represents the convolutional layer and “fcY” represents the fully-connected layer. The second column represents the number of hidden units or convolutional kernels we used in each layer.

[1] http://yann.lecun.com/exdb/mnist/

Table 4.1: LeNet5 on MNIST dataset

Layer	Hidden Units	Params	DeepIoT (Hidden Units/ Params)		DyNS	SparseSep
conv1 (5×5)	20	0.5K	10	50.0%	24.2%	84%
conv2 (5×5)	50	25K	20	20.0%	20.7%	91%
fc1	500	400K	10	0.8%	1.0%	78.75%
fc2	10	5K	10	2.0%	16.34%	70.28%
total		431K		1.98%	2.35%	72.39%
Test Error	0.85%		0.85%		0.85%	1.05%



(a) The tradeoff between testing accuracy and memory consumption by models.

(b) The tradeoff between testing accuracy and execution time.

(c) The tradeoff between testing accuracy and energy consumption.

Figure 4.2: System performance tradeoff for LeNet5 on MNIST dataset

The third column represents the number of parameters used in each layer and in total. The original LeNet-5 is trained and achieves an error rate of 0.85% in the test dataset.

We then apply DeepIoT and two other baseline algorithms, DyNS and SparseSep, to compress LeNet-5. Note that, we do not use DyNS-Ext because the network does not contain a recurrent layer. The network statistics of the compressed model are shown in Table 4.1. DeepIoT is designed to prune the number of hidden units for a more efficient network structure. Therefore, we illustrate both the remaining number of hidden units and the proportion of the remaining number of parameters in Table 4.1. Both DeepIoT and DyNS can significantly compress the network without hurting the final performance. SparseSep shows an acceptable drop of performance. This is because SparseSep is designed without fine-tuning. It has the benefit of not fine-tuning the model, but it suffers the loss in the final performance at the same time.

The detailed tradeoff between testing accuracy and memory consumption by the model is illustrated in Fig 4.2a. We compress the original neural network with different compression ratios and recode the final testing accuracy. In the zoom-in illustration, DeepIoT achieves at least ×2 better tradeoff compared with the two baseline methods. This is mainly due to two reasons. One is that the compressor neural network in DeepIoT obtains a global view of parameter redundancies and is therefore better capable of eliminating them. The other is that DeepIoT prunes the hidden units directly, which enables us to represent the compressed model parameters with a small dense matrix instead of a large sparse matrix. The sparse matrix consumes more memory for the indices of matrix elements. Algorithms such as DyNS generate models represented by sparse matrices that cause larger memory consumption.

The evaluation results on execution time of compressed models on Intel Edison, are illustrated in Fig. 4.2b. We run each compressed model on Intel Edison for 5000 times and use the mean value for generating the tradeoff curves.

DeepIoT still achieves the best tradeoff compared with other two baselines by a significant margin. DeepIoT takes 14.2ms to make a single inference, which reduces execution time by 71.4% compared with the original network without loss of accuracy. However SparseSep takes less execution time compared with DyNS at the cost of acceptable performance degradation (around 0.2% degradation on test error). The main reason for this observation is that, even though fully-connected layers occupy the most model parameters, most execution time is used by the convolution operations. SparseSep uses a matrix factorization method to covert the 2d convolutional kernel into two 1d convolutional kernels on two different dimensions. Although this method makes low-rank assumption on convolutional kernel, it can speed up convolution operations if the size of convolutional kernel is large (5×5 in this experiment). It can sometimes speed up the operation even when two 1d kernels have more parameters in total compared with the original 2d kernel. However DyNS applies a magnitude-based method that prunes most of the parameters in fully-connected layers. For convolutional layers, DyNS does not reduce the number of convolutional operations effectively, and sparse matrix multiplication is less efficient compared with regular matrix with the same number of elements. DeepIoT directly reduces the number of convolutional kernels in each layer, which reduces the number of operations in convolutional layers without making the low-rank assumption that can hurt the network performance.

The evaluation of energy consumption on Intel Edison is illustrated in Fig. 4.2c. For each compressed model, we run it for 5000 times and measure the total energy consumption by a power meter. Then, we calculate the expected energy consumption for one-time execution and use the one-time energy consumption to generate the tradeoff curves in Fig. 4.2c.

Not surprisingly, DeepIoT still achieves the best tradeoff in the evaluation on energy consumption by a significant margin. It reduces energy consumption by 73.7% compared with the original network without loss of accuracy. Being similar as the evaluation on execution time, energy consumption focuses more on the number of operations than the model size. Therefore, SparseSep can take less energy consumption compared with DyNS at the cost of acceptable loss on performance.

4.2.3 Image recognition with VGGNet

The task image recognition through low-resolution camera. During this experiment, we use CIFAR10[2] as our training and testing dataset. The CIFAR-10 dataset consists of 60000 32x32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images. It is a standard testing benchmark dataset for the image recognition tasks. While not necessarily representative of seeing objects in the wild, it offers a more controlled environment for an apples-to-apples comparison.

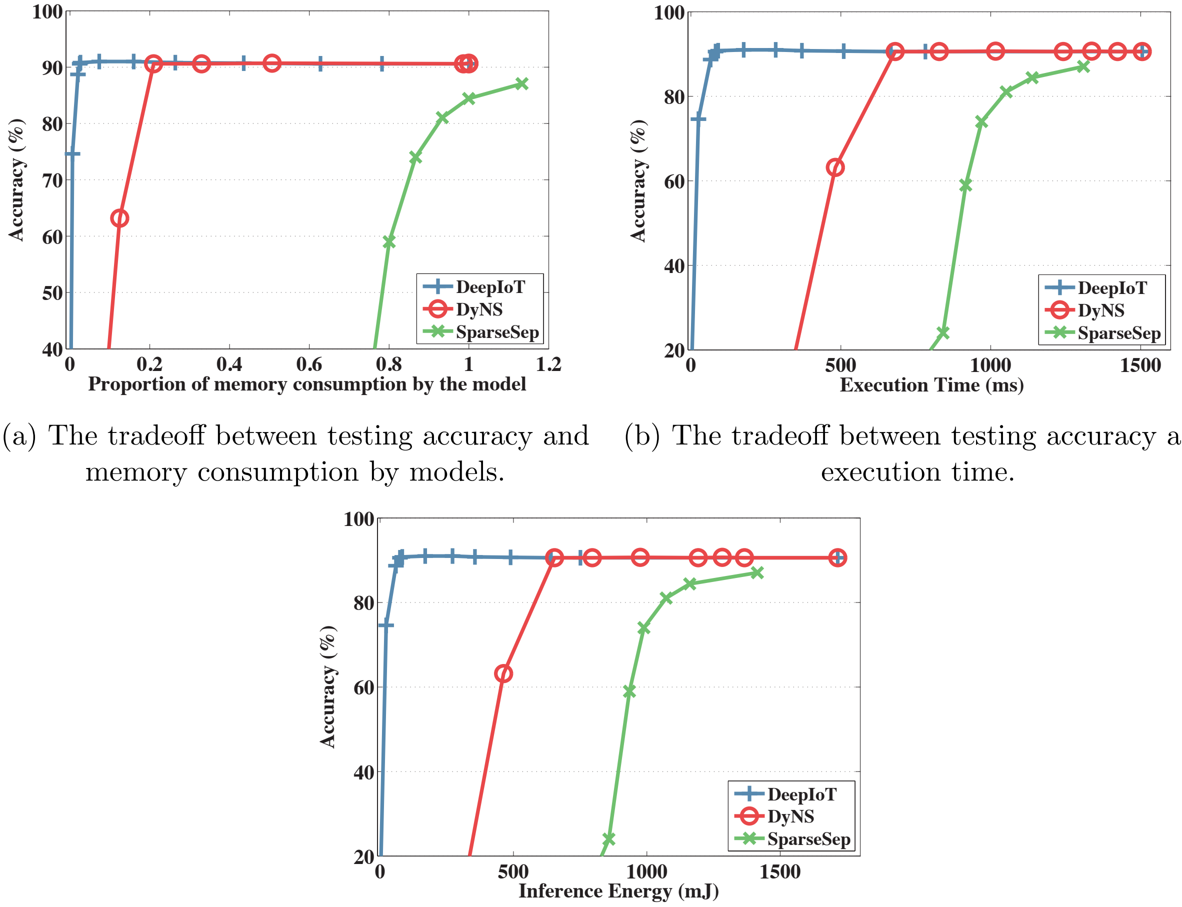
During this evaluation, we use the VGGNet structure as our original network structure. It is a huge network with millions of parameters. VGGNet is chosen to show that DeepIoT is able to compress relative deep and large network structure. The detailed structure is shown Table 4.2.

In Table 4.2, we illustrate the detailed statistics of best compressed model that keeps the original testing accuracy for three algorithms. We clearly see that DeepIoT beats the other two baseline algorithms by a significant margin. This shows that the compressor in DeepIoT can handle networks with relatively deep structure. The compressor uses a variant of the LSTM architecture to share the redundancy information among different layers. Compared with other baselines considering only local information within each layer, sharing the global information among layers helps us learn about the parameter redundancy and compress the network structure. In addition, we observe performance loss in the compressed network generated by SparseSep. It is mainly due to the fact that SparseSep avoids the fine-tuning step. This experiment shows that fine-tuning (Line 18 to Line 21 in Algorithm 4.1) is important for model compression.

[2] https://www.kaggle.com/c/cifar-10

Table 4.2: VGGNet on CIFAR-10 dataset

Layer	Hidden Units	Params	DeepIoT (Hidden Units/ Params)		DyNS	SparseSep
conv1 (3 × 3)	64	1.7K	27	42.2%	53.9%	93.1%
conv2 (3 × 3)	64	36.9K	47	31.0%	40.1%	57.3%
conv3 (3 × 3)	128	73.7K	53	30.4%	52.3%	85.1%
conv4 (3 × 3)	128	147.5K	68	22.0%	67.0%	56.8%
conv5 (3 × 3)	256	294.9K	104	21.6%	71.2%	85.1%
conv6 (3 × 3)	256	589.8K	97	15.4%	65.0%	56.8%
conv7 (3 × 3)	256	589.8K	89	13.2%	61.2%	56.8%
conv8 (3 × 3)	512	1.179M	122	8.3%	36.5%	85.2%
conv9 (3 × 3)	512	2.359M	95	4.4%	10.6%	56.8%
conv10 (3 × 3)	512	2.359M	64	2.3%	3.9%	56.8%
conv11 (2 × 2)	512	1.049M	128	3.1%	3.0%	85.2%
conv12 (2 × 2)	512	1.049M	112	5.5%	1.7%	85.2%
conv13 (2 × 2)	512	1.049M	149	6.4%	2.4%	85.2%
fc1	4096	2.097M	27	0.19%	2.2%	95.8%
fc2	4096	16.777M	371	0.06%	0.39%	135%
fc3	10	41K	10	9.1%	18.5%	90.2%
total		29.7M		2.44%	7.05%	112%
Test Accuracy	90.6%		90.6%		90.6%	87.1%



(a) The tradeoff between testing accuracy and memory consumption by models.

(b) The tradeoff between testing accuracy and execution time.

(c) The tradeoff between testing accuracy and energy consumption.

Figure 4.3: System performance tradeoff for VGGNet on CIFAR-10 dataset

Fig. 4.3a shows the tradeoff between testing accuracy and memory consumption for different models. DeepIoT achieves a better performance by even a larger margin, because the model generated by DeepIoT can still be represented by a standard matrix, while other methods that use a sparse matrix representation require more memory consumption.

Fig. 4.3b shows the tradeoff between testing accuracy and execution time for different models. DeepIoT still achieves the best tradeoff. DeepIoT takes 82.2ms for a prediction, which reduces 94.5% execution time without the loss of accuracy. DyNS uses less execution time compared with SparseSep in this experiment. There are two reasons for this. One is that VGGNet use smaller convolutional kernel compared with LeNet-5. Therefore factorizing 2d kernel into two 1d kernel helps less on reducing computation time. The other point is that SparseSep fails to compress the original network into a small size while keeping the original performance, because SparseSep avoids the fine-tuning.

Fig. 4.3c shows the tradeoff between testing accuracy and energy consumption for different models. DeepIoT reduces energy consumption by 95.7% compared with the original VGGNet without loss of accuracy. It greatly helps us to develop a long-standing application with deep neural network in energy-constrained embedded devices.

4.2.4 Speech recognition with deep Bidirectional LSTM

The task is about speech. The sensing system can take the voices of users from the microphone and automatically convert what users said into text. The previous experiment focus on the network structure with convolutional layers and fully-connected layers. We see how DeepIoT and the baseline algorithms work on the recurrent neural network in this section.

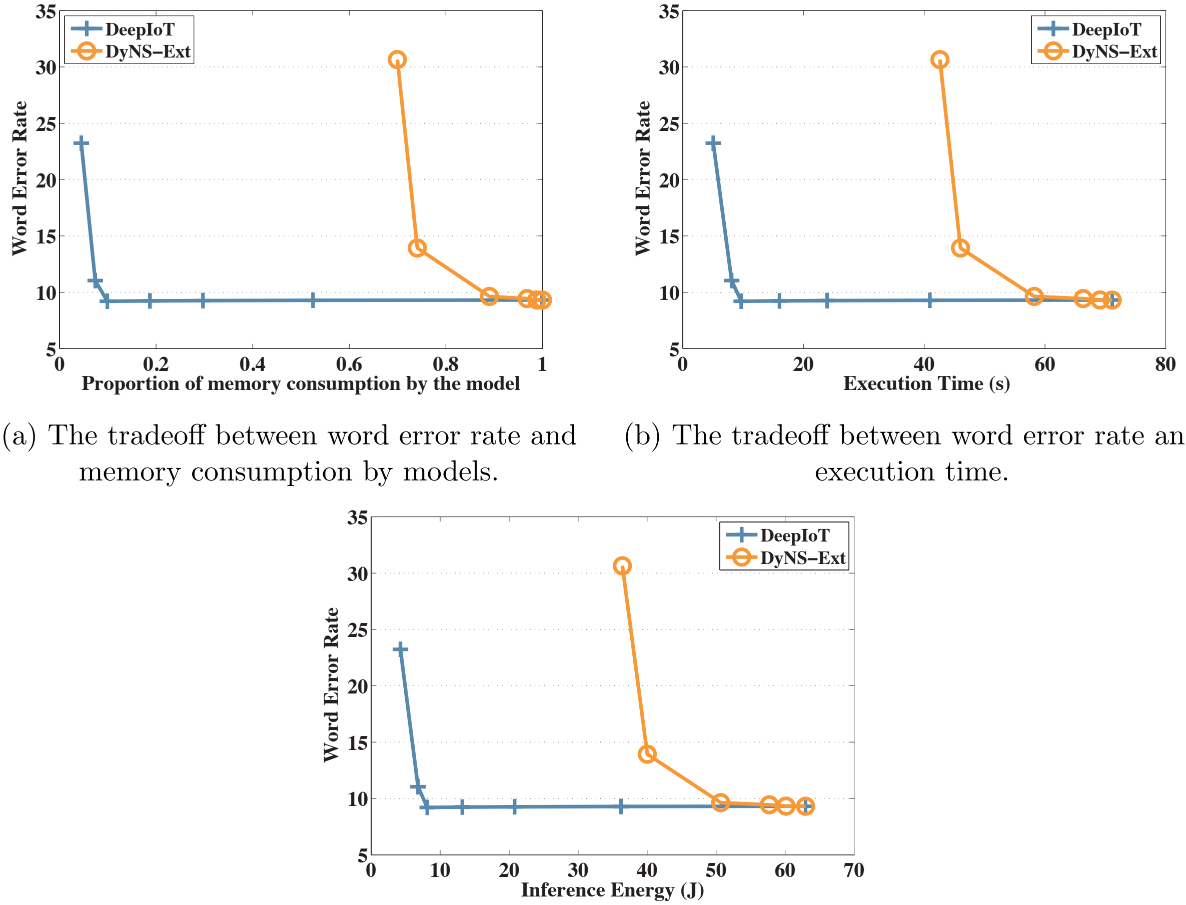
In this experiment, we use LibriSpeech ASR corpus [62] as our training and testing dataset. The LibriSpeech ASR corpus is a large-scale corpus of read English speech. It consists of 460-hour training data and 2-hour testing data.

We choose deep bidirectional LSTM as the original model [63] in this experiment. It takes mel frequency cepstral coefficient (MFCC) features of voices as inputs, and uses two 5-layer long short-term memory (LSTM) in both forward and backward direction. The output of two LSTM are jointly used to predict the spoken text. The detailed network structure is shown in the first column of Table 4.3, where “LSTMf” denotes the LSTM in forward direction and “LSTMb” denotes the LSTM in backward direction.

Two baseline algorithms are not applicable to the recurrent neural network, so we compared DeepIoT only with SyNS-Ext in this experiment. The word error rate (WER), defined as the edit distance between the true word sequence and the most probable word sequence

Table 4.3: Deep bidirectional LSTM on LibriSpeech ASR corpus

Layer		Hidden Unit		Params		DeepIoT (Hidden Units/ Params)				DyNS-Ext	
LSTMf1	LSTMb1	512	512	1.090M	1.090M	55	20	10.74%	3.91%	34.9%	18.2%
LSTMf2	LSTMb2	512	512	2.097M	2.097M	192	71	4.03%	0.54%	37.2%	23.1%
LSTMf3	LSTMb3	512	512	2.097M	2.097M	240	76	17.58%	2.06%	43.1%	27.9%
LSTMf4	LSTMb4	512	512	2.097M	2.097M	258	81	23.62%	2.35%	52.3%	40.2%
LSTMf5	LSTMb5	512	512	2.097M	2.097M	294	90	28.93%	2.78%	72.6%	61.8%
fc1		29		59.3K		29		37.5%		69.0%	
total				19.016M				9.98%		37.1%	
Word error rate (WER)		9.31				9.20				9.62	



(a) The tradeoff between word error rate and memory consumption by models.

(b) The tradeoff between word error rate and execution time.

(c) The tradeoff between word error rate and energy consumption.

Figure 4.4: System performance tradeoff for deep bidirectional LSTM on LibriSpeech ASR corpus

predicted by the neural network, is used as the evaluation metric for this experiment.

We show the detailed statistics of best compressed model that keeps the original WER in Table 4.3. DeepIoT achieves a significantly better compression rate compared with DyNS-Ext, and the model generated by DeepIoT even has a little improvement on WER. However, compared with the previous two examples on convolutional neural network, DeepIoT fails to compress the model to less than 5% of the original parameters in the recurrent neural network case (still a 20-fold reduction though). The main reason is that compressing recurrent networks needs to prune both the output dimension and the hidden dimension. It has been shown that dropping hidden dimension can harm the network performance [34]. However DeepIoT is still successful in compressing network to less than 10% of parameters.

Fig. 4.4a shows the tradeoff between word error rate and memory consumption by compressed models. DeepIoT achieves around ×7 better tradeoff compared with magnitude-based method, DyNS-Ext. This means compressing recurrent neural networks requires more information about parameter redundancies within and among each layer. Compression using only local information, such as magnitude information, will cause degradation in the final performance.

Fig. 4.4b shows the tradeoff between word error rate and execution time. DeepIoT reduces execution time by 86.4% without degradation on WER compared with the original network. With the evaluation on Intel Edision, the original network requires 71.15 seconds in average to recognize one human speak voice example with the average length of 7.43 seconds. The compressed structure generated by DeepIoT reduces the average execution time to 9.68 seconds without performance loss, which improves responsiveness of human voice recognition.

Fig. 4.4c shows the tradeoff between word error rate and energy consumption. DeepIoT reduces energy by 87% compared with the original network. It performs better than DyNS-Ext by a large margin.

4.2.5 Supporting Human-Centric Context Sensing

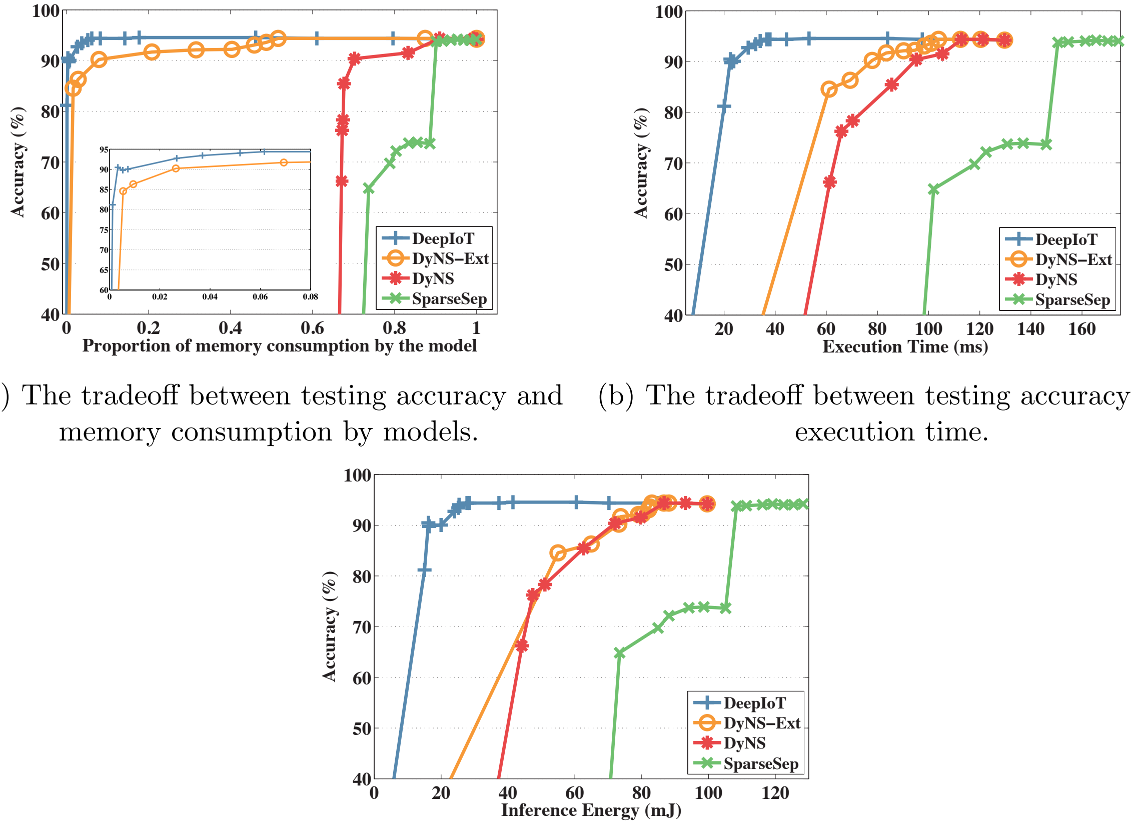
In addition to experiments about supporting basic human-centric interaction modalities, we evaluate DeepIoT on one human-centric context sensing application. We compress the state-of-the-art deep learning model, DeepSense, [2] for these problems and evaluate the accuracy and other system performance for the compressed networks. DeepSense contains all commonly used modules, including convolutional, recurrent, and fully-connected layers, which is also a good example to test the performance of compression algorithms on the combination of different types of neural network modules.

The human-centric context sensing tasks we consider is heterogeneous human activity recognition (HHAR). The HHAR task recognizes human activities with motion sensors, accelerometer and gyroscope. "Heterogeneous" means that the task is focus on the generalization ability with human who has not appeared in the training set.

In this evaluation section, we use the dataset collected by Allan et al. [11]. This dataset contains readings from two motion sensors (accelerometer and gyroscope). Readings were

Table 4.4: Heterogeneous human activity recognition

Layer		Hidden Unit		Params		DeepIoT (Hidden Units/ Params)				DyNS-Ext		DyNS		SparseSep	
conv1a	conv1b (2 × 9)	64	64	1.1K	1.1K	20	19	31.25%	29.69%	92%	95.7%	50.3%	60.0%	100%	100%
conv2a	conv2b (1 × 3)	64	64	12.3K	12.3K	20	14	9.76%	6.49%	70.1%	77.7%	25.3%	40.5%	114%	114%
conv3a	conv3b (1 × 3)	64	64	12.3K	12.3K	23	23	11.23%	7.86%	69.9%	66.2%	32.1%	35.4%	114%	114%
conv4 (2 × 8)		64		65.5K		10		5.61%		40.3%		20.4%		53.7%	
conv5 (1 × 6)		64		24.6K		12		2.93%		27.2%		18.3%		100%	
conv6 (1 × 4)		64		16.4K		17		4.98%		24.6%		12.0%		100%	
gru1		120		227.5K		27		5.8%		1.2%		100%		100%	
gru2		120		86.4K		31		6.24%		3.6%		100%		100%	
fc1		6		0.7K		6		25.83%		98.6%		99%		70%	
total				472.5K				6.16%		17.1%		74.5%		95.3%	
Test Accuracy		94.6%				94.7%				94.6%		94.6%		93.7%	



(a) The tradeoff between testing accuracy and memory consumption by models.

(b) The tradeoff between testing accuracy and execution time.

(c) The tradeoff between testing accuracy and energy consumption.

Figure 4.5: System performance tradeoff for heterogeneous human activity recognition

recorded when users execute activities scripted in no specific order, while carrying smartwatches and smartphones. The dataset contains 9 users, 6 activities (biking, sitting, standing, walking, climbStairup, and climbStair-down), and 6 types of mobile devices.

The original network structure of DeepSense is shown in the first two columns of Table 4.12. HHAR uses a unified neural network structure as introduced in Section 3. The structure contains both convolutional and recurrent layers. Since SparseSep and DyNS are not directly applicable to recurrent layers, we keep the recurrent layers unchanged while using them. In addition, we also compare DeepIoT with DyNS-Ext in this experiment.

Table 4.12 illustrates the statistics of final pruned network generated by four algorithms that have no or acceptable degradation on testing accuracy. DeepIoT is the best-performing algorithm considering the remaining number of network parameters. This is mainly due to the design of compressor network and compressor-critic framework that jointly reduce the redundancies among parameters while maintaining a global view across different layers. DyNS and SparseSep are two algorithms that can be only applied to the fully-connected and convolutional layers in the original structure. Therefore there exists a lower bound of the left proportion of parameters, *i.e.*, the number of parameters in recurrent layers. This lower bound is around 66%.

The detailed tradeoffs between testing accuracy and memory consumption by the models are illustrated in Fig. 4.5a. DeepIoT still achieves the best tradeoff for sensing applications. Other than the compressor neural network providing global parameter redundancies, directly pruning hidden elements in each layer also enables DeepIoT to obtain more concise representations in matrix form, which results in less memory consumption.

The tradeoffs between system execution time and testing accuracy are shown in Fig. 4.5b. DeepIoT uses the least execution time when achieving the same testing accuracy compared with three baselines. It takes 36.7ms for a single prediction, which reduces execution time by around 71.4% without loss of accuracy. DyNS and DyNS-Ext achieve better performance on time compared with SparseSep. As shown in Table 4.12, the original network uses 1-d filters in its structure. The matrix factorization based kernel compressing method used in SparseSep cannot help to reduce or even increase the parameter redundancies and the number of operations involved. Therefore, there are constraints on the network structure when applying matrix factorization based compression algorithm. In addition, SparseSep cannot be applied to the recurrent layers in the network, which consumes a large proportion of operations during running the neural network.

The tradeoffs between energy consumption and testing accuracy are shown in Fig. 4.5c. DeepIoT is the best-performing algorithm for energy consumption. It reduces energy by around 72.2% without loss of accuracy. Due to the aforementioned problem of SparseSep on 1-d filter, redundant factorization causes more execution time and energy consumption in the experiment.

Table 4.5: Execution time of convolutional layers with 3×3 kernel size, stride 1, same padding, and 224×224 input image size on the Nexus 5 phone.

	in_channel	out_channel	FLOPs	Time (ms)
CNN1	8	32	452.4 M	114.9
CNN2	32	8	452.4 M	300.2
CNN3	66	32	3732.3 M	908.3
CNN4	43	64	4863.3 M	751.7

4.3 THE DESIGN OF FASTDEEPIOT

In this section, we show how a better understanding of the non-linear relation between neural network structure and performance can further improve execution time and energy consumption without impacting accuracy.

4.3.1 Nonlinearities: Evidence and Exploitation

In practice, counting the number of neural network parameters and the total FLOPs does not lead to good estimates of execution time because the relation between these predictors and execution time is not proportional. On one hand, the fully-connected layer usually has more parameters but takes much less time to run compared to the convolutional layer [64]. On the other hand, one can easily find examples, where increasing the total FLOPs does not translate into added execution time. Caching effects, memory accesses, and compiler optimizations complicate the translation. Table 4.5 shows that CNN2 takes around ×2.6 the execution time of CNN1, while both have *the same* total FLOPs. Moreover, CNN3 takes *longer* to run compared to CNN4 despite having *fewer* FLOPs. These observations indicate that current rules-of-thumb for estimating neural network execution time are not the best approximations.

FastDeepIoT answers two key questions to better parameterize neural network implementations for efficient execution on mobile and embedded platforms:

1. What are the main factors that affect the execution time of neural networks on mobile and embedded devices?
2. How to guide existing structure compression algorithms to minimize the neural network execution time properly?

FastDeepIoT consists of two main modules to tackle these two challenging problems, respectively.

Profiling: Due to different code-level optimizations for different network structures within the deep learning library, the execution time of neural network layers can be extremely nonlinear

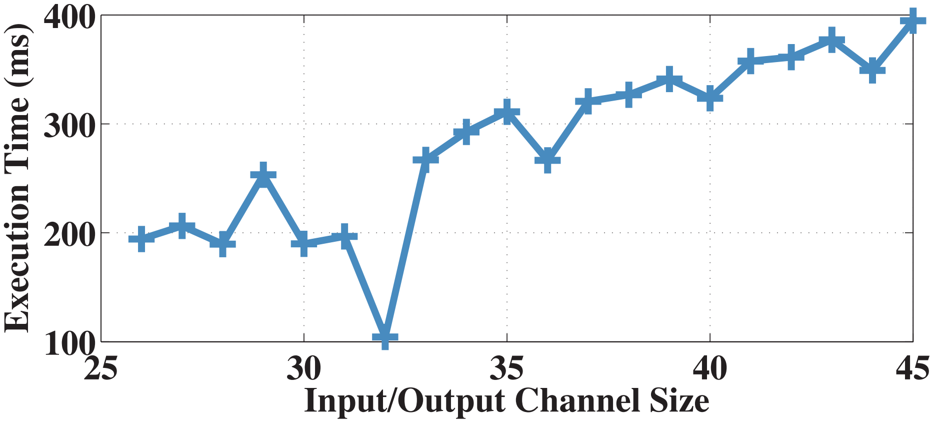


Figure 4.6: The non-linearity of neural network execution time over input/output channel.

over the structure configuration space. A simple illustration is shown in Figure 4.6, where we plot the execution time of convolutional layers when changing the size of input and output channels simultaneously. The plot reveals non-monotonic effects, featuring periodic dips in execution time as network size increases.

A simple regression model over the entire space will thus not be a good approximation. Instead, we propose a tree-structured linear regression model. Specifically, we automatically detect key conditions at which linearity is violated and arrange them into a tree structure that splits the overall modeling space into piecewise linear regions. Within each region (tree branch), we use linear regression to convert input structure information into some key explanatory variables, predictive of execution time. The splitting of the overall space and the fitting of subspaces to predictive models are done jointly, which improves both model interpretability and accuracy. The aforementioned modeling is done without specific knowledge of underlying hardware and deep learning library.

Compression: Using the results of profiling, we then propose a compression steering module that guides existing neural network structure compression methods to better minimize execution time. The execution time model leads compression algorithms to focus more on the layer that takes longer to run instead of treating all layers equally or concentrating on inaccurate total metrics. It is also better able at exploiting non-monotonicity of execution time with respect to network structure size to reduce the former without hurting application-level accuracy metrics.

4.3.2 Profiling Module

We separate this module into two parts. The first part generates diverse training structures for profiling. The second part builds an accurate and interpretable model predicting

Table 4.6: The scope of our structure configuration for fully-connected (FC), convolutional (CNN), and recurrent (RNN) layers.

Type	Structure configuration scope
FC	$in\_dim \in [1, 4096]$ $out\_dim \in [1, 4096]$
CNN	$in\_height \in [24, 225]$ $in\_width \in [24, 225]$ $kernel\_height \times kernel\_width \in \{2 \times 2, 3 \times 3, 4 \times 4, 5 \times 5, 2 \times 3\}$ $in\_channel \in [1, 256]$ $out\_channel \in [1, 256]$ $padding \in \{\text{valid}, \text{same}\}$ $stride \in \{1, 2\}$
RNN	$in\_dim \in [1, 512]$ $out\_dim \in [1, 512]$ $step \in \{8, 10, 15, 20\}$

the execution time of deep learning components for the corresponding structure information.

Neural Network Profiling

We introduce the basic system settings and the procedure of generating training structures for profiling here.

FastDeepIoT utilizes TensorFlow benchmark tool [65] to profile the execution time of all deep learning components on the target device. In order to make the profiling results fully reflect the changes on the neural network structures, we fix the frequencies of phone CPUs (processors) to be constants and stop all the power management services that can affect the processor frequency on target devices, such as fixing *mpdecision* on Qualcomm chips.

The next step is to generate diverse neural network structures for time profiling. As a deep learning component, such as a convolutional layer and recurrent layer, the combinations of its structure design choices can form an extremely huge structure configuration space. Therefore, we can only select a small proportion of structure configurations during our time profiling. The scope of our structure configuration is shown in Table 4.6, from which the network generation code chooses a random combination. Notice that we do not contain the activation function as the profiling choice, because it only occupies around 1% $\sim$ 2% execution time of a deep learning component through empirical observations. By eliminating this insignificant configuration, *i.e.*, activation_function $\in$ {ReLU, Tanh, sigmoid}, we can save the number of profiling components by the factor of 3. Except for some pre-defined cases, such as sigmoid activation function for gate outputs in recurrent layers, we set all activation functions to be ReLU, which is one of the most widely used activation functions. In addition, the order of deep learning components in the network has little impact on their execution time empirically.

In our profiling module, for each target device, we profile around 120 neural networks with about 1300 deep learning components in total. These time profiling results form a time

Table 4.7: The definition of parameter and memory information for Fully-Connected layer (FC), Convolutional layer (CNN), Gated Recurrent Unit (GRU), and Long Short Term Memory (LSTM).

Type	mem_in	mem_out	mem_inter
FC	in_dim	out_dim	0
CNN	$in\_height \times in\_width \times in\_channel$	$out\_height \times out\_width \times out\_channel$	$out\_height \times out\_width \times kernel\_height \times kernel\_width \times in\_channel$
GRU	$step \times in\_dim$	$step \times out\_dim$	$3 \times step \times out\_dim$
LSTM	$2 \times step \times in\_dim$	$2 \times step \times out\_dim$	$4 \times step \times out\_dim$

Type	$param\_size$
FC	$in\_dim \times out\_dim + out\_dim$
CNN	$kernel\_height \times kernel\_width \times in\_channel \times out\_channel + 1$
GRU	$3 \times out\_dim \times (in\_dim + out\_dim + 1)$
LSTM	$4 \times out\_dim \times (in\_dim + out\_dim + 1)$

profiling dataset, $\mathcal{D} = \{\mathcal{S}_i, y_i\}$, where $\mathcal{S}_i$ is the structure configuration and y_i the execution time.

Execution Time Model Building

Due to the code-level optimization for different component configuration choices in the deep learning library, execution-time non-linearity appears over the structure configuration space as shown in Figure 4.6. The main challenge here is to build a model that can automatically figure out the conditions that cause the execution-time non-linearity without specific knowledge of underlying library and hardware.

In order to maintain both the accuracy and interpretability, we propose a tree-structure linear regression model. The model can recursively partition the structure configuration space such that the time profiling samples fitting the same linear relationship are grouped together. The intuition behind this model is that the execution time of deep learning component under each particular code-level optimization can be formulated with a linear relationship given a set of well-designed explanatory variables. In addition, different deep learning components, *i.e.*, fully connected, convolutional, and recurrent layer, learn their own execution time models.

Each time profiling data is composed of three elements. The feature vector $\mathbf{f}$, used for identifying the condition that causes the execution-time non-linearity; the execution time y; and the explanatory variable vector $\mathbf{x}$, used for fitting the execution time y.

The basic idea of tree-structure linear regression is to find out the most significant condition causing the execution-time non-linearity within the current dataset recursively. These conditions will form a binary tree structure. In order to figure out key conditions causing the execution-time non-linearity, we take two conditioning functions into account.

1. *Range condition* $C_1(\mathbf{f}[j], \tau) := f[j] \leq \tau$: identifies execution-time non-linearity caused by cache and memory hit as well as specific implementation for a certain feature range.
2. *Integer multiple condition* $C_2(\mathbf{f}[j], \tau) := f[j] \equiv 0 \pmod{\tau}$: identifies execution-time non-linearity caused by loop unrolling, data alignment, and parallelized operations.

Assume that we are generating node m in the binary tree with dataset $\mathcal{D}_m$. The model creates a set of conditions $\{\phi\}$. Each of them can partition the dataset into two subsets $\mathcal{D}_m^{(l)}(\phi)$ and $\mathcal{D}_m^{(r)}(\phi)$. Each condition ϕ consists of three elements, $\phi = \{\mathbf{f}[j], \tau_m, k\}$, where $k \in \{1, 2\}$ is the conditioning function type.

$$\begin{aligned} \mathcal{D}_m^{(l)}(\phi) &= \mathcal{D}_m | C_k(x_j, \tau_m), \\ \mathcal{D}_m^{(r)}(\phi) &= \mathcal{D}_m n \mathcal{D}_m^{(l)}(\phi). \end{aligned} \tag{4.12}$$

Node m selects the most significant condition ϕ^* by minimizing the impurity function $G(\mathcal{D}_m, \phi)$,

$$\phi^* = \arg\min_{\phi} G(\mathcal{D}_m, \phi), \tag{4.13}$$

$$G(\mathcal{D}_m, \phi) = \frac{\left|\mathcal{D}_m^{(l)}(\phi)\right|}{|\mathcal{D}_m|} H\big(\mathcal{D}_m^{(l)}(\phi)\big) + \frac{\left|\mathcal{D}_m^{(r)}(\phi)\right|}{|\mathcal{D}_m|} H\big(\mathcal{D}_m^{(r)}(\phi)\big), \tag{4.14}$$

$$H(\mathcal{D}) = \min_{\mathbf{w}, b} \frac{1}{|\mathcal{D}|} \sum_{(\mathbf{x}, y) \in \mathcal{D}} (\mathbf{w}^\intercal \mathbf{x} + b - y)^2 \quad \text{s.t. } \mathbf{w}, b \geq 0. \tag{4.15}$$

The impurity function is designed as the weighted mean square errors of linear regressions over two sub-datasets partitioned by the condition ϕ.

Next, we describe the feature vector $\mathbf{f}$. Our choice of feature vector $\mathbf{f}$ contains three parts: the structure features, the memory features, and the parameter feature. The structure features refer to *in_dim* and *out_dim* for fully-connected and recurrent layers as well as *in_channel* and *out_channel* for convolutional layers. The memory features include the memory size of input, *mem_in*, the memory size of output, *mem_out*, and the memory size

Algorithm 4.2. execution time model building

```
**Input:** time profiling dataset $\mathcal{D}_z$, feature vector $\mathbf{f}$, two conditioning functions $f[j] \leq \tau$ and $f[j] \equiv 0 \pmod{\tau}$, and explanatory variable $\mathbf{x}_z$.
Fit $\mathcal{D}_z$ with $\mathbf{w}_r$ and $b_r$ according to (4.15).
Save root node $r = [\emptyset, \mathbf{w}_r, b_r]$
**Initialize:** que $= \big[[r, \mathcal{D}_z]\big]$.
**while** len(que) $> 0$ **do**
    $q, \mathcal{D}_q =$ que.deque()
    **if** $\mathcal{D}_q$ meets stoping condition **then**
        Continue
    **end if**
    Search for the optimal partition $\phi^* = \{f[j^*], \tau^*, k^*\}$ according to (4.13) (4.14) (4.15).
    Generate partitioned dataset $\mathcal{D}_q^{(l)}(\phi^*)$ and $\mathcal{D}_m^{(r)}(\phi^*)$ according to (4.12).
    Fit $\mathcal{D}_q^{(l)}(\phi^*)$ with $\mathbf{w}_q^{(l)}$ and $b_q^{(l)}$ according to (4.15).
    Fit $\mathcal{D}_q^{(r)}(\phi^*)$ with $\mathbf{w}_q^{(r)}$ and $b_q^{(r)}$ according to (4.15).
    Save $q$'s left child node $q^{(l)} = \big[[\text{True}, \phi^*], \mathbf{w}_q^{(l)}, b_q^{(l)}\big]$
    Save $q$'s right child node $q^{(r)} = \big[[\text{False}, \phi^*], \mathbf{w}_q^{(r)}, b_q^{(r)}\big]$
    que.enque$\big([q^{(l)}, \mathcal{D}_q^{(l)}(\phi^*)]\big)$
    que.enque$\big([q^{(r)}, \mathcal{D}_q^{(r)}(\phi^*)]\big)$
**end while**
```

of internal representations, *mem_inter*. The parameter feature refers to the size of parameters, *param_size*. The detailed definitions of memory and parameter features are shown in Table 4.7. All notations in Table 4.7 are consistent with the notations of structure configurations in Table 4.6, except for the height and width of output image, *out_height* and *out_width*, in the convolutional layer. However, we can easily calculate these two values based on other structure information, *i.e.*, *in_height*, *in_width*, *kernel_height*, *kernel_width*, *stride*, and *padding* [3].

Last, we discuss about our explanatory variable vector $\mathbf{x}$ for linear regression. In this dissertation, we build an intuitive performance model that the execution time of a program is contributed by three parts, CPU operations, memory operations, and disk I/O operations. For a deep learning component, these parts refer to FLOPs, memory size, and parameter size,

$$\mathbf{x} = [\text{FLOPs}, mem, param\_size]. \tag{4.16}$$

where $mem = mem\_in + mem\_out + mem\_inter$.

With the weight vector $\mathbf{w}$ and the bias term b, the overall execution time of a deep learning component, y, can be modelled as $y = \mathbf{w}^\intercal \mathbf{x} + b$. Since every term should have a positive contribution to the execution time, we add an additional constraint, $\mathbf{w}, b \geq 0$, as shown in (4.15).

The tree-structure linear regression model builds a binary tree that gradually picks out conditions that cause execution-time non-linearity and breaks the dataset into subsets that contain more "linearity". Our designed explanatory variable vector $\mathbf{x}$ is able to fit the dataset with linear relationships better level by level, especially for fully-connected and convolutional

[3] https://www.tensorflow.org/api_guides/python/nn\#Convolution

layer. The recurrent layers, however, still have flaws. We analyze the error and find out that recurrent layers have a constant initialization overhead or set-up time for each step. Therefore, we update explanatory variable vector $\mathbf{x}$,

$$\begin{aligned} \mathbf{x}_{fc} = \mathbf{x}_{cnn} &= [\text{FLOPs}, mem, param\_size], \\ \mathbf{x}_{rnn} &= [\text{FLOPs}, mem, param\_size, step]. \end{aligned} \tag{4.17}$$

Table 4.8: The p-values of explanatory variables.

Type	FLOPs	*mem*	*param_size*	*step*
FC	0.000	0.000	1.000	/
CNN	0.037	0.009	1.000	/
GRU	0.000	0.000	1.000	0.000
LSTM	0.000	0.000	1.000	0.000

We summarize our execution time model building process in Algorithm 4.2. There is a stopping condition in Line 7 that keeps tree-structure linear regression from growing infinitely. In our case, the stopping condition occurs when a linear regression can fit the current dataset $\mathcal{D}_q$ with a mean absolute percentage error less than 5% or when the size of current dataset is smaller than 15, $|\mathcal{D}_q| < 15$.

Execution Time Model with Statistical Analysis

In this part, we provide an illustration of the FastDeepIoT profiling module on Nexus 5 phone with statistical analysis. The module first profiles and generates the execution time profiling dataset. Then, the module builds an execution time model for each deep learning component based on the tree-structure linear regression in Algorithm 4.2. Additional evaluations on the execution time model will be shown in Section 4.4.2.

For fully-connected layers and recurrent layers, including GRU and LSTM, their execution time has a perfect linear relationship with our explanatory variable vector $\mathbf{x}_{fc}$ and $\mathbf{x}_{rnn}$. However, the execution time model of convolutional layers reflects a strong non-linearity over the structure configuration space. As shown in Figure 4.7, the execution time of convolutional layer has local minima when *in_channel* or *out_channel* is a multiple of 4.

Then we calculate the p-values to evaluate the mathematical relationship between each explanatory variable and the execution time. The p-value for each explanatory variable tests the null hypothesis that the variable has no correlation with the execution time. Results are shown in Table 4.8. The p-values of explanatory variables, FLOPs, *mem*, and *step*, are less than the significance level (0.05) for all deep learning components. So our empirical

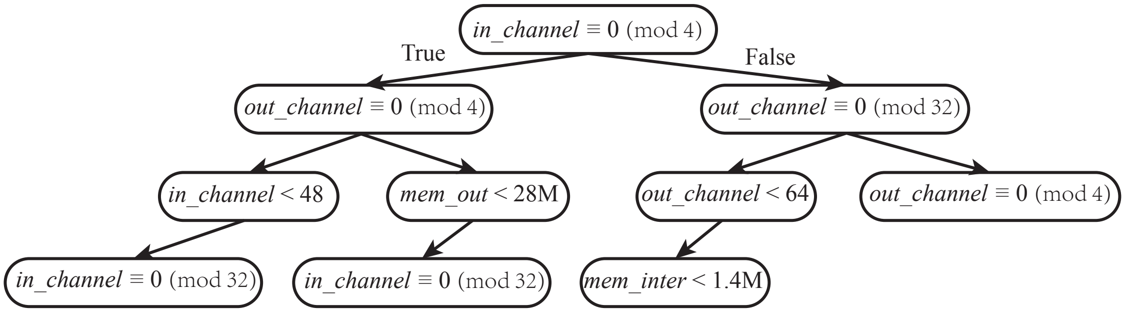


Figure 4.7: The execution time model of convolutional layers on Nexus 5.

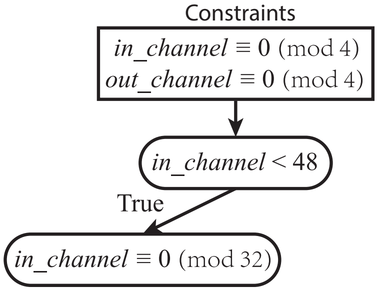


Figure 4.8: Simplified execution time model of convolutional layers on Nexus 5.

time profiling data provides enough evidence that the correlation between these explanatory variables and the execution time are statistically significant. However, the p-values for *param_size* is high for all cases, which shows that the number of parameters has limited correlation with the execution time. This experiment, again, highlights the importance of proposing a compression algorithm targeting on minimizing the execution time instead the number of parameters.

4.3.3 Compression Steering Module

Profiling and modelling deep learning execution time is not enough for speeding up the model execution. In this section, we introduce the compression steering module that is designed to empower existing deep learning structure compression algorithms to minimize model execution time properly.

We assume that $\mathcal{S} = \{\mathbf{s}_l\}$ and $\mathcal{W} = \{\mathbf{W}_l\}$ for $l = 1, \cdots, L$ is structure information and weight matrix of a neural network from layer 1 to layer L respectively. We denote our execution time model as $t_l = \mathcal{T}(\mathbf{s}_l)$, which takes the structure information $\mathbf{s}_l$ as input and predicts

Algorithm 4.3. Layer structure expansion and local minima searching

```
Input: the execution time model $\mathcal{T}()$ with root node $r$ and the layer structure $\{\mathbf{f}, \mathbf{x}\}$.
Set node = $r$, condL = [].
while ¬(node.left == None & node.right == None) do
    if node.cond is a range condition then
        condL.append(node.cond)
        if $\mathbf{f}$ obeys node.cond then
            node = node.left
        else
            node = node.right
        end if
    else
        $\hat{\mathbf{f}} = \mathbf{f}$ and $\hat{\mathbf{x}} = \mathbf{x}$.
        $\hat{\mathbf{f}}$[node.$j$] = node.$\tau \times \lceil \mathbf{f}$[node.$j$]/node.$\tau \rceil$
        Update $\hat{\mathbf{x}}$ according to $\hat{\mathbf{f}}$.
        if node.$\mathbf{w}_T^{\intercal}\hat{\mathbf{x}}$ + node.$b_T$ > node.$\mathbf{w}_F^{\intercal}\mathbf{x}$ + node.$b_F$ & $\hat{\mathbf{f}}$ obeys condL then
            $\mathbf{f} = \hat{\mathbf{f}}$ and $\mathbf{x} = \hat{\mathbf{x}}$.
            node = node.left
        else
            node = node.right
        end if
    end if
end while
Return: $\mathbf{f}$
```

the component execution time t_l. For a general neural network structure compression algorithm, we denote the original compression process as,

$$\min_{\mathcal{S},\mathcal{W}} \mathcal{L}_\theta(\mathcal{S}, \mathcal{W}), \tag{4.18}$$

where the compression algorithm minimizes a loss function, concerning prediction error or parameter size, with either the gradient descend or searching based optimization method.

In order to enable the compression algorithm to minimize the execution time, our first step is to incorporate the execution time model into the original objective function (4.18),

$$\min_{\mathcal{S},\mathcal{W}} \mathcal{L}_\theta(\mathcal{S}, \mathcal{W}) + \lambda \sum_{l=1}^{L} \mathcal{T}(\mathbf{s}_l), \tag{4.19}$$

where λ is a hyper-parameter that make the tradeoff between minimizing training loss and minimizing execution time.

Adding execution time to the compression objective function can encourage the compression algorithm to concentrate more on the layers with higher execution time, which helps to speed up the whole neural network.

However, due to the existence of execution-time local minima, compressing neural network structure is not always the optimal choice for minimizing the execution time. As shown in Figure 4.6, enlarging neural network structure can find a nearby execution-time local minimum that reduces the execution time. Notice that enlarging structure is a lossless operation. We can at least enlarge weight matrices with zeros that keeps performance the same.

In general, utilizing execution-time local minima for speeding up involves two steps:

1. Identifying an expanded structure configuration that can trigger a nearby execution-time local minimum.

2. Deciding whether the expanded structure can speed up the execution time.

For an execution time model trained with a complex method, such as neural networks, identifying a nearby execution-time local minimum can be almost impossible by blindly searching a large configuration space. However, our tree-structure linear regression can easily identify a nearby local minimum speeding up the neural network execution.

Local extrema, *i.e.*, maxima and minima, are identified by the integer multiple condition, $f[j] \equiv 0 \pmod{\tau}$, in our tree-structure linear regression model. Our compression steering module searches for the nearby local maxima by gradually expanding the structure that fits the integer multiple conditions from root node to leaf node in the execution time model.

Assume that node m is under the condition $\mathbf{f}[j_m] \equiv 0 \pmod{\tau_m}$ with two sets of linear regression parameters $\{\mathbf{w}_T, b_T\}$ and $\{\mathbf{w}_F, b_F\}$ used for fitting the dataset that obeying and against the condition respectively. A deep learning layer is denoted with the feature vector $\mathbf{f}_l$ and the explanatory variable vector $\mathbf{x}_l$. The compression steering module generates an expanded layer with feature vector $\hat{\mathbf{f}}_l$ and explanatory variable vector $\hat{\mathbf{x}}_l$ by updating the conditioning feature $\hat{\mathbf{f}}[j_m] = \tau_m \lceil \mathbf{f}[j_m]/\tau_m \rceil$. Then the module compares the values between $\mathbf{w}_T^\intercal \hat{\mathbf{x}} + b_T$ and $\mathbf{w}_F^\intercal \mathbf{x} + b_F$ to decide whether it should accept the expansion for speeding-up and go through the corresponding branch.

The layer structure expansion and local minima searching process is summarized in Algorithm 4.3. The algorithm goes through whole tree structure to find out a nearby local minimum that reduces the execution time.

For a whole neural network, each layer goes through the structure expansion and local minima searching process one by one. It is possible that conflicts exist between expanded structures of two neighbouring layers. The module solves these conflicts sequentially by choosing the one having shorter overall execution time.

In addition, we can further analyze the structure expansion process for a particular component on a particular device for a particular application settings. For example, assume that we are compressing the *in_channel* and *out_channel* of a convolutional layer on Nexus 5 with kernel size 3 × 3, input image size 24 × 24, and the same padding. We are considering the root condition *in_channel* $\equiv 0 \pmod{4}$ as shown in Figure 4.7. According to our execution time model, two linear regression models that fit the two datasets in the left and right child of the root node are:

$$\begin{aligned} \mathbf{w}_T &= [3.41 \times 10^{-8}, 4.03 \times 10^{-6}, 7.11 \times 10^{-25}] \;\; b_T = 8.11, \\ \mathbf{w}_F &= [3.11 \times 10^{-8}, 8.03 \times 10^{-6}, 1.52 \times 10^{-34}] \;\; b_F = 12.82. \end{aligned} \tag{4.20}$$

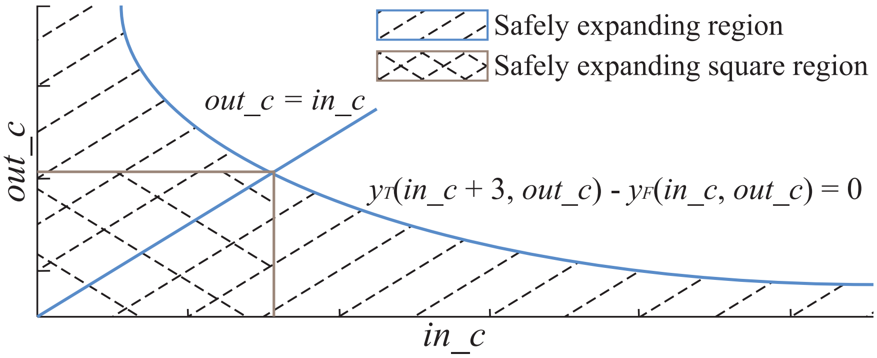


Figure 4.9: The square region of safely expanding *in_channel* for speed up.

Then we can obtain the execution time as a function of *in_channel* and *out_channel* by substituting the explanatory variable vector $\mathbf{x}$ with definitions illustrated in Table 4.7 as well as the application settings about kernel size, input image size, and padding option.

$$\begin{aligned} y_T(in\_c, out\_c) =& 3.53 \times 10^{-4} \cdot in\_c \cdot out\_c + 8.11 + \\ & 2.32 \times 10^{-2} \cdot in\_c + 2.32 \times 10^{-3} \cdot out\_c, \\ y_F(in\_c, out\_c) =& 3.23 \times 10^{-4} \cdot in\_c \cdot out\_c + 12.82 + \\ & 4.63 \times 10^{-2} \cdot in\_c + 4.63 \times 10^{-3} \cdot out\_c, \end{aligned} \tag{4.21}$$

where we denote *in_channel* and *out_channel* as *in_c* and *out_c* for simplicity.

We are interested in the region where expanding the *in_channel* to a nearby multiple of 4 can speed up the execution. This is equivalent to solving

$$y_T(in\_c + 3, out\_c) - y_F(in\_c, out\_c) < 0, \tag{4.22}$$

where its zero contour line is a hyperbola. Therefore, within the region bounded by *in_channel* axis, *out_channel* axis, and zero contour line, we can safely expand *in_channel* to a multiple of 4 to speed up the convolutional layer execution time.

In order to have a more interpretable result, as shown in Figure 4.9, we can obtain a square region by finding the intersections between the zero contour line and the function *out_channel* = *in_channel*. In this case, within the region *in_channel* × *out_channel* $\in [1, 1288] \times [1, 1288]$, we can blindly expand *in_channel* to a multiple of 4 to speed up. This region is much larger than the region we are interested in. We can keep analyzing the next condition *out_channel* $\equiv 0 \pmod 4$ and achieve similar result. Within the region *in_channel* × *out_channel* $\in [1, 808] \times [1, 808]$, we can safely expand *in_channel* and *out_channel* to a nearby multiple of 4 to speed up. In the end, we can obtain a simplified execution time model $\hat{\mathcal{T}}$ as shown in Figure 4.8.

In summary, the compression steering module compresses the neural network structure for reducing overall execution time with three steps.

1. Compressing neural network with a time-aware objective function (4.19) with execution time model $\mathcal{T}$.

2. Expanding layer structure and searching local minima for further speed up according to Algorithm 4.3 with execution time model $\mathcal{T}$ or $\hat{\mathcal{T}}$ (if available).

3. Depending on the original compression algorithm, freeze the structure and fine-tune the neural network.

4.4 THE EVALUATION OF FASTDEEPIOT

In this section, we evaluate FastDeepIoT through two sets of experiments. The first set evaluates the accuracy of the execution time model generated by our profiling module, while the second set evaluates the performance of our compression steering module. In order to evaluate execution time modeling accuracy, we compare our tree-structured linear regression model to other state-of-the-art regression models on two mobile devices. To evaluate the quality of compression, we present a set of experiments that demonstrate the speed-up of the compressed neural network obtained by the compression steering module with three human-centric interaction and sensing applications.

4.4.1 Implementation

In this section, we briefly describe the hardware, software, and architecture of FastDeepIoT.

Hardware: We test FastDeepIoT on two types of hardware, Nexus 5 phone and Galaxy Nexus phone. Two devices are profiled for each type of hardware. The Nexus 5 phone is equipped with quad-core 2.3 GHz CPU and 2 GB memory. The Galaxy Nexus phone is equipped with dual-core 1.2 GHz CPU and 1GB memory. We stop the *mpdecision* service and use *userspace* CPU governor for two hardware. We manually set 1.1GHz for the quad-core CPU on Nexus 5, and 700MHz for the dual-core CPU on Galaxy Nexus to prevent overheating caused by the constant time profiling. In addition, all profiling and testing neural network models are run solely on CPU. The execution time model building and the compression steering module are implemented on a workstation connected to two phones. All compressing steps are implemented on the workstation.

Software: FastDeepIoT utilizes TensorFlow benchmark tool [65], a C++ binary, to profile the execution time of deep learning components. For each neural network, the benchmark tool have one warm up run to initialize the model and then profile all components execution time with 20 runs without internal delay. Mean values are taken as the profiled execution time.

We install Android 5.0.1 on Nexus 5 phone and Android 4.3 on Galaxy Nexus phone. All additional background services are closed during the profiling and testing. All energy consumptions on two devices are measured by an external power meter.

Architecture: Given a target device, FastDeepIoT first queries the device and its own database for a pre-generated execution time model with device type and OS version as the key. If the query fails, the profiling module starts its function. FastDeepIoT generates random neural network structures based on the configuration scope in Table 4.6, pushes the Protocol Buffers (.pb file) to the target device, profiles the execution time of components, fetches back and processes the profiling result. Once the profiling process has finished, FastDeepIoT learns tree-structure linear regression execution time models according to Algorithm 4.2 based on the time profiling dataset. FastDeepIoT pushes the generated execution time models to the target device and its own database for storage.

Then given an original neural network structure and parameters, the compression steering module can automatically generate a compressed structure to speed up inference time for a target device. FastDeepIoT queries the target device and own database for a pre-generated execution time model, and choose a structure compression algorithm, DeepIoT as a default, to reduce the deep learning execution time according to (4.19) and Algorithm 4.3. The resulting compressed neural network is transferred to the target device used locally.

4.4.2 Execution time Model

We implement the following execution time estimation alternatives:

1. **SVR:** support vector regression with radial basis function kernel [66]. This algorithm tries to perform linear separation over a higher dimensional kernel feature space by characterizing the maximal margin.
2. **DT:** classification and regression trees [67]. This is an interpretable model. It groups and predicts execution time by the execution time itself.
3. **RF:** random forest regression [68]. This algorithm trades the interpretability of regression tree for the predictive performance by ensembling multiple trees with random feature selections.
4. **GBRT:** gradient boosted regression trees [69]. This algorithm builds an additive model in a forward stage-wise fashion, which is hard to interpret.

Table 4.9: The Mean Absolute Percentage Error (MAPE), Mean Absolute Error (MAE) in millisecond, and Coefficient of determination (R^2) of execution time models.

(a) Nexus 5-Convolutional layer

	FastDeepIoT	SVR	DT	RF	GBRT	DNN
MAPE	**7.6%**	233.8%	23.8%	19.7%	10.9%	**6.4%**
MAE	**15.2**	227.1	39.2	27.3	20.5	16.4
R^2	**0.991**	−0.229	0.969	0.985	0.988	**0.994**

(b) Nexus 5-Gated recurrent unit

	FastDeepIoT	SVR	DT	RF	GBRT	DNN
MAPE	**1.8%**	78.7%	9.4%	6.7%	4.8%	2.0%
MAE	**0.6**	23.6	2.9	1.8	1.5	0.7
R^2	**0.999**	−0.078	0.986	0.995	0.995	**0.999**

(c) Nexus 5-Long short term memory

	FastDeepIoT	SVR	DT	RF	GBRT	DNN
MAPE	**2.3%**	73.7%	9.0%	4.7%	4.1%	2.8%
MAE	**0.6**	23.7	3.0	1.4	1.6	0.9
R^2	**0.999**	−0.223	0.977	0.995	0.993	0.998

(d) Nexus 5-Fully-connected layer

	FastDeepIoT	SVR	DT	RF	GBRT	DNN
MAPE	**1.9%**	133.5%	22.5%	12.0%	**0.2%**	1.9%
MAE	**0.19**	5.98	1.18	0.38	**0.01**	0.19
R^2	**0.999**	0.065	0.977	0.996	**0.999**	**0.999**

(e) Galaxy Nexus-Convolutional layer

	FastDeepIoT	SVR	DT	RF	GBRT	DNN
MAPE	**4.1%**	164.3%	33.1%	23.0%	15.2%	14.5%
MAE	**26.8**	878.7	162.9	123.7	114.6	110.1
R^2	**0.999**	−0.246	0.969	0.980	0.982	0.983

(f) Galaxy Nexus-Gated recurrent unit

	FastDeepIoT	SVR	DT	RF	GBRT	DNN
MAPE	**2.9%**	71.5%	10.5%	8.8%	6.0%	4.1%
MAE	**1.1**	27.8	4.8	4.1	3.2	2.2
R^2	**0.997**	−0.065	0.968	0.977	0.984	0.989

(g) Galaxy Nexus-Long short term memory

	FastDeepIoT	SVR	DT	RF	GBRT	DNN
MAPE	**2.9%**	66.8%	8.4%	7.8%	6.0%	**2.9%**
MAE	**1.4**	26.2	3.0	3.3	2.7	**1.3**
R^2	**0.997**	−0.196	0.983	0.985	0.987	**0.997**

(h) Galaxy Nexus-Fully-connected layer

	FastDeepIoT	SVR	DT	RF	GBRT	DNN
MAPE	**4.0%**	55.0%	12.3%	11.3%	9.5%	4.1%
MAE	**0.3**	6.7	1.2	0.9	1.0	**0.3**
R^2	**0.996**	−0.629	0.944	0.972	0.949	**0.996**

5. **DNN:** multilayer perceptron [70]. Deep neural network is a learning model with high capacity. We build a four-layer fully connected neural network with LeRU as the activation function, except for the output layer. We fine-tune the structure and apply dropout as well as L2 regularization to prevent overfitting. DNN is a black-box model.

We train all the baseline models with the dataset generated by the profiling module in FastDeepIoT (75% for training and 25% for testing). For each deep learning component, such as CNN and LSTM, an individual model is trained. We have trained these models with feature vector $\mathbf{f}$, explanatory variable vector $\mathbf{x}$, and the concatenate of feature and explanatory variable vectors as inputs, where $\mathbf{f}$ and $\mathbf{x}$ are the same as the definitions in Section 4.3.2. We find that the model trained with explanatory variable vector $\mathbf{x}$ outperforms other choices consistently in all cases, so we only report the results of models trained with $\mathbf{x}$ for simplicity.

We evaluate these models on convolutional layer, gated recurrent unit, long short term memory, and fully-connected layer with mean absolute percentage error, mean absolute rrror, and coefficient of determination on two hardware. As shown in Table 4.9, FastDeepIoT is consistently among top 2 predictors for all experiments with all three metrics. FastDeepIoT also outperforms the highly capable deep learning model for more than half of the cases, while FastDeepIoT is much more interpretable. There are two reasons for the remarkable performance of FastDeepIoT. On one hand, FastDeepIoT captures the primary characters of deep learning execution time behaviours, which makes an interpretable and accurate model possible. On the other hand, since the profiled dataset is limited (around one thousand samples for training), complex models such as deep neural networks that require large training dataset may not be the best choice here.

4.4.3 Compression Steering Module

In this section, we evaluate the performance of our compression steering module with three sensing applications. We train the neural networks on traditional benchmark datasets as original models. Then, we compress the original models using FastDeepIoT and the three state-of-the-art baseline algorithms. Finally, we test the accuracy, execution time, and energy consumption of compressed models on mobile devices.

We compare FastDeepIoT with three baseline algorithms:

1. **DeepIoT:** This is a state-of-the-art neural structure compression algorithm [3]. The algorithm designs a compressor neural network with adaptive dropout to explore a succinct structure for the original model.

2. **DeepIoT+localMin:** We enhance DeepIoT with the ability of expanding layer for finding execution-time local minima. This method takes the compressed model of DeepIoT and expands its layers with zero-value elements that can trigger local minima according to Algorithm 4.3. We use this almost zero-effort method to show the improvement made on existing compressed models by interpreting deep learning execution time with FastDeepIoT.
3. **DeepIoT+FLOPs:** This method enhances DeepIoT by adding a term that minimizes FLOPs to the original objective function (4.18). Since a large proportion of works use FLOPs as the execution time estimation [21–23], this method shows to what extend FLOPs can be used to compress neural network for reducing execution time.

Image recognition on CIFAR-10:

This is a vision based task, image recognition based on a low-resolution camera. During this experiment, we use CIFAR-10 as our training and testing dataset. The CIFAR-10 dataset consists of 60000 32×32 colour images in 10 classes, with 6000 images per class. There are 50000 training images and 10000 test images.

During the evaluation, we use VGGNet structure as the original network structure [71]. The detailed structure is shown in Table 4.10, where we also illustrate the best compressed models that keeps the original test accuracy for all algorithms. The compressed model can be even deployed on tiny IoT devices such as Intel Edison.

As shown in Table 4.10, FastDeepIoT achieves the best performance on two hardware with their corresponding execution time models. Compared with the state-of-the-art DeepIoT algorithm, FastDeepIoT can further reduce the model execution time by 48% to 53%. DeepIoT+localMin outperforms DeepIoT on two hardware, reducing the execution time by 12% to 32%. This shows that we can decently reduce the neural network execution time by simply expanding the neural network structure to local execution-time minima. In additional, DeepIoT+FLOPs can speed up the model execution time compared with DeepIoT. However, FastDeepIoT still outperforms DeepIoT+FLOPs by a significant margin. This result highlights that FLOPs is not a proper estimation of time.

Figure 4.10a and 4.10b shows the tradeoff between testing accuracy and execution time for different algorithms. FastDeepIoT consistently outperforms other algorithms by a significant margin. Furthermore, the execution time characters on different hardware can affect the final performance. FastDeepIoT (Nexus 5/Galaxy Nexus) performs better on its corresponding hardware. DeepIoT+localMin achieves a better tradeoff compared with DeepIoT. Therefore,

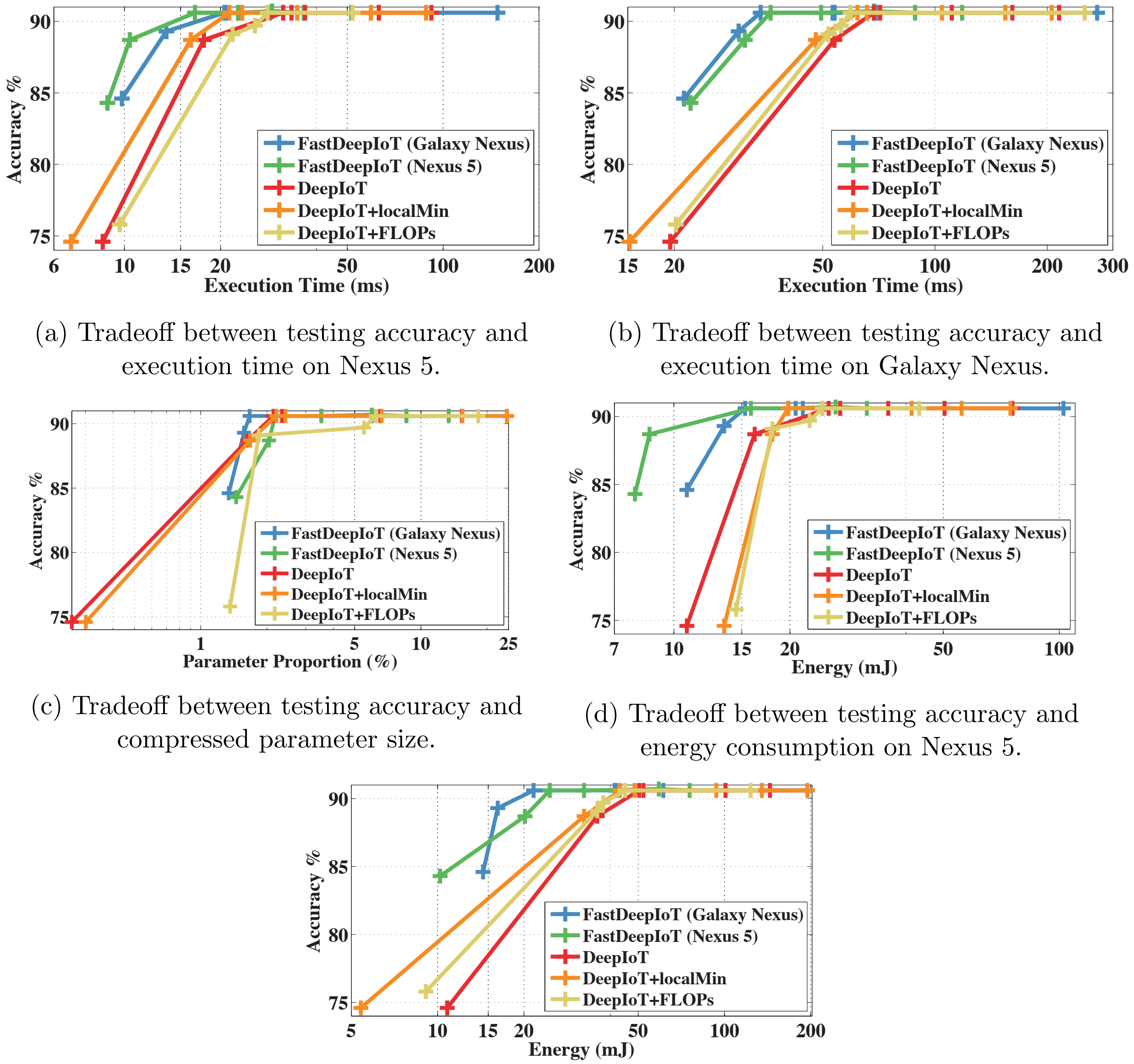
Table 4.10: VGGNet (hidden units) on CIFAR-10 dataset.

	No Execution Time Model				Nexus 5	Galaxy Nexus
Layer	Original	DeepIoT	localMin	FLOPs	FastDeepIoT	FastDeepIoT
conv1-1 (3×3)	64	27	28	19	12	16
conv1-2 (3×3)	64	47	48	17	16	24
conv2-1 (3×3)	128	53	56	33	28	36
conv2-2 (3×3)	128	68	68	50	32	44
conv3-1 (3×3)	256	104	104	89	64	72
conv3-2 (3×3)	256	97	100	79	64	56
conv3-3 (3×3)	256	89	92	77	68	72
conv4-1 (3×3)	512	122	124	115	132	96
conv4-2 (3×3)	512	95	96	112	136	80
conv4-3 (3×3)	512	64	64	112	104	120
conv5-1 (2×2)	512	128	128	143	148	116
conv5-2 (2×2)	512	112	112	132	144	108
conv5-3 (2×2)	512	146	148	182	104	92
fc1	4096	27	27	1097	132	132
fc2	4096	161	161	935	152	123
fc3	1000	10	96	72	157	167
Test accuracy	90.6%	90.6%	90.6%	90.6%	90.6%	90.6%
Execution time t (Nexus 5)	328 ms	31 ms	21 ms	28 ms	**16 ms**	23 ms
Execution time t (Galaxy)	610 ms	72 ms	63 ms	52 ms	36 ms	**34 ms**

utilizing execution-time local minima is a low-cost strategy to speed up neural network execution. In addition, since FLOPs has different degrees of execution time contribution on different hardware, DeepIoT+FLOPs are not able to achieve a better tradeoff than DeepIoT on all devices.

Figure 4.10d and 4.10e shows the tradeoff between testing accuracy and energy consumption for different algorithms. Although FastDeepIoT is not designed to minimize the energy consumption, FastDeepIoT still achieves the best tradeoff. However, we can see that the characters of energy consumption of deep neural network are different from the execution time. FastDeepIoT with the hardware-specific time models are not always the most energy-saving method on the corresponding hardware. Execution-time local minima cannot consistently help DeepIoT+localMin to outperform DeepIoT. Therefore, further studies on understanding and minimizing deep learning energy consumption are needed.

Figure 4.10c shows the tradeoff between testing accuracy and left proportion of model parameters. Since there is no algorithm targeting at minimizing model parameters, all methods show comparable performances. However, from another perspective, the execution time model learnt by FastDeepIoT empowers existing compression algorithms to reduce more execution time with almost the same amount of parameters.



(a) Tradeoff between testing accuracy and execution time on Nexus 5.

(b) Tradeoff between testing accuracy and execution time on Galaxy Nexus.

(c) Tradeoff between testing accuracy and compressed parameter size.

(d) Tradeoff between testing accuracy and energy consumption on Nexus 5.

(e) Tradeoff between testing accuracy and energy consumption on Galaxy Nexus.

Figure 4.10: System performance tradeoff for VGGNet on CIFAR-10 dataset

Large-scale image recognition on ImageNet:

This is a large-scale vision based task, image recognition based on a high-resolution camera. During this experiment, we use ImageNet as our training and testing dataset. The ImageNet dataset consists of 1.2 million 224 × 224 color images in 1000 classes with 100,000 images for testing.

During the evaluation, we still use VGGNet structure as the original network structure. The detailed structures of best compressed models without accuracy degradation of all al-

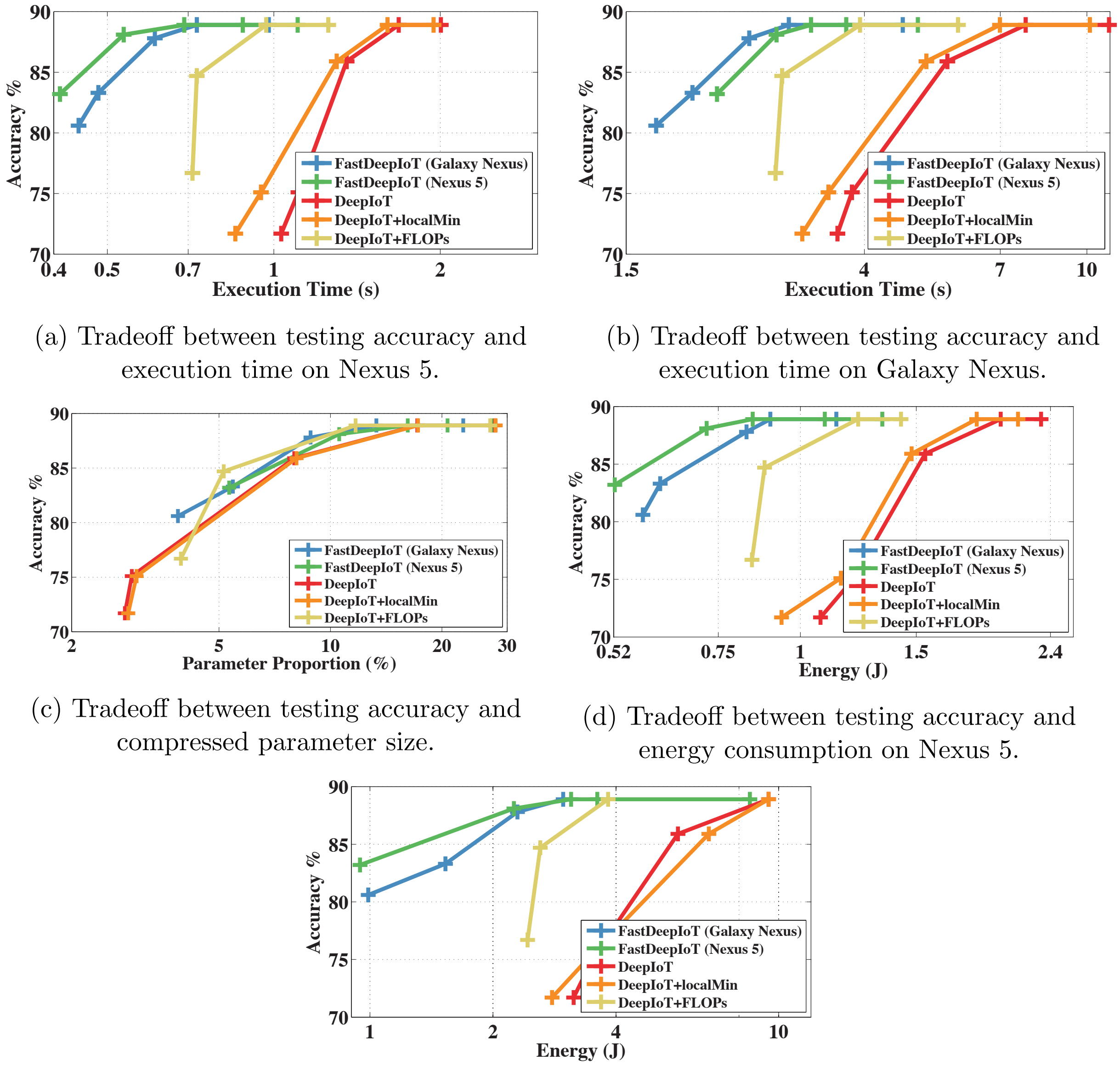
Table 4.11: VGGNet (hidden units) on ImageNet dataset.

	No Execution Time Model				Nexus 5	Galaxy Nexus
Layer	Original	DeepIoT	localMin	FLOPs	FastDeepIoT	FastDeepIoT
conv1-1 (3 × 3)	64	43	44	23	12	16
conv1-2 (3 × 3)	64	47	48	32	12	16
conv2-1 (3 × 3)	128	100	100	65	20	44
conv2-2 (3 × 3)	128	97	100	67	40	40
conv3-1 (3 × 3)	256	164	164	116	88	108
conv3-2 (3 × 3)	256	164	164	135	72	104
conv3-3 (3 × 3)	256	153	156	70	116	108
conv4-1 (3 × 3)	512	235	236	72	268	240
conv4-2 (3 × 3)	512	240	240	181	236	216
conv4-3 (3 × 3)	512	220	240	258	340	200
conv5-1 (3 × 3)	512	255	256	261	376	240
conv5-2 (3 × 3)	512	260	260	303	376	288
conv5-3 (3 × 3)	512	257	260	47	176	216
fc1	4096	436	436	1594	656	920
fc2	4096	1169	1169	824	1150	1189
fc3	1000	297	297	405	287	402
Test top-5 accuracy	88.9%	88.9%	88.9%	88.9%	88.9%	88.9%
Execution time t (Nexus 5)		1682 ms	1605 ms	968.8 ms	**688.8 ms**	725.7 ms
Execution time t (Galaxy)		7773 ms	6991 ms	3930 ms	3211 ms	**2930 ms**

gorithms are shown in Table 4.11. Note that the original VGGNet for 224 × 224 colour image input is too large for running on two testing hardware. FastDeepIoT achieves the best performance on the execution time among all methods. Compared with the state-of-the-art DeepIoT method, FastDeepIoT can further reduce the execution time by 59% to 62%. DeepIoT+localMin still outperforms DeepIoT by reducing around 5% to 10% of execution time. In addition, FastDeepIoT can further reduce 25% to 29% of execution time compared with DeepIoT+FLOPs.

Figure 4.11a and 4.11b shows the tradeoff between testing top-5 accuracy and execution time for all algorithms. FastDeepIoT consistently outperforms all other algorithms by a significant margin. With the help of execution-time local minima, DeepIoT+localMin can still outperform DeepIoT in all cases. DeepIoT+FLOPs performs better than DeepIoT in this case. As shown in Table 4.8, FLOPs still possess a certain degree of correlation with execution time. However, compared to the execution time model in FastDeepIoT, FLOPs becomes an inferior execution time estimator.

Figure 4.11d and 4.11e illustrates the tradeoff between testing top-5 accuracy and energy consumptions. FastDeepIoT outperforms all algorithms with a large margin. However, FastDeepIoT with the Galaxy Nexus execution time model is not the most energy-saving compression method on the Galaxy Nexus device. Also, DeepIoT+localMin cannot consistently outperforms DeepIoT on energy saving. These two observations witness the discrepancies



(a) Tradeoff between testing accuracy and execution time on Nexus 5.

(b) Tradeoff between testing accuracy and execution time on Galaxy Nexus.

(c) Tradeoff between testing accuracy and compressed parameter size.

(d) Tradeoff between testing accuracy and energy consumption on Nexus 5.

(e) Tradeoff between testing accuracy and energy consumption on Galaxy Nexus.

Figure 4.11: System performance tradeoff for VGGNet on ImageNet dataset

between the execution time and energy modeling on mobile devices. Figure 4.11c shows the tradeoff between testing accuracy and left proportion of model parameters. Again, all methods show the similar tradeoff, which indicates that FastDeepIoT is a parameter-efficient method on execution time reduction.

Table 4.12: DeepSense (hidden units) on HHAR dataset.

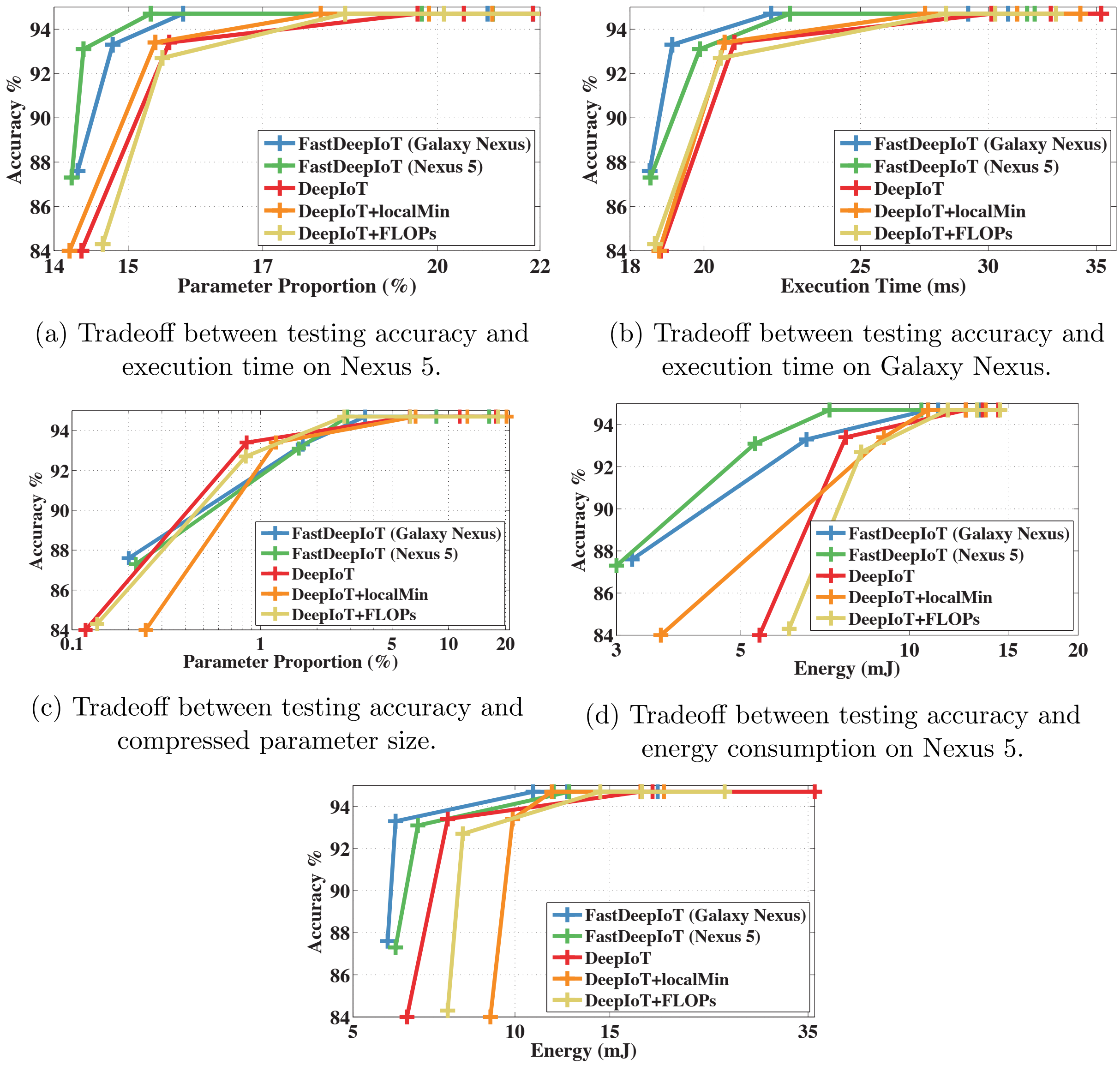
		No Execution Time Model										Nexus 5		Galaxy Nexus	
Layer		Original		DeepIoT		localMin		FLOPs		$t_{\min}$		FastDeepIoT		FastDeepIoT	
conv1a	conv1b (2 × 9)	64	64	20	19	20	20	26	25	4	4	16	8	16	16
conv2a	conv2b (1 × 3)	64	64	20	14	20	16	19	17	4	4	8	12	20	16
conv3a	conv3b (1 × 3)	64	64	23	23	24	24	22	22	4	4	16	16	16	16
conv4 (2 × 8)		64		10		12		9		4		12		16	
conv5 (1 × 6)		64		12		12		13		4		16		16	
conv6 (1 × 4)		64		17		18		18		4		12		16	
gru1		120		27		27		11		1		15		10	
gru2		120		31		31		15		1		17		10	
Test accuracy		94.6%		94.7%		94.7%		94.7%		16.7%		94.7%		94.7%	
Execution time t (Nexus 5)		26.2 ms		19.5 ms		17.9 ms		18.3 ms		14.1 ms		**15.3 ms**		15.8 ms	
$t - t_{\min}$ (Nexus 5)		12.1 ms		5.4 ms		3.8 ms		4.2 ms				**1.2 ms**		1.7 ms	
Execution time t (Galaxy)		70.9 ms		30.1 ms		27.4 ms		28.2 ms		18.4 ms		22.6 ms		**22.0 ms**	
$t - t_{\min}$ (Galaxy)		52.5 ms		11.7 ms		9.0 ms		9.8 ms				4.2 ms		**3.6 ms**	

Heterogeneous human activity recognition:

This is a human-centric context sensing application, recognizing human activities with accelerometer and gyroscope. Especially, we are considering the heterogeneous human activity recognition (HHAR). This task focuses on the generalization ability with human who has not appeared in the training dataset. During this experiment, we use the dataset collected by Allan et al. [11].

During this evaluation, we use DeepSense structure as the original network structure [2]. Table 4.12 illustrates the detailed structure of the original network and final compressed networks generated by four algorithms with no degradation on testing accuracy. As shown in Table 4.12, FastDeepIoT achieves the best performance on two devices with the corresponding execution time models. Compared with DeepIoT, FastDeepIoT can further reduce the model execution time by 22% to 42%. During the compressing process, we observe that all compressed models tend to approach a model execution time lower bound, which has not been seen in the previous two experiments. In order to obtain the lower bound, we build a DeepSense structure with all hidden units that equal to 1, and then applies Algorithm 4.3 to find the structure that triggers local minimum. The resulted structure is illustrated in Table 4.12 denoted by $t_{\min}$. If we calculate the deductible model execution time by subtracting $t_{\min}$ from the model execution time, compared with DeepIoT, FastDeepIoT can reduce the deductible execution time by 69% to 78%.

Furthermore, we can attempt to deduce the fundamental cause of the lower bound with our execution time model. As shown in (4.17), the execution time of recurrent layer is partially controlled by the number of *step*, which can be interpreted as an initialization overhead for each *step* in the recurrent layer. We can use an example to illustrate the relationship between the *step* overhead and this lower bound. In our experiment, there are 20 steps in the GRU. The coefficient of *step* on Nexus 5 is 0.666 ms. Therefore, the lower bound is



(a) Tradeoff between testing accuracy and execution time on Nexus 5.

(b) Tradeoff between testing accuracy and execution time on Galaxy Nexus.

(c) Tradeoff between testing accuracy and compressed parameter size.

(d) Tradeoff between testing accuracy and energy consumption on Nexus 5.

(e) Tradeoff between testing accuracy and energy consumption on Galaxy Nexus.

Figure 4.12: System performance tradeoff for DeepSense on HHAR dataset

$14.1 \approx 20 \times 0.666$ ms. Thus, only algorithms dealing with reducing recurrent-layer steps can help further reducing the model execution time. Unfortunately, to the best of our knowledge, there is no existing work that solves this problem. However, our empirical observation and execution time model reveal an interesting problem that requires future research.

The tradeoffs between testing accuracy and execution time for different algorithms are illustrated in Figure 4.12a and 4.12b. FastDeepIoT still achieves the best tradeoff for all cases. DeepIoT+localMin still performs better than DeepIoT with the help of our structure expanding and local minima searching algorithm. The performance of DeepIoT+FLOPs is not stable among different devices. The tradeoffs between testing accuracy and energy consumption are illustrated in Figure 4.12d and 4.12e. FastDeepIoT performs better than all other baselines in almost all cases. The tradeoffs between testing accuracy and remaining proportion of model parameters are illustrated in Figure 4.12c. All algorithms show comparable results.

CHAPTER 5: DEEP LEARNING FOR LABEL-LIMITED IOT SYSTEMS

In this section, we first introduce the technical details of the SenseGAN framework. Then, we evaluate SenseGAN with several representative and challenging IoT applications with different proportions of labelled and unlabelled data.

5.1 THE DESIGN OF SENSEGAN

In this section, we introduce our SenseGAN semi-supervised learning framework for IoT applications. We separate our descriptions into six parts. First, we give some preliminary knowledge about GAN. Next, we provide an overview of the SenseGAN framework with its internal interactions among different components. Then, we introduce our designs of generator, discriminator, and classifier in SenseGAN respectively. In the end, we discuss SenseGAN's training process.

5.1.1 Preliminaries

In this section, we give a preliminary overview of GANs [72], GANs with the Wasserstein metric [24, 73], the Gumbel-Softmax function for categorical representations [25], and deep learning classification models for IoT applications, all of which serve as key design ingredients of our SenseGAN framework.

Generative Adversarial Networks (GANs)

The idea of GANs is to design a game between two competing networks. The generator network takes the noise vectors as inputs and generates data samples. The discriminator network takes either a generated sample or a real data sample, and distinguishes between the two. The generator is trained to fool the discriminator [72].

The game between generator G and discriminator D can be formulated as the minmax objective:

$$\min_G \max_D \mathbb{E}_{\mathbf{x}\sim\mathbb{P}_r}[\log(D(\mathbf{x}))] + \mathbb{E}_{\tilde{\mathbf{x}}\sim\mathbb{P}_g}[\log(1 - D(\tilde{\mathbf{x}}))], \tag{5.1}$$

where $\mathbb{P}_r$ is the real data distribution and $\mathbb{P}_g$ the generated data distribution implicitly defined by $\tilde{\mathbf{x}} = G(\mathbf{z})$, $\mathbf{z} \sim p(\mathbf{z})$ (the input of generator $\mathbf{z}$ is sampled from a simple noise distribution, such as the uniform distribution or a spherical Gaussian distribution).

If the discriminator is optimal, the training objective function (5.1) amounts to minimizing the Jensen-Shannon (JS) divergence between $\mathbb{P}_r$ and $\mathbb{P}_g$. In practice, a stochastic lower-bound to the JS divergence is minimized.

Wasserstein GANs

Since training instability has hindered the deployment of GANs on deeper and more complex neural network structures, a great amount of research efforts have been made recently to tackle this problem. Arjovsky et al. [24] argue that Jensen-Shannon divergence are potentially not continuous and thus cannot provide a usable gradient for the generator. They propose Earth mover's distance (also called Wasserstein-1 Distance) as the training objective. Earth mover's distance is the minimum cost of transporting mass in order to transform one distribution into the other distribution, where the cost is mass times transport distance. They have shown that the Earth mover's distance is continuous everywhere, and differentiable almost everywhere, providing the desirable property for GANs training.

Wasserstein GAN (WGAN) is thus proposed, and its objective function is constructed by applying the Kantorovich-Rubinstein duality [74]

$$\min_{G} \max_{D \in \mathcal{D}} \mathbb{E}_{\mathbf{x} \sim \mathbb{P}_r}[D(\mathbf{x})] - \mathbb{E}_{\tilde{\mathbf{x}} \sim \mathbb{P}_g}[D(\tilde{\mathbf{x}})], \tag{5.2}$$

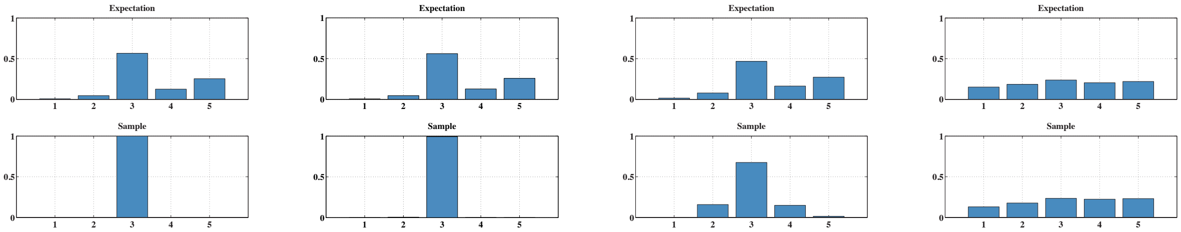
where $\mathcal{D}$ is the set of 1-Lipschitz functions, $\mathbb{P}_r$ and $\mathbb{P}_g$ are defined the same as before. Then given the optimal discriminator, WGAN's training objective function (5.2) amounts to minimizing the Earth mover's distance between $\mathbb{P}_r$ and $\mathbb{P}_g$.

Then the remaining question is how to enforce the Lipschitz constraint on the discriminator. Arjovsky et al. [24] make the Lipschitz constraint by clipping the weights within a compact space $[-c, c]$. This results in a subset of k-Lipschitz functions for k depending on the space boundary parameter c and the structure of the discriminator.

However Gulrajani et al. [73] claim that weight clipping can still cause optimization difficulties such as capacity underuse and gradients exploding or vanishing. Then an alternative is proposed: a differentiable function is 1-Lipschitz if and only if it has gradients with norm less than or equal to 1 everywhere. Since an exact constraint is not easily tractable, Gulrajani et al. enforce a soft version by making gradient penalty on sampled points. The loss function for the discriminator then becomes:

$$\mathbb{E}_{\tilde{\mathbf{x}} \sim \mathbb{P}_g}[D(\tilde{\mathbf{x}})] - \mathbb{E}_{\mathbf{x} \sim \mathbb{P}_r}[D(\mathbf{x})] + \lambda \cdot \mathbb{E}_{\hat{\mathbf{x}} \sim \mathbb{P}_{\hat{\mathbf{x}}}}\big[(\|\nabla D(\hat{\mathbf{x}})\|_2 - 1)^2\big], \tag{5.3}$$

where gradient penalties are made by taking samples $\hat{\mathbf{x}}$ from straight lines between points



(a) Categorical distribution.

(b) Gumbel-Softmax distribution with $\tau = 0.1$.

(c) Gumbel-Softmax distribution with $\tau = 1$.

(d) Gumbel-Softmax distribution with $\tau = 10$.

Figure 5.1: The expectation and sample of Categorical and Gumbel-Softmax distribution.

in the data distribution $\mathbb{P}_r$ and the generator distribution $\mathbb{P}_g$:

$$\begin{aligned} &\epsilon \sim U[0,1], x \sim \mathbb{P}_r, \tilde{x} \sim \mathbb{P}_g, \\ &\hat{\mathbf{x}} = \epsilon x + (1-\epsilon)\tilde{x}, \end{aligned} \tag{5.4}$$

where $U[0,1]$ is the uniform distribution from 0 to 1.

The WGAN with gradient penalty has achieved the state-of-the-art performance on multiple generative tasks, such as images and text generations. To the best of our knowledge, SenseGAN is the first study to adopt WGAN with gradient penalty training strategy into the semi-supervised learning.

Gumbel Softmax

Discrete variables sampled from categorical distribution is a powerful technique for representing categorical distributions. In our framework, the discriminator in SenseGAN is designed to differentiate the joint data/label distributions between partially generated samples and real samples, which naturally requires taking discrete variables for categorical representation as inputs.

However, neural networks with discrete variables involving sampling from categorical distributions are non-differentiable, making them difficult to train with the backpropagation algorithm. Existing stochastic gradient estimation requires variance reduction techniques to stabilize the training process. In order to alleviate the problem of high-variance gradient estimation in neural networks with discrete variables, a continuous relaxation of the discrete variable is needed.

Recent study on Gumbel-Softmax defines a continuous distribution over the simplex that can approximate samples from a categorical distribution [25], where the theoretical analysis has been made.

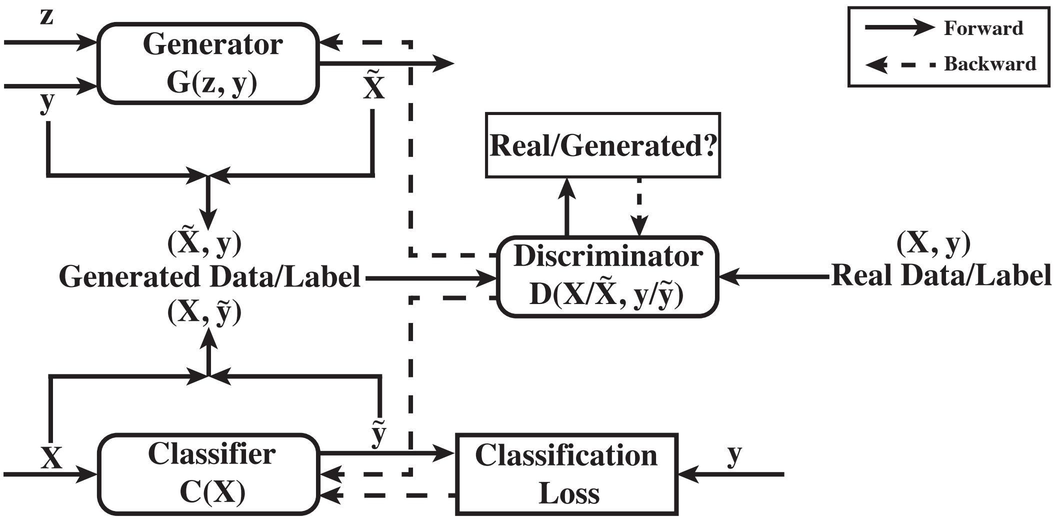


Figure 5.2: The illustration of SenseGAN components.

We assume categorical samples are encoded as l-dimensional one-hot vectors lying on the corners of the $(l-1)$-dimensional simplex, Δ^{l-1}. $o_1, \cdots, o_l$ are multinomial logits for l categories. We can therefore generate l-dimensional sample vectors y:

$$y_i = \frac{\exp((o_i + g_i)/\tau)}{\sum_{j=1}^{l} \exp((o_j + g_j)/\tau)}, \tag{5.5}$$

where $g_1, \cdots, g_l$ are i.i.d. samples drawn from Gumbel(0, 1). The Gumbel(0, 1) distribution can be sampled using inverse transform sampling by drawing $u \sim U[0,1]$ and computing $g = -\log(-\log(u))$.

As the softmax temperature τ approaches 0, samples from the Gumbel-Softmax distribution turns into one-hot representations, and its expectation approaches the corresponding categorical distribution from the uniform distribution. The expectation and sample of Gumbel-Softmax distribution with different softmax temperature τ are illustrated in Figure 5.1. In SenseGAN, the multinomial logits of the classifier go through the Gumbel-Softmax before being fed into the discriminator, which further improves the training stability and the final predictive performance.

5.1.2 SenseGAN Components Overview

The SenseGAN framework consists of three basic components. We assume the following notations: $\mathbf{X}$ denotes the input sensing data tensor, $\mathbf{y}$ the one-hot categorical representation, and $\mathbf{z}$ the random vector drawn from the random Gaussian distribution. The relationship among these three components is illustrated in Figure 5.2.

1. **Generator $\tilde{\mathbf{X}} \sim G(\mathbf{z}, \mathbf{y})$:** The generator takes a random vector and a corresponding one-hot categorical representation as the input, and generates a sensing data tensor $\tilde{\mathbf{X}}$ that "fools" the discriminator into thinking that it is the real sensing data.
2. **Classifier $\tilde{\mathbf{y}} \sim C(\mathbf{X})$:** The classifier takes a sensing data tensor as the input and generates classification result $\tilde{\mathbf{y}}$ that can "fool" the discriminator into thinking that it is the real data label. If the sensing data tensor happens to be from the limited amount of labelled data, the classification result $\tilde{\mathbf{y}}$ should also fit the supervision of label $\mathbf{y}$.
3. **Discriminator $D(\mathbf{X}/\tilde{\mathbf{X}}, \mathbf{y}/\tilde{\mathbf{y}})$:** The discriminator takes a pair of sensing data and corresponding one-hot categorical representation as the input. It gives each input pair a score for indicating whether the pair is sampled from the real labelled dataset or partially generated by other components.

The intuition of how our SenseGAN framework is able to leverage unlabelled data to enhance its predictive power is as follows. The discriminator tries to discriminate real data/label samples and the partially generated data/label samples; the generator attempts to generate and recover the real sensing inputs based on the categorical information that can fool the discriminator; and the classifier tries to predict the label of sensing inputs that can both fit the supervision and fool the discriminator. During the adversarial game among the three components under the training process, the resulting three improved components can mutually boost performance. When the training reaches the optimality, the discriminator will have learnt the true joint probability distribution of the input sensing data and their corresponding labels for both the labelled and unlabelled samples. The classifier will have learnt the true conditional probability of labels given the sensing input.

All three components are represented by neural networks. We will discuss their specific structures for dealing with the multimodal sensing inputs in detail in the following subsections. In addition, the structure of the classifier can be task-specific or unified with diverse IoT applications [2]. We therefore will not introduce the detailed structure for the classifier but only discuss its output representation. We treat the classifier as an modular and customizable component for IoT applications when using SenseGAN for semi-supervised learning.

5.1.3 SenseGAN Generator

The goal here is to generate sensing data tensor by modelling the conditional probability of sensing data given the one-hot label representation. The randomness of generated samples is controlled by the input random vectors.

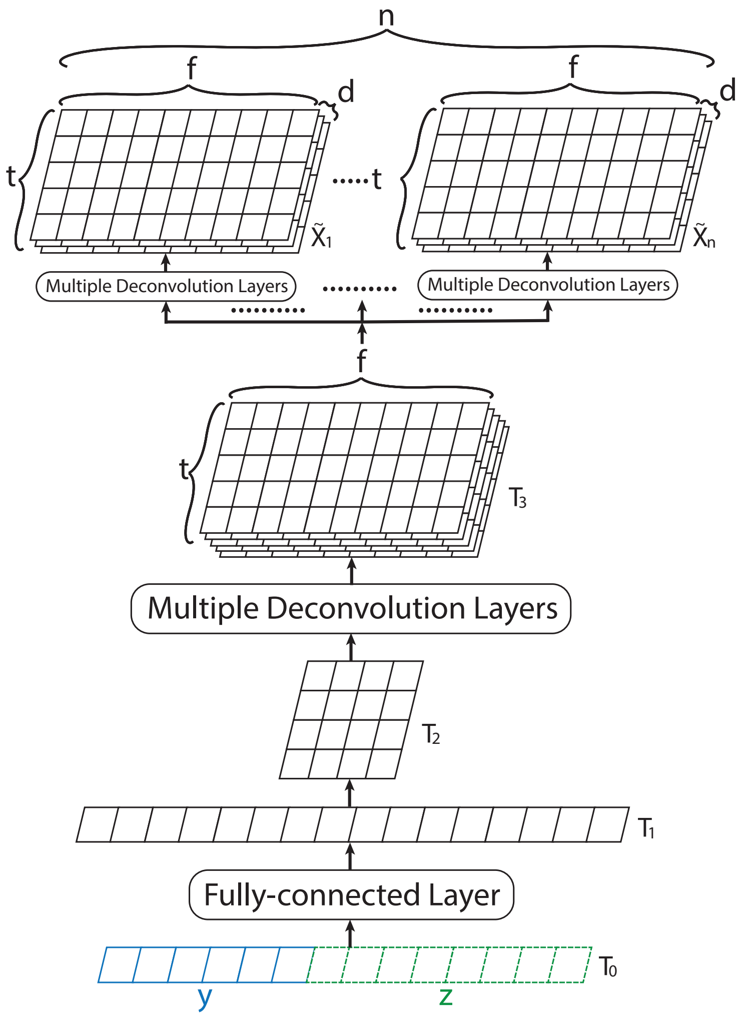

n
f
d
f
d
t
t
$\tilde{X}_1$
$\tilde{X}_n$
Multiple Deconvolution Layers
Multiple Deconvolution Layers
f
t
T_3
Multiple Deconvolution Layers
T_2
T_1
Fully-connected Layer
T_0
y
z

Figure 5.3: The neural network design of SenseGAN generator.

We denote the generated sensing data tensor as $\tilde{\mathbf{X}} \in \mathbb{R}^{n \times t \times f \times d}$, where n is the number of sensors used in an IoT application, t is the number of time steps, f is the length of frequency domain or the number of time points within a time step, and d is the feature dimension of data on each frequency or time point. We denote the one-hot label representation as $\mathbf{y} \in \mathbb{R}^{l}$, where l is the number of categories of an IoT application. We also denote the input random vector as $\mathbf{z} \in \mathbb{R}^{r}$, where r is the dimension of random vector.

The illustration of SenseGAN generator $G(\mathbf{z}, \mathbf{y})$ is shown in Figure 5.3. The input of generator $\mathbf{T}_0$ is the concatenated vector of the one-hot label representation $\mathbf{y}$ and the random vector $\mathbf{z}$. $\mathbf{T}_0$ first goes through a fully-connected layer for learning a latent representation $\mathbf{T}_1$, which is then transformed into a matrix form $\mathbf{T}_2$.

Then the generator starts the process of generating sensing data tensor as $\tilde{\mathbf{X}} \in \mathbb{R}^{n \times t \times f \times d}$.

We assume that n sensing data possess a joint latent representation, and the generation process is first going through generating along the time/frequency dimensions, t and f, with a shared structure and then going through generating the feature dimension d with individual structures for n sensors.

As shown in Figure 5.3, $\mathbf{T}_2$ goes through multiple deconvolution layers and generates a joint latent representation along the time and frequency dimensions $\mathbf{T}_3 \in \mathbb{R}^{t \times f \times c}$, where c is the number of filters in the deconvolution layer. Then the joint representation $\mathbf{T}_3$ goes through n independent multiple deconvolution layers for generating the sensing data tensor for n sensors, $\tilde{\mathbf{X}} = [\tilde{\mathbf{X}}_1, \cdots, \tilde{\mathbf{X}}_n]$.

The whole generator structure tries to learn the conditional distribution of sensing data given the label representation $P(\mathbf{X}|\mathbf{y})$. It also helps the discriminator to learn the joint data/label distribution $P(\mathbf{X}, \mathbf{y})$ and the classifier to learn the conditional distribution of label given the sensing data $P(\mathbf{y}|\mathbf{X})$ during the adversarial game by leveraging unlabelled data.

5.1.4 SenseGAN Discriminator

The SenseGAN discriminator aims to differentiate partially generated data/label tuples from the real ones by modelling the joint data/label distribution.

We again denote the sensing data as $\mathbf{X} \in \mathbb{R}^{n \times t \times f \times d}$, and the one-hot label representation as $\mathbf{y} \in \mathbb{R}^{l}$. The illustration of SenseGAN discriminator $D(\mathbf{X}, \mathbf{y})$ with two sensor inputs is shown in Figure 5.4. The structure for n sensor inputs can be similarly designed. We first need to generate input $\hat{\mathbf{X}}$ containing both data and label information. We simply expand label representation $\mathbf{y}$ into $\mathbf{Y} \in \mathbb{R}^{n \times t \times f \times l}$ with the tiling operation, and then concatenate $\mathbf{X}$ and $\mathbf{Y}$ into the generator input $\hat{\mathbf{X}} \in \mathbb{R}^{n \times t \times f \times (d+l)}$ as shown in Figure 5.4. These expansion and concatenation operations can help to merge the data and label information and to balance gradient penalty (5.3) between data and label inputs.

The design of discriminator is to first merge the information from multiple sensors and then extract important relationships along the time and frequency dimensions. Since convolution layers can learn about the long-term dependency by stacking multiple of them [75], SenseGAN uses full-convolution structure (with fully-connected layers at the end) instead of the recurrent neural network for speeding up the training process. The empirical comparison betweem CNN-based and RNN-based discriminator is shown in Section 5.2.2.

As shown in Figure 5.4, SenseGAN discriminator first learns a joint representation $\mathbf{T}_1$ from multiple sensors through a convolution layer. Then multiple convolution layers are used to learn the latent representation $\mathbf{T}_2$ along the time and frequency dimension. At

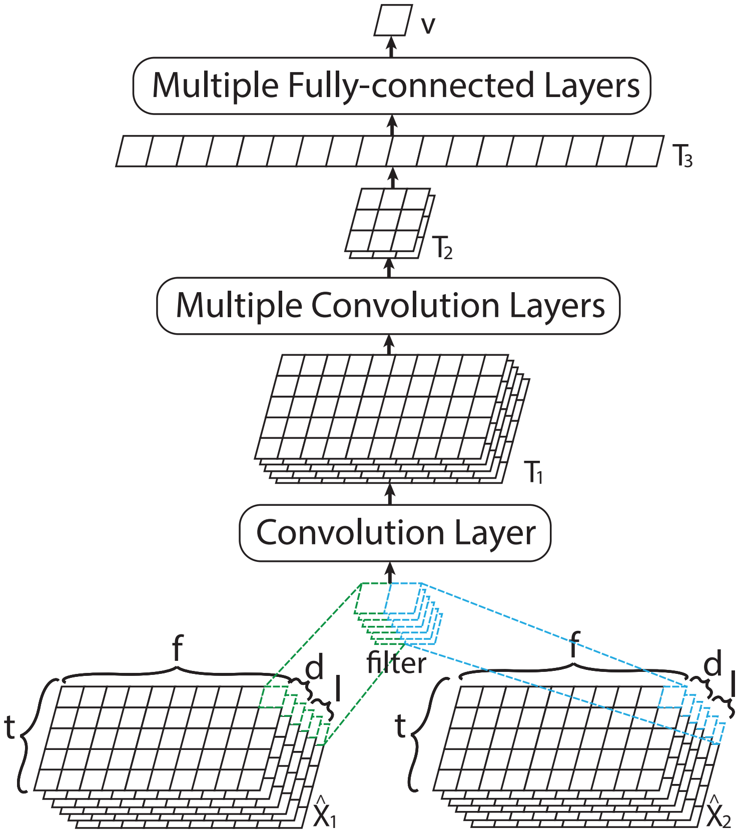


Figure 5.4: The neural network design of SenseGAN discriminator with two sensor inputs.

last, the discriminator generates the score v through multiple fully-connected layers for determining whether the input $\hat{\mathbf{X}}$ is drawn from real data samples or partially generated by other components.

The discriminator tries to differentiate whether the input is drawn from real samples or is partially generated by learning the joint data and label distribution $p(\mathbf{X}, \mathbf{y})$. The discriminator structure in Figure 5.4 tries to fit the multimodal sensing inputs that exists in IoT applications better.

5.1.5 SenseGAN Classifier

As mentioned previously, the SenseGAN classifier, by design, serves as an application-dependent customizable module, capable of taking the form of any existing neural network classifiers. The classifier can be formulated as $\mathbf{o} = C(\mathbf{X})$, where $\mathbf{X} \in \mathbb{R}^{n \times t \times f \times d}$ is the sensing data input, and $\mathbf{o} \in \mathbb{R}^{l}$ is the multinomial logits for l categories.

However as shown in Figure 5.2, two other components take categorical representations instead of multinomial logits as inputs. When the sensing data inputs $\mathbf{X}$ have no existing label, the classifier feeds the predictions $\tilde{\mathbf{y}}$ as well as the sensing data $\mathbf{X}$ into the discriminator. On the other hand, if the sensing data inputs $\mathbf{X}$ have existing labels, the classifier feeds the predictions $\tilde{\mathbf{y}}$ into classification loss with true label $\mathbf{y}$ for supervision. We denote these two predictions as $\tilde{\mathbf{y}}_d$ and $\tilde{\mathbf{y}}_l$ for discriminator input and classification loss input respectively.

Softmax function is a common practice for converting multinomial logits into categorical representations, which is also our choice for $\tilde{\mathbf{y}}_l$. However for $\tilde{\mathbf{y}}_d$, in order to prevent discriminator from differentiating real or generated samples by just looking at whether the categorical representations are one-hot representations or categorical distribution, we choose Gumbel-Softmax (5.5) for converting multinomial logits into categorical representations, which is introduced in Section 5.1.1. Compared with sampling from categorical distribution, Gumbel-Softmax can eliminate non-differentiable operations and prevent using gradient estimation method with high variance.

5.1.6 SenseGAN Training

We now discuss the SenseGAN training that involves the aforementioned three components. We denote $\{(\mathbf{X}_l, \mathbf{y}_l)\}$ as the labelled dataset and $\{(\mathbf{X}_u)\}$ as the unlabelled dataset.

The SenseGAN objective function consists of two parts. The first part captures the classifier's intent to minimize the classification loss of labelled data, which can be formulated as:

$$\min_{C} \frac{1}{m_l} \sum_{(\mathbf{X}_l, \mathbf{y}_l)} \ell\big(\mathbf{y}, C(\mathbf{X})\big) \tag{5.6}$$

where $\ell(\cdot, \cdot)$ denotes the cross entropy loss and $C(\cdot)$ the SenseGAN classifier. The second part is the adversarial game among the discriminator, the generator, and the classifier. The discriminator tries to distinguish positive data/label tuples of "real" labelled dataset from negative data/label tuples that are partially generated by the generator or classifier. At the same time, the generator and classifier try to generate data/label tuples that can fool the discriminator. Here we provide the formal definition of negative and positive data/label tuples during the training process.

There are two types of negative data/label tuples. The first type is $(\mathbf{X}_u^{(1)}, C(\mathbf{X}_u^{(1)}))$, where we sample sensing data from the unlabelled dataset and generate one-hot label with the SenseGAN classifier. The second type is $(G(\mathbf{z}, \mathbf{y}_g), \mathbf{y}_g)$, where we randomly generate one-hot label from all possible categories and generate sensing data with the SenseGAN generator.

The positive data/label tuples are $(\mathbf{X}_l, \mathbf{y}_l)$, where we directly sample data/label tuples from the labelled dataset. In order to prevent the discriminator from overfitting the limited labelled dataset by memorizing all of them, we introduce the pseudo positive data/label

Table 5.1: Definitions of positive and negative data/label tuples.

Postive Tuples		Pseudo Postive Tuples	
Data sampled from the labelled dataset	$\mathbf{X}_l$	Data sampled from the unlabelled dataset	$\mathbf{X}_u^{(2)}$
Corresponding real label	$\mathbf{y}_l$	Corresponding label generated by the Classifier	$C(\mathbf{X}_u^{(2)})$

Negative Tuples 1		Negative Tuples 2	
Data sampled from the unlabelled dataset	$\mathbf{X}_u^{(1)}$	Corresponding data generated by the Generator	$G(\mathbf{z}, \mathbf{y}_g)$
Corresponding label generated by the Classifier	$C(\mathbf{X}_u^{(1)})$	Randomly generated label	$\mathbf{y}_g$

tuples, $(\mathbf{X}_u^{(2)}, C(\mathbf{X}_u^{(2)}))$ when training the discriminator in practice. The detailed definitions of these negative and positive data/label tuples are summarized in Table 5.1.

With these definitions, we can formulate the second part of the objective function. We apply the Earth mover's distance to formulate the adversarial game, which is similar to the Wasserstein GAN (5.2).

$$\begin{aligned}
&\max_{D \in \mathcal{D}} \frac{1}{m_l + m_u^{(2)}} \Bigg(\sum_{\mathbf{X}_l, \mathbf{y}_l} D(\mathbf{X}_l, \mathbf{y}_l) + \sum_{\mathbf{X}_u^{(2)}} D\Big(\mathbf{X}_u^{(2)}, C\big(\mathbf{X}_u^{(2)}\big)\Big) \Bigg) - \frac{\alpha}{m_g} \sum_{\mathbf{y}_g} D\big(G(\mathbf{z}, \mathbf{y}_g), \mathbf{y}_g\big) \\
&\qquad - \frac{1-\alpha}{m_u^{(1)}} \sum_{\mathbf{X}_u^{(1)}} D\Big(\mathbf{X}_u^{(1)}, C\big(\mathbf{X}_u^{(1)}\big)\Big), \qquad (5.7) \\
&\min_{G,C} - \frac{\alpha}{m_g} \sum_{\mathbf{y}_g} D\big(G(\mathbf{z}, \mathbf{y}_g), \mathbf{y}_g\big) - \frac{1-\alpha}{m_u^{(1)}} \sum_{\mathbf{X}_u^{(1)}} D\Big(\mathbf{X}_u^{(1)}, C\big(\mathbf{X}_u^{(1)}\big)\Big),
\end{aligned}$$

where α is a hyper-parameter for balancing two types of negative tuples with default value 0.5; $\mathcal{D}$ is the set of 1-Lipschitz functions; m_l, $m_u^{(2)}$, $m_u^{(1)}$, and m_g are the batch sizes of positive, pseudo-positive, and two negative tuples during training. In SenseGAN, we enforce the Lipschitz constraint by making gradient penalty (5.3) with λ.

The pseudo positive tuples $(\mathbf{X}_u^{(2)}, C(\mathbf{X}_u^{(2)}))$ also work as a variance reduction method for the negative tuples $(\mathbf{X}_u^{(1)}, C(\mathbf{X}_u^{(1)}))$, which further stabilize the training process [76]. In general, the pseudo positive tuple introduces a biased distribution into the discriminator. However, the bias is controlled by the batch sizes of positive and pseudo positive tuples, m_l and $m_u^{(2)}$. In addition, the classifier can often achieve a reasonably good prediction result quickly, which further reduces the bias. The empirical evaluation of pseudo positive tuple is shown in Section 5.2.2.

By combining the two objective functions (5.6) and (5.7) with hyper-parameter γ, we can summarize our final training process. As shown in Algorithm 5.1, the training proceeds by iteratively updating the parameters of the discriminator, classifier, and generator. The update of discriminator (Line 3 - Line 8) consists of three parts: increasing the score of positive tuples, reducing the score of negative tuples, and imposing gradient penalty for

Algorithm 5.1. SenseGAN training process

```
1: Initialize: discriminator with parameter $\theta_D$, generator with parameter $\theta_G$, and classifier with parameter $\theta_C$.
2: for $k$ training iterations do
3:     for $k_d$ iterations do
4:         Sample $(\mathbf{X}_l, \mathbf{y}_l)$, $(\mathbf{X}_u^{(1)})$, $(\mathbf{X}_u^{(2)})$, $(\mathbf{y}_g)$, $\epsilon \sim U[0,1]$, and $\mathbf{z} \sim \mathcal{N}(0,1)$
5:         $\bar{\mathbf{X}} = \epsilon \cdot \mathbf{X}_l + (1-\epsilon) \cdot G(\mathbf{z}, \mathbf{y}_g)$
6:         $\bar{\mathbf{y}} = Gumbel\_softmax(\epsilon \cdot \mathbf{y}_l + (1-\epsilon) \cdot \mathbf{y}_g)$
7:         Update $\theta_D$ by descending along its gradient $\nabla_{\theta_D} \Big[ \frac{\alpha}{m_g} \sum_{\mathbf{y}_g} D\big(G(\mathbf{z}, \mathbf{y}_g), \mathbf{y}_g\big) + \frac{1-\alpha}{m_u^{(1)}} \sum_{\mathbf{X}_u^{(1)}} D\Big(\mathbf{X}_u^{(1)}, C\big(\mathbf{X}_u^{(1)}\big)\Big) - \frac{1}{m_l + m_u^{(2)}} \Big( \sum_{\mathbf{X}_l, \mathbf{y}_l} D(\mathbf{X}_l, \mathbf{y}_l) + \sum_{\mathbf{X}_u^{(2)}} D\Big(\mathbf{X}_u^{(2)}, C\big(\mathbf{X}_u^{(2)}\big)\Big)\Big) + \lambda(\|\nabla D(\bar{\mathbf{X}}, \bar{\mathbf{y}})\|_2 - 1)^2 \Big]$
8:     end for
9:     for $k_c$ iterations do
10:        Sample $(\mathbf{X}_l, \mathbf{y}_l)$ and $(\mathbf{X}_i^{(1)})$
11:        Update $\theta_C$ by descending along its gradient $\nabla_{\theta_C} \Big[ \frac{\gamma}{m_l} \sum_{(\mathbf{X}_l, \mathbf{y}_l)} CE\big(\mathbf{y}, C(\mathbf{X})\big) - \frac{1-\alpha}{m_u^{(1)}} \sum_{\mathbf{X}_u^{(1)}} D\Big(\mathbf{X}_u^{(1)}, C\big(\mathbf{X}_u^{(1)}\big)\Big) \Big]$
12:    end for
13:    for $k_g$ iterations do
14:        Sample $(\mathbf{y}_g)$ and $\mathbf{z} \sim \mathcal{N}(0,1)$
15:        Update $\theta_G$ by descending along its gradient $\nabla_{\theta_G} \Big[ -\frac{\alpha}{m_g} \sum_{\mathbf{y}_g} D\big(G(\mathbf{z}, \mathbf{y}_g), \mathbf{y}_g\big) \Big]$
16:    end for
17: end for
```

enforcing the Lipschitz constraint. The update of classifier (Line 9 - Line 12) consists of two parts: reducing the classification loss of labelled data and learning to make prediction that can obtain high score by the discriminator. The update of generator (Line 13 - Line 16) has only one part: learning to generate sensing data that can obtain high score by the discriminator so that the discriminator can be fooled to believe that the generated sensing data is real. This iterative process can gradually improve the performance of all three components by mutually boosting themselves with limited labelled supervision and abundant unlabelled data.

5.2 THE EVALUATION OF SENSEGAN

In this section, we test SenseGAN on three IoT applications with several sets of experiments evaluating the effectiveness, the design choices, and the resource consumption of SenseGAN on commodity IoT devices.

5.2.1 Experiments Overview

For the evaluation, we conduct all our experiments on three different tasks. We next briefly describe these tasks and introduce the training and testing datasets.

1. *Heterogeneous human activity recognition (HHAR):* In this task, we perform a human activity recognition task with accelerometer and gyroscope measurements. We use the dataset collected by Allan et al. [11]. This dataset contains readings from two

motion sensors (accelerometer and gyroscope). Readings were recorded when users executed activities scripted in no specific order, while carrying smartwatches and smartphones. The dataset contains 9 users, 6 activities (biking, sitting, standing, walking, climbStairup, and climbStairdown), and 6 types of mobile devices. For this task, accelerometer and gyroscope measurements are classifier inputs, and activities are used as labels.

2. *User identification with biometric motion analysis (UserID):* In this task, we perform user identification with biometric motion analysis. We classify users' identity according to accelerometer and gyroscope measurements. We use the same dataset as in the HHAR task. For this task, accelerometer and gyroscope measurements are classifier inputs, and users' unique IDs are used as labels.
3. *Wi-Fi signal based gesture recognition (Wisture):* In this task, we perform gesture recognition (swipe, push, and pull) with Received Signal Strength Indicator (RSSI) of Wi-Fi signal. We use the dataset collected by Mohamed et al. [77]. This dataset contains labeled Wi-Fi RSSI measurements corresponding to three hand gestures made near a smartphone under different spatial and data traffic scenarios. The Wi-Fi RSSI measurements are classifier inputs, and gestures are used as labels.

For all the following experiments, the HHAR task performs leave-one-user-out cross-validation. We select 1 out of 9 users as the testing dataset with the left as the training dataset. The UserID and Wisture tasks perform 10-fold cross-validation. Each time we choose 10% data as the testing dataset with the left as the training dataset. Then we further divide the training dataset into labelled and unlabelled dataset according to the specification of each experiment.

For all experiments with SenseGAN, we choose DeepSense [2] as the classifier, which is a state-of-the-art neural network structure designed for IoT applications. The generator and the discriminator usually require more training steps to achieve better performance. We therefore set k_d, k_c, and k_g in Algorithm 5.1 to be 5, 1, and 7 respectively without fine-tuning, which consistently achieves decent performance for all three tasks.

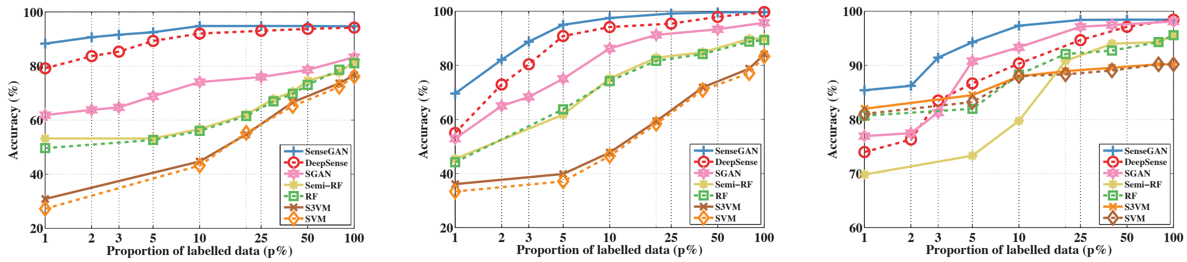
5.2.2 Effectiveness

We first illustrate the effectiveness of SenseGAN at leveraging unlabelled data for IoT applications. We evaluate the performance of SenseGAN with different proportions of labelled and unlabelled data, and compare SenseGAN with supervised deep learning algorithms and other traditional supervised/semi-supervised machine learning algorithms, which are believed to have better predictive performance with smaller datasets. The baseline algorithms are as follows:

1. *DeepSense:* A state-of-the-art supervised neural network structure designed for IoT applications [2], also the classifier used in SenseGAN. DeepSense baseline is chosen to show the performance gain of SenseGAN by leveraging unlabelled data. DeepSense takes the same input as the SenseGAN model, which divides the sensing data into equal-length time interval followed by Fourier transformation.
2. *SGAN:* A general GAN-based semi-supervised deep learning algorithm [78]. The algorithm is not specifically designed for adapting the IoT sensing data. This baseline is chosen to show the importance of design choices in SenseGAN that tailor the GAN training method for IoT applications. SGAN also takes the same input as the SenseGAN model.
3. *RF:* The random forests supervised classifier. The input of random forests is the concatenation of popular time-domain and frequency domain features from [42] and ECDF features [43].
4. *SVM:* The support vector machine supervised classifier, with the same feature selection as the RF model.
5. *Semi-RF:* A semi-supervised learning algorithm that puts a self-training wrapper on the random forest classifier [79]. Semi-RF takes the same input as the RF model.
6. *S3VM:* Semi-supervised SVMs (S3VMs) with the goal of maximizing the margin of unlabelled data for classification [80]. S3VM takes the same input as the SVM model.

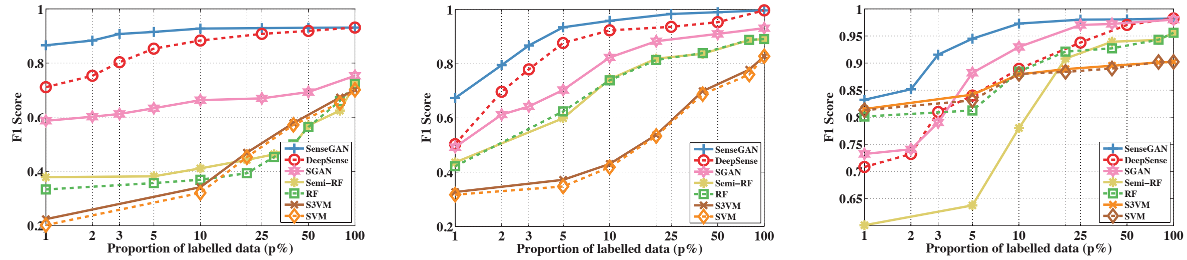
IoT Applications with Different Proportions of Labelled Data

We first conduct a series of experiments on evaluating SenseGAN against baseline algorithms with different proportions of labelled data. The original datasets for tasks are all labelled data samples. During these experiments, we randomly select $p\%$ of training samples as labelled data, and treat the rest training samples as unlabelled data. When $p = 100\%$, SenseGAN stops generating "fake" data. SenseGAN then becomes equivalent to its supervised counterpart, DeepSense. For semi-supervised algorithms, SenseGAN, Semi-RF, and S3VM, all unlabelled data is used for training. For supervised algorithms, DeepSense, RF, and SVM, only labelled data is used for training. Please notice that the testing data never appears in the unlabelled dataset.



(a) The HHAR task. (b) The UserID task. (c) The Wisture task.

Figure 5.5: The accuracy of models with $p\%$ of labelled data for three IoT applications.

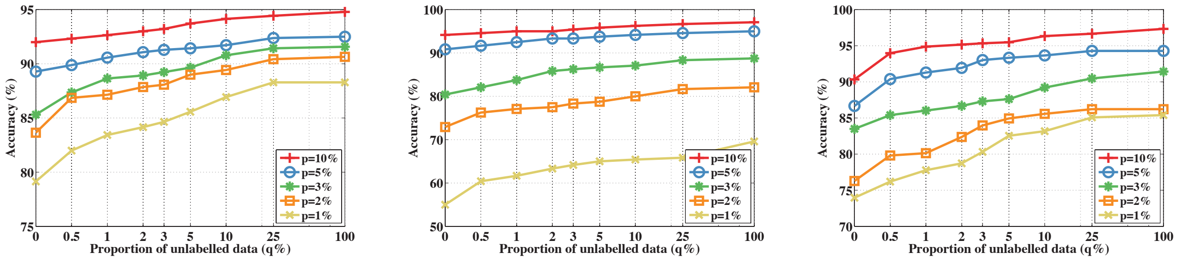


(a) The HHAR task. (b) The UserID task. (c) The Wisture task.

Figure 5.6: The F1 Score of models with $p\%$ of labelled data for three IoT applications.

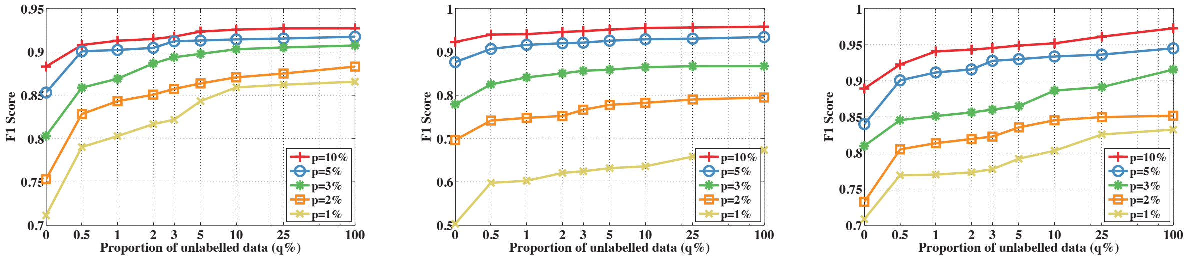
The results of three IoT tasks are shown in Figure 5.5 and 5.6. Two figures are plotted in log scale along the $p-$axis, because we are more interested in the cases with limited labelled data. The performance gains between semi-supervised and supervised algorithms can be small when the proportion of labelled data p is large.

One interesting observation is that two deep learning methods, SenseGAN and DeepSense, perform almost consistently better than the traditional machine learning methods even with tiny proportion of labelled data (with the only exception of DeepSense on the Wisture task). This is mainly attributed to their structures that can effectively handle the multi-modal sensing data in three IoT applications. Performance can be further improved by our proposed SenseGAN framework. As shown in Figure 5.5 and 5.6, SenseGAN shows a significant improvement on both accuracy and F1 score compared with all baseline algorithms. SenseGAN can achieve within 2% and 0.03 drops on accuracy and F1 score with only 10% of labelled data. In our experiments, 10% of labelled data equals around 200, 130, and 30 labelled data samples per category for the HHAR, UserID, and Wisture tasks respectively, which is easily affordable by human labelling. In addition, the existing GAN-based semi-supervised deep learning method, SGAN, consistently shows an inferior performance compared to SenseGAN in all three tasks. These experimental results empirically show the effectiveness of our SenseGAN design choices in general. More ablation study on verifying the design choices of SenseGAN will be shown latter.



(a) The HHAR task. (b) The UserID task. (c) The Wisture task.

Figure 5.7: The accuracy of models with $q\%$ of unlabelled data for three IoT applications.



(a) The HHAR task. (b) The UserID task. (c) The Wisture task.

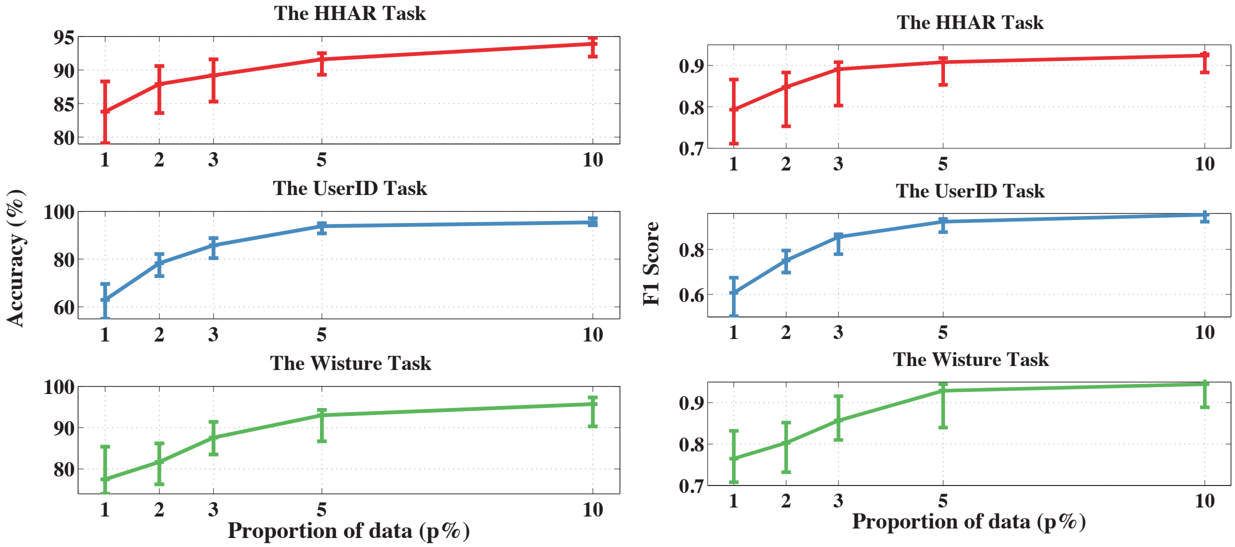
Figure 5.8: The F1 Score of models with $q\%$ of unlabelled data for three IoT applications.

SenseGAN also attains a great improvement compared with its supervised counterpart, DeepSense. However the other two traditional semi-supervised methods, Semi-RF and S3VM, only achieve tiny improvements compared with their supervised counterparts, RF and SVM. Semi-RF even performs worse than RF on the Wisture task. These observations indicate that SenseGAN can effectively leverage the unlabelled multimodal sensor data for improving the complex IoT recognition tasks.

IoT Applications with Different Proportions of Unlabelled Data

We have shown that SenseGAN can utilize labelled data efficiently. It can consistently achieve the best performance with a large margin on three IoT tasks with different proportions of labelled data. However, we have not investigated whether SenseGAN can utilize unlabelled data efficiently. Next, we conduct a series of experiments on performing semi-supervised learning with different proportions of unlabelled data to explore the effect of unlabelled-data sizes on the SenseGAN predictive performance.

For these experiments, we randomly select $p = [1, 2, 3, 5, 10]\%$ of training samples as labelled data, and randomly select $q\%$ of the remaining training samples as unlabelled data, and discard all the rest training samples. We evaluate only SenseGAN here, because Semi-RF and S3VM show little improvement on three IoT tasks even with $q = 100\%$ in Section 5.2.2.



(a) Accuracy.

(b) F1 Score.

Figure 5.9: SenseGAN-LaUL with $p\%$ labelled data as unlabelled data.

When $q = 0\%$, the model is trained with only labelled data, which is equivalent to the DeepSense model. When $q = 100\%$, all experiments are fully semi-supervised, which are the same as those in Section 5.2.2, because all data samples other than $p\%$ of labelled data are used as unlabelled data for training. As shown in Figure 5.7 and 5.8, SenseGAN with $q = 25\%$ can attain almost the same predictive performance compared with $q = 100\%$ on both accuracy and F1 score. Even when the proportion of unlabelled data is tiny, such as $q = 0.5\%$ or 1%, SenseGAN can still achieve decent improvements compared with its supervised counterpart. This indicates that SenseGAN can efficiently use both labelled and unlabelled data to effectively improve the performance of the neural network classifier. Due to the scale of y-axis in Figure 5.7 and 5.8, it seems that, when the proportion of labelled data p is large, increasing the proportion of unlabelled data q can only make limited improvement. However, the unlabelled data does play an important role in SenseGAN from the perspective of reducing predictive error. Take UserID task with $p = 10\%$ as an example, SenseGAN reduces the error of accuracy from $5.8\%(= 100\% - 94.2\%)$ to $2.9\%(= 100\% - 97.1\%)$ and the error of F1 score from $0.077(= 1 - 0.923)$ to $0.041(= 1 - 0.959)$ by increasing q from 0% to 100%, which greatly reduces the errors by around 50%.

IoT Applications with Labelled Data as Unlabelled Data (LaUL)

In a lot of proof-of-concept IoT applications, the total amount of data is limited. Researchers probably want to label all existing samples for training. However, concerns still exist that neural networks may cause performance degradation as shown in Figure 5.5c and 5.6c. In this experiment, we try to fully leverage the limited amount of data samples with SenseGAN by using them as both labelled and unlabelled data. Since the classifier is learned from both labelled supervision and joint interactions with the discriminator and generator in SenseGAN. Using labelled data as unlabelled data can enhance the discriminator and generator, and therefore boost the performance of classifier in return.

During the experiments, we randomly select $p\%$ of original training data as labelled data and discard all the rest training samples. These $p\%$ of data is used as both labelled and unlabelled data for training SenseGAN in the manner just described in Algorithm 5.1.

In Figure 5.9, we plot the results of training SenseGAN with Labelled data as UnLabelled data, called SenseGAN-LaUL, with $p = [1, 2, 3, 5, 10]\%$, each accompanied with a upper and a lower bound. The upper bound is training SenseGAN with all left training samples as unlabelled data (*i.e.*, $q = 100\%$ in Section 5.2.2). The lower bound is the DeepSense model trained on only labelled data (*i.e.*, $q = 0\%$ in Section 5.2.2).

SenseGAN-LaUL attains a considerable improvement on all the three IoT tasks compared with the upper and lower bound. SenseGAN-LaUL can almost always achieve more than 50% of the maximum gain, *i.e.*, the middle point between the upper and lower bound, on three tasks with both accuracy and F1 score. Therefore, SenseGAN-LaUL is a good choice for training IoT applications with extremely limited data. Performance agains show that SenseGAN can efficiently utilize labelled and unlabelled data to effectively improve the predictive performance of neural network classifier on IoT tasks.

Ablation Study for Design Choices

The previous experiments focus on evaluating the performance of SenseGAN for semi-supervised learning on IoT tasks. Recall that we have many design choices within the SenseGAN structure. In this subsection, we evaluate these design choices of SenseGAN against baseline models generated by deleting one design component from the SenseGAN model at a time and measuring the impact. This approach results in the following baselines:

1. *SG-noWGAN*: This model does not train the adversarial game with the Earth mover's distance, defined by Equation (5.3). Instead it uses the original less stable objective function, defined by Equation (5.1).
2. *SG-noGumbel*: This model uses softmax instead of Gumbel-Softmax when feeding the output of classifier into the discriminator, which is discussed in Section 5.1.5.
3. *SG-noPseudo*: This model deletes the pseudo positive data/label tuples $(\mathbf{X}_u^{(2)}, C(\mathbf{X}_u^{(2)}))$ in the training objective function (5.7).

Table 5.2: Different design choices with $p = 10\%$ and $q = 100\%$.

(a) Accuracy.

	SenseGAN	SG-simpG	SG-simpD	SG-rnnD	SG-noGumbel	SG-noWGAN	SG-noPseudo	DeepSense
HHAR	**0.948**	0.932	0.926	0.935	0.918	0.920	0.930	0.920
UserID	**0.971**	0.950	0.946	0.949	0.917	0.916	0.917	0.942
Wisture	**0.973**	**0.973**	**0.973**	0.943	0.943	0.953	0.914	0.903

(b) F1 Score.

	SenseGAN	SG-simpG	SG-simpD	SG-rnnD	SG-noGumbel	SG-noWGAN	SG-noPseudo	DeepSense
HHAR	**0.927**	0.915	0.903	0.894	0.882	0.878	0.913	0.883
UserID	**0.959**	0.941	0.939	0.938	0.911	0.914	0.898	0.923
Wisture	**0.973**	**0.973**	**0.973**	0.939	0.939	0.942	0.884	0.889

4. *SG-simpG*: This model uses a simple structure for the generator. As shown in Figure 5.3, instead of individual deconvolution layers for each sensor, this model uses single but larger deconvolution layers to generate outputs for all sensors.

5. *SG-simpD*: This model uses a simple structure for the discriminator. As shown in Figure 5.4, instead of individual convolution layers for each sensor, this model uses single but larger convolution layers to extract the information from multiple sensor inputs altogether.

6. *SG-rnnD*: Instead of using a full-convolution structure, this model uses the DeepSense structure (with RNN layers) [2] as the discriminator. To best of our knowledge, existing deep learning libraries do not support high-order gradient for RNN-based structure. Therefore, we use weight clipping [24] instead of the gradient penalty [73] for the WGAN training.

The baseline models SG-simpG and SG-simpD are designed such that they each have the same number of parameters as SenseGAN.

We evaluate all baseline algorithms as well as SenseGAN and DeepSense on the three IoT tasks with an illustrating case that randomly selects $p = 10\%$ of training samples as labelled data and regards all the rest as unlabelled data, *i.e.*, $q = 100\%$. SenseGAN and DeepSense work as the “upper bound” and the “lower bound” respectively. All baseline algorithms should perform better than the supervised counterpart by leveraging unlabelled data. SenseGAN should perform better than all baseline algorithms, if all design choices of SenseGAN are reasonable.

Experiment results are shown in Table 5.2. As seen, SenseGAN outperforms all baseline algorithms in all three tasks, providing strong empirical supports for our design choices. The

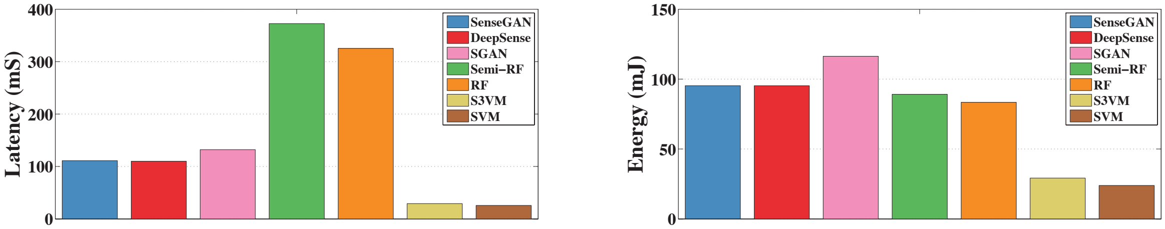
Wisture takes only one sensor as input, so the structure of SenseGAN, SG-simpG, and SG-simpD are identical in this task. These three structures therefore have the same predictive performance for the Wisture task. However, SG-noGumbel and SG-noWGAN can easily perform worse than the supervised counterpart, DeepSense. This indicates that training instability can also affect the final performance of classifier by providing noisy supervision during the adversarial training. SG-simpG and SG-simpD also suffer from performance degradation. Therefore, our specifically designed structures for handling multimodal sensing inputs are crucial to semi-supervised learning for IoT applications. SG-noPseudo also perform worse than the supervised counterpart. Thus pseudo positive data/label tuples are important to prevent discriminator from overfitting when the number of labelled data is limited. In addition, SenseGAN outperforms SG-rnnD in all three tasks, indicating the instability of RNN-based discriminator.

5.2.3 Energy and Time Efficiencies

Finally, we evaluate the energy and time efficiencies of SenseGAN when running on an IoT device, Intel Edison. Intel Edison computing platform is powered by the Intel Atom SoC dual-core CPU at 500 MHz and is equipped with 1GB memory and 4GB flash storage. For fairness, all models are run solely on CPU during experiments. Please notice that only the classifier of SenseGAN need to be loaded and executed on the IoT device during inference. The discriminator and generator only help the classifier to leverage unlabelled data during training, and they can be deleted after the training process.

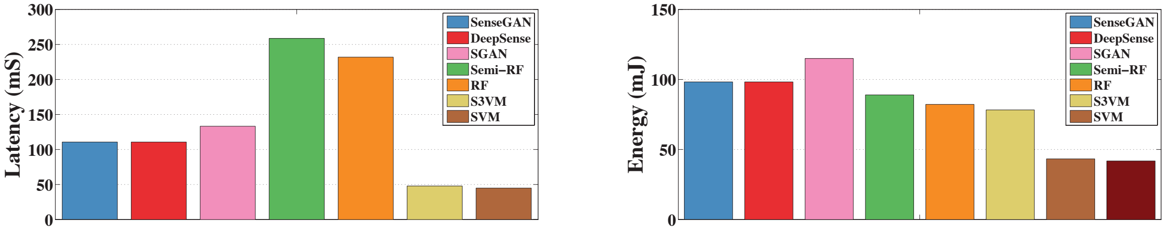
We evaluate SenseGAN and all baseline models used in Section 5.2.2 trained on 10% of labelled data by measuring their per-inference running time and energy consumption. All the measurements are reported by taking the mean of 500 experiments.

The results on three IoT tasks are illustrated in Figure 5.10, 5.11, and 5.12. SenseGAN and its supervised counterpart, DeepSense, have almost the same running time and energy consumption on three tasks, while other semi-supervised models consume a little more time and energy during inference compared their supervised counterparts. This observation highlights the feature of SenseGAN that can leverage unlabelled sensing data for training without additional time and energy consumption during inference on IoT devices. In addition, random forest based algorithms, Semi-RF and RF, take relatively long time to run. It is because random forest models consist of long decision paths that contain a large number of condition operations, which slow down the instruction pipelining in the CPU.



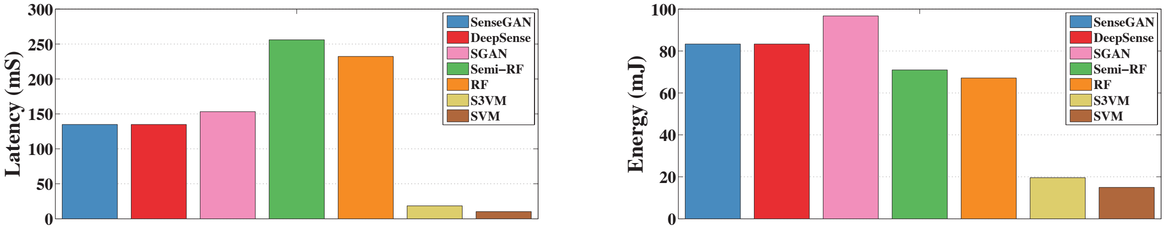
(a) Running time (b) Energy consumption

Figure 5.10: Running time and energy consumption of HHAR



(a) Running time (b) Energy consumption

Figure 5.11: Running time and energy consumption of UserID



(a) Running time (b) Energy consumption

Figure 5.12: Running time and energy consumption of Wisture

5.2.4 Hyper-parameter Tuning

The training process of SenseGAN is controlled by several hyper-parameters. α controls the tradeoff between two types of negative tuples created by the generator and the classifier. γ controls the tradeoff between loss functions generated by the supervised classification error and the adversarial game among three components. λ controls the strength of gradient penalty. In this subsection, we evaluate the final performance of SenseGAN with different choices of these three hyper-parameters on three IoT tasks.

We still evaluate with an illustrating case that randomly selects $p = 10\%$ of training samples as labelled data and regards all the rest as unlabelled data, i.e., $q = 100\%$. The default value of α is 0.5; λ is 10; and γ is 10, 150, and 100 for HHAR, UserID, and

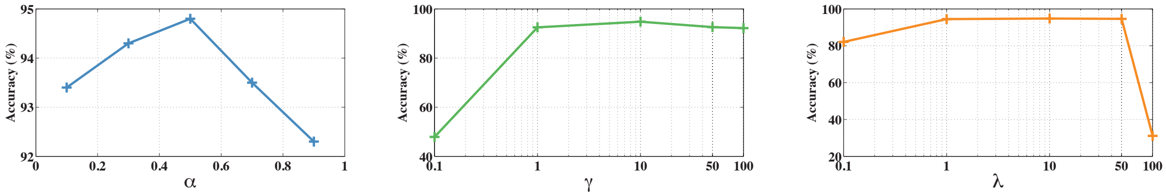


Figure 5.13: Accuracy of HHAR with hyperparameters α, γ, and λ.

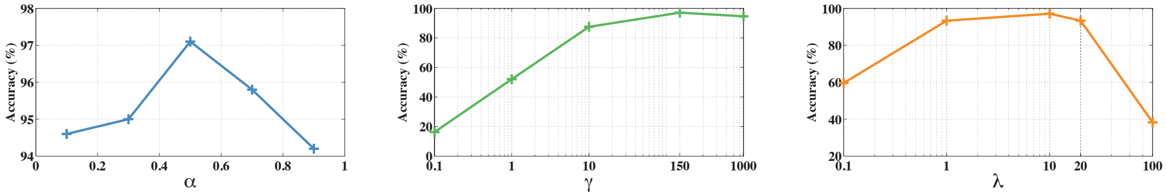


Figure 5.14: Accuracy of UserID with hyperparameters α, γ, and λ.

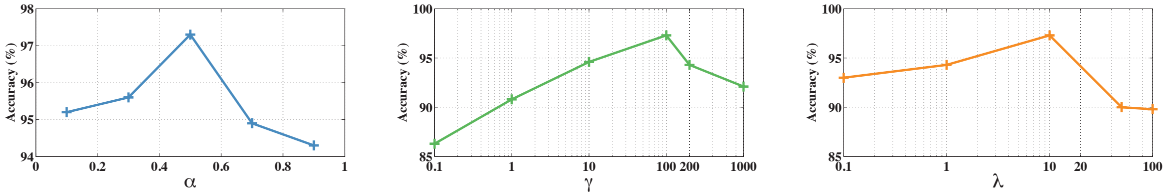


Figure 5.15: Accuracy of Wisture with hyperparameters α, γ, and λ.

Wisture respectively. During each set of experiments, we only change the value of target hyper-parameter, while keeping all other hyper-parameters as the default values. The tradeoffs between the prediction accuracy of the choices of hyper-parameters are shown in Figure 5.13, 5.14, and 5.15.

For α, it achieves the best performance with 0.5 on all three tasks. Therefore, treating two types of negative tuples equally helps to improve the predictive performance of SenseGAN. Similarly, λ consistently achieves the best performance when equals to 10 in all three tasks. These experiments show that fixed fault values work well for α and λ in practice, so we do not have to tune the value of α and λ for different tasks. However, the values of γ are different when achieving the highest accuracy in three tasks. When γ is small, the classifier cannot obtain enough learning signal from the labelled data. When γ is large, the learning signal from the labelled data can overwhelms the learning signal from the adversarial game. Therefore, we need to fine tune the value of γ to balance these two types of learning signals for different tasks.

CHAPTER 6: DEEP LEARNING FOR RELIABLE IOT SYSTEMS

In this section, we first introduce the technical details of the RDeepSense framework. Then, we evaluate RDeepSense with two representative and challenging IoT applications based on uncertainty measuring, accuracy, and resource consumption.

6.1 THE DESIGN OF RDEEPSENSE FRAMEWORK

This section elaborates on the technical details of the RDeepSense framework in three constituents. Section 6.1.1 introduces a simple yet effective recipe to build a fully-connected neural network with predictive uncertainty estimations. In Section 6.1.2, we introduce preliminary knowledge and make the theoretical analysis of RDeepSense. We prove that RDeepSense is a mathematically grounded method to obtain predictive uncertainty estimations. In Section 6.1.3, we introduce an effective and efficient approximation for RDeepSense to obtain predictive uncertainty estimations while running on the resource-constrained embedded devices.

6.1.1 RDeepSense components

RDeepSense is a simple and effective method that empowers fully-connected neural networks to output predictive uncertainty estimations. There are only two steps to convert an arbitrary fully-connected neural networks into a neural network with uncertainty estimations:

1. Insert dropout operation to each fully-connected layer.
2. Adopt a proper scoring rule as the loss function, and emit a distribution estimation instead of a point estimation at the output layer.

The following two parts describe dropout training and proper scoring rules in detail.

Dropout training Fully-connected neural networks can be formulated using the following equations:

$$\begin{aligned} \mathbf{y}^{(l)} &= \mathbf{x}^{(l)}\mathbf{W}^{(l)} + \mathbf{b}^{(l)}, \\ \mathbf{x}^{(l+1)} &= f^{(l)}\big(\mathbf{y}^{(l)}\big), \end{aligned} \tag{6.1}$$

where the notation $l = 1, \cdots, L$ is the layer index in the fully-connected neural network. For any layer l, the weight matrix is denoted as $\mathbf{W}^{(l)} \in \mathbb{R}^{d^{(l-1)} \times d^{(l)}}$; the bias vector is denoted as $\mathbf{b}^{(l)} \in \mathbb{R}^{d^{(l)}}$; the input is denoted as $\mathbf{x}^{(l)} \in \mathbb{R}^{d^{(l-1)}}$; and $d^{(l)}$ is the dimension of the l^{th} layer. In addition, $f^{(l)}(\cdot)$ is a nonlinear activation function.

However, such formulations could run into feature co-adapting and model overfitting problems. To avoid these problems, researchers introduce the concept of dropout as a regularization method [53]. "Dropout" originally refers to dropping out hidden and visible units in a neural network, which is mathematically equivalent to ignoring rows of the weight matrix $\mathbf{W}^{(l)}$. Therefore, a fully-connected neural network with dropout can be represented as follows:

$$\begin{aligned}
\mathbf{z}^{(l)}_{[i]} &\sim \text{Bernoulli}(\mathbf{p}^{(l)}_{[i]}), \\
\tilde{\mathbf{W}}^{(l)} &= \text{diag}\big(\mathbf{z}^{(l)}\big)\mathbf{W}^{(l)}, \\
\mathbf{y}^{(l)} &= \mathbf{x}^{(l)}\tilde{\mathbf{W}}^{(l)} + \mathbf{b}^{(l)}, \\
\mathbf{x}^{(l+1)} &= f^{(l)}\big(\mathbf{y}^{(l)}\big).
\end{aligned} \tag{6.2}$$

As shown in (6.2), a vector of Bernoulli variables $\mathbf{z}^{(l)} \in \{0,1\}^{d^{(l-1)}}$ forms a diagonal matrix which acts as a mask to dropout the i^{th} row of $\tilde{\mathbf{W}}^{(l)}$ with probability $\mathbf{p}^{(l)}_{[i]}$. Intuitively, the dropout operations (4.1) convert a traditional (deterministic) neural network with parameters $\{\mathbf{W}^{(l)}\}$ into a random Bayesian neural network with random variables $\{\tilde{\mathbf{W}}^{(l)}\}$, which equates a neural network with a statistical model without using the Bayesian approach explicitly. This conversion with dropout helps us to obtain predictive uncertainty estimations and avoid the computationally intensive operations used in Bayesian approaches. The detailed analysis about the equivalence will be discussed later.

Proper scoring rules Optimizing a deep neural network requires minimizing the loss function. Therefore the loss function plays a crucial role in designing an effective neural network. Many commonly used neural network loss functions are proper scoring rules, such as logistic loss and hinge loss.

Scoring rules, also known as score functions, measure the quality of predictive uncertainties [81]. Assume that $p_\theta(y|\mathbf{x})$ is the probabilistic distribution represented by a deep neural network. The scoring rule $S(p_\theta(y|\mathbf{x}), (\mathbf{x}, y))$ assigns a numerical score for the quality of predictive distribution $p_\theta(y|\mathbf{x})$ on event $(\mathbf{x}, y) \sim q(\mathbf{x}, y)$, where $q(\mathbf{x}, y)$ is the true distribution of data samples. The expected scoring rule is formulated as

$$S(p_\theta(y|\mathbf{x}), q(\mathbf{x}, y)) = \int q(\mathbf{x}, y) S(p_\theta(y|\mathbf{x}), (\mathbf{x}, y)) d\mathbf{x} dy. \tag{6.3}$$

For a proper scoring rule, the equality in $S(p_\theta(y|\mathbf{x}), q(\mathbf{x}, y)) \geq S(q(\mathbf{x}, y)), q(\mathbf{x}, y))$ holds if and only if $p_\theta(y|\mathbf{x}) = q(\mathbf{x}, y)$. Widely-adopted proper scoring rules include Log-likelihood $\log p_\theta(y|\mathbf{x})$ and Brier score $-\sum_{k=1}^{K}(\mathbb{1}_k(y) - p_\theta(y = k|\mathbf{x}))^2$.

RDeepSense employs a tunable function, the weighted sum of negative log-likelihood and mean square error (Brier score for classification problems), which is a proper scoring rule, as the loss functions for both regression and classification problems. This loss function tries to offset the effect of overestimation and underestimation caused by negative log-likelihood and mean square error respectively, which will be analyzed and evaluated later.

For regression problems, in order to optimize the neural network with negative log-likelihood, we have to emit a distribution estimation instead of a point estimation at the output layer. Therefore, we slightly change the structures of neural networks. The last output layer generates both the predictive mean $\mu(\hat{y})$ and the predictive variance $\sigma^2(\hat{y})$. According to the notation in (6.2), the output layer is represented by $\mathbf{x}^{L+1} = \left[\mu(\hat{y}),\ \sigma^2(\hat{y})\right]^\intercal = \left[\mathbf{y}_{[0]}^{(L)},\ \text{softplus}(\mathbf{y}_{[1]}^{(L)})\right]^\intercal$, where softplus function is $\log(1 + \exp(\cdot))$ enforcing the positivity constraint on the variance. Predictive mean $\mu(\hat{y})$ and predictive variance $\sigma^2(\hat{y})$ compose a Gaussian distribution $\mathcal{N}(\mu(\hat{y}), \sigma^2(\hat{y}))$ as the output predictive distribution of the neural network.

Then the final loss function of a regression problem, $\mathcal{L}_r$, is the weighted sum of mean square error $\mathcal{L}_{re}$ and negative log-likelihood $\mathcal{L}_{rl}$,

$$\begin{aligned}
\mathcal{L}_{re} &= \sum_{n=1}^{N} \big(y - \mu(\hat{y})\big)^2 + \lambda_e \sum_{l=1}^{L} \|\mathbf{W}^{(l)}\|_2^2, \\
\mathcal{L}_{rl} &= \sum_{n=1}^{N} \Big(\frac{1}{2}\log\sigma^2(\hat{y}) + \frac{1}{2\sigma^2(\hat{y})}\big(y - \mu(\hat{y})\big)^2\Big) + \lambda_l \sum_{l=1}^{L} \|\mathbf{W}^{(l)}\|_2^2, \\
\mathcal{L}_r &= (1 - \alpha) \cdot \mathcal{L}_{rl} + \alpha \cdot \mathcal{L}_{re},
\end{aligned} \tag{6.4}$$

where N is the number of training samples, the second term in the first two equations are the L_2 regularization, and α is a hyper-parameter.

As we will discuss in Section 6.1.2 and evaluate in Section 6.2.5, a larger α leads neural networks to focus more on estimating an accurate mean value, which may underestimate the true uncertainties, while a smaller α leads neural networks to estimate a larger variance during the optimization process, which may overestimate the true uncertainties. Therefore, α is a hyper-parameter that makes the bias-variance tradeoff and is tuned to generate a well-calibrated predictive uncertainty, *i.e.*, neither underestimation nor overestimation.

For the classification problem, $f^{(L)}(\cdot)$ is the softmax function that generates predictive probabilities for each category. The final loss function of a classification problem, $\mathcal{L}_c$, is the weighted sum of mean square error $\mathcal{L}_{ce}$ and negative log-likelihood $\mathcal{L}_{cl}$,

$$\begin{aligned}
\mathcal{L}_{ce} &= \sum_{n=1}^{N}\sum_{k=1}^{K}(\mathbb{1}_k(y) - p_\theta(y=k|\mathbf{x}))^2 + \lambda_e \sum_{l=1}^{L} \|\mathbf{W}^{(l)}\|_2^2, \\
\mathcal{L}_{cl} &= \sum_{n=1}^{N} -\log p_{\mathcal{W}}(\hat{y}=y|\mathbf{x}) + \lambda_l \sum_{l=1}^{L} \|\mathbf{W}^{(l)}\|_2^2, \\
\mathcal{L}_c &= (1-\alpha)\cdot\mathcal{L}_{cl} + \alpha\cdot\mathcal{L}_{ce},
\end{aligned} \tag{6.5}$$

where N is the number of training samples, K is the number of classes, the second term in the first two equations are the L_2 regularization, and α is a hyper-parameter.

In summary, the whole neural network is optimized through a tunable proper scoring rule that maximizes the quality of predictive uncertainties. The detailed theoretical backup and proof of the equivalence between RDeepSense and a statistical model will be shown in Section 6.1.2.

6.1.2 The equivalence between RDeepSense and statistical models

Uncertainty estimations are usually inferred by a statistical model, such as a gaussian process [82] and a graphical model [83]. This section provides the theoretical bases for using RDeepSense to estimate predictive uncertainties by proving the equivalence between the RDeepSense model and a statistical model. To achieve this goal, we first summarize the preliminary knowledge about the equivalence between dropout training with mean square error and a deep Gaussian process, which is proposed by Gal et al. [84]. Then we prove the equivalence between dropout with the proper scoring rule (log-likelihood) and a Gaussian or categorical distributions based on latent deep Gaussian process. Finally, we generalize the analysis to another tunable proper scoring rule, weighted sum of log-likelihood and negative mean square error, which provides the theoretical foundation for the RDeepSense.

Preliminary: Dropout with mean square error Gaussian process is a powerful statistical tool that allows us to model distribution over functions [82]. The proof optimize a variational approximation of deep Gaussian process is equivalent to optimizing an dropout neural network based on mean square error as the loss function, which is first discussed and proven by Gal et al. [84].

However, mean square error is not a proper scoring rule for regression problems, which cannot generate a well calibrated uncertainty estimations. Besides, due to the mode matching nature of KL divergence, the variational approximating usually generates a highly underestimated predictive uncertainty [85], which is also verified in our experiments in Section 6.2.4. Therefore we further discuss the case of dropout training with proper scoring rules, which enables RDeepSense to provide a high quality uncertainty estimation.

Dropout with negative log-likelihood We have introduced the previous work that treats a neural network with dropout training based on mean square error loss function as a deep Gaussian process with variational approximation. We call this method MCDrop.

However, there are two drawbacks for MCDrop. One is the underestimation of predictive distribution. Variational Bayesian used in MCDrop is known to provide underestimated posterior uncertainty, because optimizing the KL divergence will generate a low-variance estimation to a single mode of true posterior distribution [85]. In addition, the loss function of MCDrop is not a proper scoring rule that can help to mitigate the negative effect of underestimation caused by the variational Bayesian method. Underestimation is not a desirable property for mobile and ubiquitous computing applications, because it means that the deep neural network will always be over-confident about its prediction results.

The other drawback of MCDrop is the high computational burden during uncertainty estimation. Since the output of MCDrop is a stochastic point estimation, Monte Carlo sampling method is required to estimate the predictive mean and variance. Therefore we need to run the whole neural network for multiple times, *i.e.*, running k times for k samples, to generate the predictive uncertainty. Since running time and energy consumption are two crucial problems for mobile and ubiquitous computing applications, MCDrop is not a suitable solution for applications running on embedded devices.

Therefore, we integrate proper scoring rules and dropout training in RDeepSense to solve the aforementioned two drawbacks. The proper scoring rules such as log-likelihood help to reduce or even erase the underestimation effect of MCDrop, because proper scoring rule is a score function that gives higher quality uncertainty estimations more credits. In addition, since a neural network with proper score rule directly generates a predictive distribution estimation instead of a point estimation, we can efficiently obtain an approximated expectation of uncertainty estimation through dropout inference. At the same time, dropout as Bayesian approximation can provide a equivalence between the deep neural network and a statistical model, which guarantees RDeepSense to be a mathematically grounded uncertainty estimation method.

Readers can refer detailed proof in our original paper [6] that training a fully-connected neural network with dropout and negative log-likelihood loss function is equivalent to a Gaussian or categorical distribution based on the latent deep Gaussian process.

Dropout with weighted sum of negative log-likelihood and mean square error Training a neural network with a proper scoring rule, log-likelihood loss, should generate predictive uncertainty estimations that faithfully reflect the probability that the prediction will happen. However, training a neural network will log-likelihood loss solely could converge to a local optima that overestimates the true uncertainty empirically, which will be shown in our evaluation Section 6.2.4.

The intuitive explanation for this phenomenon is straight-forward. During the early phase of training a neural network with log-likelihood loss, it is relatively hard to generate an accurate estimation of predictive mean. Then increasing the value of variance estimation can consistently decrease the negative log-likelihood loss with a high probability, since there is only a logarithm term that prevents variance from increasing as shown in (6.4). Therefore, the predictive uncertainty tends to favor an estimation with large variance that overestimates the true uncertainty. As a result, although log-likelihood loss is a proper score rule that assigns more credits to predictive uncertainties with higher quality, it usually fails to achieve a good bias-variance tradeoff during training process in practice.

In order to achieve a well-calibrated uncertainty estimation, *i.e.*, an estimation that neither underestimates nor overestimates, we design a tunable proper scoring rule as the training objective function of RDeepSense. It is a weighted sum of log-likelihood and negative mean square error controlled by a hyper-parameter α,

$$(1-\alpha)\cdot\log p_{\mathcal{W}}(\hat{y}=y|\mathbf{x})-\alpha\cdot(\hat{y}-y)^2. \tag{6.6}$$

With the definition in Section 6.1.1, we can easily see that (6.6) is a proper scoring rule.

According to the analysis in the previous two subsections 6.1.2 and 6.1.2, we can see that RDeepSense, training fully-connected neural network by maximizing the weighted sum of log-likelihood and negative mean square error, is equivalent to the mixture distribution of a Gaussian or categorical distribution based on the latent deep Gaussian process and a deep Gaussian process.

Since training solely with negative mean square error or log-likelihood tends to underestimate or overestimate the predictive uncertainties respectively, it is easy to fine-tune the hyper-parameter α with the validation dataset. When the predictive uncertainty is underestimated, we decrease the value α, and vice versa.

6.1.3 RDeepSense uncertainty estimation

The previous sections prove that RDeepSense is a mathematically grounded method to estimate predictive uncertainties for fully-connected neural networks. In this section, we show that RDeepSense can efficiently estimate predictive uncertainties of fully-connected neural networks with only little computational overhead.

According to the analysis in our original paper [6], the approximated predictive distribution is

$$q(\mathbf{y}|\mathbf{x}) = \int p(\mathbf{y}|\mathbf{x}, \mathcal{W})q(\mathcal{W})d\mathcal{W} = \mathbb{E}_{q(\mathcal{W})}\big[p(\mathbf{y}|\mathbf{x}, \mathcal{W})\big], \tag{6.7}$$

where $\mathcal{W} = \{\tilde{\mathbf{W}}^{(l)}\}$ is the random variables generated by dropout operations at each layer.

$$\begin{aligned} \mathbf{z}^{(l)}_{[i]} &\sim \text{Bernoulli}(\mathbf{p}^{(l)}_{[i]}), \\ \tilde{\mathbf{W}}^{(l)} &= \text{diag}\big(\mathbf{z}^{(l)}\big)\mathbf{W}^{(l)}. \end{aligned} \tag{6.8}$$

Usually Monte Carlo estimation is used to approximate the predictive distribution $q(\mathbf{y}|\mathbf{x})$ through sampling random variables $\mathcal{W}$,

$$q(\mathbf{y}|\mathbf{x}) = \frac{1}{M}\sum_{m=1}^{M} p(\mathbf{y}|\mathbf{x}, \mathcal{W}_m). \tag{6.9}$$

For classification, (6.9) is the average of categorical distribution. For regression, (6.9) is an average of Gaussian distributions. If we assume that M Gaussian distributions are independent, the resulted average distribution can be approximated by a single Gaussian distribution according to the central limit theorem,

$$\begin{aligned} \frac{1}{M}\sum_{m=1}^{M} p(\mathbf{y}|\mathbf{x}, \mathcal{W}_m) &= \sum_{m=1}^{M} \mathcal{N}(\mu_m(\mathbf{x}), \sigma_m^2(\mathbf{x})) \\ &= \mathcal{N}(\hat{\mu}(\mathbf{x}), \hat{\sigma}^2(\mathbf{x})), \\ \hat{\mu}(\mathbf{x}) &= \frac{1}{M}\sum_{m=1}^{M} \mu_m(\mathbf{x}), \\ \hat{\sigma}^2(\mathbf{x}) &= \frac{1}{M}\sum_{m=1}^{M} \big(\sigma_m^2(\mathbf{x}) + \mu_m^2(\mathbf{x})\big) - \hat{\mu}^2(\mathbf{x}). \end{aligned} \tag{6.10}$$

The drawback of Monte Carlo estimation for embedded devices is its high energy and time consumptions. We have to run the whole neural network for M times to generate M samples, which is not suitable for embedded devices with limited resources.

Fortunately, there is a simple yet effective recipe proposed by the dropout operation that can effectively approximate the expected output value instead of using Monte Carlo estimation [53]. During test time, the dropout operation is changed from (4.1) into

$$\begin{aligned}\tilde{\mathbf{W}}^{(l)} &= \operatorname{diag}\big(\mathbf{p}^{(l)}\big)\mathbf{W}^{(l)},\\ \mathbf{y}^{(l)} &= \mathbf{x}^{(l)}\tilde{\mathbf{W}}^{(l)} + \mathbf{b}^{(l)},\\ \mathbf{x}^{(l+1)} &= f^{(l)}\big(\mathbf{y}^{(l)}\big).\end{aligned} \tag{6.11}$$

Although the approximation (6.11) is not theoretically equivalent to the Monte Carlo estimation (6.10) by assuming the zero variance of mean estimation, $\sum_{m=1}^{M} \mu_m^2(\mathbf{x}) - (\sum_{m=1}^{M} \mu_m(\mathbf{x}))^2 = 0$, the proposed approximation (6.11) turns to be an effective and efficient approximation during the evaluation in Section 6.2. In the evaluation section, we will empirically compare the biased approximation (6.11) with the unbiased Monte Carlo estimation (6.10).

Therefore, with the approximation (6.11), we can directly estimate the expected predictive mean and variance of a Gaussian distribution for regression problems and expected categorical probabilities for classification problems by just running the neural network for a single time. This makes RDeepSense a suitable candidate for deep neural networks with uncertainty estimations used in mobile and ubiquitous computing applications.

6.2 THE EVALUATION OF RDEEPSENSE

In this section, we evaluate RDeepSense on two mobile and ubiquitous computing tasks. We first introduce the experimental setup for each task, including hardware, datasets, and baseline algorithms. We then evaluate the accuracy and the quality of uncertainty estimation. Next, we evaluate the inference time and energy consumption of all algorithms on the testing hardware. At last we evaluate and analyze the effect of hyper-parameter α in the training objective function (6.6) on the model performance such as accuracy and quality of uncertainty estimation.

6.2.1 Testing hardware

Our testing hardware is based on Intel Edison computing platform [38]. The Intel Edison computing platform is powered by the Intel Atom SoC dual-core CPU at 500 MHz and is equipped with 1GB memory and 4GB flash storage. For fairness, all neural network models are run solely on CPU during evaluation for inference time and energy consumption.

Table 6.1: Statistical Information of four datasets used in evaluations

Dataset	Training Size	Validating Size	Testing Size	Mean of output	Std of output	Range of output
BPEst	1,281,098	26,689	26,689	88.74	25.01	$[50.0, 199.93]$
NYCommute	10,287,766	214,328	214,328	15.08	52.79	$[0.0, 1439.5]$
GasSen	2,839,933	59,166	59,166	94.56	145.16	$[0.0, 533.33]$
HHAR	28,314	1,686	1,686	N/A	N/A	$\{0, 1, 2, 3, 4, 5\}$

6.2.2 Evaluation tasks

We conduct four experiments related to human health and wellbeing, smart city transportation, environment monitoring, and human activity recognition We conduct two experiments on environment monitoring and human activity recognition with RDeepSense and other two state-of-the-art deep learning uncertainty measuring methods as well as a statistical model. The experimental settings of the tasks and datasets are introduced in this subsection.

The detailed statistical information of four datasets is illustrated in Table 6.1

- *BPEst: Cuffless blood pressure monitoring through photoplethysmogram.* The first task is to monitor cuffless blood pressure through photoplethysmogram from fingertip. The dataset is originally collected by patient monitors at various hospitals between 2001 and 2008. Waveform signals were sampled at the frequency of 125 Hz with at least 8 bit accuracy [86]. The photoplethysmogram from fingertip (PPG) and arterial blood pressure (ABP) signal (mmHg) is extracted by Mohamad et al. for the non-invasive cuffless blood pressure monitoring task [87].[1] The target of BPEst task is to infer the waveform of ABP based on the waveform of PPG collected from fingertips. This is a more challenging task compared with estimating the upper and lower bound of the blood pressure, which requires a more precise estimation of predictive uncertainty. During the experiment, a learning model is trained to estimate a 2-second ABP waveform (250 samples) based on the corresponding 2-second PPG waveform.

- *NYCommute: Commute time estimation of New York City.* Smart transportation is an increasingly important task within the topic of smart city. The second task is to estimate commute time in New York City through the pick-up time and location as well as the drop-off location. We use the yellow and green taxi trip records within January 2017 as the training, validation, and testing dataset.[2] The input of the learning model is a vector with 5 elements, containing the standardized longitude and latitude of pick-up and drop-off location as well as the pick-up time within a day. The output of

[1] `https://archive.ics.uci.edu/ml/datasets/Cuff-Less+Blood+Pressure+Estimation`

[2] `http://www.nyc.gov/html/tlc/html/about/trip_record_data.shtml`

the learning model is the expected commute time and its corresponding uncertainty estimation.

- *GasSen: Estimate dynamic gas mixtures from chemical sensors.* The third task is related to the environment monitoring. The task is to estimate real concentration of Ethylene and CO gas mixture from an array of low-end chemical sensors. Fonollosa et al. constructed the dataset by the continuous acquisition the signals of a sensor array with 16 chemical sensors for a duration of about 12 hours without interruption with the sampling frequency of 100 Hz [88]. [3] Gas concentrations range from $0 - 600$ parts-per-million (ppm). The learning model is trained and tested to predict the concentration Ethylene and CO gas mixtures through the vector of 16 sensor inputs.

- *HHAR: Heterogeneous human activity recognition.* This is the same task as we described in Section 3.2.1.

6.2.3 Baseline algorithms

We compare RDeepSense with other two state-of-the-algorithm deep learning uncertainty estimation algorithms, RDeepSense with Monte Carlo estimation, and Gaussian process. The algorithms with deep neural network, including RDeepSense, use the same neural network architecture. It is a 4-layer fully-connected neural network with 500 hidden dimension.

- *MCDrop:* This algorithm is based on the Monte Carlo dropout as described in Section 6.1.2 [84]. Compared with RDeepSense, the main difference is that MCDrop is not optimized by a proper scoring rule. MCDrop requires running the neural network for multiple times to generate samples during uncertainty estimation. Therefore we use MCDrop-k to represent MCDrop with k samples. Multiple samples, *i.e.*, $k > 1$, are required to generate a predictive uncertainty estimation. During the evaluation, we let k to be 3, 5, 10, and 20 to evaluate the tradeoff between the quality of uncertainty estimation and the resource consumption for MCDrop.

- *SSP:* This algorithm trains the neural network with proper scoring methods and uses the ensemble method [89]. Compared with RDeepSense, the main difference is that SSP uses the ensemble method instead of the dropout operation in each layer. SSP requires training multiple neural networks for ensemble. Therefore we use SSP-k to represent SSP by ensemble k individual neural networks. During the evaluation, we

[3] https://archive.ics.uci.edu/ml/datasets/Gas+sensor+array+under+dynamic+gas+mixtures

let k to be 1, 3, 5, and 10 to evaluate the tradeoff between the quality of uncertainty estimation and the resource consumption for SSP.

- *RDeepSense-MC:* This algorithm is basically the proposed RDeepSense algorithm. The difference is that, during the inference, RDeepSense-MC uses Monte Carlo estimation (6.10) instead of the efficient approximation (6.11) for uncertainty estimation. Therefore we use RDeepSense-MCk to present RDeepSense-MC with k samples. During the evaluation, we let k to be 3, 5, 10, and 20 to evaluate the effectiveness and efficiency of RDeepSense inference approximation (6.11) compared with the Monte Carlo estimation (6.10).

- *GP:* Gaussian process (GP) is the baseline algorithm used during the evaluation of accuracy and the quality of uncertainty estimations, but not for the evaluations of running time and energy consumption on Edison. The main reason is that the computation cost during model inference for GP is $O(N^3)$, where N is the number of data instances. This cost can be prohibitive even for moderately sized datasets on embedded devices, such as Intel Edison. In additional, GP requires $O(N^2)$ memory consumption during training. Therefore we train the GP with only a proportion of dataset on a server with 128GB memory. Notice that GP is the baseline used to illustrate the quality of uncertainty estimations generated by a statistical model, so the size of training dataset is not the main concern.

6.2.4 Accuracy of prediction and quality of uncertainty estimations

In this section, we discuss the accuracy and the uncertainty estimation quality of RDeepSense compared with the other baseline algorithms. RDeepSense is tuned with the validating dataset, and all algorithms in all experiments are tested on the testing dataset.

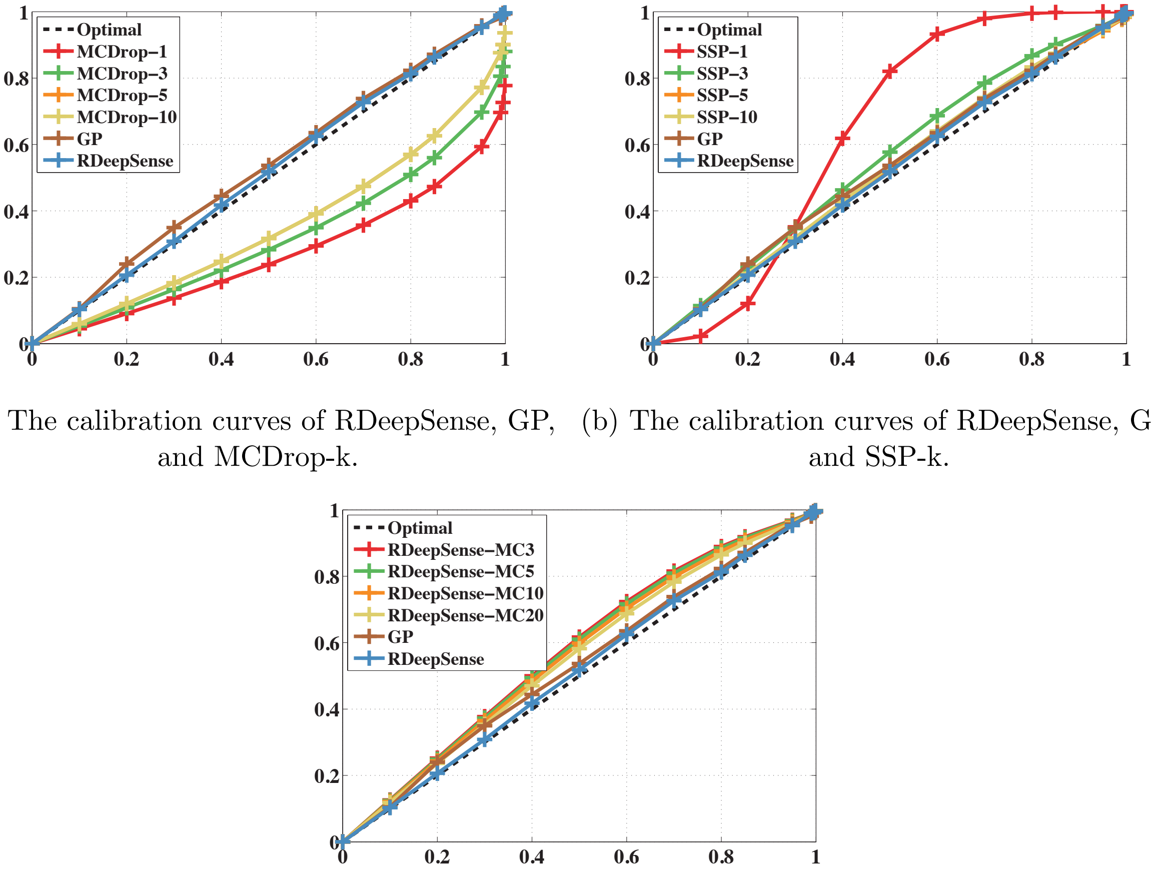
For the regression problem, two types of evaluation results will be illustrated and discussed. The first type of evaluation is based on some basic measurements including mean absolute error and negative log-likelihood. The second type of evaluation is based on the calibration curves, also known as reliability diagrams. We compute the $z\%$ confidence interval for each testing data based on predictive mean and variance of each algorithm. Then we measure the fraction of the testing data that falls into this confidence interval. For a well-calibrated uncertainty estimation, the fraction of testing data that falls into the confidence interval should be similar to $z\%$. We compute the calibration curves with $z = [10\%, 20\%, 30\%, 40\%, 50\%, 60\%, 70\%, 80\%, 85\%, 95\%, 99\%, 99.5\%, 99.9\%]$ for all three regression problems.

Table 6.2: Mean Absolute Error (MAE) and Negative Log-Likelihood (NLL) for the BPEst task. Except for RDeepSense-MC20, RDeepSense is the best-performing algorithm for NLL and is the second best-performing algorithm for MAE.

	RDeepSense	RDeepSense-MC3	RDeepSense-MC5	RDeepSense-MC10	RDeepSense-MC20
MAE	**14.18**	14.93	14.64	14.44	14.32
NLL	**3.46**	3.49	3.47	3.46	**3.45**

	SSP-1	SSP-3	SSP-5	SSP-10	GP
MAE	15.76	14.68	14.67	14.78	19.15
NLL	4.4	3.69	3.48	3.49	3.59

	MCDrop-3	MCDrop-5	MCDrop-10	MCDrop-20	
MAE	14.80	14.41	**14.09**	**14.09**	
NLL	38.1	5.28	4.00	4.00	



(a) The calibration curves of RDeepSense, GP, and MCDrop-k.

(b) The calibration curves of RDeepSense, GP, and SSP-k.

(c) The calibration curves of RDeepSense, GP, and RDeepSense-MCk.

Figure 6.1: The calibration curves of BPEst for RDeepSense, GP, MCDrop-k, SSP-k, and RDeepSense-MCk. MCDrop-k underestimates the predictive distribution. SSP-k overestimates the predictive distribution. RDeepSense is the closest curve to the optimal predictive distribution.

For the classification problem, the calibration curve is not available. Therefore, we evaluate HHAR based on accuracy, F1 Score, negative log-likelihood, and a new measurement called the mean entropy of false predictions. If the entropies of false predictions are higher, the learning algorithms show more uncertainties about the false predictions, which represents a better quality of uncertainty estimations.

BPEst

We first compare RDeepSense with four baseline algorithms based on mean absolute error (MAE) and negative log-likelihood (NLL), which is illustrated in Table 6.2, where we highlight the results of RDeepSense and the best-performing one.

From Table 6.2, we can see that, except for RDeepSense-MC20, RDeepSense is the best-performing and the second best-performing algorithm for NLL and MAE respectively, which means that RDeepSense can provide accurate estimation with high-quality predictive uncertainty. RDeepSense-MC20 only slightly beats RDeepSense on NLL, however RDeepSense-MC20 consumes around ×20 time and energy compared with RDeepSense. The performance of MCDrop-k increases when k increases. Larger k means that MCDrop algorithm generates more samples during model inference, which can provide higher-quality estimations but more resource consumptions. MCDrop-3 provides a relatively bad result for NLL, which means MCDrop does require a number of samples for uncertainty estimation with reasonable quality. The ensemble method used in SSP increases the prediction performance, but it is not consistent. SSP-10 observes the performance degradation compared with SSP-5. GP obtains a relatively large MAE. This is because GP cannot be scaled to train on the whole dataset.

The calibration curves of BPEst task is illustrated in Figure 6.1. These three figures show the quality of predictive uncertainty estimations. RDeepSense generates predictive uncertainties with the highest quality. RDeepSense even slightly out-performs the traditional statistical model, GP. As we mentioned in Section 6.1.2, MCDrop-k tends to underestimate the predictive uncertainty, while SSP-k tends to overestimate the predictive uncertainty. RDeepSense even generates predictive uncertainty with better calibration compared with RDeepSense-MCk, which indicate the effectiveness of approximation during inference. All MCDrop-k, SSP-k, and RDeepSense-MCk improve the quality of uncertainty estimations by increasing the value of k.

NYCommute

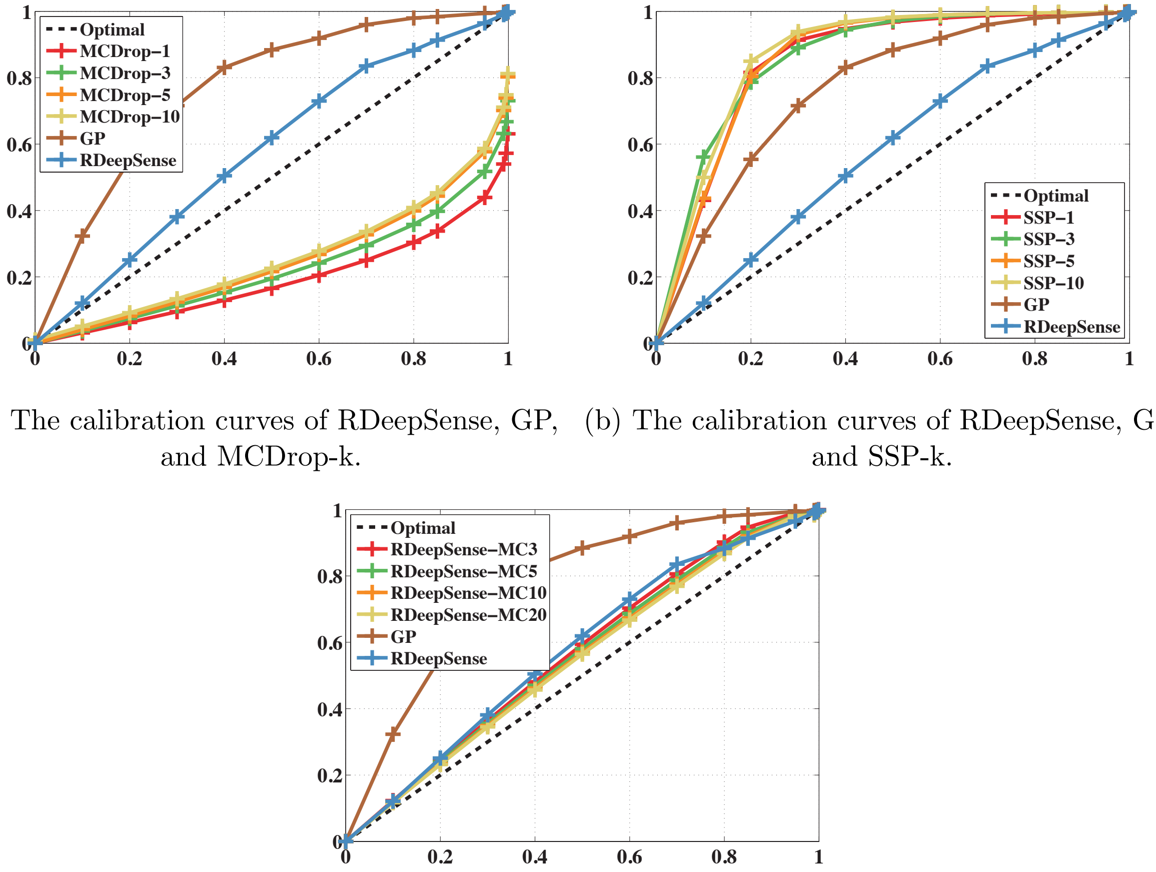
Then we compare RDeepSene with baseline algorithms for NYCommute task. The comparison based on Mean Absolute Error (MAE) and Negative Log-Likelihood (NLL) is shown in Table 6.3.

Table 6.3: Mean Absolute Error (MAE) and Negative Log-Likelihood (NLL) for the NYCommute task.

	RDeepSense	RDeepSense-MC3	RDeepSense-MC5	RDeepSense-MC10	RDeepSense-MC20
MAE	**5.64**	6.10	6.04	5.99	5.96
NLL	**7.7**	7.85	7.81	7.73	**7.7**

	SSP-1	SSP-3	SSP-5	SSP-10	GP
MAE	8.15	7.90	7.51	7.03	11.84
NLL	4.86	**4.67**	4.84	4.81	7.46

	MCDrop-3	MCDrop-5	MCDrop-10	MCDrop-20	
MAE	5.69	5.64	**5.61**	**5.61**	
NLL	19995.6	1335.73	640.35	640.35	



(a) The calibration curves of RDeepSense, GP, and MCDrop-k.

(b) The calibration curves of RDeepSense, GP, and SSP-k.

(c) The calibration curves of RDeepSense, GP, and RDeepSense-MCk.

Figure 6.2: The calibration curves of NYCommute for RDeepSense, GP, MCDrop-k, SSP-k, and RDeepSense-MCk. MCDrop-k highly underestimates the predictive distribution. SSP-k highly overestimates the predictive distribution. RDeepSense makes a tradeoff between these two and is the closest curve to the optimal predictive distribution.

In this task, RDeepSense tends to find a balance between MAE and NLL measurements. MCDrop-k shows low MAE and high NLL, while SSP-k shows high MAE and low NLL. MCDrop-k tries to minimize the mean square error, while SSP-k tries to minimize the negative log-likelihood. Therefore, MCDrop-k focuses more on the mean of predictive distribution, and SSP-k focuses more on the overall likelihood. RDeepSense combines two objective functions, mean square error and negative log-likelihood, which tries to find a balance point between these two. Still, due to the scalability problem, GP obtains a relatively larger MAE. Compared with RDeepSense-MCk, RDeepSense achieve a good performance on both MAE and NLL. Only RDeepSense-MC20 shows the same performance on the NLL measurement.

The calibration curves of NYCommute task is illustrated in Figure 6.2. Both MCDrop-k and SSP-k fail to generate high-quality uncertainty estimations by either underestimating or overestimating the predictive uncertainties. However, RDeepSense can still provide uncertainty estimations with good quality, which outperforms GP with a significant margin. Compared with RDeepSense-MCk, RDeepSense shows similar performance on generating well-calibrated predictive uncertainties, which shows that the approximation (6.11) works well in practice.

GasSen

Next we compare RDeepSense with other baseline algorithms for the GasSen task. Table 6.4 illustrates the performance of all these algorithms based on Mean Absolute Error (MAE) and Negative Log-Likelihood (NLL). Except for RDeepSense-MC20, RDeepSense is the best-performing algorithm according to these two metrics. Similarly, MCDrop-k shows low MAE and NLL, while SSP-k shows high MAE and NLL. This is due to the objective of these two types of algorithms. MCDrop-k minimizes the mean square error, while SSP-k minimizes the negative log-likelihood. Therefore, MCDrop-k focuses more on the mean of predictive distribution, and SSP-k focuses more on the overall likelihood. RDeepSense combines two objective function. Therefore, RDeepSense is able to achieve the best performance in both cases. The usage of dropout that prevents feature co-adapting is the main reason why RDeepSense achieves better NLL compared with SPP-k. The RDeepSense still achieves good performance compared with its Motel Carlo version. Only RDeepSense-MC20 slightly outperforms RDeepSense under the NLL measurement, which shows the effectiveness of the approximation used in RDeepSense.

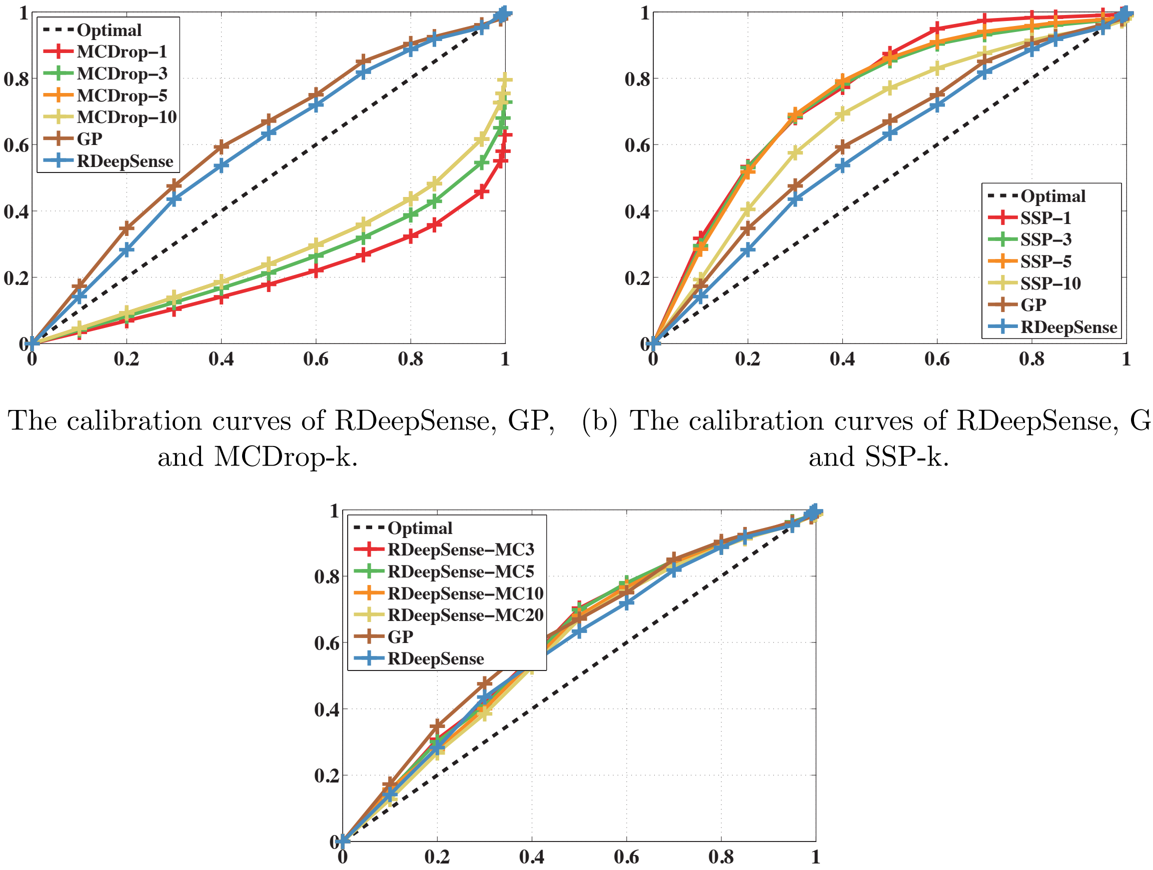
The calibration curves of GasSen task is illustrated in Figure 6.3. The calibration curves of MCDrop-k highly underestimates the predictive distribution as shown in Figure 6.3a, while the calibration curves of SSP-k highly overestimates the predictive distribution as shown

Table 6.4: Mean Absolute Error (MAE) and Negative Log-Likelihood (NLL) for the GasSen task. Except for RDeepSense-MC20, RDeepSense is the best-performing algorithm for both MAE and NLL.

	RDeepSense	RDeepSense-MC3	RDeepSense-MC5	RDeepSense-MC10	RDeepSense-MC20
MAE	**15.25**	17.21	16.44	16.34	15.61
NLL	**3.77**	4.23	4.18	3.88	**3.73**

	SSP-1	SSP-3	SSP-5	SSP-10	GP
MAE	24.40	22.53	20.75	20.68	35.74
NLL	4.76	4.34	3.92	3.81	7.76

	MCDrop-3	MCDrop-5	MCDrop-10	MCDrop-20	
MAE	21.23	20.45	19.79	19.79	
NLL	2201.95	463.94	170.45	170.45	



(a) The calibration curves of RDeepSense, GP, and MCDrop-k.

(b) The calibration curves of RDeepSense, GP, and SSP-k.

(c) The calibration curves of RDeepSense, GP, and RDeepSense-MCk.

Figure 6.3: The calibration curves of GasSen for RDeepSense, GP, MCDrop-k, SSP-k, and RDeepSense-MCk. MCDrop-k highly underestimates the predictive distribution. SSP-k highly overestimates the predictive distribution. RDeepSense is the closest curve to the optimal predictive distribution.

Table 6.5: Accuracy (Acc), Negative Log-Likelihood (NLL), Mean Entropy of False Predictions (MEFP) for the HHAR task. RDeepSense is the best-performing algorithm according to all measures.

	RDeepSense	RDeepSense-MC3	RDeepSense-MC5	RDeepSense-MC10	RDeepSense-MC20
Acc	**83.98**%	80.66%	83.07%	83.08%	83.85%
F1 Score	**0.670**	0.601	0.638	0.668	**0.671**
NLL	**0.161**	0.193	0.188	0.172	**0.159**
MEFP	**1.715**	1.604	1.621	1.626	1.628

	SSP-1	SSP-3	SSP-5	SSP-10	GP
Acc	77.15%	78.34%	79.30%	80.30%	77.29%
F1 Score	0.650	0.652	0.657	0.661	0.659
NLL	1.138	1.188	1.165	1.214	0.807
MEFP	1.619	1.629	1.672	1.708	1.218

	MCDrop-3	MCDrop-5	MCDrop-10	MCDrop-20	
Acc	79.53%	79.73%	79.73%	80.51%	
F1 Score	0.586	0.589	0.589	0.593	
NLL	0.166	0.163	0.162	**0.161**	
MEFP	0.501	0.548	0.574	0.579	

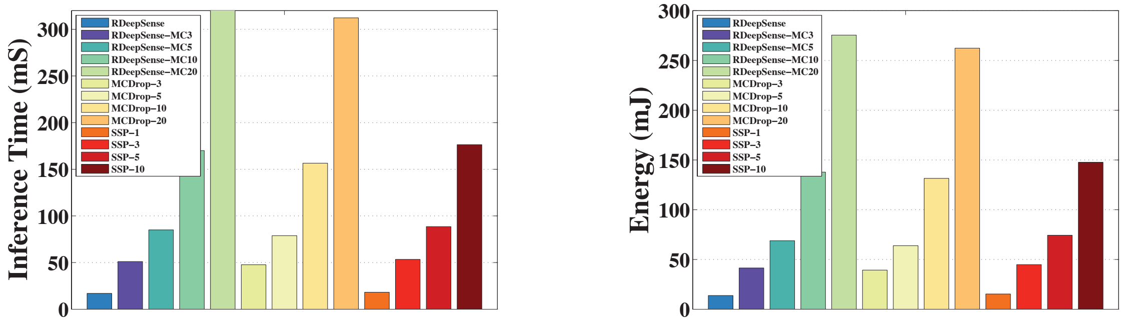
in Figure 6.3b. Although there exists a bit deviation for RDeepSense compared with the optimal calibration curve, RDeepSense greatly reduces the effect of underestimation and overestimation, and slightly outperforms the traditional statistical model, GP. Compared with unbiased RDeepSense-MCk, RDeepSense shows the similar performance. However, RDeepSense saves save a great amount of energy and time consumption as we will discuss in Section 6.2.5.

HHAR

Last we compare RDeepSense with the other baseline algorithm for the HHAR task. Table 6.5 illustrates the performance metrics of all algorithms based on Accuracy (Acc), F1 Score (F1 Score), Negative Log-Likelihood (NLL), and Mean Entropy of False Predictions (MEFP).

Except for RDeepSense-MC20, RDeepSense is the best-performing algorithm according to all measures, which means RDeepSense can provide both high prediction accuracy as well as high quality of uncertainty estimations. MCDrop-k algorithms are trained with log-likelihood. Therefore they try to minimize the negative log-likelihood, but they are over-confident about their prediction even when they make some wrong predictions according to the MEFP measure. SSP-k algorithms are trained with Brier score. Therefore they fall short to achieve smaller NLL values. Compared with RDeepSense-MCk algorithms, RDeepSense still provides a good performance in all measurements. Only RDeepSense-MC20 shows a superior performance on F1 Score and NLL measurements.

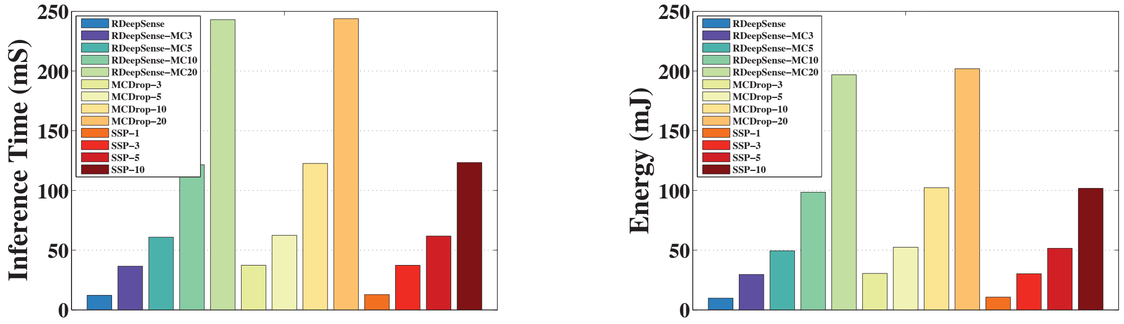
6.2.5 Inference time and energy consumption



(a) The inference time of RDeepSense, RDeepSense-MCk, MCDrop-k, and SSP-k for BPEst.

(b) The energy consumption of RDeepSense, RDeepSense-MCk,MCDrop-k, and SSP-k for BPEst.

Figure 6.4: The inference time and energy consumption of RDeepSense, RDeepSense-MCk, MCDrop-k, and SSP-k for BPEst.

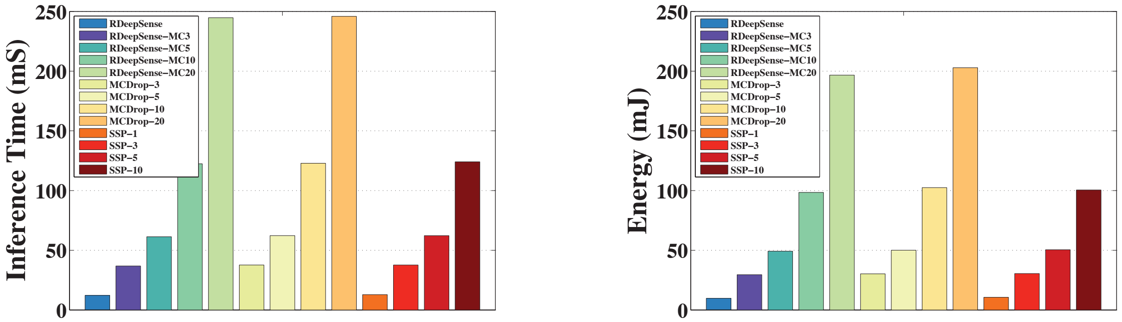


(a) The inference time of RDeepSense, RDeepSense-MCk, MCDrop-k, and SSP-k for NYCommute.

(b) The energy consumption of RDeepSense, RDeepSense-MCk, MCDrop-k, and SSP-k for NYCommute.

Figure 6.5: The inference time and energy consumption of RDeepSense, RDeepSense-MCk, MCDrop-k, and SSP-k for NYCommute.

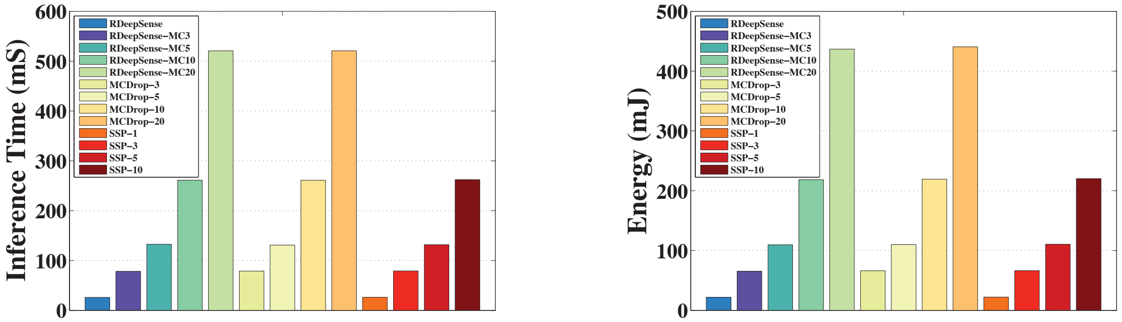
We compared the resource consumption of each algorithm including inference time and energy consumption of one-data-sample execution, which are two key issues for mobile and ubiquitous computing. All the experiments are conducted on Intel Edison with only CPU as the computing unit. No further optimization is made on any algorithms. The inference time and energy consumption of GP are not included. This is because the time complexity of GP is $O(N^3)$, where N is the size of training dataset, which is infeasible for embedded



(a) The inference time of RDeepSense, RDeepSense-MCk, MCDrop-k, and SSP-k for GasSen.

(b) The energy consumption of RDeepSense, RDeepSense-MCk, MCDrop-k, and SSP-k for GasSen.

Figure 6.6: The inference time and energy consumption of RDeepSense, RDeepSense-MCk, MCDrop-k, and SSP-k for GasSen.



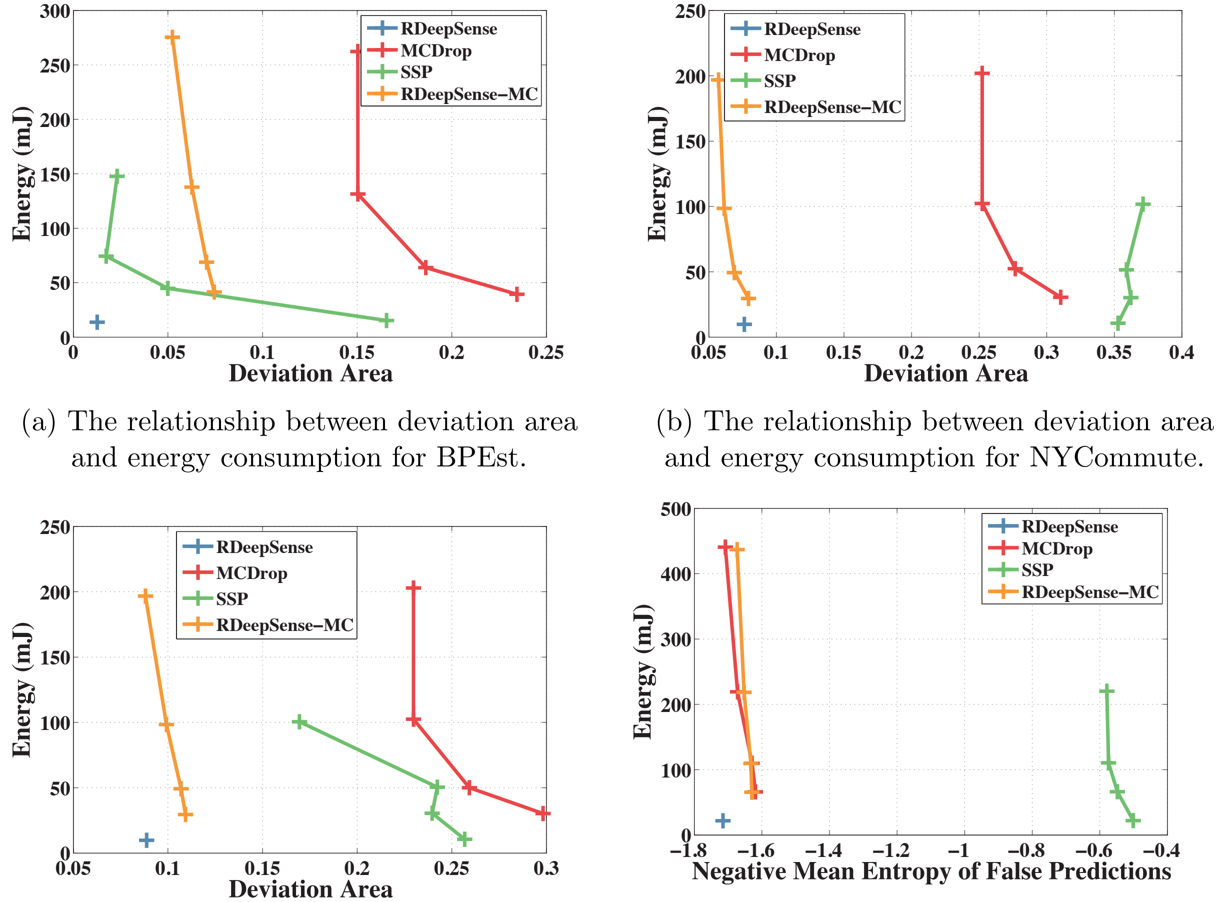
(a) The inference time of RDeepSense, RDeepSense-MCk, MCDrop-k, and SSP-k for HHAR.

(b) The energy consumption of RDeepSense, RDeepSense-MCk, MCDrop-k, and SSP-k for HHAR.

Figure 6.7: The inference time and energy consumption of RDeepSense, RDeepSense-MCk, MCDrop-k, and SSP-k for HHAR.

devices such as Intel Edison. The results of four tasks, *i.e.*, BPEst, NYCommute, GasSen, and HHAR, are illustrated in Figures 6.4, 6.5, 6.6, and 6.7 respectively.

We can clearly see that RDeepSense greatly reduces the inference time and energy consumption compared with the other deep learning uncertainty estimation algorithms. Compared with MCDrop algorithm, RDeepSense is trained according to the proper scoring rule, which can directly output the predictive distribution instead of using sampling methods. Compared with SSP algorithm, RDeepSense uses dropout regularization as an implicit ensemble method, which avoids running multiple deep learning models during model inference on embedded devices. Compared with RDeepSense-MC, RDeepSense use the approximation (6.11) to replace the computationally intensive Motel Carlo method (6.10).



(a) The relationship between deviation area and energy consumption for BPEst.

(b) The relationship between deviation area and energy consumption for NYCommute.

(c) The relationship between deviation area and energy consumption for GasSen.

(d) The relationship between negative mean entropy of false predictions and energy consumption for HHAR.

Figure 6.8: The relationship between deviation area/negative mean entropy of false predictions and energy consumption of all algorithms. RDeepSense (in the bottom-left corner) is the best-performing algorithm that uses the least energy to achieve the best uncertainty estimation quality

We further analyze the relationship between energy consumption and the quality of uncertainty estimation for each algorithms. For regression problems, we use the area between the calibration curve of an algorithm and the optimal calibration curve, called deviation area, as the quality measurement of uncertainty. The smaller deviation area is, the better quality of uncertainty the algorithm estimates. When the calibration curve of an algorithm is optimal, the deviation area is 0. For classification problems, we use the negative mean entropy of false predictions as the quality measurement of uncertainty. Smaller negative mean entropy of false predictions means is that the algorithm is more uncertain about their false predictions. The result is shown in Figure 6.8.

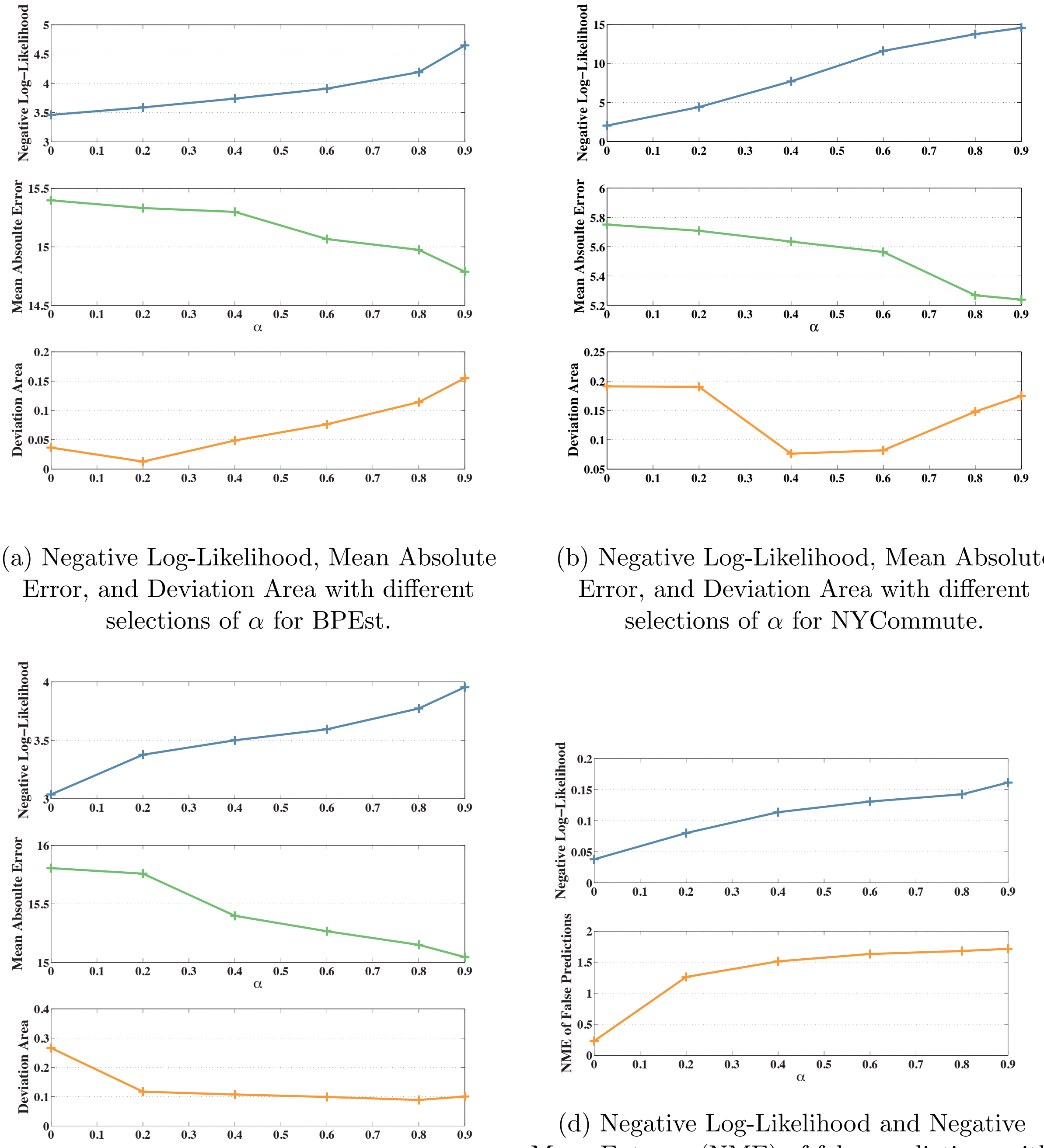
The point or line stay in the bottom-left corner of the graph represents a better tradeoff between energy and uncertainty quality, *i.e.*, using less energy to obtain better uncertainty estimations. Therefore, RDeepSense is the best-performing algorithm that uses the least amount of energy to obtain the best uncertainty estimation quality. RDeepSense-MC can achieve similar uncertainty estimation quality as RDeepSense, however it requires much more energy consumption. The results show that RDeepSense is an effective and efficient uncertainty estimation algorithm (6.11) compared with its Monte Carlo version (6.10). Other two baseline algorithms, MCDrop and SSP, usually suffer a large deviation area or become overconfidence about their false predictions while using more energy for computation. Figure 6.8 shows that RDeepSense is the most suitable algorithm for generate predictive uncertainty estimations for mobile and ubiquitous computing application on embedded devices.

Effect of hyper-parameter α on model performance

The hyper-parameter α controls the tradeoff between optimization of mean and variance within the training objective function (6.6) that can help to obtain a well-calibrated uncertainty estimation. In this subsection, we evaluate the functionality of α and also shed light on the way of tuning α.

For each task, we train RDeepSense with $\alpha = [0, 0.2, 0.4, 0.6, 0.8, 0.9]$. When $\alpha = 0.0$, RDeepSense is trained by minimizing the negative log-likelihood. When we increase the value of α, RDeepSense focuses more on the mean value estimation instead of the negative log-likelihood. In order to show the effect of the choice of α on the quality of predictive uncertainty estimation, we show the negative log-likelihood and devision area (the area between the calibration curve of an algorithm and the optimal calibration curve) for regression tasks and show the negative log-likelihood and Negative Mean Entropy (NME) of false predictions for the classification task in Figure 6.9.

A good uncertainty estimation should faithfully reflect the probability that prediction will happen. Therefore, RDeepSense targets on a well-calibrated uncertainty estimation, such as the prediction with low devision area, in stead of the prediction with low negative log-likelihood. From Figure 6.9a, 6.9b, and 6.9c, we can see that hyper-parameter α controls the tradeoff between optimization mean and variance within the training objective function (6.6). Smaller α tends to reduce negative log-likelihood by increasing the predictive variance, which tends to result the overestimation of predictive uncertainties. Larger α tends to reduce negative log-likelihood by predicting a better mean value, which tends to result the underestimation of predictive uncertainties. When tuning the hyper-parameter α, we can easily found a point that achieve the smallest devision area by grid searching α from 0 to 1. At the same time, it is not surprising that increasing α can slightly increase



(a) Negative Log-Likelihood, Mean Absolute Error, and Deviation Area with different selections of α for BPEst.

(b) Negative Log-Likelihood, Mean Absolute Error, and Deviation Area with different selections of α for NYCommute.

(c) Negative Log-Likelihood, Mean Absolute Error, and Deviation Area with different selections of α for GasSen.

(d) Negative Log-Likelihood and Negative Mean Entropy (NME) of false predictions with different selections of α for HHAR.

Figure 6.9: Negative Log-Likelihood, Mean Absolute Error, and Deviation Area/Negative Mean Entropy (NME) of false predictions with different selections of α for four tasks.

the negative log-likelihood, since $\alpha = 0$ represents regarding negative log-likelihood as the objective function. In addition, Figure 6.9d shows that increasing α can consistently increase the negative mean entropy of false predictions.

CHAPTER 7: DEEP LEARNING SERVICES ON EDGE/CLOUD

In this section, we first introduce the technical details of the RTDeepSense framework. Then, we implement a user space scheduling framework to verify the effectiveness of RTDeepIoT.

7.1 THE DESIGN OF RTDEEPIOT

In this section, we introduce 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. We keep the model in this section relatively abstract, and defer to Section 7.2 the technical details of applying this model in practice to an actual service prototype. We also ignore a few complexities first, such as the manner in which we estimate accurate utility values. Those will be discussed later, when we map the abstract model 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. We assume that communication with the server is not the bottleneck. Later, in the evaluation, we indeed show that this is the case (due to the comparatively heavier computational demand of deep neural network processing). Hence, below we model 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. We define the set $\mathcal{T}_i^{(1:l)}$ 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 we discuss later, our system uses EDF as the underlying scheduling algorithm, which is optimal for the uniprocessor scheduling model.

We consider diminishing-return utility curves, 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 [26]. The formal definition of diminishing-return utility is as follows:

Definition 7.1. *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 $l_1 > l_2$.*

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

Definition 7.2. *For n stage-wise programs, we denote the set of all stages of all programs 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 we select two T_i 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 (we use 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. We can now formulate the problem as:

$$\begin{aligned} \max_{\mathcal{S} \subseteq \mathcal{T}} \quad & U(\mathcal{S}) = \sum_{i=1}^{n} u_i^{(h_i)} \\ s.t. \quad & t_i^{(h_i)} < d_i \end{aligned} \tag{7.1}$$

As shown in Definition 7.1, the utility curve of deep learning execution stages follow the diminishing-return property. The overall utility of scheduling set, therefore, diminishes as we add more scheduling stages. We express that more formally as Lemma 7.1:

Lemma 7.1. *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 $U(\mathcal{S}) \leq U(\mathcal{W})$.*

Algorithm 7.1. 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:     $v* \leftarrow \arg\max_{v \in \mathcal{G}} \mathcal{U}(\mathcal{S}_G \cup v) - \mathcal{U}(\mathcal{S}_G)$
 5:     $\mathcal{G} \leftarrow \mathcal{G} \backslash \{v*\}$
 6:     if $\mathcal{S}_G \cup \{v*\}$ can be scheduled under deadlines $\{\tau_i\}$ then
 7:         $\mathcal{S} \leftarrow \mathcal{S}_G \cup \{v*\}$
 8:     end if
 9: end while
10: return $\mathcal{S}_G$
```

Without loss of generality, we assume $v = T_n$. According to the diminishing-return utility defined in *Definition 7.1*, $U(\mathcal{S} \cup v) - U(\mathcal{S}) \geq U(\mathcal{W} \cup v) - U(\mathcal{W})$. In addition, since the utility is positive and non-decreasing, $U(\mathcal{S}) \leq U(\mathcal{W})$.

The submodularity of objective set function motivates us to solve the scheduling problem (7.1) 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, we first design a greedy algorithm to our scheduling problem (7.1) and then analyze its performance accordingly.

The greedy algorithm for our scheduling problem is shown in Algorithm 7.1. 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 we show later, is the estimated confidence in results), and deletes $v*$ from the candidate set $\mathcal{G}$. Then, we verify whether the new $\mathcal{S}_G$ set is still feasible, 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. We can thus 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 [90] are one of these structures:

Definition 7.3. *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 $\mathcal{S} \subseteq \mathcal{T}$, we assume 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. We can formally define it 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 7.4. *An independence system* $(\mathcal{T}, \mathcal{I})$ *is called a p-independence system if for all* $\mathcal{S} \subseteq \mathcal{I}$*,*

$$\frac{\max_{\mathcal{S}' \in B(\mathcal{S})} |\mathcal{S}'|}{\min_{\mathcal{S}'' \in B(\mathcal{S})} |\mathcal{S}''|} \leq p, \tag{7.2}$$

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

Now we can easily obtain that the feasible solutions of our scheduling problem (7.1) form an independence system. Firstly, the empty set is schedulable, which is a feasible solution to (7.1). Secondly, for a feasible set $\mathcal{S}$, all its subsets $S' \subseteq S$ are also feasible to the problem. Therefore, according to *Definition 7.2*, the feasible sets of our scheduling problem (7.1) form an independence system.

Here, we provide an illustrative example for the previous two definitions. Assume that we have 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 7.4.

After identifying the structure of feasible sets, we can provide a general guaranteed performance of our greedy scheduling algorithm, Algorithm 7.1.

Theorem 7.1. *The optimal solution of the scheduling problem* (7.1) *can be approximated by the greedy algorithm, Algorithm 7.1, with a factor of* $1/(1+p)$ *in the worst case, when the feasible sets form a p-independence system,* i.e., $U(\mathcal{S}_G) \geq \max_{\mathcal{S} \subseteq \mathcal{T}} U(\mathcal{S})/(1+p)$.

Given that our scheduling problem (7.1) is a p-independent system. The theorem follows from the previous study in combinatorial optimization [90].

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 we show in the next section, this problem is solvable, but only approximately. Assume that we have an α-approximate oracle for the (next stage of the) utility curve. Hence, the following applies [90]:

Theorem 7.2. *When only an α-approximate oracle for the utility curve is available for $\alpha < 1$, the optimal solution of the scheduling problem* (7.1) *can be approximated by Algorithm 7.1, with a factor of $\alpha/(\alpha + p)$ in the worst case.*

Given that we have an α-approximate oracle for the (next stage of the) utility curve. The theorem follows directly from earlier literature [90].

The above two theorems provide Algorithm 7.1 a worst-case performance guarantee. The guarantee depends on p defined in (7.2). 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:

If we have 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 7.1. *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* (7.1) *forms a matroid,* i.e., 1*-independence system. The optimal solution of* (7.1) *can be approximated by the greedy algorithm, Algorithm 7.1, 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* (7.1) *can be approximated by Algorithm 7.1, with a factor of $\alpha/(\alpha + 1)$ in the worst case.*

If the feasible set of the scheduling problem (7.1) forms a matroid with $t(T_i^{(l)}) = w$, the whole proposition can be naturally proved with *Lemma 7.2 and 7.3*.

According to *Definition 7.1*, the feasible set of the scheduling problem (7.1) forms an independence system $(\mathcal{T}, \mathcal{I})$. We, therefore, only need to prove that, if $\mathcal{A}, \mathcal{B} \in \mathcal{I}$ and $|\mathcal{A}| > |\mathcal{B}|$, then $\exists e \in \mathcal{A}\backslash\mathcal{B}$ such that $\mathcal{B} \cup \{e\} \in \mathcal{I}$.

We assume that there are D deadline values. Without loss of generality, we sort them such that a larger index denotes a longer deadline. We denote the set of executable stages with

Algorithm 7.2. The top-k greedy submodular maximization algorithm

```
Input: $\mathcal{T}, \mathcal{U}(\cdot), \forall i : d_i$; Output: $\mathcal{S}_G$
$\mathcal{S}_G \leftarrow \emptyset, \mathcal{G} \leftarrow \mathcal{T}$
while $|\mathcal{S}_G| < k$ do
    $v* \leftarrow \arg\max_{v \in \mathcal{G}} \mathcal{U}(\mathcal{S}_G \cup v) - \mathcal{U}(\mathcal{S}_G)$
    $\mathcal{G} \leftarrow \mathcal{G} \backslash \{v*\}$
    if $\mathcal{S}_G \cup \{v*\}$ can be scheduled under deadlines $\{\tau_i\}$ then
        $\mathcal{S} \leftarrow \mathcal{S}_G \cup \{v*\}$
    end if
end while
return $\mathcal{S}_G$
```

deadline d_i in $\mathcal{A}$ and $\mathcal{B}$ as $\mathcal{A}_i$ and $\mathcal{B}_i$ respectively. We sort $\mathcal{A}$ and $\mathcal{B}$ according to the EDF algorithm. Since $|\mathcal{A}| > |\mathcal{B}|$, the total execution time of $\mathcal{A}$ is larger than the total execution time of $\mathcal{B}$ by at least w:

$$\begin{aligned} \sum_{a \in \mathcal{A}} t(a) - \sum_{b \in \mathcal{B}} t(b) \geq w, \\ d_D - \sum_{i=1}^{D} \sum_{b \in \mathcal{B}_i} t(b) \geq w. \end{aligned} \tag{7.3}$$

We try to find a element $a_D \in \mathcal{A}_D$ that $a_D \notin \mathcal{B}$. If such element a_D is found, then we prove that $\mathcal{B} \cup \{a_D\} \in \mathcal{I}$. If not, then any elements in $\mathcal{A}_D$ exists in $\mathcal{B}_D$, which indicates that $|\mathcal{A}_D| < |\mathcal{B}_D|$ and also:

$$d_{D-1} - \sum_{i=1}^{D-1} \sum_{b \in \mathcal{B}_i} t(b) \geq w. \tag{7.4}$$

Similarly, we try to find a element $a_{D-1} \in \mathcal{A}_{D-1}$ that $a_{D-1} \notin \mathcal{B}$. If such element a_{D-1} is found, then we prove that $\mathcal{B} \cup \{a_{D-1}\} \in \mathcal{I}$. If not, then any elements in $\mathcal{A}_{D-1}$ exists in $\mathcal{B}_{D-1}$, which indicates that $|\mathcal{A}_{D-1}| < |\mathcal{B}_{D-1}|$. Continuing with this recursion, we can either find a element a_i such that $\mathcal{B} \cup \{a_i\} \in \mathcal{I}$ that ends the proof or $|\mathcal{A}_i| < |\mathcal{B}_i|$ for i from D to 1. If we cannot find any element e such that $\mathcal{B} \cup \{e\} \in \mathcal{I}$, we end up with the condition that:

$$\sum_{i=1}^{D} |\mathcal{A}_i| < \sum_{i=1}^{D} |\mathcal{B}_i|, \tag{7.5}$$

which is in contradiction to the initial condition $|\mathcal{A}| > |\mathcal{B}|$. Thus, we prove the proposition.

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 7.1 with updated information. This property of constant updating causes some issues. On one hand, the

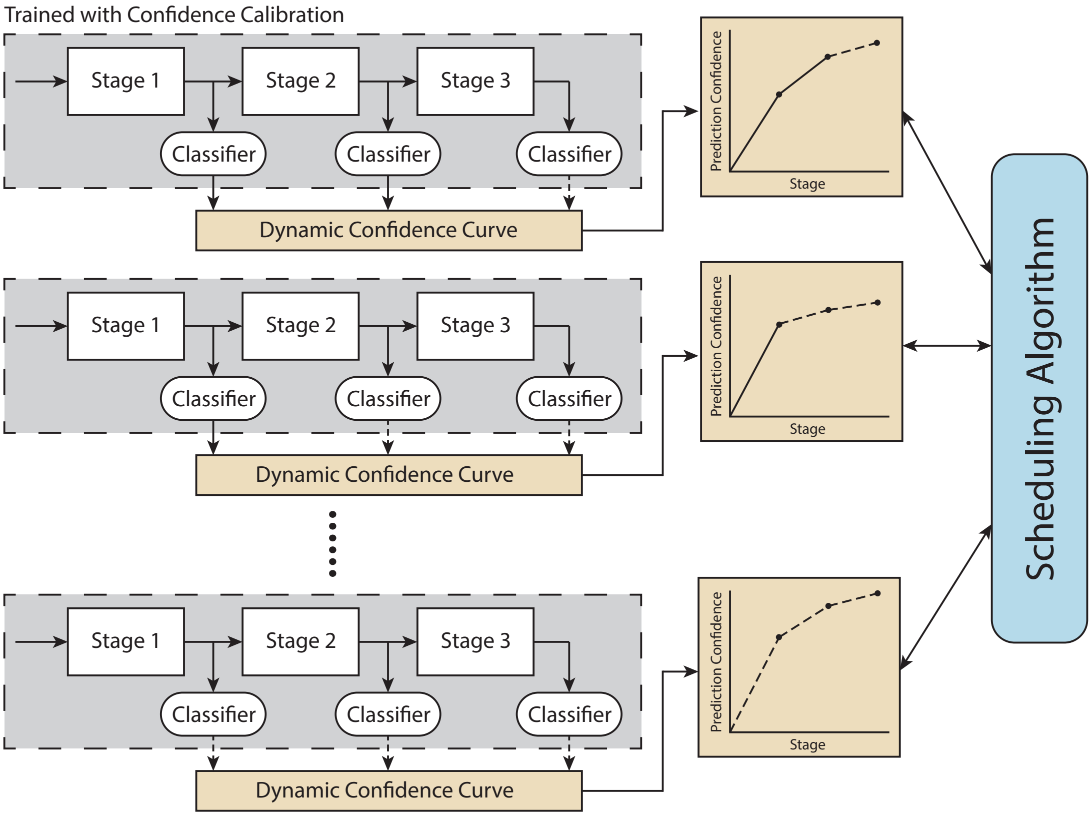


Figure 7.1: The overview of Deep Intelligence as a Service.

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, we further approximate Algorithm 7.1 by limiting the size of scheduling set. As shown in Algorithm 7.2, we change the ending condition of while loop into $|\mathcal{S}_G| < k$ (Line 2). Therefore, we will obtain a scheduling set $\mathcal{S}_G$ with size k through Algorithm 7.2, and then we schedule $\mathcal{S}_G$ with simple EDF algorithm.

7.2 DEEP INTELLIGENCE AS A SERVICE

Next, we 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 7.1.

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.

We discuss the technical details of these two modules in the following subsections.

7.2.1 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 [91].

The calibration of confidence can be visually represented by the reliability diagram [92]. As shown in Figure 7.2, 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.

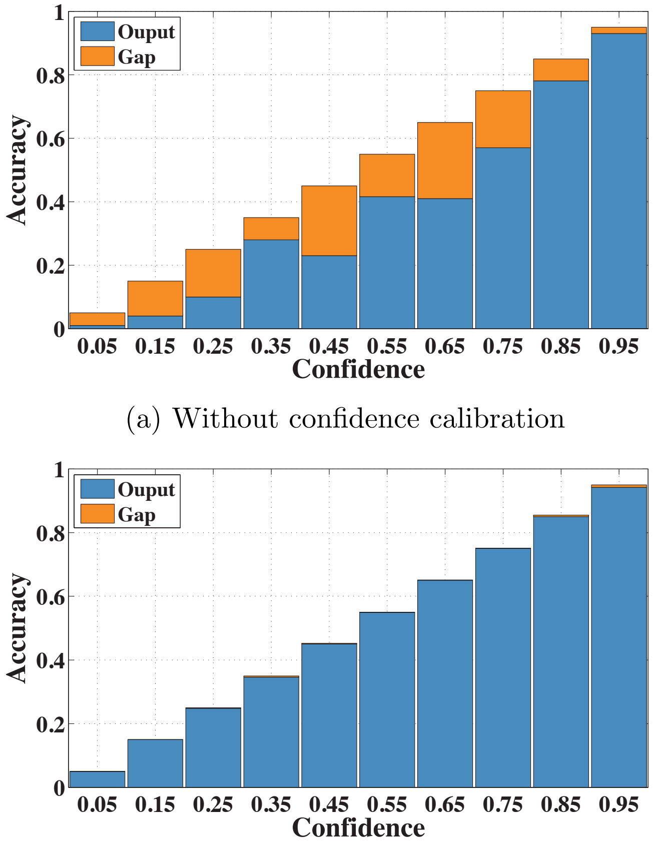
In order to represent the degree of miscalibration with a scalar that summarizes statistics of calibration, we introduce the metric, Expected Calibration Error (ECE) [93]. First, we group classification results into M bins with equal-width $1/M$. We denote $\mathcal{S}_m$ as the set of samples whose classification confidence falls into the interval $((m-1)/M, m/M]$. Then, we can define the average accuracy of $\mathcal{S}_m$ as:

$$acc(\mathcal{S}_m) = \frac{1}{|\mathcal{S}_m|} \sum_{\mathbf{S}_i \in \mathcal{S}_m} \mathbb{1}(\hat{y}_i = y_i), \tag{7.6}$$

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

$$conf(\mathcal{S}_m) = \frac{1}{|\mathcal{S}_m|} \sum_{\mathbf{S}_i \in \mathcal{S}_m} p_i, \tag{7.7}$$

where p_i is the classification confidence of sample $\mathbf{S}_i$. The ECE metric is defined as the



(a) Without confidence calibration

(b) With the entropy-based calibration.

Figure 7.2: The reliability diagrams of ResNet on CIFAR-10.

weighted average of the difference between average accuracy and confidence in M bins.

$$ECE = \sum_{m=1}^{M} \frac{|\mathcal{S}_m|}{m} \Big| acc(\mathcal{S}_m) - conf(\mathcal{S}_m) \Big|. \tag{7.8}$$

Accurate confidence estimation has drawn growing attention in recent studies [6, 89, 94]. However, existing efforts either focus mainly on regression problems [6, 89, 94] or tend to underestimate or overestimate the confidence [89, 94]. The behaviour of confidence underestimation and overestimation can be described by the formulation of average accuracy (7.6) and confidence (7.7). We denote $\mathcal{S}$ 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, we propose a simple entropy-based regularization method for confidence calibration with fine-tuning. We reformulate the loss function of the fine-tuning process as:

$$\mathcal{L} = CE(\mathbf{p}_i, \mathbf{y}_i) + \alpha \cdot H(\mathbf{p}_i), \tag{7.9}$$

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, we set $\alpha < 0$ and vice-versa.

Our confidence calibration method is simple but works well in practice. Detailed evaluation is shown in Section 7.4.1.

7.2.2 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. We do so using regression models that relate computed confidence in results of the executed stage(s) to predicted confidence in results of future stages.

Specifically, we choose the Gaussian process regression model [95]. We made this choice for 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 we can easily compute the mean value or desired confidence intervals. In this dissertation, we simply select the expected mean value, because RTDeepIoT focuses on maximizing the classification accuracy.

Specifically, we choose the Gaussian process regression model [95]. We made this choice for 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 we can easily compute the mean value or desired confidence intervals. In this dissertation, we simply select the expected mean value, because RTDeepIoT focuses on maximizing the classification accuracy.

For a three-stage neural network, as shown in Figure 7.1, we train three Gaussian process regression models, $\hat{p}_i^{(2)} = \mathcal{GP}_{1\_2}(p_i^{(1)})$, $\hat{p}_i^{(3)} = \mathcal{GP}_{1\_3}(p_i^{(1)})$, and $\hat{p}_i^{(3)} = \mathcal{GP}_{2\_3}(p_i^{(2)})$, where $p_i^{(l)}$

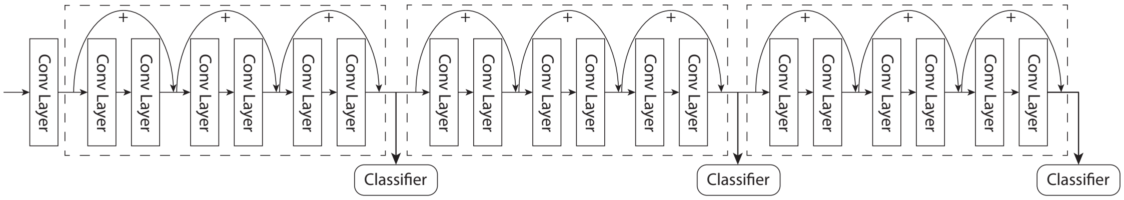


Figure 7.3: The illustration of three-stage ResNet.

denotes the classification confidence of sample i at neural network stage l. These regression models are learnt 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, we can approximate these complex Gaussian process regression models 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, we can use these computationally efficient piece-wise linear functions 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 7.4.1.

7.3 THE IMPLEMENTATION OF RTDEEPIOT

We implemented a user space scheduling framework to verify the effectiveness of RT-DeepIoT. Implementing the schedule 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, we integrate it with TensorFlow [96].

To implement classifiers for our proof-of-concept prototype, we choose an image recognition service based on a state-of-the-art convelutional neural network (CNN) structure; namely, residual neural networks (ResNet). As shown in Figure 7.3, 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.

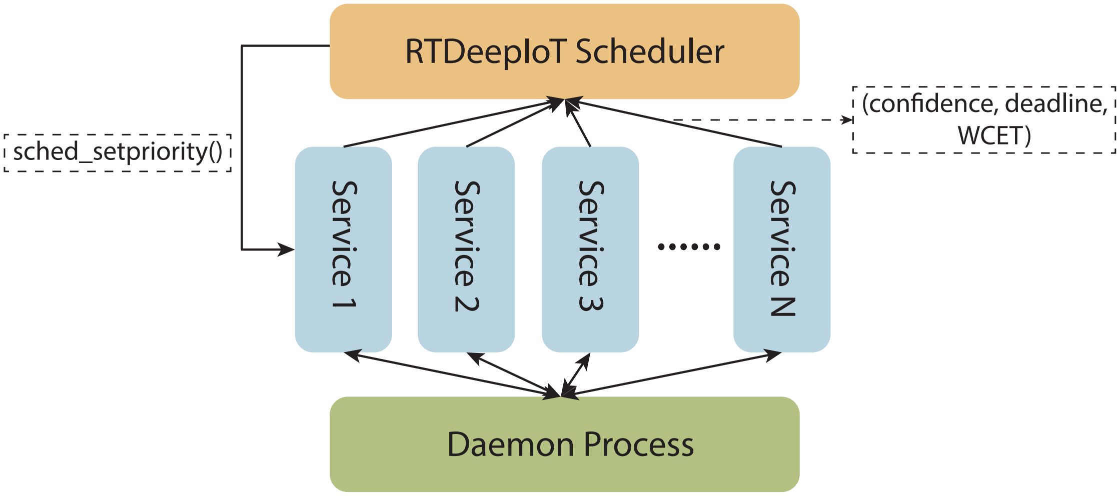


Figure 7.4: Structure of scheduling framework implementation.

Figure 7.4 demonstrates the structure of our framework. The RTDeepIoT scheduler spawns a pool of worker processes. 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. Below, we describe the workflow 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 7.2.

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.

7.4 THE EVALUATION OF RTDEEPIOT

To verify the effectiveness of our proposed scheduling algorithm, we tested the scheduler 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.

In the following subsections, we will evaluate our RTDeepIoT real-time scheduling pipeline from different perspectives, including classification confidence, scheduler overhead, pipeline bottlenecks, and overall classification accuracy under different workloads.

7.4.1 Confidence Calibration & Dynamic Confidence Updates

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

We first evaluate the quality of confidence calibration. We train the three-stage ResNet structure shown in Figure 7.3 on the CIFAR-10 dataset with following calibration method:

1. RTDeepIoT: the entropy-based confidence calibration method introduced in (7.9).

Table 7.1: 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 7.2: 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

2. RDeepSense: the state-of-the-art confidence calibration method with dropout operations [6].

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

An illustration of reliability diagrams is shown in Figure 7.2. 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 (7.8), is shown in Table 7.1. 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.

Next, we evaluate the quality of our dynamic confidence updates predicted for three-stage ResNet, 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 7.2. 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. In the following sections, we show that the dynamic confidence update method provides better

Table 7.3: The averaged overhead of RTDeepIoT scheduler and daemon process for 10000 images with different number of worker processes.

	3 procs	4 procs	5 procs
Scheduling Runtime (ms)	24.82	33.72	37.56
Classification Runtime (ms)	196.60	256.67	314.67
Total Runtime (ms)	221.42	290.39	352.23
Overhead	11.2%	11.6%	10.7%

overall classification accuracy than simple heuristics.

7.4.2 Scheduler Overhead

In this subsection, we evaluate the overhead of our framework. Since the framework contains a user space scheduler and a daemon process running at highest priority, we need to make sure these two modules do not impose significant overhead. We measure the portion of time spent on scheduling framework and daemon process.

As shown in Table 7.3, RTDeepIoT scheduling framework takes roughly 11% of total runtime to calculate and adjust priority of worker processes. Implementing our schedule in user space on top of the TensorFlow framework, therefore, has significant costs. Scheduling overhead per task can be as large as a few dozen milliseconds and the percentage overhead reaches the low double-digits. While more efficient implementation of the scheduler are possible, we opted for ours 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 TensorFlow overheads, which is the price paid for compatibility/portability. As we show later, despite this overhead, we still achieve a higher total utility because we are able to allocate more judiciously the resources remaining after overhead is paid.

7.4.3 Transmission vs. Computation Bottleneck

The deep-intelligence-as-a-serive scheduler presented in this dissertation 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

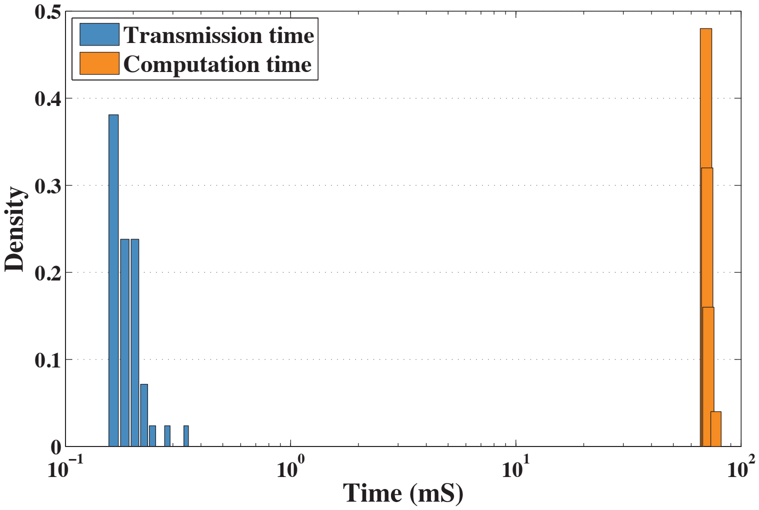


Figure 7.5: Histogram of transmission and computation time.

completion and this work is not needed.

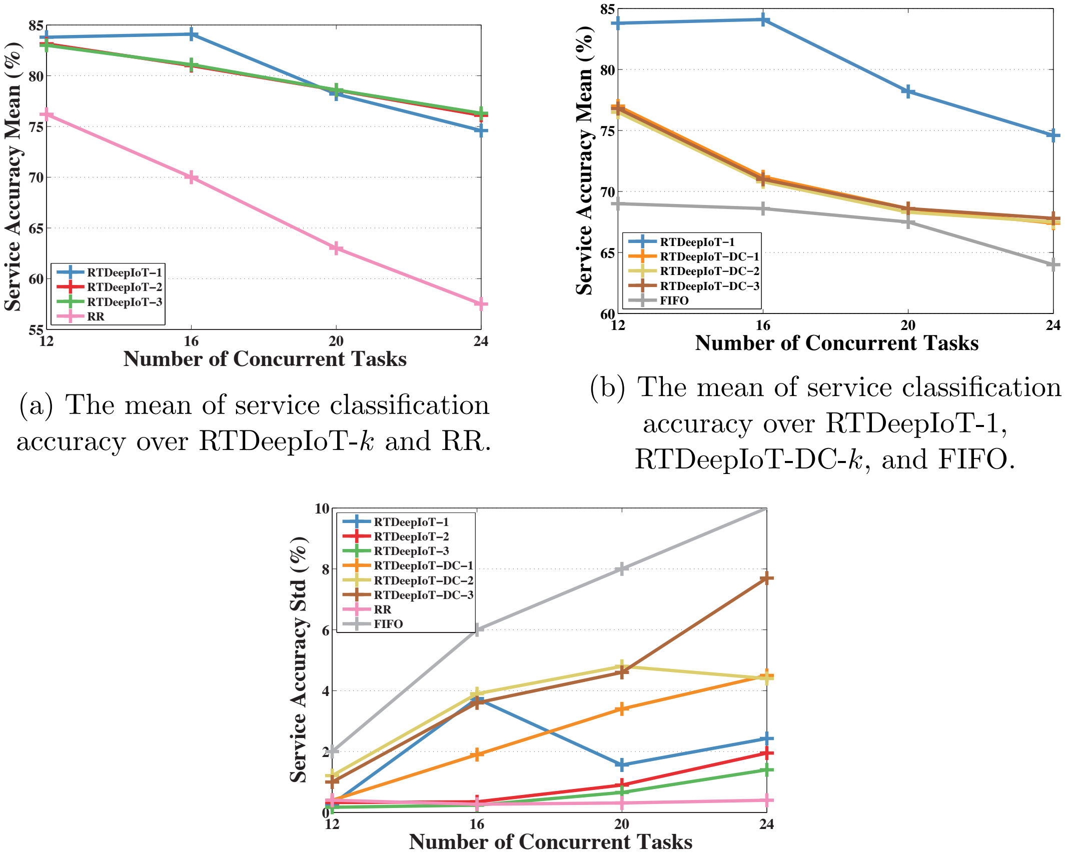
In this subsection, we compare empirical measurements of transmission and computation delays from our prototype of deep-learning based image recognition services. A local desktop constantly transmits images from CIFAR-10 dataset to a remote workstation through a secure copy protocol. The trained ResNet, as shown in Figure 7.3, takes the received images from the local desktop as input and runs the whole three stages to classify the image content. We measure the distributions of transmission and computation delays and plot Figure 7.5 on a log scale along the time axis. We can see 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.

7.4.4 Deep Intelligence as a Service

In this subsection, we evaluate the deep learning based image recognition services on the workstation. In order to illustrate the effectiveness of our RTDeepIoT pipeline, we take the following algorithms as the backend scheduler.

1. RTDeepIoT-k: this is our proposed scheduling pipeline, where k denotes the size of scheduling set we choose in Algorithm 7.2. During the whole experiments, we select k to be $\{1, 2, 3\}$.

2. RTDeepIoT-DC-k: this is a variant of our scheduling pipeline. Instead of using dynamic confidence updates, we assume that the confidence increases with the same slope. Therefore, we use the confidence gain of the current stage as the gain of future stages. We still select k to be $\{1, 2, 3\}$.



(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.

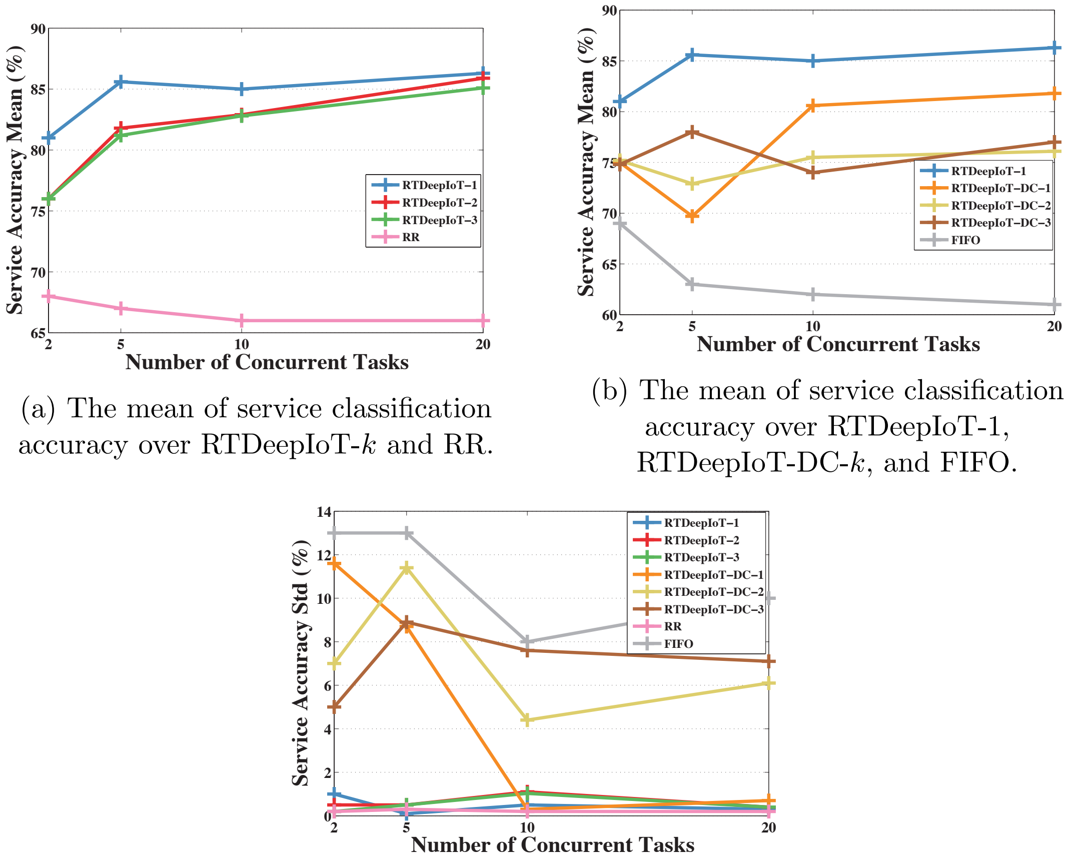
Figure 7.6: The intensity test for scheduling algorithms with ResNet on CIFAR-10.

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.

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, we also consider two kind of service workloads. In the first one, we keep the deadline of deep learning services unchanged but increase the number of concurrent deep learning tasks. In the second one, we increase the service deadline and the number of concurrent deep learning tasks proportionally. Each deep learning task will log its classification results over stages with timestamps. We will select the last result before its deadline as the final classification result.

In the first set of experiments, we set the deadline to 1.0s, while increasing the number of concurrent deep learning tasks from 12 to 24 with step 4. We illustrate the mean value and the standard deviation of classification accuracy over concurrent tasks for the scheduling



(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 7.7: The scalability test for scheduling algorithms with ResNet on CIFAR-10.

algorithms in Figure 7.6. From Figure 7.6c, we can clearly see that 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. We believe that such phenomenon is caused by two reasons. On one hand, although *Lemma 7.3* and *Proposition 7.1* can provide us the worst case guarantee with Algorithm 7.1 even when we only have an approximately correct utility curve, Algorithm 7.1 may not be the optimal choice for a normal case, which is our experiment setting. On the other hand, Algorithm 7.1 assumes the underlying scheduling policy to be EDF, which is not the best choice under the intensive workload. However, as we increase intensity of workload, 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, we can achieve better classification accuracy through utilizing the heterogeneous input data complexity. When comparing RTDeepIoT-k to RTDeepIoT-DC-k, we can find that RTDeepIoT-k consistently achieves better performance with a large margin. Therefore, 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, we proportionally increase the deadline and the number of concurrent tasks from 0.14s and 2 to 0.35s and 5, 0.7s and 10, as well as 1.4s and 20. We illustrate the mean value and the standard deviation of classification accuracy over concurrent tasks for the scheduling algorithms in Figure 7.7. The standard deviation of classification accuracy, shown in Figure 7.7c, 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 7.2 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 we increase the scale of the workload.

When using algorithm 7.2 as backend, *i.e.*, RTDeepIoT-k and RTDeepIoT-DC-k, the overall classification accuracy among deep learning services increases as we scale up the workload. 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 we scale the workload. Therefore, the scalability of these algorithms is poor, while our RTDeepIoT does better as we scale the workload. This again illustrates the importance of taking neural network depth as a new dimension for scheduling.

CHAPTER 8: RELATED WORK

8.1 DEEP LEARNING FOR SENSOR-RICH IOT SYSTEMS

Recently, deep learning [32] has become one of the most popular methodologies in AI-related tasks, such as computer vision [29], speech recognition [97], and natural language processing [36]. Lots of deep learning architectures have been proposed to exploit the relationships embedded in different types of inputs. For example, Residual nets [29] introduce shortcut connections into CNNs, which greatly reduces the difficulty of training super-deep models. However, since residual nets mainly focus on visual inputs, they lose the capability to model temporal relationships, which are of great importance in time-series sensor inputs. LRCNs [98] apply CNNs to extract features for each video frame and combine video frame sequences with LSTM [30], which exploits spatio-temporal relationships in video inputs. However, it does not consider modeling multimodal inputs. This capability is important to mobile sensing and computing tasks, because most tasks require collaboration among multiple sensors. Multimodal DBMs [99] merge multimodal inputs, such as images and text, with Deep Boltzmann Machines (DBMs). However, the work does not model temporal relationships and does not apply tailored structures, such as CNNs, to effectively and efficiently exploit local interactions within input data. To the best of our knowledge, DeepSense is the first architecture that possesses the capability for both (i) modelling temporal relationships and (ii) fusing multimodal sensor inputs. It also contains specifically designed structures to exploit local interactions in sensor inputs.

There are several illuminating studies, applying deep neural network models to different mobile sensing applications. DeepEar [10] uses Deep Boltzmann Machines to improve the performance of audio sensing tasks in an environment with background noise. RBM [44] and MultiRBM [45] use Deep Boltzmann Machines and Multimodal DBMs to improve the performance of heterogeneous human activity recognition. IDNet [47] applies CNNs to the biometric gait analysis task. However, these studies do not capture the temporal relationships in time-series sensor inputs, and, with the only exception of MultiRBM, lack the capability of fusing multimodal sensor inputs. In addition, these techniques focus on classification-oriented tasks only. To the best of our knowledge, DeepSense is the first framework that directly solves both regression-based and classification-based problems in a unified manner.

The impressive achievements in image classification using deep neural networks at the turn of the decade [100] precipitated a re-emergence of intereset in deep learning. Deep neural networks have achieved significant accuracy improvements in a broad spectrum of areas, including computer vision [26, 71], natural language processing [36, 101], and network analysis [102, 103]. However, none of the aforementioned IoT-inspired efforts addressed the customization of learning machinery to a different signal space inspired by the physics of measured processes; namely, the frequency domain.

To fill the above gap, recent work in machine learning focused on extending deep neural networks to complex numbers and spectral representations. Trabelsi et al. propose deep complex networks, investigating the complex-value neural network structure [18]. However, they mainly concentrate on the problems of initialization, normalization, and activation functions when extending real-valued operations directly into the complex-value domain. Their designs focus more on complex-value representations than spectral representations, and do not take the properties of spectral data into consideration. Rippel et al. study spectral representations for convolutional neural networks [104]. However, their study focuses on spectral parametrizing of standard CNNs, instead of designing operations customized for spectral data. In addition, their work treats input data fully from the frequency perspective instead of the time-frequency perspective. DeepSense takes short-time Fourier transformed data as inputs [2]. Yet their design uses traditional CNNs and RNNs, combining the real and imagery parts of complex-value inputs as additional features.

To the best of our knowledge, STFNet is the first work that integrates neural networks with traditional time-frequency analysis, and designs fundamental spectral-compatible operations for Fourier-transformed representations. Our study shows that the approach leads to improved accuracy compared to the state of the art. It implies that integrating neural networks with domain-inspired transformation techniques (in our case, the Fourier Transform of physical time-series signals) projects input signals into a space that significantly facilitates the learning process.

8.2 DEEP LEARNING FOR RESOURCE-CONSTRAINED IOT SYSTEMS

Recent studies focused on compressing deep neural networks for embedded and mobile devices. Han et al. proposed a magnitude-based compression method with fine-tuning, which illustrated promising compression results [58]. This method removes weight connections with low magnitude iteratively; however, it requires additional implementation of sparse matrix with more resource consumption. In addition, the aggressive pruning method increases the potential risk of irretrievable network damage. Guo et al. proposed a compression algorithm with connection splicing, which provided the chance of rehabilitation with a certain threshold [20]. However, the algorithm still focuses on weight level instead of structure level. Other than the magnitude-based method, another series of works focused on the factorization-based method that reduced the neural network complexity by exploiting low-rank structures in parameters. Denton et al. exploited various matrix factorization methods with fine-tunning to approximate the convolutional operations in order to reduce the neural network execution time [105]. Lane et al. applied sparse coding based and matrix factorization based method to reduce complexity of fully-connected layer and convolutional layer respectively [61]. However, factorization-based methods usually obtain lower compression ratio compared with magnitude-based methods, and the low-rank assumption may hurt the final network performance. Wang et al. applied the information of frequency domain for model compression [106]. However, additional implementation is required to speed-up the frequency-domain representations, and the method is not suitable for modern CNNs with small convolution filter sizes. Hinton et al. proposed a teacher-student framework that distilled the knowledge in an ensemble of models into a single model [107]. However, the framework focused more on compressing model ensemble into a single model instead of structure compression.

To the best of our knowledge, DeepIoT is the first framework for neural network structure compressing based on dropout operations and reducing parameter redundancies, where dropout operations provide DeepIoT the chance of rehabilitation with a certain probability. DeepIoT generates a more concise network structure for transplanting large-scale neural networks onto resource-constrained embedded devices.

In addition, all these previous compression algorithms focus on reducing the model parameters, while taking execution time speed-up as a by-product. Therefore, these compression methods inevitably show inferior performance on execution time reduction. There are some preliminary studies on designing time-efficient neural network for mobile and embedded devices [21–23], but all these works design the network structure based on their own personal experience. There is little understanding about the impact of different network structures on system performance, such as the execution time. In addition, these works usually use some biased metrics, such as FLOPs, for evaluating the model execution time. To the best of our knowledge, FastDeepIoT is the first framework to understand the impact of changing neural network structure on model execution time, and to empower existing compression algorithms to reduce the execution time on mobile and embedded devices properly.

8.3 DEEP LEARNING FOR LABEL-LIMITED IOT SYSTEMS

The idea of GANs is to design a game between two competing networks. The generator network takes the noise vectors as inputs and generates data samples. The discriminator network takes either a generated sample or a real data sample, and distinguishes between the two. The generator is trained to fool the discriminator [72].

Since training instability has hindered the deployment of GANs on deeper and more complex neural network structures, a great amount of research efforts have been made recently to tackle this problem. The WGAN with gradient penalty has achieved the state-of-the-art performance on multiple generative tasks, such as images and text generations [73]. To the best of our knowledge, SenseGAN is the first study to adopt WGAN with gradient penalty training strategy into the semi-supervised learning.

In addition, deep learning is emerging as a powerful learning component in IoT applications [1]. These highly capable models are good at making sophisticated mappings between unstructured data such as sensor inputs and target quantities, which can hardly be achieved by traditional machine learning models. Lane et al. [10] build a multilayer perceptron improving the performance of multiple audio sensing tasks. Yao et al. [2] design a unified deep learning framework for IoT tasks that can fuse multiple sensory modalities and extract temporal relationships along sensor inputs. However, to the best of our knowledge, all the previous works focus on the supervised learning scenario, which fails to utilize the abundant unlabelled data in IoT applications.

8.4 DEEP LEARNING FOR RELIABLE IOT SYSTEMS

Recently there are some illuminating works from the machine learning community that tries to provide deep neural networks with uncertainty estimations. Gal et al. [84] provide the first theoretical proof of the linkage between dropout training with deep Gaussian process called MCDrop. However, the proposed method tends to underestimate the uncertainty due to the nature of variational inference. Lakshminarayanan et al. [89] propose a solution SSP based on proper scoring rules and ensemble methods. However, the proposed method tends to overestimate the uncertainty on real datasets.

Since these previous works do not consider the scenario of mobile and ubiquitous computing, all these proposed methods require the operations with high computational cost during model inference, *i.e.*, sampling methods or ensemble methods. These computationally intensive operations aggravate the time and energy consumption problems in the embedded devices, which is one of the key issues of mobile and ubiquitous computing.

To the best of our knowledge, RDeepSense is the first work that provides a simple yet effective solution to estimate the uncertainties of deep neural networks for mobile and ubiquitous computing applications. RDeepSense uses proper scoring rules to mitigate the underestimation effect of MCDrop, and applies dropout training as implicit ensemble to avoid the computationally intensive ensemble method used in SSP.

8.5 DEEP LEARNING SERVICES ON EDGE/CLOUD

RTDeepIoT is the first to develop a real-time scheduler for a service motivated by machine intelligence needs of IoT applications. 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 [108], architecture [109], energy efficient computing [110], machine learning [111], algorithm design [112] [113]. Our work resembles approximate computing in that we aim to provide better quality of service (QoS) in the context of offering machine intelligence as the service paradigm, where we 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 [114–118]. Our work resembles imprecise computations in that we use 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 [116], 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 [119,120]. Lee et al. extended the QoS management analysis with the discrete and non-concave utility functions [121]. Abdelzaher et al. proposed a real-time QoS negotiation model for maximizing system utility with guaranteed performance [122]. Curescu et al. presented a QoS optimization scheme for mobile networks [123]. Koliver et al. designed a fuzzy-control approach for QoS adaptation [124]. However, all of the previous studies assume a known utility function beforehand. In this dissertation, we 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 TensorFlow - 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 9: CONCLUSION AND DISCUSSION

We have only scratched the surface of the research landscape on Deep Learning for IoT. Fundamentally, interest in deep learning will evolve as a means to bridge the ever-growing gap between the exponentially increasing planet-wide data generation rate on one hand (thanks to the proliferation of IoT devices), and the flat human ability to consume the data, on the other (since our cognitive capacity and population do not increase at the same exponential rate). Deep learning empowers automation that takes the human out of the data processing loop and more to a supervisory capacity.

9.1 DEEP LEARNING FOR SENSOR-RICH IOT SYSTEMS

The past decade witnessed a reemergence of interest in deep learning with significant contributions to *human-like* perception modalities including computer vision, natural language processing, and speech processing. In the next decade, however, growth of IoT-device-sourced data will significantly outpace the growth of human-sources data, due to the proliferation of such devices at rates that far outpace human population growth on the planet. As a consequence, I envision that a growing research interest will shift to modeling and analyzing "IoT big data" using deep neural networks. This is not only due to the sheer volume of data created by the growing number of IoT devices, but also due to the unique problem space that IoT data offers. IoT data are generated by physical, social, and spatio-temporal processes that have different dynamics, correlations, and internal structure compared to bits in a video, or words in an article. Researchers have gained much experience designing neural networks for human-like perception tasks, inspired by the way our brain processes information. STFNet suggests that while the human brain evolved to excel at such perception tasks, it is not optimized for discerning the internal physics of a process, or the spatiotemporal dynamics of a planet-wide phenomenon. As such today's brain-inspired neural networks are not well-suited for inference from IoT-sourced big data. For example, my work shows that learning in the frequency domain leads to improved results, when it comes to inference from physical sensor data. Starting with existing transform domains for physical measurements (e.g., Wavelet Transform, Fourier Transform, etc) and dimensionality reduction techniques for big data (e.g., SVD, PCA, etc), I will develop hybrid solutions that empower deep learning in feature spaces that are better suited to the inherent big IoT data properties and dynamics (compared to learning from direct observations, as inspired by perception physiology of the human brain). As a result, my new deep hybrid networks will inherently perceive, capture, and distinguish the internal characteristics of big data sets the way our brains perceive external shapes and color patterns of objects, leading to significant advances in big IoT data analysis tasks.

9.2 DEEP LEARNING FOR RESOURCE-CONSTRAINED IOT SYSTEMS

DeepIoT and FastDeepIoT are frameworks for understanding and minimizing neural network execution time on mobile and embedded devices. We proposed a tree- structured linear regression model to figure out the causes of execution-time nonlinearity and to interpret execution time through explanatory variables. Furthermore, we utilized the execution time model to rebalance the focus of existing structure compression algorithms to reduce the overall execution time properly.

They are just the first few steps into the exploration of neural network compression for performance optimization. More profiling results are needed with the different choices of hardware, OS versions, load factors, power scaling, and deep learning libraries. Currently, FastDeepIoT can only support deep learning structure compression algorithms. More work is needed to support other deep learning compression methods, such as parameter quantization and pruning. The execution time model shows that the setup overhead of recurrent layers imposes a lower bound on efficacy of compression. It is a function of recurrent neural network steps, offering another dimension to compress for speeding up recurrent layers. These insights offer avenues for future research on system performance oriented neural network compression for sensing applications.

9.3 DEEP LEARNING FOR LABEL-LIMITED IOT SYSTEMS

SenseGAN separates the functionalities of discriminator and classifier into two neural networks, designs specific generator and discriminator structures for handling multimodal sensing inputs, and stabilizes and enhances the adversarial training process by WGAN with gradient penalty as well as Gumbel-Softmax for categorical representations. The evaluation empirically shows that SenseGAN can efficiently leverage both labelled and unlabelled data to effectively improve the predictive power of the classifier without additional time and energy consumption during the inference. Several improvement opportunities remain. First, our architecture for fusing multi-sensor data is not yet optimal. For example, we can use attention-based structures [125] to guide the discriminator and the generator to focus on data segments containing more representative information. Second, SenseGAN focuses on the classification problem in the IoT scenario. However, regression is another common IoT task that needs to be considered. To solve the regression problem with SenseGAN, we can transform the range of continuous target outputs into a set of intervals that can be used as discrete classes. By treating regression problems as classification problems, SenseGAN can be applied in the manner described in this dissertatuib. However, we still need further studies to formally solve this problem. Third, the learning process of SenseGAN is computationally intensive. Therefore, more studies are needed to enable online adaptive learning with streaming unlabelled sensing data on low-power IoT devices. Finally, more evaluation is needed in the context of deployed application scenarios to better understand the feasibility of needed (albeit minimal) labeling and the limitations of the approach.

9.4 DEEP LEARNING FOR RELIABLE IOT SYSTEMS

RDeepSense focuses on empowering neural networks to generate high-quality predictive uncertainty estimations in a theoretically-grounded and energy-efficient manner for mobile and ubiquitous computing tasks. Currently, RDeepSense can only support fully-connected neural networks. It is possible to extend the solution to convolutional and recurrent neural networks by replacing the original dropout operation with convolutional dropout and recurrent dropout. But additional efforts are needed to 1) theoretically prove that the extended two-step solution can equate an arbitrary neural network with a statistical model, and 2) empirically show that the extended two-step solution can provide high-quality uncertainty estimations on the real datasets.

9.5 DEEP LEARNING SERVICES ON EDGE/CLOUD

This dissertation 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. We focused 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. We show 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] S. Yao, Y. Zhao, A. Zhang, S. Hu, H. Shao, C. Zhang, L. Su, and T. Abdelzaher, "Deep learning for the internet of things," *Computer*, vol. 51, no. 5, pp. 32–41, 2018.

[2] 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, pp. 351–360.

[3] 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.

[4] S. Yao, Y. Zhao, H. Shao, S. Liu, D. Liu, L. Su, and T. Abdelzaher, "Fastdeepiot: Towards understanding and optimizing neural network execution time on mobile and embedded devices," in *Proceedings of the 16th ACM Conference on Embedded Network Sensor Systems.* ACM, 2018.

[5] S. Yao, Y. Zhao, H. Shao, C. Zhang, A. Zhang, S. Hu, D. Liu, S. Liu, L. Su, and T. Abdelzaher, "Sensegan: Enabling deep learning for internet of things with a semi-supervised framework," *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, vol. 2, no. 3, p. 144, 2018.

[6] 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.

[7] S. Yao, Y. Zhao, H. Shao, C. Zhang, A. Zhang, D. Liu, S. Liu, L. Su, and T. Abdelzaher, "Apdeepsense: Deep learning uncertainty estimation without the pain for iot applications," in *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS).* IEEE, 2018, pp. 334–343.

[8] D. H. Hubel and T. N. Wiesel, "Receptive fields and functional architecture of monkey striate cortex," *The Journal of physiology*, vol. 195, no. 1, pp. 215–243, 1968.

[9] R. C. Gonzalez, R. E. Woods et al., "Digital image processing," 2002.

[10] N. D. Lane, P. Georgiev, and L. Qendro, "Deepear: robust smartphone audio sensing in unconstrained acoustic environments using deep learning," in *UbiComp*, 2015.

[11] A. Stisen, H. Blunck, S. Bhattacharya, T. S. Prentow, M. B. Kjærgaard, A. Dey, T. Sonne, and M. M. Jensen, "Smart devices are different: Assessing and mitigatingmobile sensing heterogeneities for activity recognition," in *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems.* ACM, 2015, pp. 127–140.

[12] S. Hemminki, P. Nurmi, and S. Tarkoma, “Accelerometer-based transportation mode detection on smartphones,” in *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems.* ACM, 2013, p. 13.

[13] W. Wang, A. X. Liu, M. Shahzad, K. Ling, and S. Lu, “Understanding and modeling of wifi signal based human activity recognition,” in *Proceedings of the 21st annual international conference on mobile computing and networking.* ACM, 2015, pp. 65–76.

[14] Q. Pu, S. Gupta, S. Gollakota, and S. Patel, “Whole-home gesture recognition using wireless signals,” in *Proceedings of the 19th annual international conference on Mobile computing & networking.* ACM, 2013, pp. 27–38.

[15] S. Gupta, D. Morris, S. Patel, and D. Tan, “Soundwave: using the doppler effect to sense gestures,” in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems.* ACM, 2012, pp. 1911–1914.

[16] K.-Y. Chen, D. Ashbrook, M. Goel, S.-H. Lee, and S. Patel, “Airlink: sharing files between multiple devices using in-air gestures,” in *Proceedings of the 2014 ACM International Joint Conference on Pervasive and Ubiquitous Computing.* ACM, 2014, pp. 565–569.

[17] T. Li, Q. Liu, and X. Zhou, “Practical human sensing in the light,” in *Proceedings of the 14th Annual International Conference on Mobile Systems, Applications, and Services.* ACM, 2016, pp. 71–84.

[18] C. Trabelsi, O. Bilaniuk, Y. Zhang, D. Serdyuk, S. Subramanian, J. F. Santos, S. Mehri, N. Rostamzadeh, Y. Bengio, and C. J. Pal, “Deep complex networks,” *arXiv preprint arXiv:1705.09792*, 2017.

[19] J. O. Smith, *Mathematics of the discrete Fourier transform (DFT): with audio applications.* Julius Smith, 2007.

[20] Y. Guo, A. Yao, and Y. Chen, “Dynamic network surgery for efficient dnns,” in *Advances In Neural Information Processing Systems*, 2016, pp. 1379–1387.

[21] X. Zhang, X. Zhou, M. Lin, and J. Sun, “Shufflenet: An extremely efficient convolutional neural network for mobile devices,” *CoRR*, vol. abs/1707.01083, 2017.

[22] A. G. Howard, M. Zhu, B. Chen, D. Kalenichenko, W. Wang, T. Weyand, M. Andreetto, and H. Adam, “Mobilenets: Efficient convolutional neural networks for mobile vision applications,” *CoRR*, vol. abs/1704.04861, 2017.

[23] F. N. Iandola, M. W. Moskewicz, K. Ashraf, S. Han, W. J. Dally, and K. Keutzer, “Squeezenet: Alexnet-level accuracy with 50x fewer parameters and ¡1mb model size,” *CoRR*, vol. abs/1602.07360, 2016.

[24] M. Arjovsky, S. Chintala, and L. Bottou, “Wasserstein gan,” *arXiv preprint arXiv:1701.07875*, 2017.

[25] E. Jang, S. Gu, and B. Poole, "Categorical reparameterization with gumbel-softmax," *arXiv preprint arXiv:1611.01144*, 2016.

[26] 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.

[27] 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.

[28] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv:1502.03167*, 2015.

[29] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *arXiv:1512.03385*, 2015.

[30] K. Greff, R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber, "Lstm: A search space odyssey," *arXiv:1503.04069*, 2015.

[31] J. Chung, C. Gulcehre, K. Cho, and Y. Bengio, "Empirical evaluation of gated recurrent neural networks on sequence modeling," *arXiv:1412.3555*, 2014.

[32] I. G. Y. Bengio and A. Courville, "Deep learning," 2016, book in preparation for MIT Press.

[33] M. Schuster and K. K. Paliwal, "Bidirectional recurrent neural networks," *IEEE Trans Sig. Process.*, 1997.

[34] W. Zaremba, I. Sutskever, and O. Vinyals, "Recurrent neural network regularization," *arXiv:1409.2329*, 2014.

[35] T. Cooijmans, N. Ballas, C. Laurent, and A. Courville, "Recurrent batch normalization," *arXiv:1603.09025*, 2016.

[36] D. Bahdanau, K. Cho, and Y. Bengio, "Neural machine translation by jointly learning to align and translate," *arXiv:1409.0473*, 2014.

[37] "Qualcomm snapdragon 800 processor," https://www.qualcomm.com/products/snapdragon/processors/800.

[38] "Intel edison compute module," http://www.intel.com/content/dam/support/us/en/documents/edison/sb/edison-module_HG_331189.pdf.

[39] N. D. Lane, S. Bhattacharya, P. Georgiev, C. Forlivesi, L. Jiao, L. Qendro, and F. Kawsar, "Deepx: A software accelerator for low-power deep learning inference on mobile devices," in *IPSN*, 2016.

[40] G. Milette and A. Stroud, *Professional Android sensor programming.* John Wiley & Sons, 2012.

[41] S. Hu, L. Su, S. Li, S. Wang, C. Pan, S. Gu, M. T. Al Amin, H. Liu, S. Nath et al., "Experiences with enav: a low-power vehicular navigation system," in *UbiComp*, 2015.

[42] D. Figo, P. C. Diniz, D. R. Ferreira, and J. M. Cardoso, "Preprocessing techniques for context recognition from accelerometer data," *Pers. Ubiquit. Comput.*, 2010.

[43] N. Y. Hammerla, R. Kirkham, P. Andras, and T. Ploetz, "On preserving statistical characteristics of accelerometry data using their empirical cumulative distribution," in *ISWC*, 2013.

[44] S. Bhattacharya and N. D. Lane, "From smart to deep: Robust activity recognition on smartwatches using deep learning," in *Pervasive Computing and Communication Workshops (PerCom Workshops), 2016 IEEE International Conference on.* IEEE, 2016, pp. 1–6.

[45] V. Radu, N. D. Lane, S. Bhattacharya, C. Mascolo, M. K. Marina, and F. Kawsar, "Towards multimodal deep learning for activity recognition on mobile devices," in *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing: Adjunct.* ACM, 2016, pp. 185–188.

[46] H. M. Thang, V. Q. Viet, N. D. Thuc, and D. Choi, "Gait identification using accelerometer on mobile phone," in *Control, Automation and Information Sciences (ICCAIS), 2012 International Conference on.* IEEE, 2012, pp. 344–348.

[47] M. Gadaleta and M. Rossi, "Idnet: Smartphone-based gait recognition with convolutional neural networks," *arXiv preprint arXiv:1606.03238*, 2016.

[48] L. Zhang, B. Tiwana, Z. Qian, Z. Wang, R. P. Dick, Z. M. Mao, and L. Yang, "Accurate online power estimation and automatic battery behavior based power model generation for smartphones," in *CODES+ISSS*, 2010.

[49] F. Yu and V. Koltun, "Multi-scale context aggregation by dilated convolutions," *arXiv preprint arXiv:1511.07122*, 2015.

[50] D. Halperin, W. Hu, A. Sheth, and D. Wetherall, "Tool release: Gathering 802.11 n traces with channel state information," *ACM SIGCOMM Computer Communication Review*, vol. 41, no. 1, pp. 53–53, 2011.

[51] Y. Gal and Z. Ghahramani, "Bayesian convolutional neural networks with bernoulli approximate variational inference," *arXiv preprint arXiv:1506.02158*, 2015.

[52] Y. Gal and Z. Ghahramani, "A theoretically grounded application of dropout in recurrent neural networks," 2016.

[53] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov, "Dropout: a simple way to prevent neural networks from overfitting." *Journal of Machine Learning Research*, vol. 15, no. 1, pp. 1929–1958, 2014.

[54] P. W. Glynn, "Likelihood ratio gradient estimation for stochastic systems," *Communications of the ACM*, vol. 33, no. 10, pp. 75–84, 1990.

[55] J. Peters and S. Schaal, "Policy gradient methods for robotics," in *2006 IEEE/RSJ International Conference on Intelligent Robots and Systems*. IEEE, 2006, pp. 2219–2225.

[56] A. Mnih and K. Gregor, "Neural variational inference and learning in belief networks," 2014.

[57] S. Gu, S. Levine, I. Sutskever, and A. Mnih, "Muprop: Unbiased backpropagation for stochastic neural networks," *arXiv preprint arXiv:1511.05176*, 2015.

[58] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," *CoRR, abs/1510.00149*, vol. 2, 2015.

[59] "Loading debian (ubilinux) on the edison," https://learn.sparkfun.com/tutorials/loading-debian-ubilinux-on-the-edison.

[60] Theano Development Team, "Theano: A Python framework for fast computation of mathematical expressions," *arXiv e-prints*, vol. abs/1605.02688, May 2016. [Online]. Available: http://arxiv.org/abs/1605.02688

[61] S. Bhattacharya and N. D. Lane, "Sparsification and separation of deep learning layers for constrained resource inference on wearables," in *Proceedings of the 14th ACM Conference on Embedded Network Sensor Systems CD-ROM*. ACM, 2016, pp. 176–189.

[62] V. Panayotov, G. Chen, D. Povey, and S. Khudanpur, "Librispeech: an asr corpus based on public domain audio books," in *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*. IEEE, 2015, pp. 5206–5210.

[63] A. Graves and N. Jaitly, "Towards end-to-end speech recognition with recurrent neural networks." in *ICML*, vol. 14, 2014, pp. 1764–1772.

[64] R. Rigamonti, A. Sironi, V. Lepetit, and P. Fua, "Learning separable filters," in *Computer Vision and Pattern Recognition (CVPR), 2013 IEEE Conference on*. IEEE, 2013, pp. 2754–2761.

[65] "Tensorflow benchmark tool," https://github.com/tensorflow/tensorflow/tree/r1.4/tensorflow/tools/benchmark.

[66] H. Drucker, C. J. Burges, L. Kaufman, A. J. Smola, and V. Vapnik, “Support vector regression machines,” in *Advances in neural information processing systems*, 1997, pp. 155–161.

[67] L. Breiman, *Classification and regression trees.* Routledge, 2017.

[68] L. Breiman, “Random forests,” *Machine learning*, vol. 45, no. 1, pp. 5–32, 2001.

[69] J. H. Friedman, “Greedy function approximation: a gradient boosting machine,” *Annals of statistics*, pp. 1189–1232, 2001.

[70] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *nature*, vol. 521, no. 7553, p. 436, 2015.

[71] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *arXiv preprint arXiv:1409.1556*, 2014.

[72] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, “Generative adversarial nets,” in *Advances in neural information processing systems*, 2014, pp. 2672–2680.

[73] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. Courville, “Improved training of wasserstein gans,” *arXiv preprint arXiv:1704.00028*, 2017.

[74] C. Villani, *Optimal transport: old and new.* Springer Science & Business Media, 2008, vol. 338.

[75] Y. Dauphin, A. Fan, M. Auli, and D. Grangier, “Language modeling with gated convolutional networks,” in *ICML*, 2017.

[76] E. Greensmith, P. L. Bartlett, and J. Baxter, “Variance reduction techniques for gradient estimates in reinforcement learning,” *Journal of Machine Learning Research*, vol. 5, no. Nov, pp. 1471–1530, 2004.

[77] M. A. A. Haseeb and R. Parasuraman, “Wisture: Rnn-based learning of wireless signals for gesture recognition in unmodified smartphones,” *arXiv preprint arXiv:1707.08569*, 2017.

[78] A. Odena, “Semi-supervised learning with generative adversarial networks,” *arXiv preprint arXiv:1606.01583*, 2016.

[79] I. Triguero, S. García, and F. Herrera, “Self-labeled techniques for semi-supervised learning: taxonomy, software and empirical study,” *Knowledge and Information Systems*, vol. 42, no. 2, pp. 245–284, 2015.

[80] K. P. Bennett and A. Demiriz, “Semi-supervised support vector machines,” in *Advances in Neural Information processing systems*, 1999, pp. 368–374.

[81] T. Gneiting and A. E. Raftery, "Strictly proper scoring rules, prediction, and estimation," *Journal of the American Statistical Association*, vol. 102, no. 477, pp. 359–378, 2007.

[82] C. E. Rasmussen, "Gaussian processes for machine learning," 2006.

[83] D. Koller and N. Friedman, *Probabilistic graphical models: principles and techniques*. MIT press, 2009.

[84] Y. Gal and Z. Ghahramani, "Dropout as a bayesian approximation: Representing model uncertainty in deep learning," in *ICML*, 2016.

[85] D. M. Blei, A. Kucukelbir, and J. D. McAuliffe, "Variational inference: A review for statisticians," *Journal of the American Statistical Association*, no. just-accepted, 2017.

[86] A. L. Goldberger, L. A. Amaral, L. Glass, J. M. Hausdorff, P. C. Ivanov, R. G. Mark, J. E. Mietus, G. B. Moody, C.-K. Peng, and H. E. Stanley, "Physiobank, physiotoolkit, and physionet," *Circulation*, vol. 101, no. 23, pp. e215–e220, 2000.

[87] M. Kachuee, M. M. Kiani, H. Mohammadzade, and M. Shabany, "Cuff-less high-accuracy calibration-free blood pressure estimation using pulse transit time," in *Circuits and Systems (ISCAS), 2015 IEEE International Symposium on*. IEEE, 2015, pp. 1006–1009.

[88] J. Fonollosa, S. Sheik, R. Huerta, and S. Marco, "Reservoir computing compensates slow response of chemosensor arrays exposed to fast varying gas concentrations in continuous monitoring," *Sensors and Actuators B: Chemical*, vol. 215, pp. 618–629, 2015.

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

[90] 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.

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

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

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

[94] 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.

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

[96] 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.

[97] G. E. Dahl, D. Yu, L. Deng, and A. Acero, "Context-dependent pre-trained deep neural networks for large-vocabulary speech recognition," *IEEE TASLP*, 2012.

[98] J. Donahue, L. Anne Hendricks, S. Guadarrama, M. Rohrbach, S. Venugopalan, K. Saenko, and T. Darrell, "Long-term recurrent convolutional networks for visual recognition and description," in *CVPR*, 2015.

[99] N. Srivastava and R. R. Salakhutdinov, "Multimodal learning with deep boltzmann machines," in *NIPS*, 2012.

[100] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[101] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa, "Natural language processing (almost) from scratch," *Journal of Machine Learning Research*, vol. 12, no. Aug, pp. 2493–2537, 2011.

[102] B. Perozzi, R. Al-Rfou, and S. Skiena, "Deepwalk: Online learning of social representations," in *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining.* ACM, 2014, pp. 701–710.

[103] T. N. Kipf and M. Welling, "Semi-supervised classification with graph convolutional networks," *arXiv preprint arXiv:1609.02907*, 2016.

[104] O. Rippel, J. Snoek, and R. P. Adams, "Spectral representations for convolutional neural networks," in *Advances in neural information processing systems*, 2015, pp. 2449–2457.

[105] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus, "Exploiting linear structure within convolutional networks for efficient evaluation," in *Advances in Neural Information Processing Systems*, 2014, pp. 1269–1277.

[106] Y. Wang, C. Xu, S. You, D. Tao, and C. Xu, "Cnnpack: packing convolutional neural networks in the frequency domain," in *Advances in Neural Information Processing Systems*, 2016, pp. 253–261.

[107] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," *arXiv preprint arXiv:1503.02531*, 2015.

[108] 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.

[109] 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.

[110] 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.

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

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

[113] 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.

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

[115] 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.

[116] 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.

[117] 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.

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

[119] 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.

[120] 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.

[121] 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.

[122] 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.

[123] 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.

[124] 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.

[125] K. Xu, J. Ba, R. Kiros, K. Cho, A. Courville, R. Salakhudinov, R. Zemel, and Y. Bengio, “Show, attend and tell: Neural image caption generation with visual attention,” in *International Conference on Machine Learning*, 2015, pp. 2048–2057.