ENABLING DEEP INTELLIGENCE ON EMBEDDED SYSTEMS

Seulki Lee

A dissertation submitted to the faculty at the University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill
2021

ABSTRACT

Seulki Lee: Enabling Deep Intelligence on Embedded Systems
(Under the direction of Shahriar Nirjon)

As deep learning for resource-constrained systems become more popular, we see an increased number of intelligent embedded systems such as IoT devices, robots, autonomous vehicles, and the plethora of portable, wearable, and mobile devices that are feature-packed with a wide variety of machine learning tasks. However, the performance of DNNs (deep neural networks) running on an embedded system is significantly limited by the platform's CPU, memory, and battery-size; and their scope is limited to simplistic inference tasks only.

This dissertation proposes on-device deep learning algorithms and supporting hardware designs, enabling embedded systems to efficiently perform deep intelligent tasks (i.e., deep neural networks) that are high-memory-footprint, compute-intensive, and energy-hungry beyond their limited computing resources. We name such on-device deep intelligence on embedded systems as *Embedded Deep Intelligence*. Specifically, we introduce resource-aware learning strategies devised to overcome the four fundamental constraints of embedded systems imposed on the way towards Embedded Deep Intelligence, i.e., *in-memory multitask learning via introducing the concept of Neural Weight Virtualization, adaptive real-time learning via introducing the concept of SubFlow, opportunistic accelerated learning via introducing the concept of Neuro.ZERO, and energy-aware intermittent learning*, which tackles the problems of the small size of memory, dynamic timing constraint, low-computing capability, and limited energy, respectively.

Once deployed in the field with the proposed resource-aware learning strategies, embedded systems are not only able to perform deep inference tasks on sensor data but also update and re-train their learning models at run-time without requiring any help from any external system. Such an on-device learning capability of Embedded Deep Intelligence makes an embedded intelligent system real-time, privacy-aware, secure, autonomous, untethered, responsive, and adaptive without concern for its limited resources.

To my family

ACKNOWLEDGEMENTS

This dissertation is about adventure, challenge, sweat, and tears. My journey in the Ph.D. program started at a late age amid a mixture of anticipation and worry. At the moment, I had no academic background and research experience in computer science. It was great people at UNC Computer Science who helped me win through in the end.

I could not have completed this dissertation without tremendous support from my advisor, Shahriar Nirjon, who taught me everything about being an independent researcher. I was fortunate to be under his direction during my academic journey, which prepared me to find and solve interesting problems independently. He has demonstrated the high quality of a professional researcher, mentor, and sometimes friend, which will keep guiding me in my future career.

Although the author of this dissertation seems to be only one, there are many people whom I have to thank.

First and foremost, I should say that my beloved wife, Jieun, who always has helped me focus on my work throughout my academic journey and took care of our adorable children with devotion, should be the co-author of this dissertation. If I could, I would insist that the Ph.D. degree should be awarded to her as well, considering the significant amount of support she has put to me for the last several years.

It also ought to include the names of my classmates and colleagues in the Ph.D. program, without whom, I can say with certainty, I would not have succeeded in this endeavor: my direct collaborators, Chong Shao, Shiwei Fang, Tamzeed Islam, Bashima Islam, Yubo Luo, Kyle Asher, and Austin Wang. Their collaboration and tutelage of the intricacies of embedded intelligent systems made it possible for me to fill in the gaps in my educational background, while as conversational partners, they served as meaningful sounding boards for new research ideas and directions.

Special mention is due to my summer collaborators, Chun-Nam Yu and Karun Nithi, who set me along the path of machine learning during my internship at Nokia Bell Labs. I am also profoundly grateful to Naveen Sangeneni and my teammates, with whom I experienced the future transportation systems at Mercedes-Benz Research & Development North America over a few months.

Finally, I want to acknowledge my friends at UNC Computer Science, Namhoon Kim, Young-Woon Cha, Hyounghun Kim, Jaehyun Han, Jae Sung Park, YoungJoong Kwon, and Jaemin Cho, who have stayed with me with nadir and zenith of my Ph.D. experience.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1: INTRODUCTION

1.1 Confluence of Embedded Systems and Artificial Intelligence

Along with the deepening development in computing technologies and the surge of embedded, mobile, and IoT (Internet of Things) devices, more and more data is created by widespread and geographically distributed embedded and IoT devices. For example, 45% of the 40 zettabytes global internet data is expected to be generated by embedded devices in 2024 (Ericsson, 2019). Meanwhile, Artificial Intelligence (AI), defined as intelligence exhibited by machines, is thriving with the breakthroughs in machine learning algorithms such as deep neural networks (DNNs) (Goodfellow et al., 2016) due to their superiority in solving complex machine learning problems (Young et al., 2018; Schroff et al., 2015; Krizhevsky et al., 2012), e.g., autonomous driving (Bojarski et al., 2017; Chen et al., 2017c, 2016b, 2015a), natural language processing (Socher et al., 2012; Deng and Liu, 2018; Khan et al., 2016), and healthcare applications (Miotto et al., 2017; Jiang et al., 2017; Litjens et al., 2017) with the help of billions of bytes of data generated at the embedded devices. Considering that AI is functionally necessary for quickly analyzing vast volumes of data and extracting insights, there exists a strong demand to integrate embedded devices and AI, which gives the birth of a brand-new paradigm called *Embedded Intelligence* that performs intelligent tasks on the device directly without offloading massive data from the device to the cloud (Deng et al., 2019).

1.2 Embedded Deep Intelligence

Embedded Intelligence is not a simple combination of embedded systems and AI. The subject of embedded intelligence is tremendous and enormously sophisticated, covering many concepts and technologies, which are interwoven together in a complicated manner. Currently, the formal and acknowledged definition of Embedded Intelligence is non-existent. To deal with the problem, some researchers put forward their definitions. For example, Zhou et al. (2019) argues that the scope of Embedded Intelligence should not be restricted to running AI models solely on the cloud servers or devices but in the manner of the collaboration of device and cloud. They define six levels of Embedded Intelligence, from cloud-device co-inference (level 1) to all on-device (level 6). In this research, we focus on all on-device machine learning on embedded systems (Li et al., 2018a; Chauhan et al., 2018; Yao et al., 2017b), the level-6 embedded intelligence defined in (Zhou et al., 2019), especially for *deep neural networks*, which we call *Embedded Deep Intelligence*.

1.3 Benefits of On-Device Learning

In fact, the offloading solutions were popular back in the days when Wireless Sensor Networks (WSNs) were deployed to collect data from the sensor nodes, only to be analyzed later on a remote base station (Shaikh and Zeadally, 2016; Akhtar and Rehmani, 2015; Shaikh and Zeadally, 2016; Lu et al., 2015). Compared to the sensor motes of those WSNs, today's embedded systems are far more advanced in terms of CPU and memory, and their energy efficiency has improved by several orders of magnitude. For instance, the latest mixed-signal microcontrollers from Texas Instruments (i.e., TI MSP430 series) comes with up to 16-bit/25 MHz CPU, 512 KB flash memory, 66 KB RAM, and 256 KB non-volatile FRAM—which are comparable to the 16-bit Intel x86 microprocessors of the early 80s which ran MS-DOS. These devices are quite capable of executing simple machine learning workloads that perform on-device classification of sensor data (Gobieski et al., 2018b) as well as training of the model. In general, there are several advantages of on-device learning over relaying data to a base station:

- *Data Transmission Cost and Latency.* Data communication between a device and a base station introduces delays and increases energy cost per bit transmission. Using back-scatter communication (Lu et al., 2018) apparently lower the energy cost, but the dependency on an external entity and the unpredictable delay in wireless communication still remains, which we want to avoid by design.

- *Privacy and Security.* Private and confidential data, such as health vitals from a wearable device, can be safely learned on-device – without exposing them to external entities. Security problems caused by side-channel and man-in-the-middle attacks (Aziz and Hamilton, 2009; Kügler, 2003) are avoided by design when we adopt on-device processing of sensitive data.

- *Precision Learning and Resource Management.* Many human-in-the-loop machine learning applications running on wearable and implantable systems benefit from run-time adaptation as different persons have different preferences and different expectations from the same application. On-device learning helps a system adjust itself at run-time to satisfy each individual's needs and to optimize its own resource management.

- *Adaptability and Lifelong Learning.* Lifelong learning Chen and Liu (2016) is an emerging concept in robotics and autonomous systems where the vision is to create intelligent machines that learn and adapt throughout their lifetime. On-device machine learning enables true lifelong learning by liberating these devices from being stationary and connected to power sources, to mobile, ubiquitous, and autonomous.

1.4 Challenges of Embedded Deep Intelligence

Unfortunately, limitations in the computational capabilities of resource-scarce embedded systems inhibit the implementation of machine learning algorithms on them, including deep learning algorithms that need large amounts of input data and substantial computational power to generate results. The major challenges of Embedded Deep Intelligence are:

- *Small Size of Memory.* State-of-the-art deep neural networks require between hundreds of KB to thousands of MB of main memory (Simonyan and Zisserman, 2014; He et al., 2016a; Szegedy et al., 2017). On the other hand, low-power SoCs and microcontrollers and state-of-the-art embedded GPUs typically contain 8KB–512MB of RAM (TexasInstruments, 2018; Holton and Fratangelo, 2012; He et al., 2016b). Hence, the maximum number of deep neural networks that can reside in the main memory is quite limited, and packing multiple learners into extremely scarce memory of an embedded system still remains an open problem.

- *Dynamic Timing Constraint.* The time constraints of many embedded systems in the real-world dynamically change at run-time, making deep neural networks more challenging to be executed as a real-time task. Such dynamic time constraints are found in many modern embedded systems such as autonomous cars (Taş et al., 2016; Pongpunwattana and Rysdyk, 2004; Shiller et al., 1991), drones (Chen et al., 2017b; Nägeli et al., 2017; Soto et al., 2007), and smartphones (He et al., 2015; Wanpeng and Wei, 2014; Balog et al., 2002) where the system with limited resources must deal with online changes such as run-time application requirements, resource availability, energy level, failures, and re-configurations. Such changes consequently cause variations in the time requirements of related-tasks (Stewart and Khosla, 1991); e.g., data-dependent requirements where the periods depend on the input sensor data; time-dependent requirements where the actual deadline becomes known only at run-time when setting the actuators.

- *Low-Computing Capability.* High performance of machine learning or deep learning algorithms requires massive computation capability to deal with complex training and inference methodologies and large datasets (LeCun et al., 2015). Although embedded systems are a good source of extensive data and appealing targets for machine learning applications, they are struggling to run machine learning algorithms due to their limited computing capability within specific limitations such as form factor size which is far behind the necessary level for many state-of-the-art learning models, e.g., low-end microcontrollers (TexasInstruments, 2018) or embedded GPUs (NVIDIA, 2019a) with limited performance.

- *Limited Energy.* Most embedded devices are powered by batteries that will die eventually, which inhibits continuous learning throughout their lifetime. Given only a fixed amount of battery power during their lifetime, they have to maximize the power efficiency to increase the duration usage as much as possible, which results in limiting the performance and throughput of learning algorithms that usually require a large amount of energy for their compute-intensive workloads.

1.5 Limitations of Existing Work

The existing approaches, such as compression and pruning of deep neural networks (Ullrich et al., 2017; Parashar et al., 2017; Han et al., 2015a; Abbasi-Asl and Yu, 2017; He et al., 2014; Hassibi and Stork, 1993) that are currently used to fit large sizes of DNNs into resource-constrained embedded systems, do not entirely solve the aforementioned challenges of Embedded Deep Intelligence for the following reasons.

- Those compression algorithms require significant re-training and fine-tuning of individual deep neural network models in order to achieve memory-saving, which is not scalable to a massive number of embedded devices.
- Their run-time execution is not flexible to adapt to the dynamic timing constraints of the system due to their fixed network architecture and computation.
- They do not provide a fundamental solution to the limited energy and computing capability. Although the size of deep neural networks can be significantly decreased by compression, their power consumption and computational workloads are barely reduced as much as their network size reduction, as shown in many existing works (Chen, 2018).

- They do not learn new data and thus do not update their DNN models at run-time but only execute an inference task of the DNN (i.e., no on-device training) based on the task pipelines fixed at compile-time.
- They do not benefit from knowledge transfer as they are trained in isolation and thus do not achieve the benefit of multitask learning that increases the robustness and generalization of multiple learners running on the same system. Although by sharing network structure between deep neural network models (typically, the first few layers), the existing multitask learning methods (Caruana, 1997; Ruder, 2017; Zhang and Yang, 2017a,b) allow deep intelligence on embedded systems (He et al., 2018a), its primary goal is to increase the performance of correlated and similar-structured learners without considering resource limitation. Thus, it does not solve executing multiple heterogeneous learners on embedded systems in a resource-efficient manner.

1.6 Thesis Statement

"Embedded systems can perform on-device deep learning, not dependent on external systems, enabling multitasking, real-time, dynamic, enhanced, and lifelong learning on the device beyond their scarce resources, i.e., insufficient memory, dynamic execution time, low-computing capability, and limited energy. Such on-device deep learning is enabled by overcoming embedded systems' resource-constraints via the proposed in-memory multitask learning, adaptive real-time learning, opportunistic accelerated learning, and energy-aware intermittent learning."

1.7 Contributions

This dissertation proposes Embedded Deep Intelligence and answers the related research challenges listed above, which need to be solved to enable high-performing machine learning algorithms, especially deep neural networks, on resource-constrained embedded systems. The contributions of this dissertation are summarized as follows:

- *On-Device Deep Learning Algorithms.* This research proposes efficient, effective, and lightweight learning algorithms of deep neural networks, which can be performed under the four fundamental constraints of embedded systems, i.e., the small size of memory, dynamic timing constraint, low-computing power, and limited energy.

- *Software Frameworks (Open-Source).* This research implements the proposed deep intelligent algorithms that can be applied to a variety of embedded platforms ranging from low-end microcontrollers to high-end embedded GPUs. The implemented frameworks is open-sourced at a public repository to facilitate the related researches.

- *Novel Hardware Prototypes.* Along with the software frameworks, this research devises new form factors of embedded sensing and inference systems, which supports Embedded Deep Intelligence from the system and hardware level.

- *Real-World Applications.* Based on the proposed algorithms, software frameworks, and hardware prototypes, this research develops and deploy real-world applications of Embedded Deep Intelligence such as an air quality monitoring system, traffic sign recognizer, voice command listener, and autonomous mobile robot.

CHAPTER 2: BACKGROUND

In this chapter, we discuss background materials and terminologies related to this dissertation. We begin by introducing the concept of deep neural networks (DNNs) and the details of convolutional neural networks (CNNs) on reference to O'Shea and Nash (2015) and Goodfellow et al. (2016), which are the primary target learning models considered in this dissertation. We then discuss the basics of embedded systems on reference to (Jiménez et al., 2013), (Lee et al., 2011), and (Berger, 2001), which are the main hardware platforms running DNNs and CNNs with their constrained resources in this dissertation.

2.1 Deep Neural Networks (DNNs)

Artificial Neural Networks (ANNs) are computational processing systems heavily inspired by the way biological nervous systems (such as the human brain) operate. DNNs are mainly comprised of a high number of interconnected computational nodes (referred to as neurons), of which work entwine in a distributed fashion to collectively learn from the input in order to optimize its final output. The basic structure of an DNN can be modeled as shown in Figure 2.1. We would load the input, usually in the form of a multidimensional vector, to the input layer, which will distribute it to the hidden layers. The hidden layers will then make decisions from the previous layer and weigh up how a stochastic change within itself detriments or improves the final output, and this is referred to as the process of learning. Having multiple hidden layers stacked upon each-other is commonly called deep neural networks (DNNs) (Goodfellow et al., 2016; O'Shea and Nash, 2015).

The two key learning paradigms in deep learning tasks are supervised and unsupervised learning.

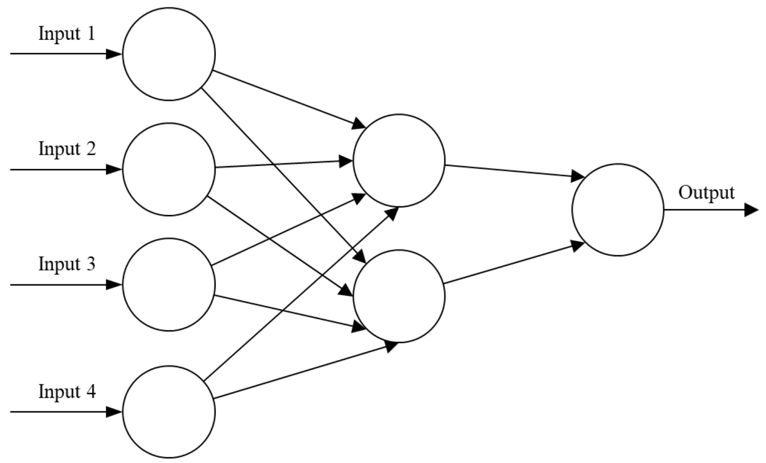


Figure 2.1: A simple three layered feed-forward neural network (FNN), comprised of a input layer, a hidden layer and an output layer. This structure is the basis of many common DNN architectures, included but not limited to Feed-forward Neural Networks (FNNs), Convolutional Neural Networks (CNNs), and Recurrent Neural Networks (RNNs). Figure from O'Shea and Nash (2015).

- Supervised learning is learning through pre-labeled inputs, which act as targets. There will be a set of input values (vectors) for each training example and one or more associated designated output values. This form of training aims to reduce the models overall classification error by correct calculation of the output value of training example by training.
- Unsupervised learning differs in that the training set does not include any labels. Success is usually determined by whether the network is able to reduce or increase an associated cost function. However, it is important to note that most pattern-recognition tasks usually depend on classification using supervised learning.

2.2 Convolutional Neural Networks (CNNs)

Convolutional Neural Networks (CNNs) are analogous to traditional DNNs in that they are comprised of neurons that self-optimize through learning. Each neuron will still receive input and perform an operation (such as a scalar product followed by a non-linear function) - the basis of countless DNNs. From the input raw image vectors to the final output of the class score, the entire network will still express a single perceptive score function (the weight). The last layer will contain loss functions associated with the classes, and all of the regular tips and tricks developed for traditional DNNs still apply.

The only notable difference between CNNs and traditional DNNs is that CNNs are primarily used in the field of pattern recognition within images. This allows us to encode image-specific features into the architecture, making the network more suited for image-focused tasks - while further reducing the parameters required to set up the model.

One of the largest limitations of traditional forms of DNN is that they tend to struggle with the computational complexity required to compute image data. Common machine learning benchmarking datasets such as the MNIST dataset (LeCun et al., 1998) of handwritten digits are suitable for most forms of DNN, due to its relatively small image dimensionality of just 28x28. With this dataset a single neuron in the first hidden layer will contain 784 weights ($28\times28\times1$ where 1 bare in mind that MNIST is normalised to just black and white values), which is manageable for most forms of DNN.

If you consider a more substantial colored image input of 64×64, the number of weights on just a single neuron of the first layer increases substantially to 12,288. Also, take into account that to deal with this scale of input, the network will also need to be a lot larger than one used to classify color-normalized MNIST digits, then you will understand the drawbacks of using such models.

As noted earlier, CNNs primarily focus on the basis that the input will be comprised of images. This focuses the architecture on being set up in a way to best suit the need for dealing with the specific type of data.

One of the key differences is that the neurons that the layers within the CNN are comprised of neurons organized into three dimensions, the spatial dimensionality of the input (height and the width) and the depth. The depth does not refer to the total number of layers within the DNN, but the third dimension of an activation volume. Unlike standard ANNS, the neurons within any given layer will only connect to a small region of the layer preceding it. In practice, this would mean that for the example given earlier, the input 'volume' will have a dimensionality of

64×64×3 (height, width, and depth), leading to a final output layer comprised of dimensionality of 1×1×n (where n represents the possible number of classes) as we would have condensed the full input dimensionality into a smaller volume of class scores filed across the depth dimension.

CNNs are comprised of three types of layers. These are convolutional layers, pooling layers, and fully-connected layers. When these layers are stacked, a CNN architecture has been formed. A simplified CNN architecture for MNIST classification is illustrated in Figure 2.2.

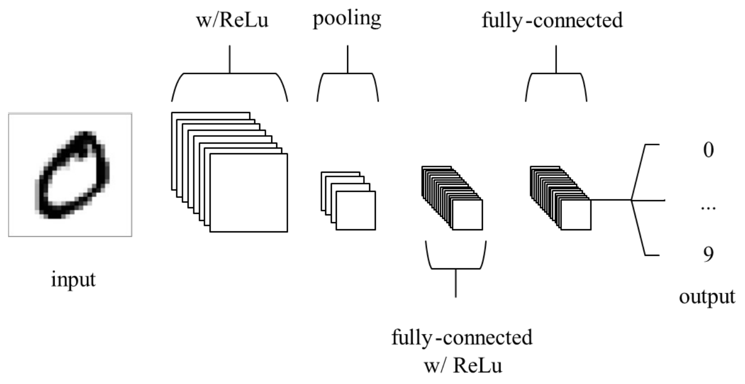


Figure 2.2: A simple CNN architecture, comprised of just five layers. Figure from O'Shea and Nash (2015).

The basic functionality of the example CNN above can be broken down into four key areas.

- As found in other forms of ANN, the input layer will hold the image's pixel values.
- The convolutional layer will determine the output of neurons connected to local regions of the input by calculating the scalar product between their weights and the region connected to the input volume. The rectified linear unit (commonly shortened to ReLu) aims to apply an 'elementwise' activation function such as sigmoid to the output of the activation produced by the previous layer.
- The pooling layer will then simply perform downsampling along with the given input's spatial dimensionality, further reducing the number of parameters within that activation.

- The fully-connected layers will then perform the same duties found in standard ANNs and attempt to produce class scores from the activations to be used for classification. It is also suggested that ReLu may be used between these layers to improve performance.

Through this simple transformation method, CNNs can transform the original input layer by layer using convolutional and downsampling techniques to produce class scores for classification and regression purposes.

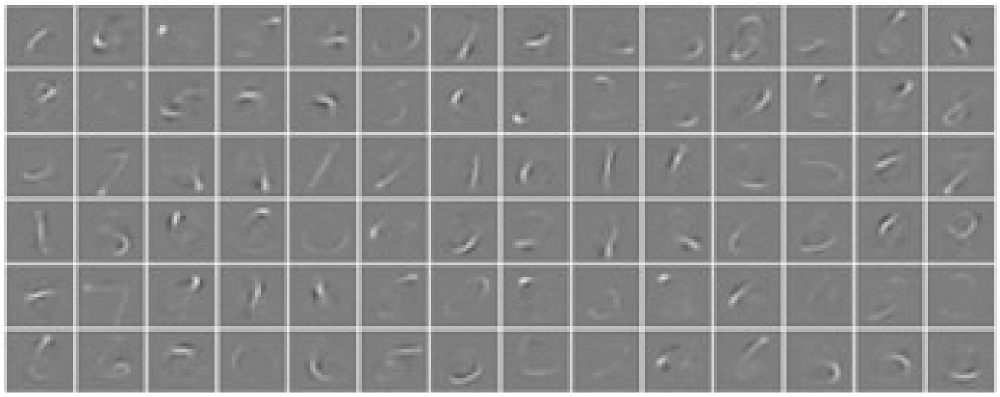


Figure 2.3: Activations taken from the first convolutional layer of a simplistic deep CNN, after training on the MNIST dataset of handwritten digits. You can see that the network has successfully picked up on characteristics unique to specific numeric digits if you look carefully. Figure from O'Shea and Nash (2015).

However, it is important to note that simply understanding the overall architecture of a CNN architecture will not suffice. The creation and optimization of these models can take quite some time and can be quite confusing. We will now explore in detail the individual layers, detailing their hyperparameters and connectivities.

2.3 Basics of Embedded Systems

An embedded system can be broadly defined as a device that contains tightly coupled hardware and software components to perform a single function, forms part of a larger system, is not intended to be independently programmable by the user, and is expected to work with minimal or no human interaction. Two additional characteristics are very common in embedded systems: reactive operation and heavily constrained (Jiménez et al., 2013).

Most embedded systems interact directly with processes or the environment, making decisions on the fly based on their inputs. This makes it necessary that the system be reactive, responding in real-time to process inputs to ensure proper operation. Besides, these systems operate in constrained environments where memory, computing power, and power supply are limited. Moreover, production requirements, in most cases due to volume, place high-cost constraints on designs.

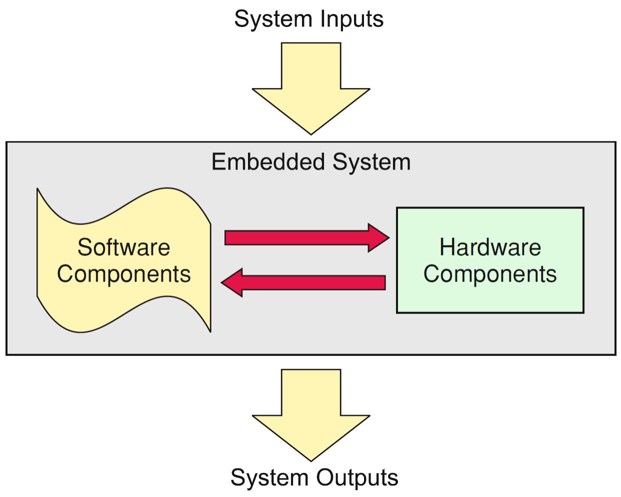


Figure 2.4: General view of an embedded system. Figure from Jiménez et al. (2013).

2.3.1 Structure of Embedded Systems

Regardless of the function performed by an embedded system, the broadest view of its structure reveals two major, tightly coupled sets of components: a set of hardware components that include a central processing unit, typically in the form of a microcontroller; and a series of software programs, typically included as firmware that gives functionality to the hardware. Figure 2.4 depicts this general view, denoting these two major components and their interrelation. Typical inputs in an embedded system are process variables and parameters that arrive via sensors and input/output (I/O) ports. The outputs are in the form of control actions on system actuators or processed information for users or other subsystems within the application. In some instances, input-output information exchange occurs with users via a user interface that might include keys and buttons, sensors, light-emitting diodes (LEDs), liquid crystal displays (LCDs), and other types of display devices, depending on the application.

The software is the most abstract part of the system and as essential as the hardware itself. It includes the programs that dictate the sequence in which the hardware components operate. When someone decides to prepare a pre-programmed meal in a microwave oven, the software picks the keystrokes in the oven control panel, identifies the user selection, decides the power level and cooking time, initiates and terminates the microwave irradiation on the chamber, the plate rotation, and the audible signal letting the user know that the meal is ready. While the meal is cooking, the software monitors the meal temperature and adjusts power and cooking time while also verifying the correct operation of the internal oven components. In the case of detecting a system malfunction, the program aborts the oven operation to prevent catastrophic consequences. Despite our choice of describing this example from a system-level perspective, the tight relation between application, hardware, and software becomes evident. In the sections below, we take a closer view of the hardware and software components that integrate an embedded system.

Hardware Components. When viewed from a general perspective, an embedded system's hardware components include all the electronics necessary for the system to perform the function it was designed for. Therefore, a particular system's specific structure could substantially differ from another, based on the application itself. Despite these dissimilarities, three core hardware components are essential in an embedded system (Figure 2.5): The Central Processing Unit (CPU), the system memory, and a set of input-output ports. The CPU executes software instructions to process the system inputs and make decisions that guide the system operation. Memory stores programs and data necessary for system operation. Most systems differentiate between program and data memories. The program memory stores the software programs executed by the CPU. Data memory stores the data processed by the system. The I/O ports allow conveying signals between the CPU and the world external to it. Beyond this point, a number of other supporting and I/O devices needed for system functionality might be present, depending on the application.

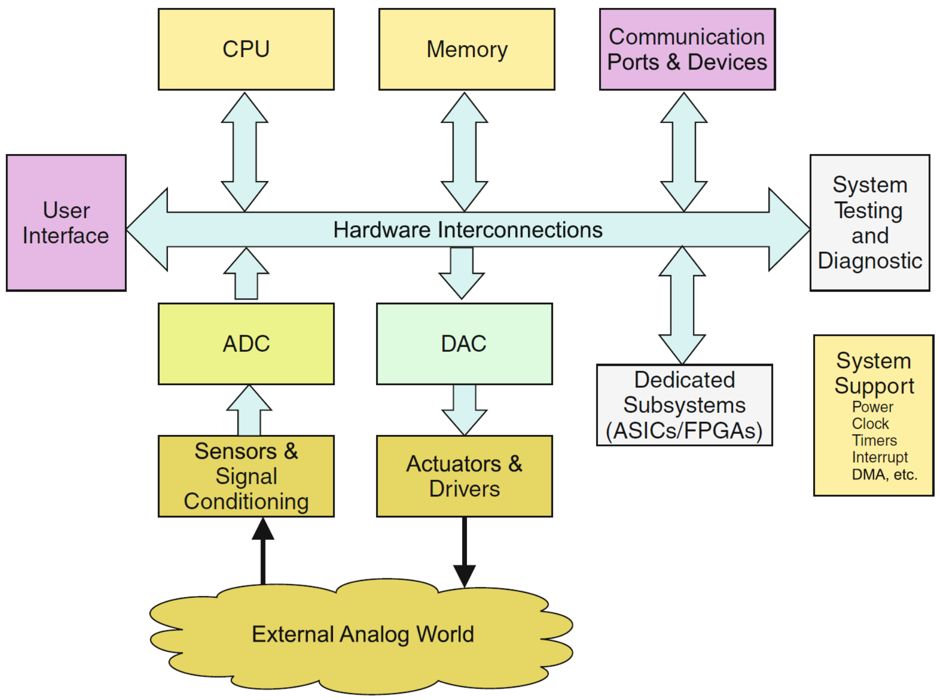

CPU
Memory
Communication Ports & Devices
User Interface
Hardware Interconnections
System Testing and Diagnostic
ADC
DAC
Dedicated Subsystems (ASICs/FPGAs)
System Support
Power
Clock
Timers
Interrupt
DMA, etc.
Sensors & Signal Conditioning
Actuators & Drivers
External Analog World

Figure 2.5: Hardware elements in an embedded system. Figure from Jiménez et al. (2013).

Software Components. The software components of an embedded system include all the programs necessary to give functionality to the system hardware. These programs, frequently referred to as the system firmware, are stored in some non-volatile memory. Firmware is not meant to be modifiable by users, although some systems could provide means of performing upgrades. System programs are organized around some form of operating system and application routines. The operating systems can be simple and informal in small applications, but as the application complexity grows, the operating system requires more structure and formality. In some of these cases, designs are developed around Real-Time Operating Systems (RTOS). Figure 2.6 illustrates the structure of embedded system software. The major components identified in a system software include:

- *System Tasks.* The application software in embedded systems is divided into a set of smaller programs called Tasks. Each task handles a distinct action in the system and requires the use of specific System Resources. Tasks submit service requests to the kernel in order to perform their designated actions. In our microwave oven example, the system operation

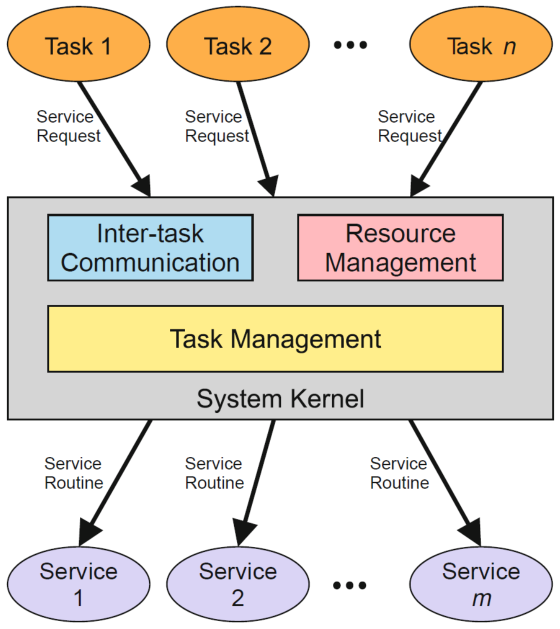


Figure 2.6: Hardware elements in an embedded system. Figure from Jiménez et al. (2013).

can be decomposed into a set of tasks that include reading the keypad to determine user selections, presenting information on the oven display, turning on the magnetron at a certain power level for a certain amount of time, just to mention a few. Service requests can be placed via registers or interrupts.

- *System Kernel.* The software component that handles the system resources in an embedded application is called the Kernel. System resources are all those components needed to serve tasks. These include memory, I/O devices, the CPU itself, and other hardware components. The kernel receives service requests from tasks, and schedules them according to the priorities dictated by the task manager. When multiple tasks contend for a common resource, a portion of the kernel establishes the resource management policy of the system. It is not uncommon finding tasks that need to exchange information among them. The kernel provides a framework that enables reliable inter-task communication to exchange information and to coordinate collaborative operation.

- *Services.* Tasks are served through Service Routines. A service routine is a piece of code that gives functionality to a system resource. In some systems, they are referred to as device drivers. Services can be activated by polling or as interrupt service routines (ISR), depending on the system architecture.

2.3.2 Classification of Embedded Systems

The three pioneering microprocessor developments at the beginning of the 1970s, besides initiating the modern era of embedded systems, inadvertently created two defining categories that we can use to classify embedded systems in general: Small and High-performance.

Small Embedded Systems. The MSP430 microcontroller series (TexasInstruments, 2018), which has become the cornerstone component of this type of embedded systems, which is by far, the most common type. This class is typically centered around a single microcontroller chip that commands the whole application. These systems are highly integrated, adding only a few analog components, sensors, actuators, and user-interface, as needed. These systems operate with minimal or no maintenance, are very low cost, and are produced in mass quantities. The software in these systems is typically single-tasked and rarely requires an RTOS. Examples of these systems include tire pressure monitoring systems, microwave oven controllers, toaster controllers, and electronic toy controllers, to mention just a few.

High-Performance Embedded Systems. This type of embedded system represents the class of highly specialized embedded systems requiring fast computations, robustness, fault tolerance, and high maintainability. These systems usually require dedicated GPUs NVIDIA (2019a) or ASICS and might include DSPs and FPGAs as part of the basic hardware. In many cases, the complexity of their software makes mandatory the use of RTOS' to manage the multiplicity of tasks. They are produced in small quantities, and their cost is very high. These are the type of embedded systems used in military or aerospace applications, such as flight controllers, missile guidance systems, and spacecraft navigation systems. The categories in this classification are not mutually exclusive. Among them, we can find "gray zones" where the characteristics of two or the three of them overlap, and applications might become difficult to associate to a single specific class. However, if we look at the broad range of embedded applications, in most cases, it becomes generally easy to identify the class to which a particular application belongs.

2.3.3 Design Constraints

A vast majority of embedded systems applications end up in the heart of mass-produced electronic applications. Home appliances such as microwave ovens, toys, and dishwasher machines, automobile systems such as anti-lock brakes and airbag deployment mechanisms, and personal devices such as cellular phones and media players are only a few representative examples. These are systems with a high-cost sensitivity to the resources included in a design due to the high volumes in which they are produced. Moreover, designs need to be completed, manufactured, and launched in time to hit a market window to maximize product revenues. These constraints shape the design of embedded applications from beginning to end in their life cycle. Therefore, the list of constraints faced by designers at the moment of conceiving an embedded solution to a problem comes from different perspectives. The most salient constraints in the list include:

- *Functionality.* Every embedded system design is expected to have a functionality that solves the problem it was designed for. More than a constraint, this is a design requirement.

- *Performance.* Performance in embedded systems usually refers to the system's ability to perform its function on time. Therefore, a measure of the number of operations per unit time will somehow always be involved. Sometimes, performance is associated with issues such as power consumption, memory usage, and even cost.

- *Power and Energy.* Power in embedded systems has become a critical constraint, not only in portable, battery-operated systems, but for every system design. The average power dissipation of an embedded design defines the rate at which the system consumes energy. In battery-powered applications, this determines how long it takes to deplete the capacity of its batteries. But aside from battery life, power affects many other issues in embedded systems design.

- *Reliability and Maintainability.* Maintainability in embedded systems can be defined as a property that allows the system to be acted upon, to guarantee a reliable operation throughout the end of its useful life. This property can be regarded as a design constraint because, for maintainability to be enabled, it has to be planned from the system conception itself. The maintainability constraint can have different levels of relevance depending on the type of embedded system being considered.
- *Size.* Physical space taken by a system solution.
- *Cost.* The amount of resources needed to conceive, design, produce, maintain, and discard an embedded system.

CHAPTER 3: LEARNING STRATEGIES FOR EMBEDDED DEEP INTELLIGENCE

The goal of this dissertation is to design and implement *Embedded Deep Intelligence* algorithms and the related hardware which will enable on-device learning, including both inference and training, under four significant constraints commonly imposed on many embedded systems, which make on-device learning challenging, i.e., *the small size of memory, dynamic timing constraint, low-computing capability, and limited energy*. We define each constraint as an independent research problem and conduct an in-depth study to provide a novel solution, i.e., *in-memory multitask learning, adaptive real-time learning, opportunistic accelerated learning, and energy-aware intermittent learning*, as illustrated in Figure 3.1. The following sections provide an overview of each proposed research.

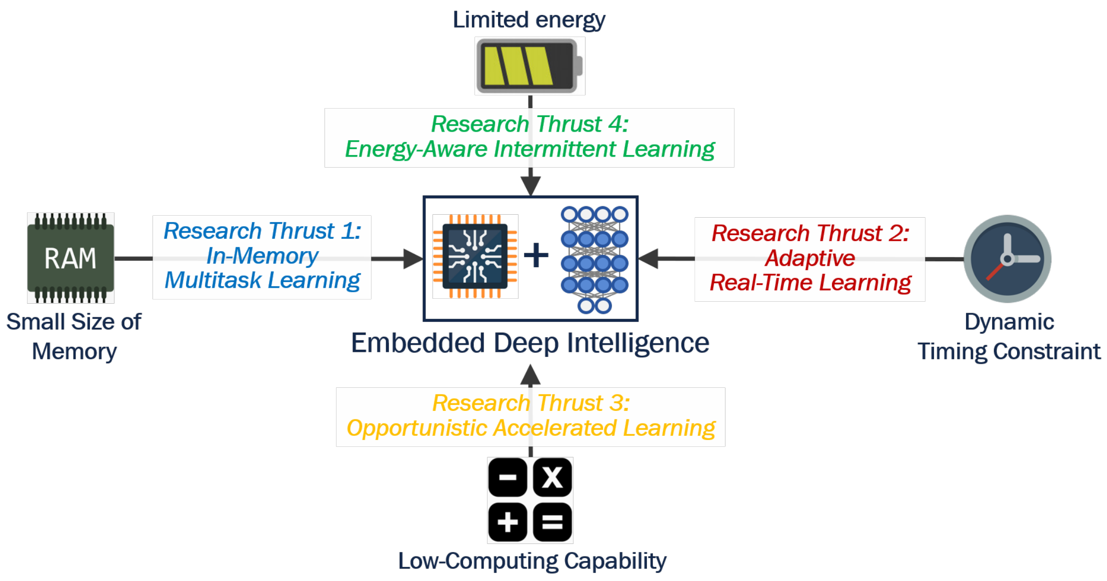


Figure 3.1: An overview of the proposed research: Embedded Deep Intelligence consists of four dimensions of resource-aware learning algorithm, i.e., in-memory multitask learning, adaptive real-time learning, opportunistic accelerated learning, and energy-aware intermittent learning, which enables on-device deep learning on resource-constrained embedded systems.

3.1 In-Memory Multitask Learning

We propose *in-memory multitask learning* based on the concept of *Neural Weight Virtualization* (Lee and Nirjon, 2020a) – which enables fast and scalable in-memory multitask deep learning on memory-constrained embedded intelligent systems. The goal of neural weight virtualization is two-fold: 1) packing multiple DNNs into a fixed-sized main memory whose combined memory requirement is larger than the main memory, and 2) enabling fast in-memory execution of the DNNs. To this end, we propose a two-phase approach: 1) *virtualization of weight parameters* for fine-grained parameter sharing at the level of weights that scales up to multiple heterogeneous DNNs of arbitrary network architectures, and 2) in-memory data structure and run-time execution framework for *in-memory execution and context-switching* of DNN tasks. We implement two multitask learning systems: 1) an embedded GPU-based mobile robot, and 2) a microcontroller-based IoT device. We thoroughly evaluate the proposed algorithms as well as the two systems that involve ten state-of-the-art DNNs. Our evaluation shows that weight virtualization improves memory efficiency, execution time, and energy efficiency of the multitask learning systems by 4.1x, 36.9x, and 4.2x, respectively.

Our approach to in-memory deep multitask learning is to virtualize a portion of the main memory, which stores a carefully generated set of constant numbers that represent the weight parameters of one or more DNNs. We call this *weight virtualization* as opposed to virtualization of main memory since the memory locations, along with their content (offline-computed fixed numbers representing DNN weights) are virtualized. For example, a memory block, B_0, may simultaneously represent K consecutive weights of the L_i^{th} layer of the first DNN as well as K consecutive weights of the L_j^{th} layer of another DNN. Weight virtualization requires us to find a set of values to be stored in the main memory such that 1) each block of memory represents a block of weights of one or more DNNs, and 2) significant weights (if not all) of all DNNs are mapped to a weight page.

Chapter 4 describes the proposed in-memory multitask learning in detail.

3.2 Adaptive Real-Time Learning

We propose *adaptive real-time learning* based on the concept of *SubFlow* (Lee and Nirjon, 2020c)—a dynamic adaptation and execution strategy for a deep neural network (DNN), which enables real-time DNN inference and training on embedded systems of limited computing resources. The goal of SubFlow is to complete the execution of a DNN task within a timing constraint, which may dynamically change while ensuring comparable performance to executing the full network by executing a subset of the DNN at run-time. To this end, we propose two online algorithms that enable SubFlow: 1) *dynamic construction* of a sub-network which constructs the best sub-network of the DNN in terms of size and configuration, and 2) *time-bound execution* which executes the sub-network within a given time budget for both inference and training.

We implement and open-source SubFlow by extending TensorFlow with full compatibility by adding SubFlow operations for convolutional and fully-connected layers of a DNN. We evaluate SubFlow with three popular DNN models (LeNet-5, AlexNet, and KWS), which shows that it provides flexible run-time execution and increases the utility of a DNN under dynamic timing constraints, e.g., 1x–6.7x range of execution times with average -3% of performance (inference accuracy) difference. We also implement an autonomous robot as an example system that uses SubFlow and demonstrate that its obstacle detection DNN is flexibly executed to meet a range of deadlines that varies depending on its running speed.

SubFlow enables the execution of DNN inference and training tasks in such a way that the task is completed under dynamically varying time constraints while retaining comparable performance to executing the original full-size DNN. The flexible execution increases the utility of a DNN by letting it meet a range of deadlines at run-time, which conventional DNNs cannot. SubFlow also facilitates flexible scheduling of multitask learning where new tasks can be accommodated by dynamically updating the deadline of existing ones. The schedulability of a system running multiple DNNs can be improved by taking into account the flexible execution time of DNNs in the scheduling decision at run-time, which increases the total system utilization.

Chapter 5 describes the proposed adaptive real-time learning in detail.

3.3 Opportunistic Accelerated Learning

We propose *opportunistic accelerated learning* based on the concept of *Neuro.ZERO* (Lee and Nirjon, 2019)—a co-processor architecture consisting of a main microcontroller (MCU) that executes scaled-down versions of a deep neural network[1] (DNN) inference task, and an accelerator microcontroller that is powered by harvested energy and follows the intermittent computing paradigm Lucia et al. (2017). The goal of the accelerator is to enhance the inference performance of the DNN that is running on the main microcontroller. Neuro.ZERO opportunistically accelerates the run-time performance of a DNN via one of its four acceleration modes: *extended inference*, *expedited inference*, *ensemble inference*, and *latent training*. To enable these modes, we propose two sets of algorithms: 1) *energy and intermittence-aware DNN inference and training algorithms*, and 2) *a fast and high-precision adaptive fixed-point arithmetic* that beats existing floating-point and fixed-point arithmetic in terms of speed and precision, respectively, and achieves the best of both.

To evaluate Neuro.ZERO, we implement low-power image and audio recognition applications and demonstrate that their inference speedup increases by $1.6\times$ and $1.7\times$, respectively, and the inference accuracy increases by 10% and 16%, respectively, when compared to battery-powered single-MCU systems.

Chapter 6 describes the proposed opportunistic accelerated learning in detail.

3.4 Energy-Aware Intermittent Learning

In order to realize embedded intelligent systems that perform lifelong learning in a prolonged period of time without the concern for the limited battery capacity, we propose *energy-aware intermittent learning* (Lee et al., 2019) that makes energy-harvested batteryless systems capable of

[1]The Deep Neural Network (DNN), by definition, refers to neural networks having more than one hidden layers Hanin (2017); Lu et al. (2017); Hornik (1991); Lee and Nirjon (2019). Thus, a wide variety of networks qualify as a DNN in the existing literature. DNNs considered in this study have up to 10^5 neurons and weights combined. They fit into 256KB memory of an MCU; have convolutional, ReLU, pooling, and fully-connected structures as regular DNNs; and perform on-device inference Gobieski et al. (2019a, 2018c).

executing lightweight machine learning tasks intermittently based on the availability of harvested energy. The notion of intermittent learning is similar to the intermittent computing paradigm with the primary difference that the program that runs on the microcontroller executes a machine learning task—involving both *training* and *inferring*.

Although it may appear to be that all machine learning tasks are merely pieces of codes that could very well be run on platforms that support intermittent computing, for several reasons, a machine learning task in an intermittent computing setup is quite different. The fundamental difference between a machine learning task and a typical task on a batteryless system (e.g., sensing and executing an offline-trained classifier) lies in the data and application semantics, which requires special treatment for effective learning under an extreme energy budget. Existing works on intermittent computing address important problems, such as ensuring atomicity Maeng et al. (2017); Colin and Lucia (2016), consistency Maeng et al. (2017); Colin and Lucia (2016); Lucia and Ransford (2015), programmability Hester et al. (2017), timeliness Hester et al. (2017), and energy-efficiency Colin et al. (2018); Hester et al. (2015b); Buettner et al. (2011), which enable efficient code execution of general-purpose tasks. We propose to complement existing literature and specialize in a batteryless system on efficient and effective on-device learning by explicitly considering the *utility of sensor data* and *the execution order of different modules* of a machine learning task.

To complement and advance the state-of-the-art of the batteryless machine learning systems, we propose the intermittent learning framework which explicitly takes into account the dynamics of a machine learning task, in order to improve the energy and learning efficiency of an intermittent learner in a systemic fashion. The fundamental difference between the proposed framework and the existing literature is that, besides *improving the efficiency of on-device inference*, the intermittent learning framework enables *on-device training* to improve the effectiveness and accuracy of the learner over time.

Chapter 7 describes the proposed energy-aware intermittent learning in detail.

CHAPTER 4: IN-MEMORY MULTITASK LEARNING

As deep learning algorithms for resource-constrained systems continue to become more efficient and more accurate (Yao et al., 2017b, 2018b, 2017a), we see an increased number of intelligent embedded systems such as home IoT devices, social robots, and the plethora of portable, wearable, and mobile devices that are feature-packed with a wide variety of machine learning tasks running on the same device (Taniguchi et al., 2018; Kawsar et al., 2018; Ota et al., 2017; Billinghurst and Starner, 1999; Bariya et al., 2018; Majumder et al., 2017). Home hubs like Amazon Echo Show and Google Nest Hub nowadays are performing speech recognition (Google, 2019c; Kim et al., 2017), speaker identification (Google, 2019f), gesture recognition (Google, 2019b), face recognition (Google, 2019d), facial expression and emotion recognition (Google, 2019a) in order to closely imitate human assistants. Similar classifiers are running on social, domestic, and personal robots (Spyridon and Eleftheria, 2012; Prassler and Kosuge, 2008; Gates, 2007; Bohren et al., 2011; Siegwart et al., 2003), which also execute robotic application-specific learning tasks such as object recognition (Maturana and Scherer, 2015; Redmon and Angelova, 2015), obstacle detection (Xie et al., 2017), scene understanding (Liao et al., 2016), self-localization (Sarikaya et al., 2017), and navigation (Bojarski et al., 2016; Giusti et al., 2015). While a naive approach to enable multiple classifiers on a device would be to train and execute each classifier independently, state-of-the-art multitask learning approaches suggest jointly training more than one correlated task in order to increase the accuracy of each learner by exploiting the commonalities and differences across different tasks (Caruana, 1997; Ruder, 2017; Zhang and Yang, 2017a,b).

Unfortunately, multitask deep learning on mobile and embedded systems is not quite as effective as running the classifiers on a high-end machine due to their limited CPU and memory.

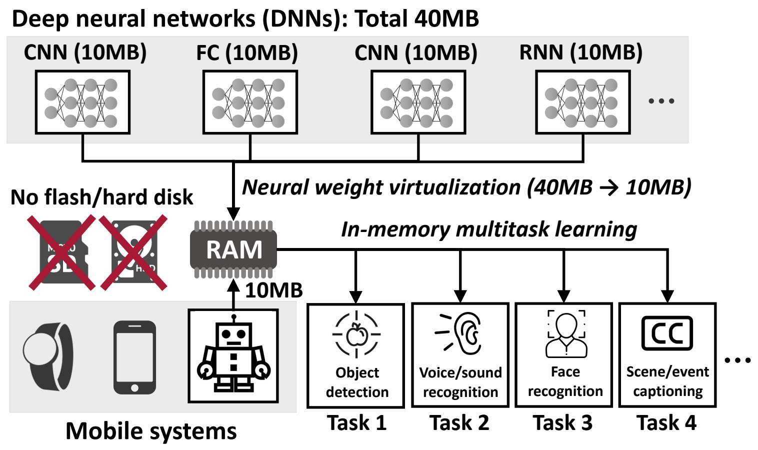


Figure 4.1: Neural weight virtualization packs multiple DNNs into the main memory of a system where the total size of the DNNs is larger than the capacity of the main memory (40MB vs. 10MB). It performs complete *in-memory* storage and execution of DNNs at run-time, which enables fast and real-time multitask learning on resource-constrained systems.

Using powerful processors and/or larger (or external) memory is not feasible in these systems due to cost, space, heating, latency, and design constraints (Henzinger and Sifakis, 2006). In general, lack of *scalability* and sluggish *response time* are the two major challenges to effective deep multitask learning on CPU and memory-constrained embedded systems:

• *Scalable Packing.* State-of-the-art DNNs require between hundreds of KB to hundreds of MB of main memory (Simonyan and Zisserman, 2014; He et al., 2016a; Szegedy et al., 2017). On the other hand, state-of-the-art embedded GPUs and low-power SoCs and MCUs typically contain 8KB–512MB of RAM (TexasInstruments, 2018; Holton and Fratangelo, 2012; He et al., 2016b). Hence, the maximum number of DNNs that can reside in the main memory is quite limited. Commonsense approaches such as compression and pruning (Ullrich et al., 2017; Parashar et al., 2017; Han et al., 2015a; Abbasi-Asl and Yu, 2017; He et al., 2014; Hassibi and Stork, 1993) DNNs do not quite solve the problem since these algorithms are applied on each DNN separately, they require significant fine-tuning, and more importantly, as opposed to multitask learning, these networks do not benefit from knowledge transfer as they are trained in isolation. Although by sharing network structure (typically, the first few layers), multitask learning achieves limited compression (He et al., 2018a), its primary goal is to increase robustness and generalization of *correlated* and *similar-structured* learners. Thus, packing multiple *heterogeneous* learners into extremely scarce memory of an embedded system still remains an open problem.

• *Low-latency Context Switching.* A practical limitation of multitask learning *systems* in the wild – which is often overlooked by the main-stream deep multitask learning literature – is the overhead of switching DNN tasks at run-time. In memory-constrained multitask learning systems, where some of the DNN models must reside in the flash or the hard disk, context switching overhead is extremely high as memory operations are typically 10–100x faster than accessing flash or hard disks, and DNN models are large. Hence, the overhead of frequent swapping in and out of DNNs to and from the main memory causes severe latency, which in turn, degrades the responsiveness and usability of the system.

To address these challenges, we introduce the concept of *Neural Weight Virtualization* – which treats *consecutive memory locations containing weights of neural networks* as resources that can be virtualized, and thus, shared by more than one DNN. Weight virtualization enables scalable packing of *any*[1] number of DNNs into the main memory while achieving the fastest possible deep multitask learning on an embedded system that incurs near-zero context switching overhead due to complete in-memory storage and execution. An illustration of weight virtualization is shown in Figure 4.1, where four DNNs, requiring a total of 40MB memory, are packed into 10MB RAM of a mobile system.

Packing multiple DNNs into the main memory via weight virtualization is motivated by empirical observations that *only a small fraction of DNN weights have significant impacts on the inference result*, and *these high-significance weights are concentrated in a few blocks in the main memory*. This guides us in designing a scalable DNN packing algorithm that matches similar blocks of weights across multiple DNNs and combines them to construct a single new block of *virtual* weights that is shared by the DNNs. The matching process is followed by an optimization (retraining) process to gain back any loss of the inference accuracy of the DNNs. Efficient

[1]Although the proposed approach can pack an unlimited number of DNNs, ensuring a minimum accuracy for each DNN does impose a limit on this number–which is still larger than alternatives such as (Mallya and Lazebnik, 2018; Chou et al., 2018a; Kaiser et al., 2017; Aytar et al., 2017; Levi and Hassner, 2015; He et al., 2018a; Ma et al., 2019; Misra et al., 2016; Ma et al., 2018).

in-memory data structures and run-time framework for task management, execution, scheduling, and context-switching have been implemented to support scalable and fast in-memory deep multi-task learning on embedded systems. The approach is architecture agnostic – it applies to any type of DNN, including fully-connected, convolutional, and recurrent layers.

We implement two deep multitask learning systems involving ten state-of-the-art DNNs: 1) a mobile robot that executes six DNNs (i.e., MobileNet (Sandler et al., 2018; Howard et al., 2017), FaceNet (Schroff et al., 2015), Place205 (Zhou et al., 2014b), UrbanSound8K (Salamon et al., 2014), GSC (Warden, 2018), and ShowAndTell (Vinyals et al., 2015)) on Jetson Nano embedded GPU platform (NVIDIA, 2019a)), and 2) an extremely resource-constrained IoT device having an MSP430 (TexasInstruments, 2018) microcontroller that executes five compressed DNNs (i.e., GTSRB (Stallkamp et al., 2011a), GSC (Warden, 2018), SVHN (Netzer et al., 2011), MNIST (LeCun et al., 1998), and CIFAR-10 (Krizhevsky et al., 2009)). Our experimental results demonstrate that with weight virtualization, these systems successfully packs all DNNs with a 4x compression ratio while achieving an improved latency of 36x and an improved energy efficiency of 4.34x due to in-memory operations and multitask learning. We have made our software open-source at a public repository[2].

4.1 Overview

The general idea of virtualization technology is to create the illusion of having an extended resource given a limited physical resource – such as hardware, storage, and networks (Plessl and Platzner, 2004; Song and Hai, 2003; Chowdhury and Boutaba, 2010). We formulate the *in-memory multitask learning problem* into a virtualization problem where multiple DNNs must reside in the same fixed-sized RAM. We assume that the main memory is at least as large as the largest DNN, but we do not impose any limit on the total memory required by all DNNs. We further assume that the DNNs are already trained, and they can be of any arbitrary network structures.

[2]`https://github.com/learning1234embed/NeuralWeightVirtualization`

Our approach to in-memory deep multitask learning is to virtualize a portion of the main memory, which stores a carefully generated set of constant numbers that represent the weight parameters of one or more DNNs. We call this *weight virtualization* as opposed to virtualization of main memory since the memory locations, along with their content (offline-computed fixed numbers representing DNN weights) are virtualized. For example, a memory block, B_0, may simultaneously represent K consecutive weights of the L_i^{th} layer of the first DNN as well as K consecutive weights of the L_j^{th} layer of another DNN. Weight virtualization requires us to find a set of values to be stored in the main memory such that 1) each block of memory represents a block of weights of one or more DNNs, and 2) significant weights (if not all) of all DNNs are mapped to a weight page.

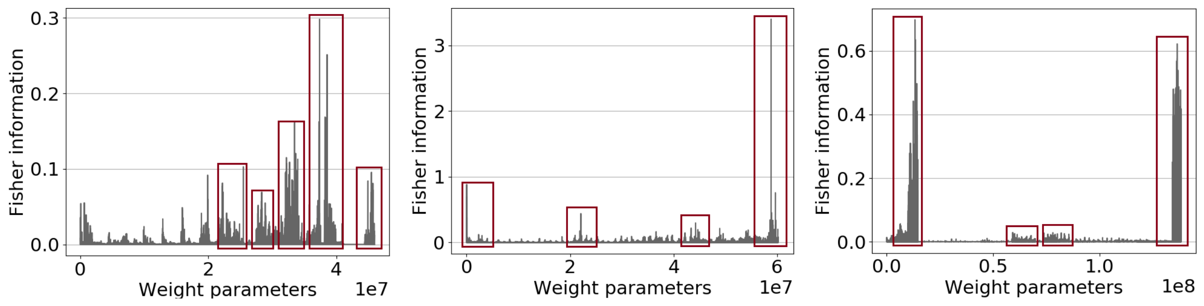
4.1.1 Feasibility of Weight Virtualization

Observations. Enabling weight virtualization to pack multiple DNNs in the main memory is motivated by two key observations on the *significance* of weight parameters towards the classification result of a DNN. While studying 13 popular DNNs used in state-of-the-art audio and visual learning tasks, we observe the following:

• *Disparity in weights' significance is globally sparse.* It is known that the significance (aka sensitivity or importance) of different weights of a neural network toward the classification result is different (LeCun et al., 1990; Han et al., 2015a). When the significance is quantified (e.g., using Fisher information (Lehmann and Casella, 2006)), we observe a large disparity — only a fraction of the weight parameters' significance is markedly higher than the rest. This happens primarily due to the inherent redundancy in most state-of-the-art DNN models (Cheng et al., 2017; Han et al., 2015a). This observation hints us that only a small fraction of high-significance weights per DNN must be stored unaltered in the main memory, while the rest can be altered and stored if there is room.

• *Disparity in weights' significance is locally dense.* Although only a fraction of the weight parameters is of high significance, they tend to be located in the vicinity of each other. In other words, when a DNN's weights are stored in the computer memory, a high-significance weight is more likely to be located next to another high-significance weight, and the same is true for low-significance weights. This is analogous to human brains, where only the neurons from a specific locality get activated by certain stimuli or tasks. This observation hints us that when sharing weights among multiple DNNs, we can consider an entire block of memory for possible sharing (which we call a *weight-page*), as opposed to bookkeeping each memory cell individually, and thus, expedite the memory sharing and management process.

Empirical Evidence. Of the 13 DNNs we studied, Figure 4.2 shows the significance score (Fisher information) of individual weights of three popular DNNs[3] on ImageNet classification (Deng et al., 2009), i.e.,— Inception-v4 (Szegedy et al., 2017), ResNet-152 (He et al., 2016a), and VGG-16 (Simonyan and Zisserman, 2014). The weights are listed in the order of layers, i.e., the weights of the first layer are followed by the second layer's, and so on.



(a) Inception-v4 (b) ResNet-152 (c) VGG-16

Figure 4.2: The importance of individual weight parameter to the inference accuracy measured by Fisher information (Lehmann and Casella, 2006): Inception-v4 (Szegedy et al., 2017), ResNet-152 (He et al., 2016a), and VGG-16 (Simonyan and Zisserman, 2014).

As shown in the figure, a few groups of neighboring weights (marked with red boxes) dominate in terms of their significance. For instance, the top 5% weights contribute to 90% Fisher score in Inception-v4, i.e., 95% of its memory space can be used by other DNNs if need be while retaining 90% accuracy of Inception-v4. Table 4.1 shows the percentages of weights of the 13 DNNs required to attain certain amounts of significance score.

[3]The remaining DNNs follow the same trend and are used in the evaluation section.

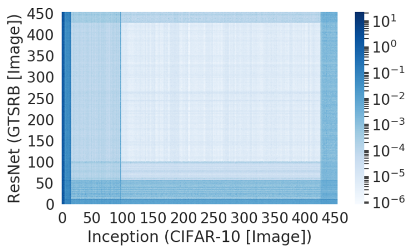
DNN	**Dataset**	**≥70%**	**≥80%**	**≥90%**	**≥95%**	**≥99%**
Inception-v4 (Szegedy et al., 2017)	**ImageNet** (Deng et al., 2009)	1.0%	2.1%	5.0%	9.3%	23.1%
Inception-ResNet-v2 (Szegedy et al., 2017)	**ImageNet** (Deng et al., 2009)	0.06%	0.19%	0.9%	3.2%	14.9%
RestNet-152 (He et al., 2016a)	**ImageNet** (Deng et al., 2009)	2.5%	5.0%	11.1%	18.9%	38.5%
VGG-16 (Simonyan and Zisserman, 2014)	**ImageNet** (Deng et al., 2009)	0.4%	0.7%	1.7%	2.8%	5.8%
PNASNet-5 (Liu et al., 2018a)	**ImageNet** (Deng et al., 2009)	0.1%	0.3%	1.4%	4.7%	20.2%
MobileNet-v2 (Sandler et al., 2018)	**ImageNet** (Deng et al., 2009)	0.3%	0.5%	0.8%	1.2%	2.0%
AlexNet (Krizhevsky et al., 2012)	**CIFAR-10** (Krizhevsky et al., 2009)	0.1%	0.4%	3.0%	8.9%	29.6%
GoogleNet (Wang et al., 2015)	**Place205** (Zhou et al., 2014b)	3.3%	5.8%	11.1%	17.6%	35.9%
FaceNet (Schroff et al., 2015)	**VGGFace2** (Cao et al., 2018)	1.9%	4.7%	10.5%	16.6%	30.3%
ShowAndTell (Vinyals et al., 2015)	**MS COCO** (Lin et al., 2014)	0.1%	0.8%	4.4%	10.7%	28.5%
KWS (Sainath and Parada, 2015)	**GSC** (Sainath and Parada, 2015)	0.01%	0.16%	3.1%	12.3%	47.9%
LeNet-5 (LeCun et al., 1998)	**MNIST** (LeCun et al., 1998)	3.9%	6.0%	9.7%	13.3%	20.6%
Boosted-LeNet-4 (LeCun et al., 1995)	**GTSRB** (Stallkamp et al., 2011a)	9.5%	14.8%	25.4%	36.9%	61.4%

Table 4.1: The percentage of weight parameters required to attain different percentages of total significance (Fisher information (Lehmann and Casella, 2006)) in the 13 state-of-the-art DNNs; e.g., 5.0% of weight parameters are required to attain 90% of the total significance score in Inception-v4.

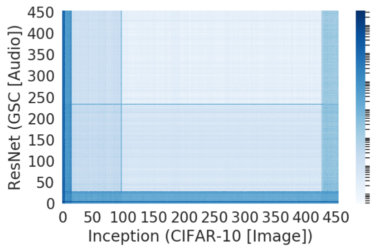
Weight Virtualization Landscape. Using three DNNs (Table 4.2 – Upper) and five datasets (Table 4.2 – Lower), we conduct an experiment where we compare the similarity of memory blocks of different pairs of DNNs. In Figure 4.3, we show three representative cases out of all $\binom{13}{2}$ pairs. In each figure, the darkness of the coordinate (x, y) represents the dissimilarity of memory block x of a DNN (X-axis) and the memory block y of another DNN (Y-axis). The dissimilarity score is computed using Equation 4.1 (explained later in Section 4.2), where a higher score or a darker dot represents larger differences between the memory blocks.

Figures 4.3a and 4.3b show that 1) DNNs of similar architectures, e.g., Inception and ResNet, tend to have similar memory blocks across the entire network except for the very beginning and the end, and 2) the number of similar blocks decreases when two DNNs perform different tasks. As shown in the figure, the area having paler color is smaller when the DNNs perform different tasks, i.e., image vs. audio classification (Figure 4.3b) than the case when they perform similar tasks, i.e., two image recognition tasks (Figure 4.3a).

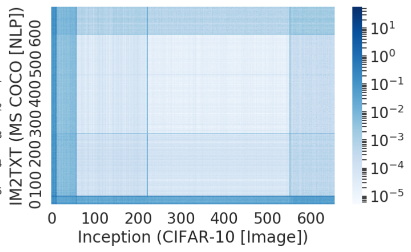
Figure 4.3c shows that DNNs of different architectures, e.g., Inception and IM2TXT, performing different tasks (image vs. NLP) tend to have memory blocks containing similar weights at specific and limited areas due to their architectural differences, i.e., convolutional (and residual) vs. recurrent structure.



(a) Inception(CIFAR10)/ResNet(GTSRB)



(b) Inception(CIFAR10)/ResNet(GSC)



(c) Inception(CIFAR10)/IM2TXT(MS)

Figure 4.3: The disparity in weights in the memory blocks of two DNNs having (a) similar architecture and similar tasks, (b) similar architecture but different tasks, and (c) different architecture and different tasks.

DNN	Network Architecture
Inception	Convonlution+Residual (Szegedy et al., 2017)
ResNet	Convonlution+Residual (He et al., 2016a)
IM2TXT	Convonlution+Recurrent (Vinyals et al., 2015)

Task	Dataset
Image	CIFAR-10 (Krizhevsky et al., 2009), GTSRB (Stallkamp et al., 2011a), MNIST (LeCun et al., 1998)
Audio	GSC (Sainath and Parada, 2015)
NLP	MS COCO (Lin et al., 2014) (Image caption)

Table 4.2: The DNN architecture and dataset.

4.1.2 Weight Virtualization Framework

We propose a two-phase weight virtualization framework (Figure 4.4) having an offline and an online phase.

Weight Virtualization (Offline). Weight virtualization is divided into three steps. First, for each DNN, its weight parameters are split into fixed-sized weight-pages - which is the basic unit of the weight virtualization. In Figure 4.4, the page size is 100, and each DNN has up to four weight-pages as the main memory size is 400. Second, the weight-pages of all DNNs are matched, and similar weight-pages are grouped together. The matching process minimizes a cost function (defined later by Equation 4.1) that reduces the performance loss caused by weight sharing. Figure 4.4 shows four groups of weight-pages. Third, the matched weight-pages in each group are combined to create a single *virtual* weight-page by optimizing (re-training) all or some of the

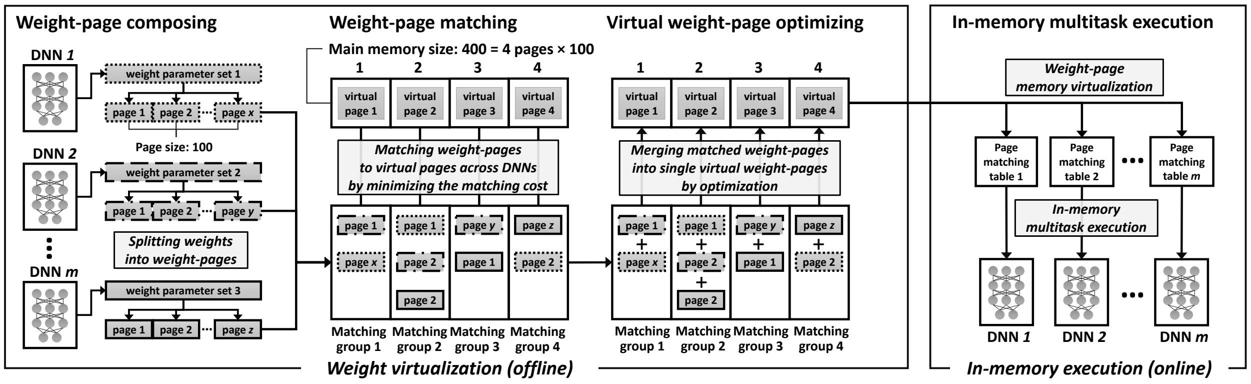

Weight-page composing
Weight-page matching
Virtual weight-page optimizing
In-memory multitask execution
Main memory size: 400 = 4 pages × 100
Page size: 100
Splitting weights into weight-pages
Matching weight-pages to virtual pages across DNNs by minimizing the matching cost
Merging matched weight-pages into single virtual weight-pages by optimization
Weight-page memory virtualization
In-memory multitask execution
Weight virtualization (offline)
In-memory execution (online)

Figure 4.4: The proposed weight virtualization framework has two phases: 1) An offline weight virtualization phase, consisting of weight-page composing, matching, and optimizing steps, and 2) an online in-memory execution phase.

DNNs. The goal of this optimization is to retain the accuracy of each DNN that shares one or more virtual weight-pages with others. Their algorithmic details are described in Section 4.2.

In-Memory Execution (Online). The virtual weight-pages are loaded into the main memory of the system. For each DNN, a page matching table that points a set of memory addresses of virtual weight-pages required by the DNN is generated. At run-time, a DNN is executed in the main memory with the necessary weight parameters obtained by dereferencing its page matching table. Further details of this phase are in Section 4.3.

4.1.3 Benefits of Weight Virtualization

Improved Multitask Learning. Unlike existing works on multitask learning (Caruana, 1997; Ruder, 2017; Yang and Hospedales, 2016b; Zhou and Zhao, 2015; Long et al., 2017) which are limited to similar network architectures and related tasks (Caruana, 1997), neural weight virtualization allows multitask learning involving heterogeneous tasks that may have completely different network architectures. Because the structure sharing happens at the granularity of weights, weight virtualization can pack any types of networks, including fully-connected (FC), convolutional (CNN), and recurrent neural networks (RNN). By isolating the memory blocks used for weights from the network structure (graph), weight virtualization preserves the original network structure of the input DNNs – which is not supported by the state-of-the-art (Ruder, 2017;

Zhang and Yang, 2017b). This latter aspect is particularly important since nowadays there are many DNN models available for use, and neural weight virtualization is a practical way to integrate these DNNs to a system without requiring any structural modifications to them (Chou et al., 2018b).

Efficient Parameter Representation. Virtual weight-pages not only reduce the amount of information required to represent the weight parameters of a DNN but also enables efficient unpacking and execution of DNNs at run-time. For example, to represent 1,000,000 weights using virtual weight-pages of size 100, only 10,000 page-pointers are required, which is only 1% of the original size of the weights (10,000 vs. 1,000,000). The larger the page-size, the less information is needed to represent the same amount of weight parameters. Hence, by suitably choosing a page-size, we can optimize the efficiency of parameter representation.

Efficient Task Management. In-memory multitask learning enables efficient management and scheduling of DNNs. Since all DNNs reside in the main memory, tasks can be executed and switched in real-time, enabling fast and responsive execution of multiple DNNs. By eliminating the need for repetitive resource allocation, e.g., loading DNNs from the flash or the hard disk, task management such as context switching and scheduling becomes simpler.

4.2 Weight Virtualization

Weight virtualization is a three-step process consisting of weight-page composing, matching, and optimizing, which collectively virtualize (combine) the weight parameters of DNNs to virtual weights that fit into the main memory. This is an offline process.

4.2.1 Weight-Page Composing

Weight-Page Composing. The first step of weight virtualization is to compose weight-pages for each DNN. Given a set of pre-trained DNN tasks, τ_i for $1 \leq i \leq m$, where m is the number of tasks, we define a *weight-page* as a sequence of s weights stored in consecutive memory locations of τ_i, where s denotes the page-size. Note that although the weights of a DNN are mathematically expressed as matrices when stored in the computer memory, they reside in a linear address space. Thus, weight-pages represent fixed-sized logical partitions of the portion of the main memory that contains the weight matrices. We define a *weight-page set*, P_i as all the weight-pages of DNN τ_i. Figure 4.5 illustrates the weight-page composition for a fully-connected layer. The process is similar for other architectures, such as convolutional and recurrent layers.

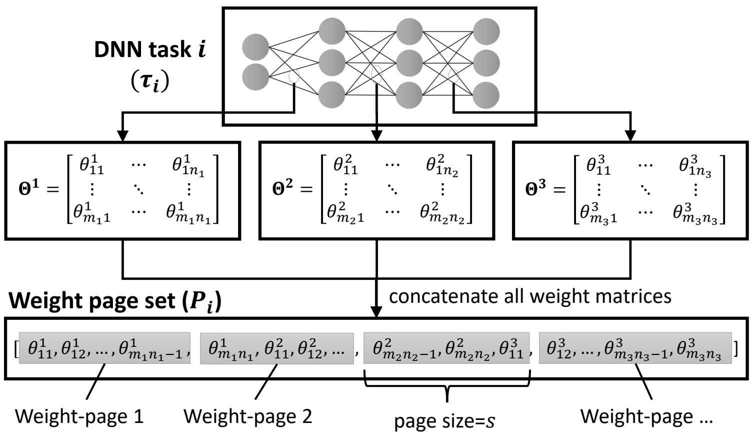

DNN task i (τ_i)
$\Theta^1 = \begin{bmatrix} \theta^1_{11} & \cdots & \theta^1_{1n_1} \\ \vdots & \ddots & \vdots \\ \theta^1_{m_1 1} & \cdots & \theta^1_{m_1 n_1} \end{bmatrix}$
$\Theta^2 = \begin{bmatrix} \theta^2_{11} & \cdots & \theta^2_{1n_2} \\ \vdots & \ddots & \vdots \\ \theta^2_{m_2 1} & \cdots & \theta^2_{m_2 n_2} \end{bmatrix}$
$\Theta^3 = \begin{bmatrix} \theta^3_{11} & \cdots & \theta^3_{1n_3} \\ \vdots & \ddots & \vdots \\ \theta^3_{m_3 1} & \cdots & \theta^3_{m_3 n_3} \end{bmatrix}$
concatenate all weight matrices
Weight page set (P_i)
$[\theta^1_{11}, \theta^1_{12}, \ldots, \theta^1_{m_1 n_1 -1}, \theta^1_{m_1 n_1}, \theta^2_{11}, \theta^2_{12}, \ldots, \theta^2_{m_2 n_2 -1}, \theta^2_{m_2 n_2}, \theta^3_{11}, \theta^3_{12}, \ldots, \theta^3_{m_3 n_3 -1}, \theta^3_{m_3 n_3}]$
Weight-page 1
Weight-page 2
page size=s
Weight-page …

Figure 4.5: The weight-page set P_i for DNN τ_i is composed by segmenting s consecutive weights in memory. The matrix, $\mathbf{\Theta}$ is an arbitrary weight or bias matrix, where θ_{ij} is an individual weight parameter.

Weight-Page-based Virtualization. We virtualize weights at the granularity of weight-pages for two reasons. First, it retains the functional and spatial locality of a DNN inside a weight-page, which increases the reusability of common locality patterns between DNNs. Second, it enables efficient and flexible management of weight parameters when sharing and accessing them. A DNN task can easily reconstruct its weight parameters by locating weight-pages in the main memory, which is much more efficient in terms of memory and computation than finding individual weights one by one, considering the massive number of weight parameters in many state-of-the-art DNNs (e.g., ResNet-50 (He et al., 2016a) and VGG-16 (Simonyan and Zisserman, 2014) have 26 and 138 million weights, respectively).

4.2.2 Weight-Page Matching

The next step of weight virtualization is to match each page within the weight-page sets of the DNNs to each page of the *virtual weight-page set*, P_0 where they are eventually merged. For multitask learning scenarios having more than two tasks, we keep a task, τ_0 as the final combined task and P_0 as the final virtual weight-page set, to which, $\tau_i(P_i)$'s are merged iteratively one by one, for $i > 0$. Hence, our goal is to find the best match between the virtual weight-page set P_0 of τ_0 and weight-page set P_i of the newly-added task τ_i – based on a one-to-one matching of each page.

Matching Objective. Given two sets of weight-pages, P_i and P_0, and a weight-page matching cost function, $C : P_i \times P_0 \to \mathbb{R}$, which is defined by Equation 4.1, the goal of the matching step is to find an injective mapping function $f_i : P_i \to P_0$ such that $\sum_{p \in P_i} C(p, f_i(p))$ is minimized, i.e., finding the closest set of $(p, f_i(p))$ pairs. Figure 4.6 illustrates a matching between task weight-page set P_i and the virtual weight-page set P_0.

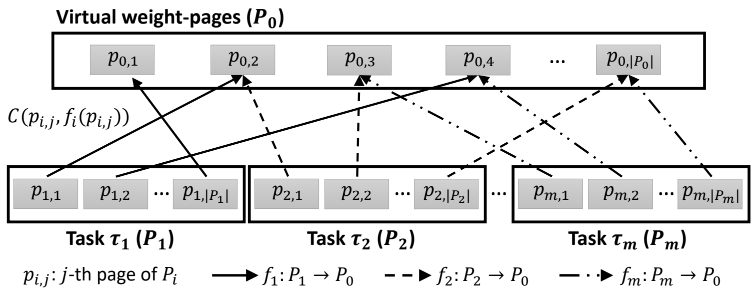


Figure 4.6: Each weight-page of task $\tau_i(P_i)$ is matched to a subset of the virtual weight-page set P_0. Each matching edge (arrows in the figure) incurs a matching cost of $C(p_{i,j}, f_i(p_{i,j}))$ between the matched weight-pages, whose summation is minimized.

Matching Cost Function. The *weight-page matching cost function* between two pages p and q is given by:

$$C(p, q) = \kappa \sum_{(\theta_p \in p, \theta_q \in q)} (\theta_p - \theta_q)^2 (\tilde{F}(\theta_p|\tau_p) + \tilde{F}(\theta_q|\tau_q)) \tag{4.1}$$

where κ is the matching regularizer, θ_p is a weight in p, θ_q is a weight in q, τ_p is the task of p, τ_q is the task of q, and $\tilde{F}(\theta_p|\tau_p)$ and $\tilde{F}(\theta_q|\tau_q)$ are Fisher information (Lehmann and Casella, 2006) of θ_p and θ_q, respectively (defined by Equation 4.2). The matching cost is minimized when two weights become similar, and the summation of Fisher information gets smaller. The matching cost function in Equation 4.1 primarily encourages virtualization of weights with similar values so that weight-pages of similar patterns can be combined together. However, at the same time, the summation of Fisher information works as a regularizer that discourages too much concentration of highly-important weight parameters on a few weight-pages in order to ensure the performance of virtualized DNNs by distributing the overall importance of weights over the entire memory.

Fisher Information. We use *Fisher Information* (Lehmann and Casella, 2006) in the matching cost function (Equation 4.1) since the difference of weights by itself does not carry the information on how impactful the change is to the final outputs of tasks (Kirkpatrick et al., 2017; Eskin et al., 2004). The Fisher information of weight, θ in $\tau_i(P_i)$ given data $X = \{ x_1, x_2, ..., x_n \}$ is defined by:

$$\tilde{F}(\theta|\tau_i) = \frac{1}{n}\sum_{j=1}^{n}\left(\frac{\partial}{\partial\theta}\log f_i(x_j|\theta)\right)^2 \tag{4.2}$$

where $f_i(x_j|\theta)$ is τ_i's probability density, i.e., the likelihood of data x_j conditioned on θ. Three key properties of Fisher score make it suitable for estimating the significance of a weight parameter (Kirkpatrick et al., 2017; Tu et al., 2016; Pascanu and Bengio, 2013): first, it is equivalent to the second derivative of the loss near a minimum, second, it can be easily computed from the first-order derivatives alone and is thus easy to calculate for large models, and third, it is guaranteed to be positive semi-definite.

Weight-Page Matching Problem. Given m DNN tasks, the weight-page matching is formulated as a combinatorial optimization problem, called the assignment problem (Schrijver, 2003) aka maximum weighted bipartite matching problem (West et al., 1996), as follows:

$$\begin{aligned} \min \sum_{i=1}^{m} & \sum_{(p,q)\in P_i \times P_0} C(p,q) \cdot x_{ipq} \\ \text{s.t.} \sum_{p \in P_i} & x_{ipq} = 1 \quad 1 \le i \le m \text{ and } q \in P_0 \\ \sum_{q \in P_0} & x_{ipq} = 1 \quad 1 \le i \le m \text{ and } p \in P_i \\ & x_{ipq} \in \{0,1\} \quad 1 \le i \le m, p \in P_i \text{ and } q \in P_0 \end{aligned} \tag{4.3}$$

where, P_i is the weight-page set of task τ_i, P_0 is the weight-page set for the combined task τ_0, and $C(p,q)$ is the weight-page matching cost function between weight-page p and q in Equation 4.1. The variable x_{ipq} is one, if page p and q are matched, and zero, otherwise. Although $\mathcal{O}(n^4)$ polynomial-time optimal algorithm (Munkres, 1957) exists for two tasks, where n is the number of weight-pages, for a large number of weight-pages, the computational overhead of such matching is too high. For more than two tasks, the problem becomes a multiple-choice multiple-assignment problem (Chen and Lu, 2007; Spieksma, 2000) of $\mathcal{O}(n^{nm})$ complexity, which is computationally intractable.

Greedy Matching Algorithm. To solve the weight-page matching problem in Equation 4.3 which is computationally challenging, we propose an *iterative greedy matching algorithm* that starts with an unmatched weight-page, $p \in P_i$ having the largest Fisher information (i.e., weights that impact the final outputs more) and greedily finds an unmatched page, $q \in P_0$ that minimizes the matching cost $C(p,q)$ in Equation 4.1. It is based on the observation that Fisher information of most weights has near-zero values except only a few having significant magnitudes. Algorithm 1 describes the proposed greedy weight-page matching algorithm. For m tasks having n weight-pages each, it has $\mathcal{O}(mn^2/p)$ of computational complexity, when computing weight-page matching cost in p-way parallel (line 10–15) using a GPU.

Algorithm 1: Greedy Weight-Page Matching

```
Input: Weight-page set $P_i$ for $0 \leq i \leq m$, Fisher information set $F_i$ for $0 < i \leq m$
Output: $f_i : P_i \to P_0$ for $1 \leq i \leq m$
1  $(c_k)_{k=1}^{|P_0|} = (0, ...)$, $(u_k)_{k=1}^{|P_0|} := (P_0)$, $(s_k) = (v_k) = ()$, $n := 0$;
2  for $i \leftarrow 1$ to $m$ do
3      forall $(f, p) \in (F_i, P_i)$ do
4          $n := n + 1$, $s_n := ||f||_1$, $v_n := p$;
5      end
6  end
7  sort $(v_k)_{k=1}^{n}$ in descending order of $(s_k)_{k=1}^{n}$;
8  for $i \leftarrow 1$ to $n$ do
9      $(c'_i)_{k=1}^{|P_0|} := (c_i)_{k=1}^{|P_0|}$, $c_{\min} := \infty$ , $\text{idx}_{\min} := 0$;
10     for $j \leftarrow 1$ to $|P_0|$ do in parallel
11         $c_j := c_j + C(v_i, u_j)$;
12         if $c_j < c_{\min}$ then
13             $c_{\min} := c_j$, $\text{idx}_{\min} := j$;
14         end
15     end
16     $c_{\text{idx}_{\min}} := c_{\min}$;
17     $t :=$ task number of $v_i$, $f_t(v_i) := u_{\text{idx}_{\min}}$;
18 end
19 return $f_i$ for $1 \leq i \leq m$
```

4.2.3 Weight-Page Optimization

The last step of weight virtualization is to combine the matched weight-pages into single pages and optimize (retrain) the tasks to retain any accuracy loss due to the changed weight parameters.

Virtual Weight-Page. Once the task weight-pages are matched, they are combined to obtain single pages called *virtual weight-pages* by optimizing (retraining) all or some of the tasks. To retain the accuracy of tasks, we minimize the side-effects of combining on the set of to-be combined weight-pages $\mathrm{Q}_i = \bigcup_{p \in P_i} \mathrm{Q}_{i,p}$ when optimizing task τ_i, which is given by:

$$\mathrm{Q}_{i,p} = \{\, q \in \bigcup_{j=1, j \neq i}^{m} P_j \mid \exists f_j^{-1}(f_i(p)) \,\} \tag{4.4}$$

where, p is the page in the weight-page set P_i of τ_i, m is the number of tasks, P_j is the weight-page set of τ_j, f_j^{-1} is the inverse page mapping function of τ_j, and f_i is the page mapping function of τ_i obtained from the weight-page matching step in Section 4.2.2.

Optimization. Task τ_i is optimized by minimizing the summation of 1) the original loss function of τ_i, and 2) the total weight-page matching cost of Q_i in Equation 4.4, which is given by:

$$\mathcal{L}(\tau_i) = \mathcal{L}_o(\tau_i) + \kappa \sum_{p \in P_i} \frac{1}{|\mathrm{Q}_{i,p}|} \sum_{q \in \mathrm{Q}_{i,p}} \sum_{\theta \in q} (\theta - \theta^*)^2 \tilde{F}(\theta|\tau_q) \tag{4.5}$$

where $\mathcal{L}_o(\tau_i)$ is the original loss of τ_i, κ is the matching regularizer, Q_i is the weight-pages matched to the page $f_i(P_i)$ defined in Equation 4.4, θ is the weight in the page q, θ^* is the virtual weight in $f_i(p)$ being optimized and shared between tasks, τ_q is the task having page q, and $\tilde{F}(\theta|\tau_q)$ is the Fisher information of θ in τ_q. By jointly optimizing tasks with the additional loss on weight-page matching, which is expressed by the second term in Equation 4.5, it tries to find common local minima for the desired performance of all tasks. Even when the tasks are sequentially optimized one-by-one, the virtual weights tend to stay in a low-risk region, maintaining the performance of the previously optimized tasks (Kirkpatrick et al., 2017).

Flexible Weight Sharing. Weight virtualization enables flexible weight sharing and achieves better performance than existing works such as continual learning (Parisi et al., 2019; Kirkpatrick et al., 2017; Zenke et al., 2017) that tries to solve the catastrophic forgetting problem (Goodfellow et al., 2013; French, 1999). While existing works can only share a fixed combination of weights in the same networks, weight virtualization allows weight sharing on arbitrary combinations of weights between tasks of different architectures. Furthermore, the accuracy of DNNs with weight virtualization is expected to be higher since the weight-pages are first matched to minimize the matching cost before optimization (retraining). Thus, similar weight-pages are more likely to be shared – which improves the performance of weight sharing between tasks.

4.3 In-Memory Execution

Weight virtualization enables fast, in-memory execution of multiple DNNs at run-time by efficiently accessing the virtualized weights supported by an efficient memory model and data structure.

4.3.1 Memory Model and Data Structure

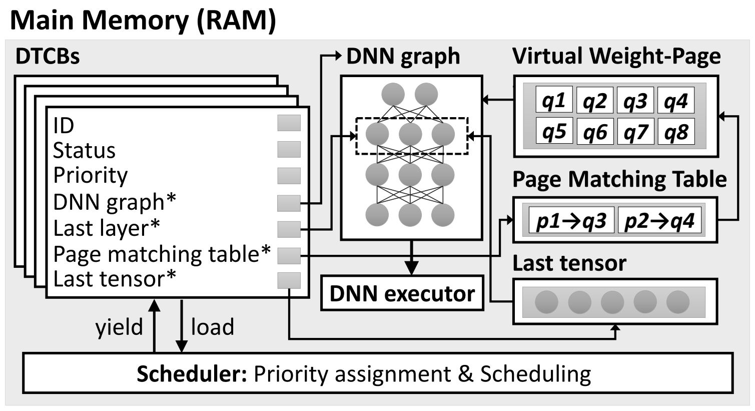


Figure 4.7: DNN Task memory map illustrating DTCB, DNN graph, DNN executor, virtual weight-pages, and page matching table. Virtual weight-pages and DNN executor are shared among tasks, while the other elements are exclusive to each task.

Memory Model. Figure 4.7 shows the memory model that supports neural weight virtualization, consisting of *DTCB* (DNN Task Control Block), *DNN graph, DNN executor, virtual weight-pages*, and *page matching table*. Unlike conventional multitasking models (Uyeda, 2009) that allocate private memory spaces to individual tasks, it allows memory overlapping between tasks for weight sharing via virtual weight-pages. Also, unlike traditional virtual memory (Toy and Zee, 1986) that expands to second-level memory, i.e., swapping DNNs with a flash or hard disk, it implements the entire memory hierarchy into the main memory since all DNN tasks fit into it.

DTCB. Similar to the Process Control Blocks (PCB) in modern operating systems (Silberschatz et al., 2012), each DNN task is managed by in-memory *DNN Task Control Block (DTCB)* that is updated by the system at run-time. Each DTCB corresponds to a DNN task which contains the necessary task information, i.e., ID, status (e.g., running, suspended), priority (for scheduling), and pointers to DNN graph, page matching table, last layer, and last tensor as shown in the left side of Figure 4.7.

Virtual Weight-Pages. The *virtual weight-pages* generated by the weight virtualization in Section 4.2 are located in the main memory. They are shared by the DNN tasks. A task accesses to a subset of the virtual weight-pages by using its page matching table that performs the weight-page translation.

Page Matching Table. For each DNN task, a *page matching table* that translates the weight-pages of the DNN to the virtual weight-pages based on the matching result of Section 4.2.2 is provided. It consists of 1) a mapping list between weight-pages of the DNN and the virtual weight-pages, and 2) memory addresses of the matched virtual weight-pages. Figure 4.8 illustrates the access process of virtual weight-pages via a page matching table where a task locates the memory addresses of virtual weight-pages matched to its weight-pages.

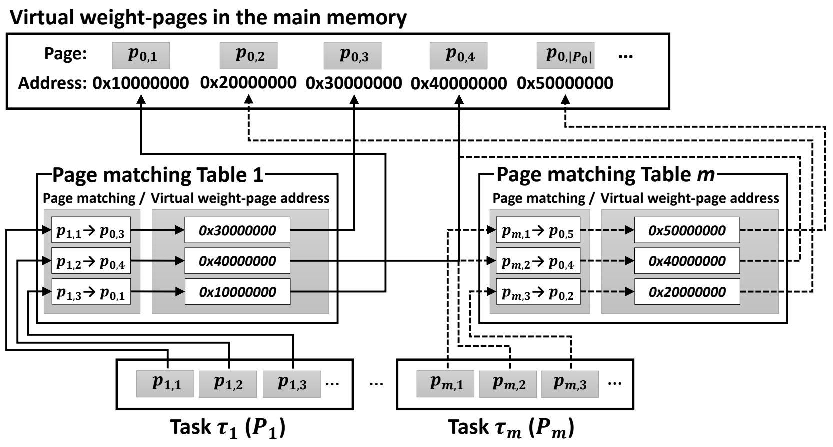

Virtual weight-pages in the main memory
Page: $p_{0,1}$ $p_{0,2}$ $p_{0,3}$ $p_{0,4}$ $p_{0,|P_0|}$ ...
Address: 0x10000000 0x20000000 0x30000000 0x40000000 0x50000000
Page matching Table 1
Page matching / Virtual weight-page address
$p_{1,1} \rightarrow p_{0,3}$ 0x30000000
$p_{1,2} \rightarrow p_{0,4}$ 0x40000000
$p_{1,3} \rightarrow p_{0,1}$ 0x10000000
Page matching Table m
Page matching / Virtual weight-page address
$p_{m,1} \rightarrow p_{0,5}$ 0x50000000
$p_{m,2} \rightarrow p_{0,4}$ 0x40000000
$p_{m,3} \rightarrow p_{0,2}$ 0x20000000
$p_{1,1}$ $p_{1,2}$ $p_{1,3}$ ...
...
$p_{m,1}$ $p_{m,2}$ $p_{m,3}$ ...
Task τ_1 (P_1)
Task τ_m (P_m)

Figure 4.8: A page matching table translates weight-pages of a DNN task into virtual weight-pages in the memory.

DNN Graph and Executor. Each task has metadata called the *DNN graph* that defines its network architecture. The *DNN executor* is a common module, shared between all tasks, which executes a part or entire DNN graph given input tensors and weights.

4.3.2 Task Execution

Task Execution. For the in-memory execution of a task, a task first unfolds its DNN graph and locates the last layer that was completed at the previous execution using its DTCB. Then, the last tensor values (also saved at the previous execution) and the necessary weight parameter values obtained via page matching table are passed to the DNN executor for execution. To improve memory efficiency, only the unfinished part of the graph from the last execution is unfolded and executed. It also allows fitting large DNNs into limited memory by unfolding and running a part of the graph in stages when the entire DNN does not fit into the memory.

Task Scheduling and Switching. DNN tasks are scheduled after every execution of a tensor. After executing a tensor of a task, the DNN executor gives control back to the scheduler that selects the next task based on its scheduling policy and task priorities. When a task is context-switched, the system saves the last layer of the current task completed by the DNN executor, including the corresponding tensor that is passed to the DNN executor when the task is scheduled to execute again. It allows DNN tasks to be executed by various schedulers, e.g., non-preemptive scheduling such as cooperative multitasking (Bartel, 2011) or preemptive scheduling (Silberschatz et al., 2018) that requires back-and-forth execution of tasks. In this chapter, we use cooperative multitasking.

In-Memory Execution. Task execution and DNN context-switches – all happening in the main memory – improves the response time and the end-to-end execution time of tasks since 1) access time of the main memory is consistent and much faster (10X–100X) than bulk storage modules such as flash/hard disks, and 2) the main memory is far more flexible and efficient for random access. On the contrary, a system using secondary storage experiences not only significant but also unpredictable overhead, e.g., disk-writes in some storage modules such as flash and solid-state drives, have to erase an entire block before writing to it. By eliminating this overhead, in-memory execution enables fast and responsive back-to-back execution of multiple DNNs.

4.3.3 Response Time Analysis

The in-memory execution improves the response time of a DNN task which is defined as the time until the task starts execution (Koopman, 2016). For example, the response time of tasks scheduled by a preemptive scheduler is given by:

$$\begin{aligned} R_{0,0} &= S_0,\ R_{i,0} = S_0 + C_0 + S_i \text{ for } i > 0 \\ R_{i,k+1} &= \sum_{j=0}^{i-1} \left\lfloor \frac{R_{i,k}}{P_j} + 1 \right\rfloor \cdot (S_j + C_j) \end{aligned} \tag{4.6}$$

where i is task number (i=0 denotes the highest priority task), $R_{i,k}$ is the response time of task i for its k-th execution, S_i is DNN context-switch time, C_i is worst-case execution time, and P_i is period of task i (worst case). It is assumed that task i never tries to execute again until the previous execution runs to completion.

As S_i reduces to S'_i due to in-memory execution, the response time also decreases. For example, the response time of task 1 for its first execution, i.e., $R_{1,1}$ is reduced to $R'_{1,1}$, and the ratio is given by:

$$\frac{R'_{1,1}}{R_{1,1}} \simeq \frac{(S'_0 + C_0 + S'_1 + P_0)(S'_0 + C_0)}{(S_0 + C_0 + S_1 + P_0)(S_0 + C_0)} \tag{4.7}$$

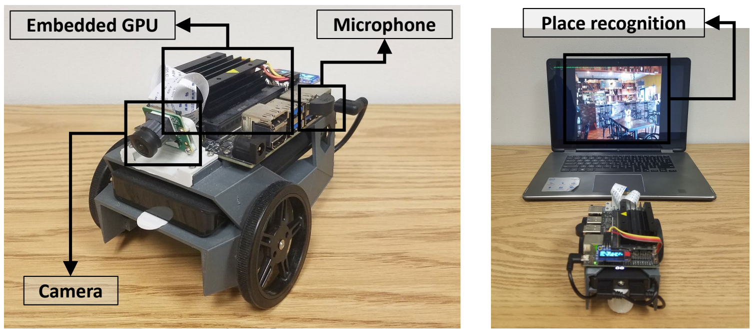
where S'_i is the reduced context-switch time. If we let $r_i = S_i/C_i$ and $q_i = P_i/C_i$, and assume $S'_i \ll S_i$ and $C_i \leq P_i$, Equation 4.7 is bounded as follows:

$$\frac{R'_{1,1}}{R_{1,1}} \geq \frac{2C_0}{C_0(r_0 + 2) + C_1 r_1} \cdot \frac{1}{r_o + 1} \tag{4.8}$$

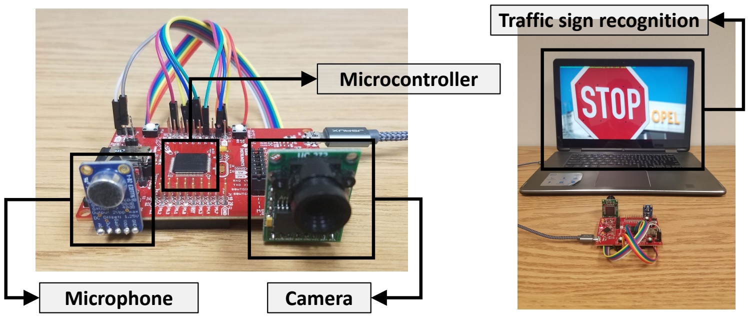
which implies that the response time is decreased more with bigger r_i and smaller q_i. In general, $R'_{i,k}/R_{i,k}$ gets smaller as $k \to \infty$ due to the recursive effect of S_i in Equation 4.6. For example, the response time is improved by at least 4x in an embedded GPU system that requires the same amount of time for DNN execution and context-switching. The response times for other scheduling algorithms can be deduced similarly.

4.4 System Implementation

We develop two real systems that implement neural weight virtualization: 1) an embedded GPU-based multitask learning mobile robot, and 2) a microcontroller-based low-power multitask learning IoT device. These systems are packed with 5-6 learning tasks. Figure 4.9 shows these systems performing an image recognition task (i.e., place and traffic sign recognition).



(a) Deep multitask learning mobile robot



(b) Deep multitask learning IoT device

Figure 4.9: Two application systems of neural weight virtualization: Multitask learning mobile robot and multitask learning IoT device.

4.4.1 Deep Multitask Learning Mobile Robot

System Overview. Intelligent robots are often tasked with multiple high-level perception tasks. For example, a social robot (Bohren et al., 2011; Siegwart et al., 2003) has to move around and assist a user – based on various human- or environment-generated information, such as voice and facial expression. As the first application system, we implement a multitask learning mobile robot (Figure 4.9a), representing an intelligent social agent, that assists humans in various situations by running six state-of-the-art DNNs, i.e., objection recognition (MobileNet-v2 (Sandler et al., 2018)), face identification (FaceNet (Schroff et al., 2015)), visual scene interpretation (ShowAndTell (IM2TXT) (Vinyals et al., 2015)), place classification (Place205 (Zhou et al., 2014b)), environmental sound classification (UrbanSound8K (Su et al., 2019)), and voice recognition (GoogleSpeechCommands (GSC) (Sainath and Parada, 2015)).

Hardware Platform. The robot is implemented using Jetson Nano (NVIDIA, 2019a), an embedded GPU platform having an NVIDIA Maxwell GPU and an ARM A57 CPU. The robot has a camera in the front, a microphone on the side, and two motors and wheels on both sides. We print the body of the robot with a 3D printer and install the components on it. For the offline phase of weight virtualization, we use an NVIDIA RTX 2080 Ti GPU. For run-time in-memory multitask DNN execution, we use the embedded GPU of the robot.

Software Platform. We extend TensorFlow (Abadi et al., 2016) to support weight virtualization algorithms and in-memory multitask execution.

Memory Budget. We allocate 146MB of GPU RAM to the virtual weight parameters and pack six DNN tasks in the GPU RAM whose total memory requirement is 604MB. The memory size of each DNN is listed in Figure 4.16 and Table 4.5.

4.4.2 Deep Multitask Learning IoT Device

System Overview. Recently, microcontroller-based embedded systems have started to support the execution of lightweight versions of state-of-the-art DNNs (Lee and Nirjon, 2019; Yao et al., 2018b; Wang et al., 2018; Gobieski et al., 2018a; Lane et al., 2017). For example, Google recently released their TensorFlow Lite library targeting microcontrollers (MCUs) (Google, 2019e), and Amazon Alexa services are now supported on MCUs such as ARM Cortex M (Cortex, 2004) that have less than 1MB memory (Amazon, 2019). However, the number of DNNs that runs on an embedded system (Chauhan et al., 2018; Gobieski et al., 2018a; Bhattacharya and Lane, 2016) is limited by the capability of the MCU, memory-size, and battery-capacity. As the second application system, we implement a low-power multitask learning IoT device that performs multitask DNN learning on an MCU, having an extremely limited memory (256KB). The system is shown in Figure 4.9b. It packs five compressed DNNs, i.e., traffic sign recognition (German Traffic Sign Recognition Benchmark (GTSRB) (Stallkamp et al., 2011a)), voice command recognition (GoogleSpeechCommands (GSC) (Sainath and Parada, 2015)), house/plate number classification (Street View House Numbers (SVHN) (Netzer et al., 2011)), digits classification (MNIST (LeCun et al., 1998)), and object recognition (CIFAR-10 (Krizhevsky et al., 2009)).

Hardware Platform. The system is implemented on an MSP430 MCU (TexasInstruments, 2018) that consumes little energy ($\leq$1.8mA). For in-memory multitask DNN execution, we use 256KB FRAM (Buck, 1952) built in the MCU package. The FRAM is a non-volatile memory, performing read/write operations in nanoseconds (Cypress, 2017). For sensing, we connect a camera and a microphone to the MCU. For the offline phase of weight virtualization, we use an NVIDIA RTX 2080 Ti GPU.

Software Platform. We extend TensorFlow (Abadi et al., 2016) for weight virtualization algorithms and implement the in-memory multitask execution framework using C language for efficient task execution.

Memory Budget. We allocate 129 KB of FRAM for weight virtualization and pack five DNNs whose total memory requirement is 523KB. The memory size is shown in Figure 4.21 and Table 4.7.

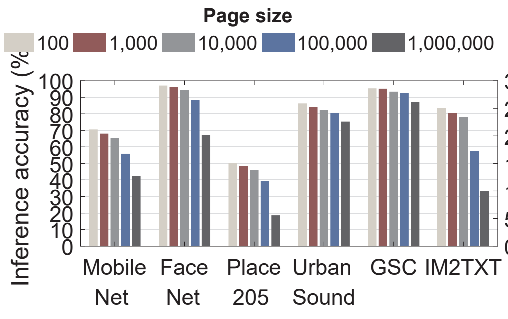
4.5 Algorithm Evaluation

We first evaluate the weight virtualization phase described in Section 4.2 for the two systems we implement in Section 4.4. We use an NVIDIA RTX 2080 Ti GPU and an Intel Core i9-9900K CPU.

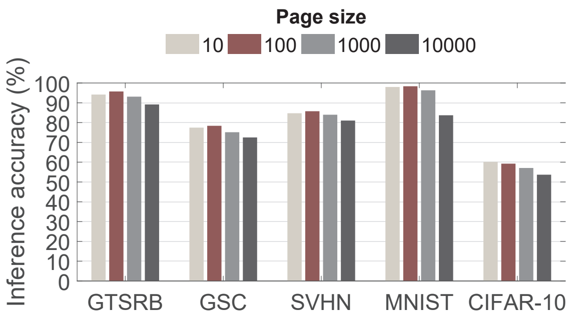
Training and Evaluation Datasets. For the training and evaluation of DNNs in the multitask mobile robot, we use ImageNet (Deng et al., 2009), VGGFace2 (Cao et al., 2018), LFW Face (Huang et al., 2008), Microsoft COCO (Lin et al., 2014), Place205 (Zhou et al., 2014b), UrbanSound8K (Salamon et al., 2014), and GoogleSpeechCommands (GSC) (Warden, 2018) dataset. For the multitask IoT device, we use GTSRB (Stallkamp et al., 2011a), GSC (Warden, 2018), SVHN (Netzer et al., 2011), MNIST (LeCun et al., 1998), and CIFAR-10 (Krizhevsky et al., 2009) dataset.

4.5.1 Weight-Page Matching

Weight-Page Size. We evaluate the effect of weight-page size on weight virtualization. Figure 4.10 plots the inference accuracy of the two application systems over different weight-page sizes. In general, larger pages result in decreased accuracy for both systems, e.g., the accuracy of FaceNet for the mobile robot drops from 97% to 67% when the page size is increased from 100 to one million. Since smaller pages are more finely matched with each other, higher accuracy is achieved.



(a) Mobile Robot



(b) IoT Device

Figure 4.10: The inference accuracy for various weight-page size for multitask learning mobile robot and the IoT device.

However, smaller page sizes increase the total number of virtual weight-pages and page matching time, which increases both the run-time and compile-time cost, as shown in Table 4.3 and 4.4. Thus, a suitable page size should be chosen based on the trade-off between the performance (e.g., inference accuracy) and computation/memory cost.

Weight-Page Matching vs. Random Matching. We evaluate the performance of the weight-page matching algorithm by comparing the weight-page matching cost defined in Equation 4.1 and the final inference accuracy of the two systems against random weight-page matching, as

Weight-Page Size	Total Number of Virtual Weight-Pages	Total Number of Virtual Weights	Matching Time (s)
100	383,510	38,351,000	916.21
1,000	38,351	38,351,000	89.77
10,000	3,836	38,350,000	16.66
100,000	384	38,400,000	14.24
1,000,000	39	39,000,000	10.28
10,000,000	4	40,000,000	10.47

Table 4.3: The number of weight-pages, weights, and weight-page matching time over different page sizes for the multitask mobile robot.

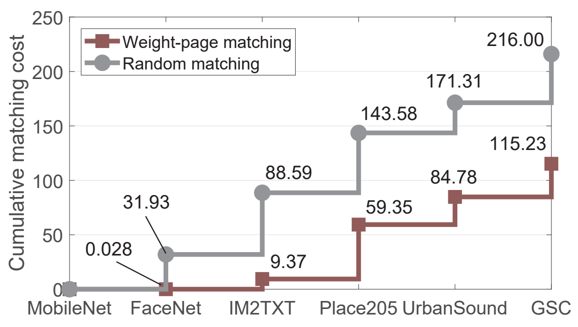
Weight-Page Size	Total Number of Virtual Weight-Pages	Total Number of Virtual Weights	Matching Time (s)
10	6,648	66,480	0.4822
100	665	66,500	0.1493
1,000	67	67,000	0.1076
10,000	7	70,000	0.1077

Table 4.4: The number of weight-pages, weights, and weight-page matching time over different page sizes for the multitask IoT device.

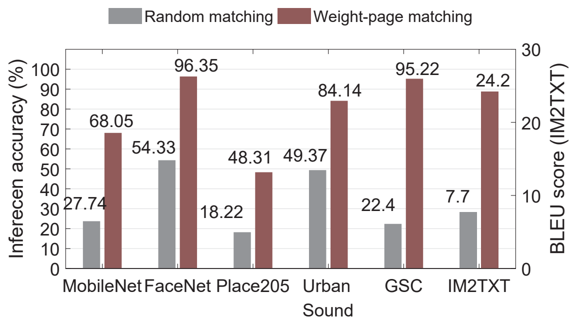
shown in Figures 4.11 and 4.12. In random matching, the weight-pages of DNNs are randomly matched for optimization without considering their similarity, as done in existing works (Kirkpatrick et al., 2017; Zenke et al., 2017). For both systems, the weight-page matching outperforms random matching, i.e., a maximum 1,140x less matching cost that results in a maximum of 72% inference accuracy improvement in the mobile robot (GSC in Figure 4.11b). It demonstrates that weight-page matching is an essential step for performance retention before performing weight-page optimization (combining), which significantly decreases the chance of combining weight-pages of big difference.

4.5.2 Weight-Page Optimization

Joint vs. Sequential Optimization. We evaluate two methods of weight-page optimization, i.e., joint vs. sequential optimization, by comparing their final inference accuracy. While all the DNN tasks are optimized together in the joint optimization, only a single task is optimized at a time after completing the optimization of the prior task in the sequential optimization. Figure 4.13 and 4.14 show the results for the two systems, respectively. Sequential optimization exhibits a pattern that a DNN recently optimized achieves its best accuracy while DNNs optimized prior to it ex-

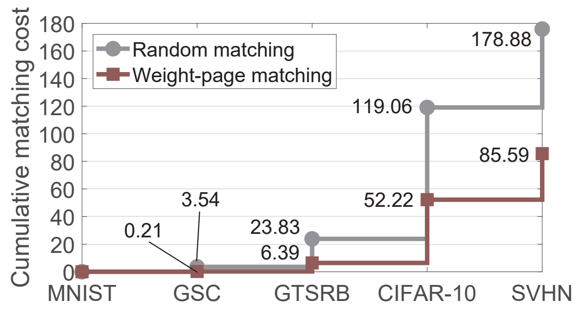


(a) Weight-page matching cost

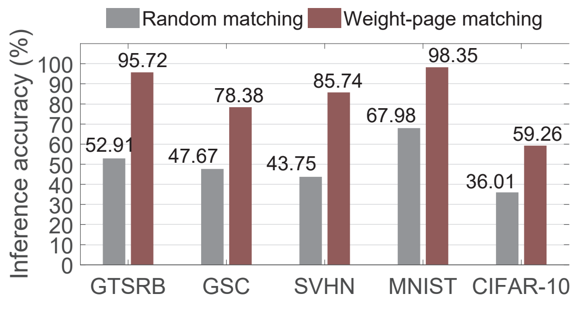


(b) Inference accuracy

Figure 4.11: The weight-page matching cost and inference accuracy of the multitask learning mobile robot: The proposed weight-page matching vs. Random matching.



(a) Weight-page matching cost

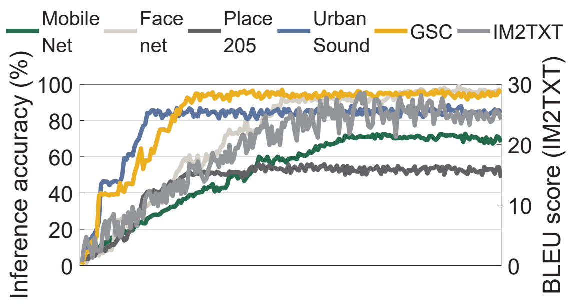


(b) Inference accuracy

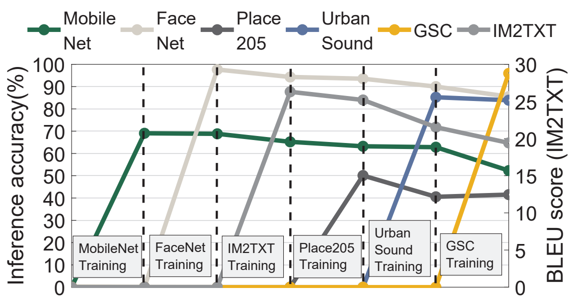
Figure 4.12: The weight-page matching cost and inference accuracy of the multitask learning IoT device: The proposed weight-page matching vs. Random matching.

perience accuracy degradation, e.g., from 69% to 52% for MobileNet for the mobile robot. On the other hand, joint optimization keeps the best accuracy of all DNNs, and sometimes achieves *higher* accuracy than the individual training of a DNN due to less overfitting achieved during joint training, e.g., from 69% to 78% for GSC in the multitask learning IoT device, by optimizing them together during the entire optimization iterations.

Matching Regularizer. We evaluate the effect of matching regularizer, κ in Equation 4.1, and 4.5, which determines the extent of matching cost that works as a penalty in optimization, i.e., the larger κ, the more joint performance of all DNNs are considered during optimization. Figure 4.15 plots the inference accuracy with different matching regularizers, which shows that the inference accuracy of the DNNs tends to increase as the regularizer increases but starts to decrease after some point. That is because too large regularizer leads the optimizer to minimize the matching cost too much, preventing it from minimizing the original losses of DNNs.

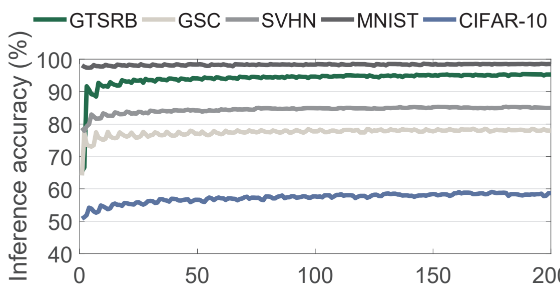


(a) Joint optimization

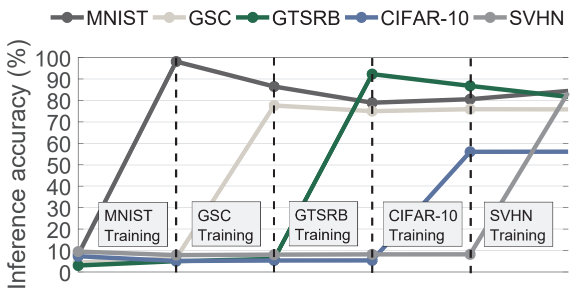


(b) Sequential optimization

Figure 4.13: The inference accuracy of the multitask learning mobile robot: Joint vs. Sequential optimization.

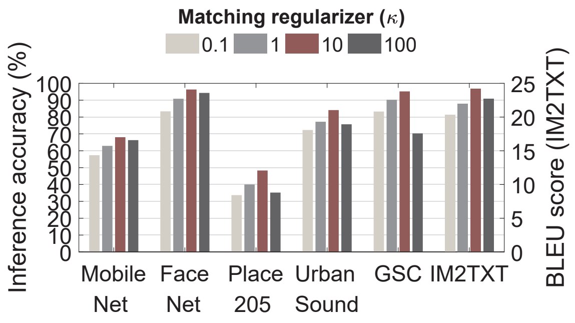


(a) Joint optimization

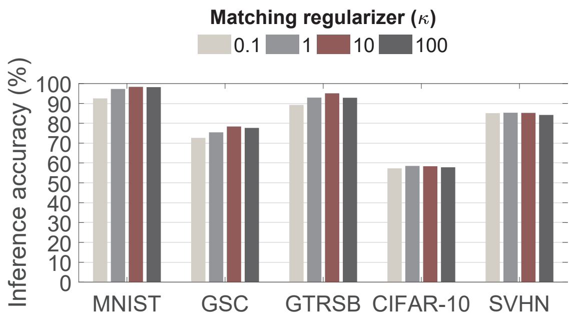


(b) Sequential optimization

Figure 4.14: The inference accuracy of the multitask learning IoT device: Joint vs. Sequential optimization.



(a) Multitask mobile robot



(b) Multitask IoT device

Figure 4.15: The inference accuracy over the regularizer κ.

4.6 System Evaluation

In this section, we evaluate the run-time performance of in-memory multitask execution for the two multitask learning systems described in Section 4.4 (i.e., mobile robot and IoT device). The evaluation is performed on the same dataset used in Section 4.5.

4.6.1 Deep Multitask Learning Mobile Robot

Memory Packing Efficiency. We evaluate the memory efficiency of weight virtualization by measuring the memory usage of the DNNs, which shows the packing ratio of multiple DNNs that reside in the limited memory. Figure 4.16 and Table 4.5 present the number of weights and memory usage for the mobile robot, where a total of 144M weight parameters (604MB) are virtualized to 38M virtual weights (146MB). It achieves a 4.13x packing ratio when compared to baseline DNNs that are not virtualized.

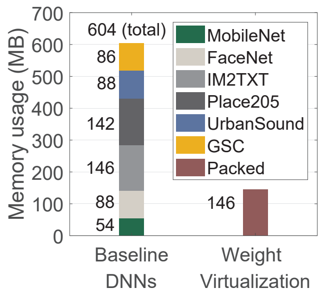


Figure 4.16: The memory usage of weight parameters: Baseline DNNs vs. Weight virtualization for the multitask learning mobile robot.

Baseline DNNs	Total Weights	Memory (MB)
MobileNet (Sandler et al., 2018; Howard et al., 2017; Deng et al., 2009)	14M	54
FaceNet (Schroff et al., 2015; Cao et al., 2018; Huang et al., 2008)	22M	88
IM2TXT (Vinyals et al., 2015; Lin et al., 2014)	37M	142
Place205 (Zhou et al., 2014b,a, 2016)	38M	146
UrbanSound (Su et al., 2019; Zhang et al., 2018b; Salamon et al., 2014)	23M	88
GSC (Sainath and Parada, 2015; Chen et al., 2014a; Warden, 2018; Li and Zhou, 2017)	22M	86
Total	**144M**	**604**
Weight Virtualization	**Packed Weights**	**Memory (MB)**
MobileNet+FaceNet +IM2TXT+Place205 +UrbanSound+GSC	**38M**	**146**
Packing Ratio	**4.13x**	**4.13x**

Table 4.5: Weight memory packing of the multitask learning mobile robot with weight virtualization.

Inference Accuracy. We evaluate the inference accuracy of the virtualized DNNs, as shown in Figure 4.17. Compared to the state-of-the-art baseline DNNs that are not virtualized, they achieve comparable performance with a maximum of 3% accuracy drop (MobileNet) or achieve better accuracy for some DNNs (0.4% up in UrbanSound and 0.72% up in GSC). The accuracy increase can be explained by the fact that weight parameter sharing via virtualization leads to 1) reduce the risk of overfitting (Baxter, 1997), and 2) provide better generalization (Ruder, 2017) with weight-page optimization.

Execution Time. We evaluate the execution time of the virtualized DNNs against baseline DNNs that are not virtualized. Figure 4.18 shows the end-to-end execution time of consecutive DNNs for 1) baseline DNNs that use eMMC memory as a secondary storage module for loading

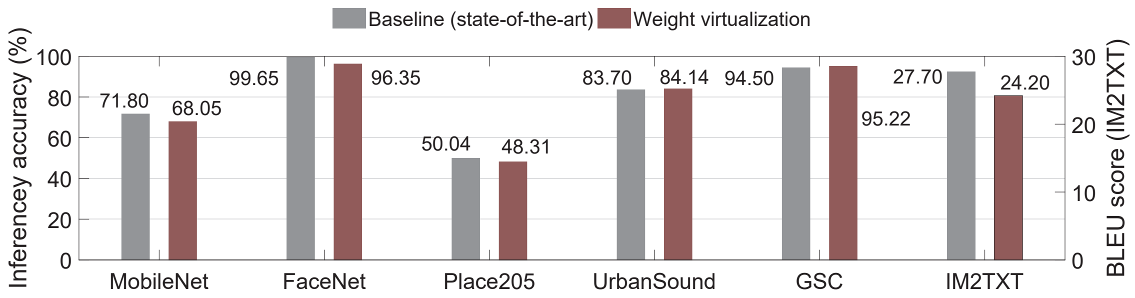


Figure 4.17: The inference accuracy of multitask mobile robot: Baseline (state-of-the-art) vs. Weight virtualization.

DNNs to GPU RAM, and 2) virtualized DNNs that perform in-memory loading and execution entirely in GPU RAM. As shown in the figure, the execution time is significantly improved when executing DNNs consecutively, e.g., 36.9x faster when executing all six DNNs successively (39.12 vs. 1.06 seconds). Next, we breakdown the end-to-end execution time into two parts, i.e., 1) actual DNN execution and 2) DNN-switching (loading) time, and measure each of them separately. Figure 4.19 shows the execution time and switching (loading) time of each DNN. All the virtualized DNNs accelerate their switching (loading) by a maximum of 87x (9.16 vs. 0.105 seconds for IM2TXT) compared to the non-virtualized DNNs (the baselines), which demonstrates that in-memory execution yields much faster response time for multitasking DNNs.

Comparison to Alternatives. We compare the inference accuracy and execution time (total time for inference plus switching) of the virtualized DNNs against two alternative methods: 1) multitask learning (MTL) and 2) individual model compression. Both use the same total amount of 146 MB memory as the virtualized DNNs. For MTL, the baseline DNNs are trained together using the state-of-the-art *K for-the-price-of 1* (Mudrakarta et al., 2018) approach, whose memory size is limited to 146 MB. For model compression, the baseline DNNs are individually compressed using pruning methods that are applicable to specific architectures (He et al., 2018b; Wu et al., 2017; Han et al., 2015b,a; Zhang et al., 2018a). Table 4.6 provides the memory usage of the baseline (uncompressed) and compressed DNNs. Note that the combined memory usage of the six compressed DNNs is 146 MB.

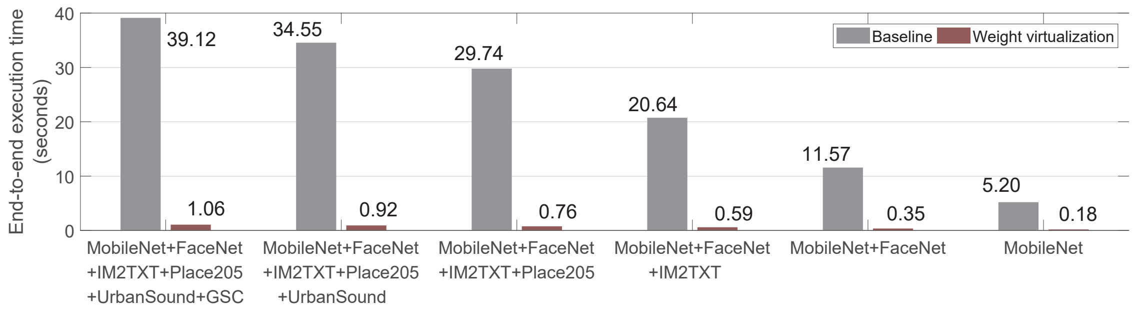
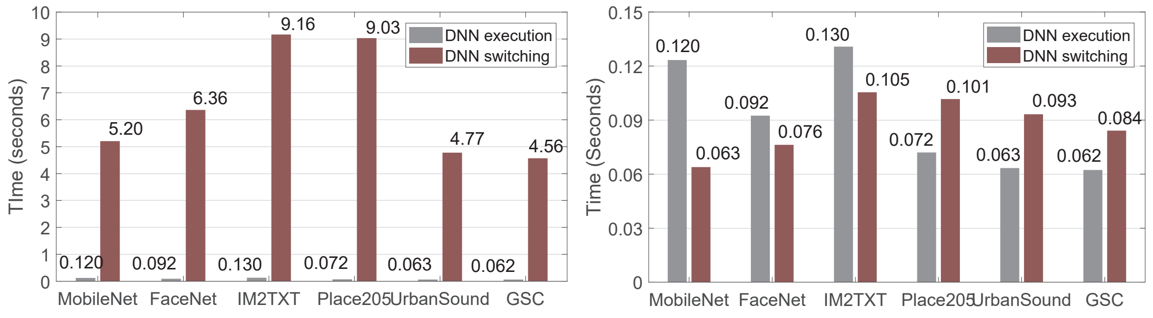


Figure 4.18: The end-to-end execution time of DNNs consecutively executed on the mobile robot: Baseline (no virtualization) vs. Weight virtualization.



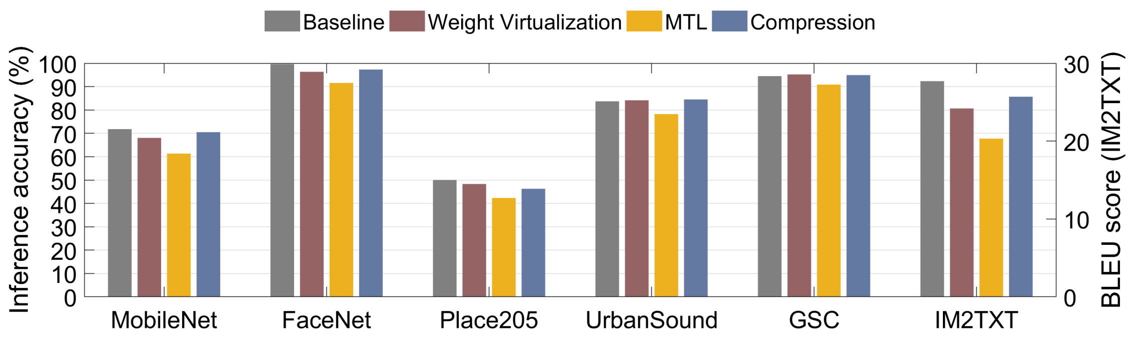
(a) Baseline (b) Weight virtualization

Figure 4.19: The DNN execution and switching time of the mobile robot: Baseline vs. Weight virtualization.

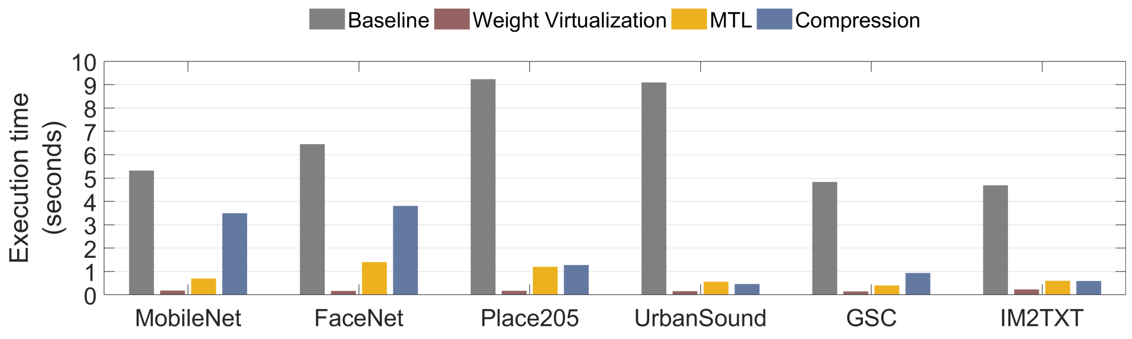
	MobileNet	FaceNet	IM2TXT	Place205	UrbanSound	GSC
Vanilla	54 MB	88 MB	142 MB	146 MB	88 MB	54 MB
Compressed	36 MB (He et al., 2018b)	52 MB (Wu et al., 2017)	19 MB (Zhang et al., 2018a)	4 MB (Han et al., 2015a)	16 MB (Han et al., 2015a)	18 MB (Han et al., 2015b)

Table 4.6: The memory usage of the baseline DNNs (uncompressed vanilla models), and individually compressed DNNs for the multitask learning robot. The compression algorithms are cited inside the brackets.

Figure 4.20a shows their inference accuracy where the virtualized DNNs achieve comparable accuracy to the MTL and individually compression approach, i.e., they perform slightly worse than the compression approach (-0.3%) but better than the MTL (+2.4%). However, the virtualized DNNs run significantly faster than the other two, i.e., 10.1x on average, as shown in Figure 4.20b. This is because 1) MTL executes the entire network for all tasks, and 2) individually compressed DNNs incur a high model switching overhead and are barely expedited due to reduced network size (Chen, 2018). This result demonstrates that weight virtualization achieves the best of both worlds, i.e., high accuracy and low execution latency.



(a) Inference accuracy with fixed memory size (146 MB)



(b) Execution time with fixed memory size (146 MB)

Figure 4.20: Comparison against multitask learning (MTL) and individually compressed DNNs.

4.6.2 Deep Multitask Learning IoT Device

Memory Packing Efficiency. Figure 4.21 and Table 4.7 show the number of weights and memory usage for the low-power multitask learning IoT device, where a total of 268,692 weights (523KB) are virtualized to 66,475 virtual weights (129KB). It achieves a 4.04x packing ratio when compared to the baseline DNNs not virtualized.

Inference Accuracy. Figure 4.22 shows the inference accuracy of the virtualized DNNs. Compared to the baseline DNNs that are not virtualized, they all achieve higher accuracy, e.g., the accuracy of GSC increases from 69.86% to 76.38% since weight virtualization decreases overfitting in the individual DNN models.

Execution Time. Figure 4.23 shows the end-to-end execution time of consecutive DNNs for 1) baseline DNNs that use an SD card as the secondary storage for loading DNNs to FRAM, and 2) virtualized DNNs that perform in-memory loading and execution entirely in FRAM. As shown in

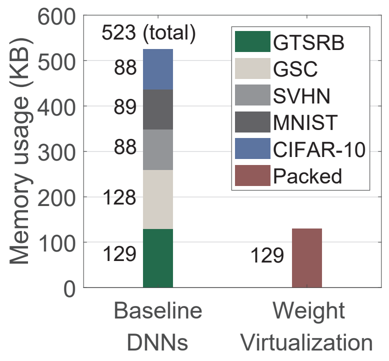


Figure 4.21: The memory usage of weight parameters: Baseline DNNs vs. Weight virtualization for the IoT device.

Baseline DNNs	**Total Weights**	**Memory (KB)**
GTSRB (Lee and Nirjon, 2019; Stallkamp et al., 2011a)	66,475	129
GSC (Chen et al., 2014a; Warden, 2018)	65,531	128
SVHN (Lee and Nirjon, 2019; Netzer et al., 2011)	45,490	88
MNIST (Gobieski et al., 2019b; LeCun et al., 1998)	45,706	89
CIFAR-10 (Yao et al., 2017b; Krizhevsky et al., 2009)	45,490	88
Total	**268,692**	**523**
Weight Virtualization	**Packed Weights**	**Memory (KB)**
GTSRB+GSC +SVHN+MNIST +CIFAR-10	**66,475**	**129**
Packing Ratio	**4.04x**	**4.04x**

Table 4.7: Weight memory packing with weight virtualization for the IoT device.

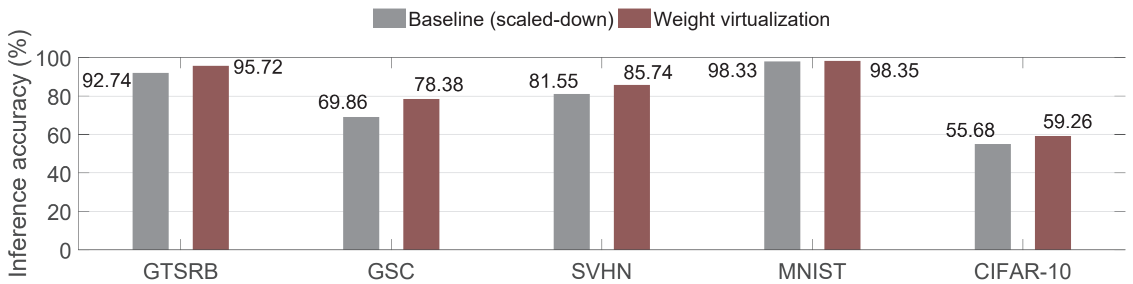


Figure 4.22: The inference accuracy of multitask IoT device: Baseline (scaled-down DNNs) vs. weight virtualization.

the figure, the execution time is improved when executing DNNs consecutively, e.g., 1.76x faster when executing all the five DNNs successively. Figure 4.24 shows the execution and switching (loading) time of each DNN. All the virtualized DNNs accelerate their switching (loading) by a maximum of 1,268x (2.41 vs. 0.0019 seconds in GTSRB) compared to the non-virtualized DNNs, which demonstrates that in-memory execution yields fast and responsive execution.

Energy Consumption. We measure the energy consumption of DNN executions on the multitask learning IoT device against the baseline that does not use weight virtualization by using EnergyTrace (Instrument, 2018) of MSP430 board. Figure 4.25 shows the total energy consumption of consecutive DNN executions for 1) the baseline DNNs that use an SD card, and 2) the virtualized DNNs that perform in-memory loading and execution in FRAM. In-memory execution reduces the total energy consumption by 4.24x when executing all five DNNs successively

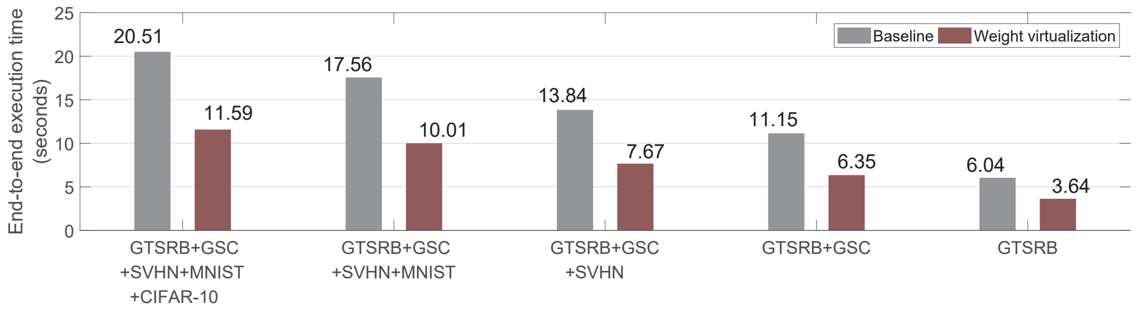


Figure 4.23: The end-to-end execution time of DNNs consecutively executed on the multitask learning IoT device: Baseline (no virtualization) vs. Weight virtualization.

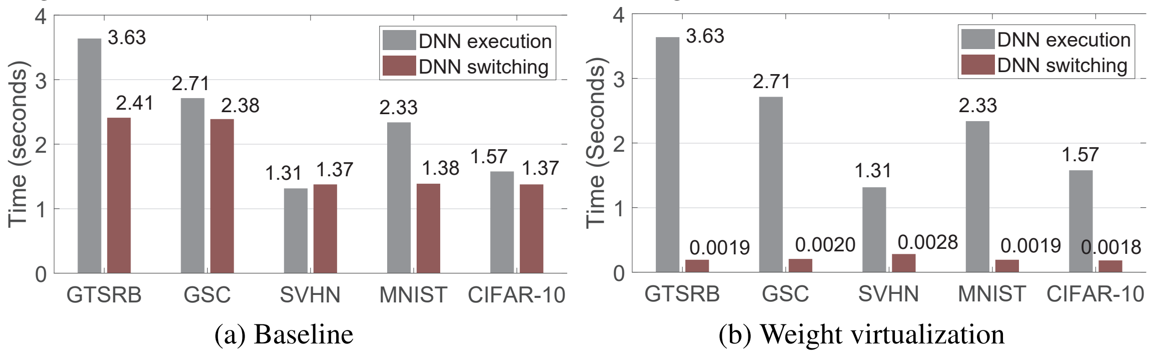


Figure 4.24: The DNN execution and DNN-switching time of the IoT device: Baseline (no virtualization) vs. Weight virtualization. The bars of DNN-switching time in (b) are drawn lager than the real size.

(432.95 vs. 102.09 mJ). Next, we breakdown the total energy consumption into two parts, i.e., 1) actual DNN execution and 2) DNN-switching (loading) energy, and measure each of them. Figure 4.26 shows the execution and switching energy for each DNN. All the virtualized DNNs improve their switching (loading) energy efficiency by a maximum of 7,413x (88.96 vs. 0.012 mJ in GTSRB) compared to the non-virtualized DNNs (the baselines).

Comparison to Alternatives. To evaluate the performance of virtualized DNNs against alternatives, we compare them with 1) a multitask learning (MTL) algorithm (Fang et al., 2018) and 2) individually compressed DNNs using (Han et al., 2015a) where the memory size of MTL and the combined memory size of all compressed DNNs is set to 129 KB for each. Table 4.8 shows the memory usages of compressed DNNs.

Figure 4.27a shows that the virtualized DNNs achieve comparable accuracy to the MTL that achieves the highest accuracy among all methods. The accuracy of MTL is average 3% higher

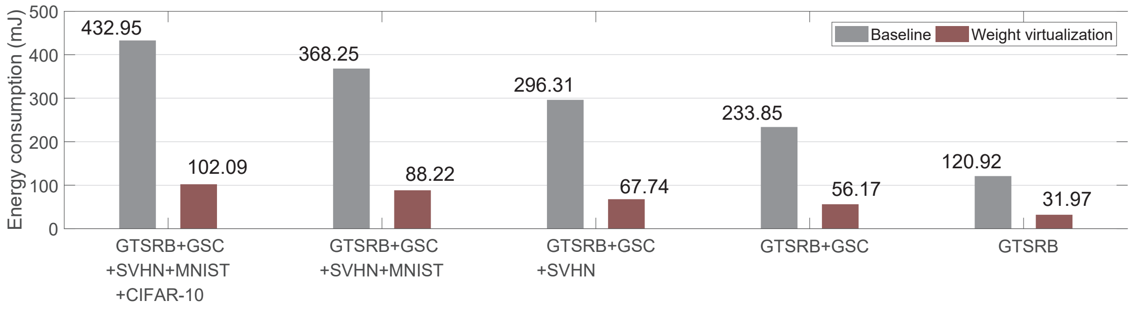


Figure 4.25: The total energy consumption of DNNs consecutively executed on the IoT device: Baseline (no virtualization) vs. Weight virtualization.

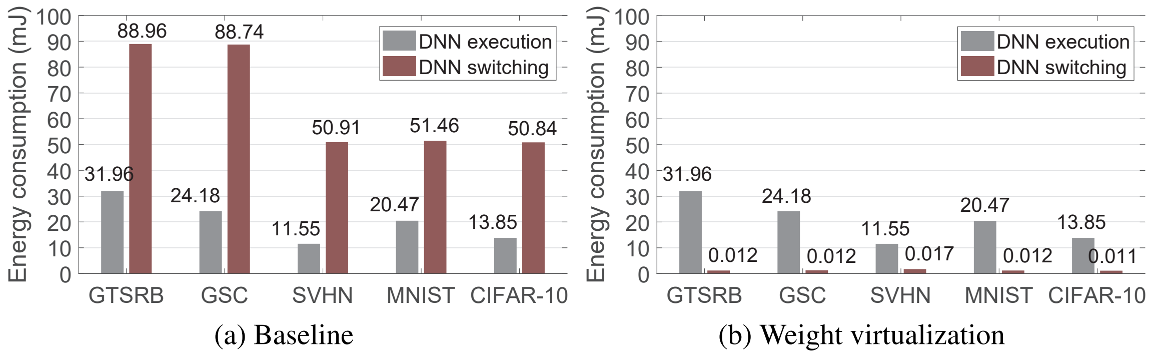


Figure 4.26: The energy consumption of DNN execution and DNN-switching for the IoT device: Baseline vs. Weight virtualization. The bars of DNN-switching energy in (b) are drawn larger than real.

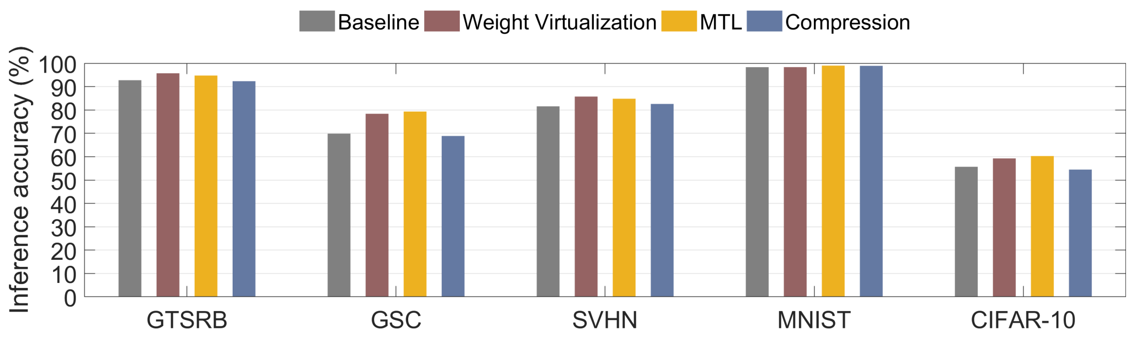
	GTSRB	GSC	SVHN	MNIST	CIFAR-10
Vanilla	129 KB	128 KB	88 KB	89 KB	88 KB
Compressed	27 KB (Han et al., 2015a)	34 KB (Han et al., 2015a)	25 KB (Han et al., 2015a)	10 KB (Han et al., 2015a)	30 KB (Han et al., 2015a)

Table 4.8: The memory usage of the baseline DNNs (uncompressed vanilla models), and individually compressed DNNs for the multitask learning IoT device. The compression algorithms are cited inside the brackets.

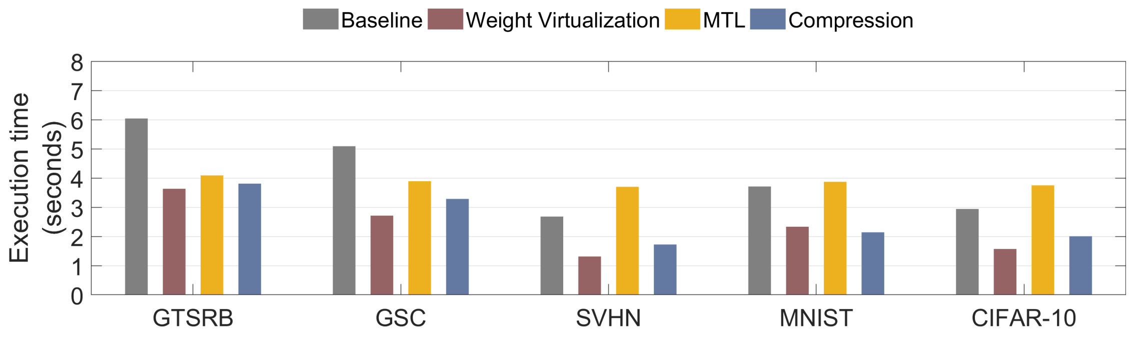
than the baseline DNNs, and the virtualized DNNs provides 0.1% lower accuracy on average than the MTL. Also, the virtualized DNNs execute up to 2.8x faster than the two alternatives, as shown in Figure 4.27b. This result demonstrates that weight virtualization provides the benefit of multitask learning, i.e., accuracy improvement via joint training while enabling fast in-memory execution at the same time.

4.7 Discussion

Fisher Information and Inference Accuracy. Fisher information (Lehmann and Casella, 2006) of weight parameters is used to optimize the loss function of a DNN, which is known as the



(a) Inference accuracy with fixed memory size (129 KB)



(b) Execution time with fixed memory size (129 KB)

Figure 4.27: Comparison against alternatives: multitask learning (MTL) and individually compressed DNNs.

natural gradient descent optimization (Amari et al., 1992; Amari, 1997, 1998; Park et al., 2000; Ollivier et al., 2017; Pascanu and Bengio, 2013). Since the weight parameters are proportionally updated to the Fisher information matrix in the natural gradient descent optimization in order to minimize the loss function that is closely related to the end-to-end accuracy, one can assume that reasonable accuracy is retained when the weight parameters of high Fisher information either remain the same or change very little. To understand their relationship, general proofs and theoretical assessment of Fisher information regarding the end-to-end inference accuracy and divergence from the optimal solution should be further studied. However, the exact mathematical relationship between the Fisher information and end-to-end accuracy is difficult to derive, which makes it challenging to provide a guaranteed inference accuracy when a large number of models are virtualized together.

Expansion to Secondary Storage. The proposed neural weight virtualization technique is primarily designed to operate purely in the main memory – without involving any secondary storage. An alternative design choice would be to extend the memory hierarchy of weight virtualization to include secondary storage or disks in the similar manner as in modern operating systems. It enables the system to retain the desired level of inference accuracy when the total model size is too large to fit into a small memory of the system in return for partial in-memory operation. By allowing disk-level weight virtualization, multitask learning of a more significant number of DNNs can be enabled with limited memory at scale without experiencing accuracy degradation.

4.8 Prior Work and Their Limitations

Multitask Learning. Multitask learning jointly trains correlated tasks to increase the accuracy of each learner by exploiting the commonalities and differences across tasks (Caruana, 1997; Ruder, 2017; Zhang and Yang, 2017a,b). Typical approaches include common feature learning (Liu et al., 2017; Zhang et al., 2016; Misra et al., 2016; Liu et al., 2015b; Mrkšić et al., 2015; Li et al., 2014b; Zhang et al., 2014), low-rank parameter search (Yang and Hospedales, 2016b; Han and Zhang, 2016; McDonald et al., 2014; Agarwal et al., 2010; Chen et al., 2009; Zhang et al., 2008), task clustering (Zhou and Zhao, 2015; Barzilai and Crammer, 2015; Han and Zhang, 2015; Kumar and Daume III, 2012; Kang et al., 2011; Thrun and O'Sullivan, 1996), and task relation learning (Long et al., 2017; Lee et al., 2016; Zhang and Yeung, 2013a,b). Although, by sharing network structure (typically, the first few layers) they achieve limited compression (He et al., 2018a), their primary goal is to increase robustness and generalization of correlated and similar-structured learners. Thus, packing multiple heterogeneous learners into extremely scarce memory of an embedded system, as well as managing, context switching, and executing different DNN tasks efficiently at run-time, are challenging to them, both of which are achieved by weight virtualization.

DNN Combining (Packing). Stacked neural networks (SNN) (Mohammadi and Das, 2016; Sridhar et al., 1999, 1996) combine multiple DNNs by adding a layer on top of the features extracted from the DNNs. Although it may achieve better accuracy, the weights of each DNN needs to be maintained, which does not reduce the memory size, and the DNNs are not trained with multi-task learning. PackNet (Mallya and Lazebnik, 2018) packs multiple DNNs to a single network with iterative pruning based on redundancies in DNNs to free up weights that can be employed for new tasks. However, the number of DNNs can be limited when free weights fall short as more DNNs are packed, and only a single network is maintained. Although (Chou et al., 2018a) merges DNNs by integrating convolutional layers, it works with only two DNNs and requires to align layers to merge them. Learn-them-all (Kaiser et al., 2017; Aytar et al., 2017; Levi and Hassner, 2015) trains a single complex DNN to handle multiple tasks simultaneously. However, it is hard to choose a suitable architecture for learning all the tasks well in advance. Besides, learning from a large training data from different types or sources are demanding.

DNN Weight Sharing. For a single DNN, soft weight sharing approaches (Nowlan and Hinton, 1992; Ullrich et al., 2017) such as the Dirichlet process (Roth and Pernkopf, 2018), k-means clustering (Son et al., 2018; Han et al., 2015a), or quantization (Köksal et al., 2001) have been proposed. These techniques do not provide much benefit in terms of memory usage since the assignments of weights to connections must be stored additionally. There are several studies on weight sharing between multiple DNNs. MultiTask Zipping (He et al., 2018a) merges DNNs for cross-model compression with a layer-wise neuron sharing. Sub-Network Routing (Ma et al., 2019) modularizes the shared layers into multiple layers of sub-networks. Cross-stitch Networks (Misra et al., 2016) apply weight sharing (Duong et al., 2015) after pooling and fully-connected layers of two DNNs. MMoE (Ma et al., 2018) learns to model task relationships from data by sharing the sub-models and weights across tasks. Tensor factorization (Yang and Hospedales, 2016a) divides each set of parameters into shared and task-specific parts. However, their scope and methods of weight sharing are limited by network architecture and task type, unlike neural weight virtualization, where weights are shared without imposing a limitation on the network structure.

DNN Compression. The need to deploy DNNs on mobile systems motivated several techniques that reduce memory and computational costs, including knowledge distillation (Chen et al., 2017a; Hinton et al., 2015; Romero et al., 2014), low-rank factorization (Ioannou et al., 2015; Tai et al., 2015; Sainath et al., 2013), pruning (LeCun et al., 1990; Polyak and Wolf, 2015; Li et al., 2016; Yu et al., 2018; Guo et al., 2016), quantization (Li et al., 2017a; Wu et al., 2016; Han et al., 2015a), compression with structured matrices (Cheng et al., 2015; Sindhwani et al., 2015), network binarization (Li et al., 2017b; Rastegari et al., 2016; Courbariaux et al., 2016), and hashing (Chen et al., 2015b). However, they do not provide cross-DNN compression that trains multiple DNNs together, unlike neural weight virtualization. Instead, each DNN is compressed individually with different compression methods, which is not scalable and does not achieve the benefits of multitask learning. Furthermore, a significantly compressed DNN does not run nearly as significantly faster since most parameters are pruned in fully-connected layers while convolutional layers consume most computation time, as shown in (Guo et al., 2016; Park et al., 2016b; Han et al., 2015a).

4.9 Summary

We introduces neural weight virtualization that enables scalable and fast multitask learning of DNNs on resource-constrained systems. By virtualizing network weights, it packs multiple DNNs into a limited size main memory. In-memory multitask learning enabled by weight virtualization improves the execution and response time, as well as the energy efficiency of the system. We implement two multitask learning systems: a mobile robot and an IoT device, and demonstrate that memory efficiency, execution time, and energy efficiency increases with weight virtualization.

CHAPTER 5: ADAPTIVE REAL-TIME LEARNING

Recently, DNNs (deep neural networks) (Goodfellow et al., 2016; LeCun et al., 2015; Schmidhuber, 2015) have been increasingly used in many real-life applications due to their superiority in solving complex machine learning problems (Young et al., 2018; Schroff et al., 2015; Krizhevsky et al., 2012), e.g., autonomous cars (Bojarski et al., 2017; Chen et al., 2017c, 2016b, 2015a), natural language processing (Socher et al., 2012; Deng and Liu, 2018; Khan et al., 2016), and healthcare applications (Miotto et al., 2017; Jiang et al., 2017; Litjens et al., 2017). However, their long and unpredictable execution time resulting from a significant amount of computation often limits their deployment on real-time systems. Although high-performance hardware such as multi-core CPUs or GPUs efficiently process the massive workload of a DNN in parallel, the complexity and proprietary architecture of these platforms make effective scheduling of deadline-aware DNN tasks challenging, as shown in many previous works (Dong et al., 2017b; Elliott and Anderson, 2014; Zhou and Liu, 2014; Elliott et al., 2013; Elliott and Anderson, 2013, 2012; Kato et al., 2012, 2011; Rossbach et al., 2011; Augonnet et al., 2010; Luk et al., 2009).

Moreover, the time constraints of many practical systems dynamically change at run-time, making DNNs more challenging to be executed as a real-time task. Such dynamic time constraints are found in many modern embedded systems such as autonomous cars (Taş et al., 2016; Pongpunwattana and Rysdyk, 2004; Shiller et al., 1991), drones (Chen et al., 2017b; Nägeli et al., 2017; Soto et al., 2007), and smartphones (He et al., 2015; Wanpeng and Wei, 2014; Balog et al., 2002) where the system must deal with online changes such as run-time application requirements, resource availability, energy level, failures, and re-configurations. Such changes consequently cause variations in the time requirements of related-tasks (Stewart and Khosla, 1991); e.g., data-dependent requirements where the periods depend on the input sensor data; time-dependent

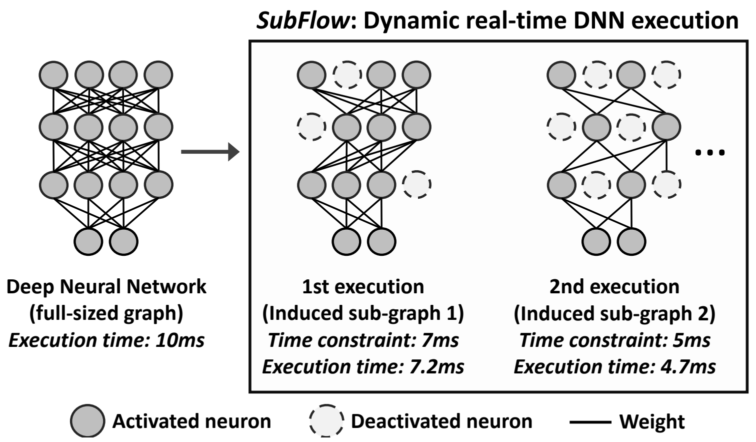


Figure 5.1: *SubFlow* enables real-time inference and training of a DNN by *dynamically executing* a sub-graph of the DNN according to the timing constraint changing at run-time. For each inference or training execution, an *induced sub-graph* of the DNN, whose execution is completed in time, is constructed and executed by activating only necessary neurons, enabling *time-aware utilization* of the DNN.

requirements where the actual deadline becomes known only at run-time when setting the actuators. For example, autonomous vehicles impose dynamic time constraints on tasks in reaction to a variety of road situations—a lower latency for obstacle detection is expected when traveling at higher speeds or when a pedestrian makes a sudden appearance. Failure of a scheduler also introduces variability in timing constraints, which reduces the amount of allowed execution time. A task scheduler in a complex and dynamic system may fail to start a task at its latest allowed start time and miss the deadline.

Although DNN compression techniques such as (Chen et al., 2017a; Hinton et al., 2015; Romero et al., 2014; Ioannou et al., 2015; Tai et al., 2015; Sainath et al., 2013; LeCun et al., 1990; Polyak and Wolf, 2015; Li et al., 2016; Yu et al., 2018; Guo et al., 2016) reduce the execution time to some extent, they are not directly applicable to DNNs having dynamic timing constraints since 1) they generate only one compressed network from the original DNN, which does not dynamically adapt once deployed, 2) most of them primarily focus on reducing memory usage as opposed to speedup, and 3) most compression methods are time-consuming as they require multiple training iterations and fine-tuning, and are limited to specific types of DNNs.

To enable the execution of DNNs with dynamic deadline constraints, we introduce *SubFlow*—an online DNN sub-graph strategy that constructs and executes a sub-graph of a DNN called the *sub-network* that completes the inference or training tasks within timing constraints that may change at run-time. The process is shown in Figure 5.1. For each execution of the DNN inference/training task, a different sub-network of different size and composition is constructed on-the-fly and executed within the time budget. In this way, the system ensures time-aware execution of a DNN with a flexible time budget–which also improves the CPU utilization and schedulability.

SubFlow consists of two run-time algorithms: 1) *dynamic construction* and 2) *time-bound execution* of a sub-network. For construction, it composes the proper sub-network based on *the importance of neurons* in such a way that the execution time is expected to match the time budget while the performance loss due to the reduced size of sub-network is minimized. For execution, we propose *time-bound inference* and *training* of convolutional and fully-connected layers of a DNN, which are two main building blocks of a DNN. We name them time-bound since their inference and training times are bounded by the architecture and size of the sub-network.

We implement SubFlow by extending TensorFlow (Abadi et al., 2016), one of the most popular DNN frameworks, and we open-source it[1]. We develop four custom operations and libraries for time-bound feed-forward and back-propagation (Rumelhart et al., 1985) of dynamic sub-networks, i.e., *sub-convolution*, *sub-multiplication*, *sub-convolution-gradient*, and *sub-multiplication-gradient*. They support both CPUs and GPUs, and are implemented by using Eigen (Guennebaud et al., 2010) and CUDA (NVIDIA, 2019b) libraries, respectively. DNNs designed with TensorFlow are easily adapted to SubFlow by simply applying SubFlow operations to the model, without requiring any architectural modifications, which makes SubFlow universal and applicable to existing DNNs. Since a new sub-network is constructed at run-time as opposed to constructing and saving a set of sub-networks offline, no additional memory is needed in SubFlow for this purpose.

[1]SubFlow Project: `https://github.com/learning1234embed/SubFlow`

We evaluate SubFlow for three standard DNN architectures, i.e., LeNet-5 (LeCun et al., 1998), AlexNet (Krizhevsky et al., 2012), and KWS (Key-Word Spotting) (Sainath and Parada, 2015) on three popular datasets, i.e., MNIST (LeCun et al., 1998), CIFAR-10 (Krizhevsky et al., 2009), and GSC (Google Speech Commands) (Warden, 2018), respectively. Experiments are conducted on various hardware platforms: CPU (x86 and ARM) and GPU (RTX 2080 Ti and Jetson Nano (NVIDIA, 2019a)). The evaluation results show that both inference and training time of DNNs change dynamically according to the size and configuration of the sub-networks while achieving comparable performance to the full-sized network. For instance, the execution speed of AlexNet (Krizhevsky et al., 2012) dynamically changes between 1x and 6.7x while only a 3% inference accuracy drop is observed on average.

We also implement a mobile robot using an embedded GPU platform (Jetson Nano (NVIDIA, 2019a)) as an example real system that uses SubFlow where the latency requirement for obstacle detection changes due to the traveling speed of the robot. In this real-life experiment, the robot runs at various speeds and the execution of the DNN (Chakravarty et al., 2017) that detects obstacles is adapted dynamically depending upon varying deadlines.

The main contributions of this chapter are:

- We introduce *SubFlow*, a real-time DNN execution strategy enabling flexible time-bound inference and training that is completed within a dynamic time budget by constructing and executing a sub-network of the DNN at run-time.
- We propose an online sub-network construction algorithm to determine the proper sub-graph of a DNN with a minimum performance loss based on induced sub-graph (Diestel, 2006) method whose execution time is matched to a dynamic time budget.
- We propose time-bound feed-forward and back-propagation of convolutional and fully-connected layers of a DNN, where the total computation time is bounded by the size and configuration of the sub-network.

• We implement and open-source SubFlow[1] by developing four custom operations of TensorFlow with full compatibility, which allows the DNN designers to transform their DNNs into real-time dynamic DNNs easily.

• We develop and demonstrate a mobile robot as an example real system that uses SubFlow. The robot executes a depth estimation DNN for obstacle detection with its time constraint that dynamically changes based on the speed of the robot.

5.1 Overview

The goal of SubFlow is to enable execution of DNN inference and training tasks in such a way that the task is completed under dynamically varying time constraints while retaining comparable performance to executing the original full-size DNN. The flexible execution increases the utility of a DNN by letting it meet a range of deadlines at run-time, which conventional DNNs cannot. SubFlow also facilitates flexible scheduling of multitask learning where new tasks can be accommodated by dynamically updating the deadline of existing ones. The schedulability of a system running multiple DNNs can be improved by taking into account the flexible execution time of DNNs in the scheduling decision at run-time, which increases the total system utilization.

An unbounded trade off of inference accuracy for real-time execution of a DNN is not desirable in most systems. Hence, to limit the maximum loss of accuracy above a certain level, SubFlow limits the execution of sub-networks whose expected accuracy is lower than the desired level. SubFlow enables this by imposing a limit on the minimum network utilization parameter (defined in Section 5.2) that essentially defines the size of the sub-network. The minimum network utilization parameter is empirically determined and is set by the developer or the system admin.

5.1.1 SubFlow Operations

SubFlow has three major operations, which are shown in Figure 5.2. A brief description of each operation follows.

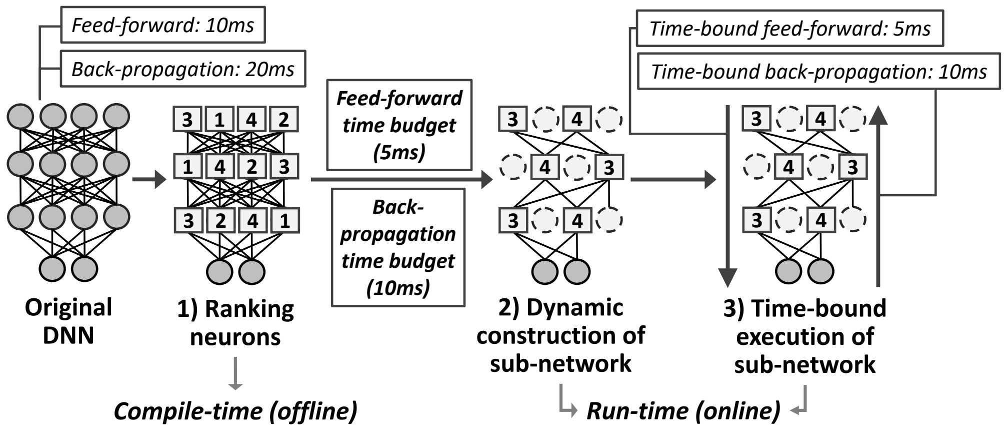

Feed-forward: 10ms
Back-propagation: 20ms
Original DNN
1) Ranking neurons
Feed-forward time budget (5ms)
Back-propagation time budget (10ms)
2) Dynamic construction of sub-network
Time-bound feed-forward: 5ms
Time-bound back-propagation: 10ms
3) Time-bound execution of sub-network
Compile-time (offline)
Run-time (online)

Figure 5.2: SubFlow operations: SubFlow consists of three steps: ranking neurons, dynamic construction, and time-bound execution of a sub-network. Ranking of neurons is done at compile-time. At run-time, inference or training jobs with different time budgets are executed by forming and executing dynamic sub-networks.

1) Ranking Neurons. Given a trained DNN on which we want to apply SubFlow, the utility/contribution of each neuron to the performance (inference accuracy) of the DNN is computed. This is calculated only once at compile-time. The details of this step are described in Section 5.3.

2) Dynamic Construction of Sub-Network. At run-time, a sub-network of the DNN is dynamically constructed for each job of a DNN task according to the given time budget. For example, an image classification task releases a job (say, every 500ms) where the job is to classify an image taken with the camera. For every job, an induced sub-graph (Diestel, 2006) with a different subset of neurons (vertices) is constructed based on their importance calculated at compile-time. The construction of a sub-network is described in Section 5.3.

3) Time-Bound Execution of Sub-Network. Each sub-network corresponding to a job is executed and completed within the given time budget. The execution time of a job is bounded by the size and configuration of sub-networks. To enable the time-bound execution of the sub-network, we propose time-bound feed-forward and back-propagation (Rumelhart et al., 1985) algorithms, which are described in Section 5.4.

5.1.2 An Example Application

As an application of SubFlow, we describe an autonomous mobile robot which is one of many application-specific systems where SubFlow is applicable. We identify two real-time inference tasks of the robot (i.e., obstacle detection and sensor-based control) that have dynamic timing constraints. In section 5.7, as a proof of concept system, we implement and evaluate the obstacle detection task on an embedded GPU-enabled autonomous mobile robot.

Obstacle Detection. In most autonomous cars and robots of today, data captured by cameras and other on-board sensors are processed by convolutional neural networks (CNNs) (Li et al., 2019; Pan et al., 2018; Gao et al., 2018; Al-Qizwini et al., 2017; Drews et al., 2017) to detect obstacles and to take timely measures to avoid collisions. For example, Tesla's autopilot (Ingle and Phute, 2016) constructs depth maps using cameras to create 3D point maps of their surroundings, measuring objects' distance (Wang et al., 2019; Chakravarty et al., 2017; Chen et al., 2016b; Mancini et al., 2016; Eigen et al., 2014). The real-time requirement of obstacle detection task in these systems becomes tighter when the vehicle is moving at a relatively higher speed – requiring the CNN to complete its processing faster. In contrast, when the vehicle is moving at a lower speed, the timing requirements are relaxed, allowing more time for the CNN to complete execution.

Sensor-based Control. Real-time requirements for sensor-based control systems of a mobile robot may change dynamically at run-time (Stewart and Khosla, 1991). For example, tactile sensors on a mobile robot measure the force (and torque) exerted on its body (Kappassov et al., 2015; Badreddin, 1992), which helps collision avoidance (Badreddin, 1992). Depending on whether the robot is likely to be in contact with an object, it can adapt its sampling frequency of the sensors and thus scale its computation accordingly. Such dynamics not only changes the timing requirements of the tactile sensing and collision inference task but also affects the timing requirements of other inference tasks that are concurrently running in the system.

5.1.3 Programmability

SubFlow provides a set of DNN operations fully compatible with the existing TensorFlow operations, which allows a programmer to easily design and execute a DNN with SubFlow. Listing 5.1 and 5.2 show an example code of TensorFlow and SubFlow written for a convolutional layer, respectively. The implementation of SubFlow can be found in our GitHub[1].

Listing 5.1: TensorFlow programming example.

```
# DNN designing (a convolution layer)
output = tensorflow.nn.conv2d(input, filters, ...)
# DNN execution
sess.run(..., feed_dict={...})
```

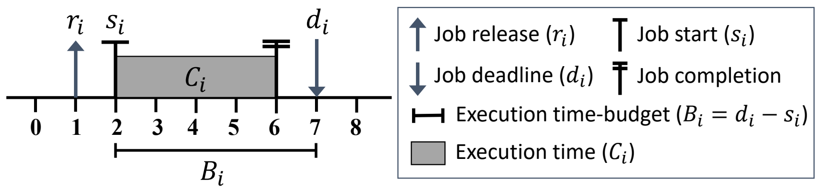
Listing 5.2: SubFlow programming example.

```
# DNN designing (a convolution layer)
output = subflow.conv2d(input, filters, ..., activation)
# DNN execution
 activation_vector = get_activation ( network_utilization )
sess.run(..., feed_dict={..., activation: activaiton_vector})
```

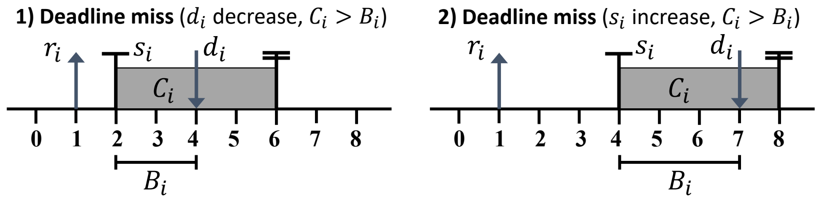
5.2 Background and Terminologies

SubFlow regards an inference or training task of a DNN as a real-time task τ with period T, execution time C, release time r, and relative and absolute deadline D and d, which is scheduled along with other tasks in the system. A DNN task, τ releases a sequence of jobs, J corresponding to the execution of a single iteration of inference or training that needs to be completed within the deadline, D as shown in Figure 5.3a.

Dynamic Execution Time Budget. We define *execution time budget* for the i-th job J_i as $B_i = d_i - s_i$, where d_i is the absolute deadline and s_i is the start time of J_i. Since B_i for different J_i may be different, we call it a dynamic execution time budget. It is equivalent to the maximum allowed execution time for J_i to meet the deadline. Obviously, J_i meets its deadline if $B_i \geq C_i$, where C_i is the execution time of J_i. On the other hand, if $B_i < C_i$, J_i cannot meet the deadline. The latter case, i.e., $B_i = d_i - s_i < C_i$ happens in two situations: 1) d_i has decreased due to the system or application induced run-time variations in timing requirements, and 2) s_i has increased due to a scheduling failure or unavailable resources, causing J_i to be executed too late to complete within the deadline. Figure 5.3b illustrates an example of these two cases.

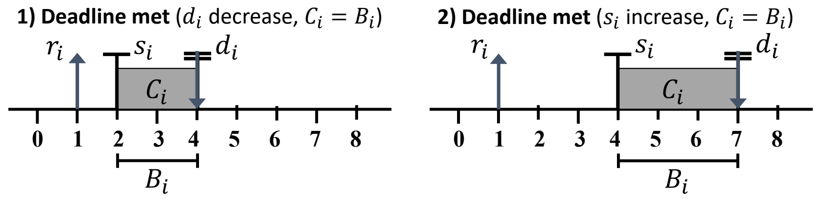


(a) An example job execution of a task with $T = 6$ and $C = 4$.



1) Deadline miss (d_i decrease, $C_i > B_i$)
2) Deadline miss (s_i increase, $C_i > B_i$)
r_i
s_i
d_i
C_i
0 1 2 3 4 5 6 7 8
B_i

(b) Two cases of dynamic execution time budget, which causes deadline miss unless execution time, C_i is adapted.



1) Deadline met (d_i decrease, $C_i = B_i$)
2) Deadline met (s_i increase, $C_i = B_i$)
r_i
s_i
d_i
C_i
0 1 2 3 4 5 6 7 8
B_i

(c) The job meets the deadline by adjusting C_i to B_i for both cases.

Figure 5.3: An example of a dynamic execution time budget: Given a dynamic execution time budget, SubFlow allows the job to meet the deadline by adjusting its execution time according to the budget.

Sub-Network and Execution Time. SubFlow enables a DNN task, τ to complete J_i within B_i even if $B_i < C_i$ by reducing the execution time to the given time budget, i.e., $C_i \rightarrow B_i$. For each

J_i, SubFlow constructs and executes a sub-network, τ_i^s that is a subset of the full-size network, τ, which is able to complete its execution within B_i. Figure 5.3c shows an example in which the job meets the deadline by adjusting C_i to B_i.

Network Utilization. We quantify the size of a sub-network using *network utilization*, $u_i \in (0, 1]$ which is the relative size ratio of a sub-network constructed for the job, J_i to the full-size network as defined in Equation 5.1. Note that it is different from the task utilization used in scheduling, i.e., $U_i = C_i/T_i$.

Properties of DNNs. The construction of sub-networks having different execution times is enabled by three unique properties of DNNs: 1) many different configurations and sizes of DNN graphs often result in a similar performance (Hecht-Nielsen, 1992; Sussmann, 1992), 2) the over-sized architecture of modern DNNs, i.e., the massive number of neurons and parameters allows incorporating multiple sub-networks into a single larger network (Zhu et al., 2017), and 3) most DNNs repeat the same computation for each layer, i.e., convolution, matrix multiplication, etc., which makes the estimation of computation time for a sub-network feasible (Justus et al., 2018).

Induced Sub-Graph. SubFlow constructs a sub-network based on the induced sub-graph (Diestel, 2006) of a DNN graph. Given a graph $G = (V, E)$ where $S \subset V$ be any subset of vertices of G, the induced sub-graph $G[S]$ is defined as the graph whose vertex set is S and whose edge set consists of all of the edges in E that have both endpoints in S.

In SubFlow, the neurons at each layer are considered as vertices, and connections between two neurons with a weight parameter are considered as edges. By activating the right subset of neurons and using only the edges connected to them as an induced sub-graph, a sub-network with the desired size and configuration is constructed. We use induced sub-graph since it is based on the selection of vertices (neurons), not edges (weight parameters). For most DNNs, the number of neurons is several orders of magnitude smaller than weight parameters, which enables efficient construction of sub-networks.

5.3 Dynamic Construction of Sub-Network

The first two steps of SubFlow are the ranking of neurons and the dynamic construction of a sub-network. This section first describes the construction step starting from the definition of sub-network and then discusses how neurons are ranked for the construction of a sub-network.

5.3.1 Definition of Sub-Network

Basic DNN Operation. Before defining the sub-network, we describe the basic operation of DNNs. Given a training or test dataset for a DNN, τ having n instances, we denote the entire dataset as $\{(\mathbf{x}_j, \mathbf{y}_j)\}_{j=1}^{n}$. For the j-th instance, the input and output neurons of layer l is denoted as $\mathbf{o}_j^{l-1} \in \mathbb{R}^{m^{l-1}}$ and $\mathbf{o}_j^{l} \in \mathbb{R}^{m^{l}}$, respectively. The output of layer l, $\mathbf{o}_j^{l}$ is obtained by performing $\mathbf{o}_j^{l} = \sigma\left(\mathbf{y}_j^{l}\right)$, where $\mathbf{y}_j^{l} = \mathbf{o}_j^{l-1^{\top}}\mathbf{W}^{l} + \mathbf{b}^{l}$ with $\mathbf{W}^{l} \in \mathbb{R}^{m^{l-1} \times m^{l}}$ being the weight parameter and $\mathbf{b}^{l} \in \mathbb{R}^{m^{l}}$ being the bias for layer l. For nonlinearity, an nonlinear function $\sigma(\cdot)$ such as sigmoid function (Han and Moraga, 1995) is applied to $\mathbf{y}_j^{l}$. Following the recent trend in state-of-the-art DNNs such as (Szegedy et al., 2015; Simonyan and Zisserman, 2014; Krizhevsky et al., 2012), we use the rectified linear unit (ReLU) (Glorot et al., 2011) as our $\sigma(\cdot)$. Although this is a formulation for a fully-connected layer, it also applies to convolution layers by converting a kernel operation with input into a matrix product as in (Yao et al., 2017b).

Sub-Output Neuron. To construct a sub-network, τ_i^{s} for the job J_i from a DNN task, τ, we first compose *sub-output neuron*, $\tilde{\mathbf{o}}_j^{l} \in \mathbb{R}^{m^{l}}$ for each layer l, which is a sparse vector of the same length with the output neuron, $\mathbf{o}_j^{l} \in \mathbb{R}^{m^{l}}$ at the l-th layer of τ. It consists of a subset of $\mathbf{o}_j^{l}$ and zeros. Having $\tilde{\mathbf{o}}_j^{l}$ for all layers, we create a sub-network by connecting only the non-zero elements of $\tilde{\mathbf{o}}_j^{l}$ (activated vertices) based on the induced sub-graph construction (Diestel, 2006). Depending on $\tilde{\mathbf{o}}_j^{l}$, only a subset of weight parameter elements (edges) in $\mathbf{W}^{l}$ connected between $\tilde{\mathbf{o}}_j^{l-1}$ and $\tilde{\mathbf{o}}_j^{l}$ is activated for the sub-network. The elements of $\tilde{\mathbf{o}}_j^{l}$ are obtained by multiplying $\mathbf{o}_j^{l}$ with a binary vector, $\mathbf{a}_i^{l} \in \mathbb{R}^{m^{l}}$ called *activation vector* that determines whether the corresponding neuron element of $\mathbf{o}_j^{l}$ is activated or not in $\tilde{\mathbf{o}}_j^{l}$ as a vertex of induced sub-graph. In summary, the sub-output neuron of layer l, $\tilde{\mathbf{o}}_j^l$ for the j-th input instance is given by:

$$\begin{aligned}
&\mathbf{a}_i^l \in \{0,1\}^{m^l} \text{ s.t. } \left\|\mathbf{a}_i^l\right\|_1 = \left\lfloor u_i^l \cdot m^l \right\rfloor \text{ or } u_i^l = \frac{\left\|\mathbf{a}_i^l\right\|_1}{m^l} \\
&\tilde{\mathbf{y}}_j^l = \tilde{\mathbf{o}}_j^{l-1^\top} \mathbf{W}^l + \mathbf{b}^{l^\top} \\
&\tilde{\mathbf{o}}_j^l = \mathbf{a}_i^l \circ \sigma\left(\tilde{\mathbf{y}}_j^l\right) = \sigma\left(\mathbf{a}_i^l \circ \tilde{\mathbf{y}}_j^l\right) \text{ since } \sigma\left(\cdot\right) \text{ is ReLU}
\end{aligned} \tag{5.1}$$

where $\mathbf{a}_i^l$ is the activation vector, m^l is the length of $\tilde{\mathbf{o}}_j^l$ and $\mathbf{o}_j^l$, $\|\cdot\|_1$ is ℓ_1-norm, u_i^l is the network utilization, $\mathbf{W}^l$ is the weight parameter, $\mathbf{b}^l$ is the bias, $\sigma\left(\cdot\right)$ is the nonlinear function (ReLU), and $\circ$ is Hadamard (element-wise) product (Davis, 1962).

Definition of Sub-Network. We define sub-network, τ_i^s for the job J_i as an induced sub-graph of a DNN, τ with total L layers, where its vertices are composed of the sub-output neurons of all layers, $\{\tilde{\mathbf{o}}_j^l | 1 \leq l \leq L\}$ defined in Equation 5.1.

5.3.2 Construction of Sub-Network

Construction Objectives. Given a dynamic execution time budget B_i for the job J_i, a sub-network having total layers L, τ_i^s is constructed to achieve two objectives: 1) finding the maximum network utilization for each layer, $\mathbf{u}_i = [u_i^1, u_i^2, ..., u_i^L]$ in such a way that their total execution time is equivalent to or less than B_i, and 2) finding the activation vector, $\mathbf{a}_i^l$ for each layer l to determine sub-output neuron, $\tilde{\mathbf{o}}_j^l$ in which the number of elements of value one in $\mathbf{a}_i^l$, i.e., $\left\|\mathbf{a}_i^l\right\|_1$ is equivalent to $\left\lfloor u_i^l \cdot m^l \right\rfloor$ as defined in Equation 5.1 in such a way that the error between $\tilde{\mathbf{o}}_j^l$ and $\mathbf{o}_j^l$ is minimized.

Finding Network Utilization. To find the network utilization $\mathbf{u}_i$, we define execution time $C(\mathbf{u}_i)$ of a sub-network, τ_i^s as:

$$C(\mathbf{u}_i) \propto \kappa \sum_{l=1}^{L} u_i^l f^l\left(\mathbf{o}_j^l\right) \propto \kappa N_i \sum_{l=1}^{L} f^l\left(\mathbf{o}_j^l\right) \text{ if } u_i^l = N_i \tag{5.2}$$

where κ is a constant, $u_i^l \in (0, 1]$ is the network utilization of layer l, and $f^l\left(\mathbf{o}_j^l\right)$ is the execution time function of layer l, i.e., the time required to compute $\mathbf{o}_j^l$. We apply the same network utilization, N_i to all layers, i.e., $u_i^l = N_i$ since finding a set of optimal u_i^l is NP-hard, i.e., the search space for all combinations of u_i^l is exponential. Also, by activating the same proportion of neurons at each layer, the resulting sub-network is not to be cut or broken. Hence, $\mathbf{u}_i$ can be obtained by finding the maximum N_i satisfying $C(N_i) \leq B_i$ in Equation 5.2.

Layer-Wise Error. To obtain sub-output neuron, $\tilde{\mathbf{o}}_j^l$ of layer l that minimizes the error from $\mathbf{o}_j^l$, we define layer-wise error between $\tilde{\mathbf{o}}_j^l$ and $\mathbf{o}_j^l$ for both the j-th training instance and the total n of training instances, similar to (Dong et al., 2017a). They are denoted as $E_j^l(\tilde{\mathbf{o}}_j^l)$ and E^l, respectively, which are defined as:

$$\begin{aligned}
\hat{\mathbf{o}}_j^l &= \tilde{\mathbf{o}}_j^l - \mathbf{a}_i^l \circ \mathbf{o}_j^l \\
E_j^l(\tilde{\mathbf{o}}_j^l) &= \hat{\mathbf{o}}_j^{l\top} \hat{\mathbf{o}}_j^l = \left\|\tilde{\mathbf{o}}_j^l - \mathbf{a}_i^l \circ \mathbf{o}_j^l\right\|_2^2 \\
E^l &= \frac{1}{n}\sum_{j=1}^{n} E_j^l(\tilde{\mathbf{o}}_j^l) = \frac{1}{n}\sum_{j=1}^{n}\left\|\tilde{\mathbf{o}}_j^l - \mathbf{a}_i^l \circ \mathbf{o}_j^l\right\|_2^2
\end{aligned} \tag{5.3}$$

where $\circ$ is Hadamard (element-wise) product, $\|\cdot\|_2$ is ℓ_2-norm, and $\hat{\mathbf{o}}_j^l = \tilde{\mathbf{o}}_j^l - \mathbf{a}^l \circ \mathbf{o}_j^l$ is the output error vector of layer l.

Error Bound. From Equation 5.1 and $\|\sigma(\mathbf{x}) - \sigma(\mathbf{y})\|_2 \leq \|\mathbf{x} - \mathbf{y}\|_2$, the property of the ReLU, $E_j^l(\tilde{\mathbf{o}}_j^l)$ is bounded by:

$$\begin{aligned}
E_j^l(\tilde{\mathbf{o}}_j^l) &= \left\|\tilde{\mathbf{o}}_j^l - \mathbf{a}^l \circ \mathbf{o}_j^l\right\|_2^2 \\
&= \left\|\sigma\left(\mathbf{a}^l \circ \tilde{\mathbf{y}}_j^l\right) - \sigma\left(\mathbf{a}^l \circ \mathbf{y}_j^l\right)\right\|_2^2 \\
&\leq \left\|\mathbf{a}^l \circ \tilde{\mathbf{y}}_j^l - \mathbf{a}^l \circ \mathbf{y}_j^l\right\|_2^2 \\
&= \left\|\mathbf{a}^l \circ \left(\left(\tilde{\mathbf{o}}_j^{l-1} - \mathbf{a}^{l-1} \circ \mathbf{o}_j^{l-1} + \mathbf{a}^{l-1} \circ \mathbf{o}_j^{l-1} - \mathbf{o}_j^{l-1}\right)^\top \mathbf{W}^l\right)\right\|_2^2 \\
&\leq \left\|\mathbf{a}^l \circ \left(\hat{\mathbf{o}}_j^{l-1\top} \mathbf{W}^l\right)\right\|_2^2 + \left\|\mathbf{a}^l \circ \left(\left(\left(\mathbf{1} - \mathbf{a}^{l-1}\right) \circ \mathbf{o}_j^{l-1}\right)^\top \mathbf{W}^l\right)\right\|_2^2
\end{aligned} \tag{5.4}$$

where $\|\cdot\|_2$ is ℓ_2-norm, and $\mathbf{1}$ is a vector whose all elements are equal to 1. As shown in the last inequality, the upper bound of $E_j^l(\tilde{\mathbf{o}}_j^l)$ is determined by two results of the previous layer $l-1$, i.e., 1) the error vector, $\hat{\mathbf{o}}_j^{l-1}$ in the first term, and 2) the not activated elements of the output neuron, $\left(\mathbf{1}-\mathbf{a}^{l-1}\right)\circ\mathbf{o}_j^{l-1}$ in the second term. Hence, the bound of $E_j^l(\tilde{\mathbf{o}}_j^l)$ of layer l can be obtained by recursively computing them for the previous layers, i.e., from the first to the $(l-1)$-th layer.

Minimizing Error. Since sub-output neuron of the last layer L, $\tilde{\mathbf{o}}_j^L$ determines the performance of a sub-network, we minimize E^L. From Equation 5.1, 5.2 and 5.3, minimizing E^L given time budget, B_i is reduced to finding the activation vector, $\mathbf{a}_i^L$.

$$\underset{\mathbf{a}_i^L\in\{0,1\}^{m^L}}{\operatorname{argmin}} E^L \text{ s.t. } C(N_i)\leq B_i \text{ and } \left\|\mathbf{a}_i^L\right\|_1=\left\lfloor N_i\cdot m^L\right\rfloor \tag{5.5}$$

Since E^L is obtained from the errors of previous layers as shown in Equation 5.4, it is also minimized by finding $\{\mathbf{a}_i^l|1\leq l\leq L\}$ that minimizes the error of each layer l, i.e., $E_j^l(\tilde{\mathbf{o}}_j^l)$, which is achieved by minimizing their summation, i.e., $\sum_{l=1}^{L}E_j^l(\tilde{\mathbf{o}}_j^l)$. Hence, from Equation 5.1 and 5.3, the problem of constructing a sub-network given B_i is reformulated as:

$$\begin{aligned}&\underset{\mathbf{a}_i^l\in\{0,1\}^{m^l}}{\operatorname{argmin}}\frac{1}{n}\sum_{j=1}^{n}\sum_{l=1}^{L}E_j^l(\tilde{\mathbf{o}}_j^l)\\&=\underset{\mathbf{a}_i^l\in\{0,1\}^{m^l}}{\operatorname{argmin}}\frac{1}{n}\sum_{j=1}^{n}\sum_{l=1}^{L}\left\|\mathbf{a}_i^l\circ\sigma\left(\tilde{\mathbf{y}}_j^l\right)-\mathbf{a}_i^l\circ\sigma\left(\mathbf{y}_j^l\right)\right\|_2^2\\&\qquad\text{s.t. } C(N_i)\leq B_i \text{ and } \left\|\mathbf{a}_i^l\right\|_1=\left\lfloor N_i\cdot m^l\right\rfloor\end{aligned} \tag{5.6}$$

Hence, a sub-network for the job J_i which is completed within B_i with minimum error is dynamically constructed by finding N_i and $\{\mathbf{a}_i^l|1\leq l\leq L\}$ in Equation 5.6.

5.3.3 Neuron Ranking for Sub-Network Construction

Importance-based Ranking. While the network utilization, N_i is easily obtained by finding the maximum N_i satisfying $C(N_i)\leq B_i$ in Equation 5.2, the activation vector, $\mathbf{a}_i^l$ that selects the elements of sub-output neuron, $\tilde{\mathbf{o}}_j^l$ from $\mathbf{o}_j^l$ is required to minimize the error $E_j^l(\tilde{\mathbf{o}}_j^l)$ in Equation 5.3. We determine $\mathbf{a}_i^l$ based on the importance of each neuron of $\mathbf{o}_j^l$, which represents the increased error when it is removed. Then, $E_j^l(\tilde{\mathbf{o}}_j^l)$ is minimized by composing the binary elements of $\mathbf{a}_i^l$ such that $\tilde{\mathbf{o}}_j^l$ consists of $\lfloor N_i \cdot m^l \rfloor$ number of neurons in $\mathbf{o}_j^l$ having largest importance and zero for all the other elements.

To measure the importance, we compute the second-order derivatives of the error $E_j^l(\mathbf{o}_j^l)$ w.r.t. the output neuron, $\mathbf{o}_j^l$ for each layer using Optimal Brain Surgeon algorithm (Hassibi and Stork, 1993). We use it since the heuristic methods such as magnitude-based method (Li et al., 2016; Hu et al., 2016; Han and Zhang, 2015) may eliminate wrong neurons (LeCun et al., 1990; Hassibi and Stork, 1993), resulting in large error and poor performance (Sharma et al., 2017).

Error Approximation. For a DNN trained to a local minimum, the error, $E_j^l(\tilde{\mathbf{o}}_j^l)$ can be approximated with Taylor series as in (Hassibi and Stork, 1993) and (Arfken and Weber, 1999) w.r.t. the output neuron, $\mathbf{o}_j^l$ as follows:

$$\begin{aligned}\delta E_j^l &= E_j^l(\tilde{\mathbf{o}}_j^l) - E_j^l(\mathbf{o}_j^l) \\ &= \left(\frac{\partial E_j^l}{\partial \mathbf{o}_j^l}\right)^\top \delta\mathbf{o}_j^l + \frac{1}{2}\delta\mathbf{o}_j^{l\top}\mathbf{H}^l\delta\mathbf{o}_j^l + O(\left\|\delta\mathbf{o}_j^l\right\|^3) \\ &\approx \frac{1}{2}\delta\mathbf{o}_j^{l\top}\mathbf{H}^l\delta\mathbf{o}_j^l\end{aligned} \tag{5.7}$$

where δ is a perturbation of corresponding variable, $\mathbf{H}^l \equiv \partial^2 E_j^l/\partial(\mathbf{o}_j^l)^2$ is the Hessian matrix (Upton and Cook, 2014), and $O(\|\delta\mathbf{o}_j^l\|^3)$ is the third and all higher-order terms. With the error function defined in Equation 5.3, the first and third terms are eliminated (Dong et al., 2017a). To minimize the increase in error, δE_j^l, we set the q-th element of $\mathbf{o}_j^l$, denoted as o_{jq}^l, to zero, expressed as:

$$\delta o_{jq}^l + o_{jq}^l = 0 \text{ or more generally } \mathbf{e}_q^{l\top}\delta\mathbf{o}_j^l + o_{jq}^l = 0 \tag{5.8}$$

where $\mathbf{e}_q^l$ is the unit vector whose q-th element is 1 and others are all 0. With o_{jq}^l being removed from $\mathbf{o}_j^l$ as shown in Equation 5.8, we minimize Equation 5.7 as follows:

$$\min_q \frac{1}{2}\delta\mathbf{o}_j^{l\top}\mathbf{H}^l\delta\mathbf{o}_j^l \text{ s.t. } \mathbf{e}_q^{l\top}\delta\mathbf{o}_j^l + o_{jq}^l = 0 \tag{5.9}$$

Computing Importance. To compute the importance of the q-th element of $\mathbf{o}_j^l$ at layer l, o_{jq}^l, we solve Equation 5.9 with a Lagrangian function, $\mathcal{L}^l$, which is given by:

$$\mathcal{L}^l = \frac{1}{2}\delta\mathbf{o}_j^{l\top}\mathbf{H}^l\delta\mathbf{o}_j^l + \lambda(\mathbf{e}^{l\top}\delta\mathbf{o}_j^l + o_{jq}^l) \tag{5.10}$$

where λ is a Lagrange multiplier. By taking derivatives, employing Equation 5.8, and using matrix inversion, the optimal increase in error and output change of o_{jq}^l are obtained by:

$$s_q^l = \frac{1}{2}\frac{(o_{jq}^l)^2}{[\mathbf{H}^{-1}]_{qq}^l} \text{ and } \delta\mathbf{o}_j^l = -\frac{o_{jq}^l}{[\mathbf{H}^{-1}]_{qq}^l}[\mathbf{H}^{-1}]^l\mathbf{e}_q^l \tag{5.11}$$

We call s_q^l as the *importance* of the q-th neuron element of layer l, o_{jq}^l– the amount of increase in error when it is removed from a DNN. Based on s_q^l, the activation vector, $\mathbf{a}_i^l$ is determined to activate $\lfloor N_i \cdot m^l \rfloor$ number of neuron elements in $\mathbf{o}_j^l$ having the largest importance by setting the corresponding elements of $\mathbf{a}_i^l$ to 1 and others to 0, which provides the sub-output neuron, $\tilde{\mathbf{o}}_j^l$ for construction of a sub-network with minimum error. s_q^l is computed only once at compile-time.

5.4 Time-Bound Execution of Sub-Network

The last step of SubFlow is to execute a newly-constructed sub-network within the execution time budget. This section describes the two run-time execution operations of SubFlow, i.e., time-bound feed-forward and back-propagation, which enables the time-bound completion of a sub-network.

5.4.1 Time-bound Feed-Forward

Time-Bound Feed-Forward. The inference of a DNN is achieved by executing the DNN layer by layer, which is called the feed-forward. Given a sub-network, an inference job, J_i completes the feed-forward within the time budget, B_i by performing computation in Equation 5.1 only for the non-zero elements of a sub-output neuron, $\tilde{\mathbf{o}}_j^l$. The amount of computation, as well as the execution time, are expected to be proportional to the number of non-zero elements of $\tilde{\mathbf{o}}_j^l$, which is determined by the activation vector, $\mathbf{a}_i^l$. Computation related to zero neuron elements is skipped since multiplication by zero results in a zero, which should require no work. We name it as *time-bound feed-forward* since the feed-forwarding time is bounded by the size and configuration of a sub-network depending on a set of $\mathbf{a}_i^l$.

Existing Feed-Forward. Unfortunately, the current feed-forward algorithms, such as the lowering method (Sze et al., 2017) do not support the sparse-neuron-aware feed-forwarding. They always perform the same amount of computation based on the fixed sequence of calculation regardless of the number of non-zero neuron elements. To enable time-bound feed-forward, we propose *sub-convolution* and *sub-multiplication* for a convolutional and fully-connected layer, respectively in a similar way to the direct sparse convolution (Park et al., 2016a).

Sub-Convolution. For a convolutional layer l, layer output, $\mathbf{O}^l \in \mathbb{R}^{n \times c' \times h' \times w'}$ is computed by taking input, $\mathbf{O}^{l-1} \in \mathbb{R}^{n \times c \times h \times w}$ from the previous layer $l-1$, where n is the size of the input batch; c', h', and w' denote the channel, height, and width of the output. For input $\mathbf{O}^{l-1}$, c, h, and w denote its channel, height, and width. For convolution, convolutional filter (weight parameter) denoted as $\mathbf{W}^l \in \mathbb{R}^{c' \times c \times y \times x}$ is applied to input, $\mathbf{O}^{l-1}$, where c', c, y, and x denote the size of the output channel, input channel, height, and width of filter.

Given sub-input neuron, $\tilde{\mathbf{O}}^{l-1}$ and sub-output neuron, $\tilde{\mathbf{O}}^{l}$ composed by $\mathbf{a}_i^{l-1}$ and $\mathbf{a}_i^{l}$, respectively, the computational complexity of convolution operation at layer l with filter $\mathbf{W}^l$, denoted as $f_c^l(\cdot)$, is given by:

$$f_c^l(\mathbf{a}_i^{l-1}, \mathbf{a}_i^{l}, \mathbf{W}^l) = \mathcal{O}(\left|\mathbf{a}_i^l\right| \left|\mathbf{W}^l\right| - \left\|\mathbf{1} - \mathbf{a}_i^{l-1}\right\|_1 - \left\|\mathbf{1} - \mathbf{a}_i^{l}\right\|_1 \left|\mathbf{W}^l\right|) \tag{5.12}$$

where $\|\cdot\|_1$ denotes ℓ_1-norm, $|\cdot|$ denotes the number of elements, and $\mathbf{1}$ is a vector whose all elements are 1. The first term indicates the total amount of computation at the l-th layer of the full-size network, while the second and third term indicates the amount of computation reduced by the sub-input and sub-output neuron, respectively. The activation vector of the previous layer, $\mathbf{a}_i^{l-1}$ also determines the complexity since an output of one layer is the input of the next layer.

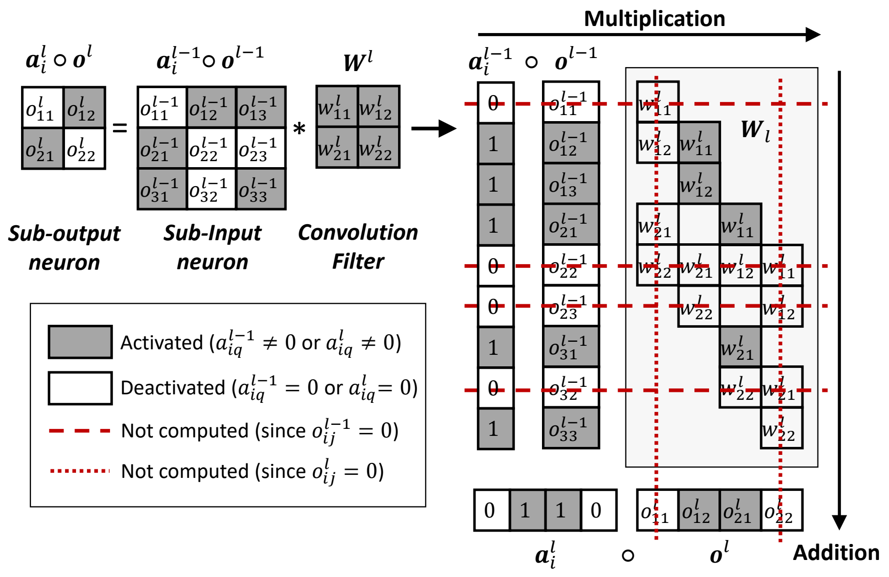
We define *sub-convolution* as the convolution of a sub-network whose computational complexity is determined by $\mathbf{a}_i^{l-1}$ and $\mathbf{a}_i^{l}$ as shown in Equation 5.12. Equation 5.13 is an example of sub-convolution with $\tilde{\mathbf{O}}^{l-1} \in \mathbb{R}^{1\times1\times3\times3}$, $\tilde{\mathbf{O}}^{l} \in \mathbb{R}^{1\times1\times2\times2}$, and $\mathbf{W}^l \in \mathbb{R}^{1\times1\times2\times2}$, where only two and five elements are activated as sub-output and sub-input neuron.

$$\begin{bmatrix} \cancel{o_{11}^{l}} & o_{12}^{l} \\ o_{21}^{l} & \cancel{o_{22}^{l}} \end{bmatrix} = \begin{bmatrix} \bcancel{o_{11}^{l-1}} & o_{12}^{l-1} & o_{13}^{l-1} \\ o_{21}^{l-1} & \bcancel{o_{22}^{l-1}} & \bcancel{o_{23}^{l-1}} \\ o_{31}^{l-1} & \bcancel{o_{32}^{l-1}} & o_{33}^{l-1} \end{bmatrix} * \begin{bmatrix} w_{11}^{l} & w_{12}^{l} \\ w_{21}^{l} & w_{22}^{l} \end{bmatrix} \tag{5.13}$$

Here, $*$ denotes the convolution, $o_{ij}^{l-1} \neq 0$ and $o_{ij}^{l} \neq 0$ are the non-zero neuron elements, whereas $\bcancel{o_{ij}^{l-1}} = 0$ and $\cancel{o_{ij}^{l}} = 0$ are the zero neuron elements. With vectorization, Equation 5.13 can be rewritten as matrix multiplication, which is given by:

$$\begin{bmatrix} \cancel{o_{11}^{l}} & o_{12}^{l} & o_{21}^{l} & \cancel{o_{22}^{l}} \end{bmatrix} = \begin{bmatrix} w_{11}^{l} & w_{12}^{l} & w_{21}^{l} & w_{22}^{l} \end{bmatrix} \begin{bmatrix} \xcancel{o_{11}^{l-1}} & o_{12}^{l-1} & o_{21}^{l-1} & \xcancel{o_{22}^{l-1}} \\ \cancel{o_{12}^{l-1}} & o_{13}^{l-1} & \bcancel{o_{22}^{l-1}} & \xcancel{o_{23}^{l-1}} \\ \cancel{o_{21}^{l-1}} & \bcancel{o_{22}^{l-1}} & o_{31}^{l-1} & \xcancel{o_{32}^{l-1}} \\ \xcancel{o_{22}^{l-1}} & \bcancel{o_{23}^{l-1}} & \bcancel{o_{32}^{l-1}} & \cancel{o_{33}^{l-1}} \end{bmatrix} \tag{5.14}$$

Figure 5.4 illustrates the sub-convolution of Equation 5.14, where only four out of sixteen multiplications are performed. It efficiently performs only the necessary computation by check-



Multiplication
$a_i^l \circ o^l$
$a_i^{l-1} \circ o^{l-1}$
W^l
Sub-output neuron
Sub-Input neuron
Convolution Filter
Activated ($a_{iq}^{l-1} \neq 0$ or $a_{iq}^l \neq 0$)
Deactivated ($a_{iq}^{l-1} = 0$ or $a_{iq}^l = 0$)
Not computed (since $o_{ij}^{l-1} = 0$)
Not computed (since $o_{ij}^l = 0$)
W_l
Addition

Figure 5.4: An example of sub-convolution: Given a sub-input, sub-output neuron, and convolution filter, the sub-convolution is performed by walking through the sub-input (vertical direction) and sub-output (horizontal direction) only once to see if the elements are zero or not. By skipping computation related to zero-elements, the total computation time becomes proportional to the number of non-zeros. The final output requires only four out of sixteen multiplications, i.e., $\tilde{o}_{12}^{l-1} \cdot w_{11}^l$, $\tilde{o}_{13}^{l-1} \cdot w_{12}^l$, $\tilde{o}_{21}^{l-1} \cdot w_{11}^l$, and $\tilde{o}_{31}^{l-1} \cdot w_{21}^l$.

ing the sub-input and sub-output neurons only once to see whether they are zero or not with linear complexity, i.e., $\mathcal{O}(|\tilde{\mathbf{O}}^{l-1}| + |\tilde{\mathbf{O}}^l|)$, while a naive algorithm takes $\mathcal{O}(|\tilde{\mathbf{O}}^l||\mathbf{W}^l|)$. For example, a sub-convolution between 100×100 input and 10×10 filter, which results in 91×91 output, requires only $18,281$ zero-element checks. On the contrary, a naive algorithm requires $828,100$ zero-check (i.e., 45$\times$ less efficient).

Sub-Multiplication. For a fully-connected layer l, layer output $\mathbf{O}^l \in \mathbb{R}^{n \times m^l}$ is computed by taking the input, $\mathbf{O}^{l-1} \in \mathbb{R}^{n \times m^{l-1}}$ from the previous layer $l-1$, where n is the size of the input batch, m^{l-1} and m^l are the input and the output size, respectively. The output, $\mathbf{O}^l$ is obtained by multiplying a weight parameter $\mathbf{W}^l \in \mathbb{R}^{m^{l-1} \times m^l}$ to the input, $\mathbf{O}^{l-1}$.

Given sub-input $\tilde{\mathbf{O}}^{l-1}$ and sub-output $\tilde{\mathbf{O}}^{l}$ composed by $\mathbf{a}_i^{l-1}$ and $\mathbf{a}_i^{l}$, respectively, the computational complexity of matrix multiplication with weight parameter $\mathbf{W}^l$, $f_m^l(\cdot)$ is given by:

$$f_m^l(\mathbf{a}_i^{l-1}, \mathbf{a}_i^{l}, \mathbf{W}^l) = \mathcal{O}\left(\left\|\mathbf{a}_i^{l-1}\right\|_1 \left|\mathbf{a}_i^{l}\right| + \left\|\mathbf{a}_i^{l}\right\|_1 \left|\mathbf{W}^l\right| - \left\|\mathbf{a}_i^{l}\right\|_1 \left\|\mathbf{a}_i^{l-1}\right\|_1\right) \tag{5.15}$$

where $\|\cdot\|_1$ is ℓ_1-norm, and $|\cdot|$ is the number of elements. Equation 5.15 is proportional to the number of non-zero elements in the weight matrix used for multiplication. The first and second term indicate the number of row-wise and column-wise elements in the matrix, respectively. The last term cancels out the overlapped elements between the first and second term.

We define *sub-multiplication* as the matrix multiplication of a sub-network whose computational complexity is determined by $\mathbf{a}_i^{l-1}$ and $\mathbf{a}_i^{l}$ as shown in Equation 5.15. Equation 5.16 is an example of sub-multiplication with 1×3 sub-input, 1×3 sub-output neuron, and 3×3 weight.

$$\begin{bmatrix} \cancel{o_{11}^l} \; o_{12}^l \; o_{13}^l \end{bmatrix} = \begin{bmatrix} o_{11}^{l-1} \; \bcancel{o_{12}^{l-1}} \; o_{13}^{l-1} \end{bmatrix} \begin{bmatrix} w_{11}^l \; w_{12}^l \; w_{13}^l \\ w_{21}^l \; w_{22}^l \; w_{23}^l \\ w_{31}^l \; w_{32}^l \; w_{33}^l \end{bmatrix} \tag{5.16}$$

With o_{12}^{l-1} and o_{11}^{l} being zero in sub-input and sub-output neuron, respectively, (1×3) by (3×3) matrix multiplication reduces to (1×2) by (2×2), as follows:

$$\begin{bmatrix} o_{12}^l \; o_{13}^l \end{bmatrix} = \begin{bmatrix} o_{11}^{l-1} \; o_{13}^{l-1} \end{bmatrix} \begin{bmatrix} w_{12} \; w_{14} \\ w_{32} \; w_{34} \end{bmatrix} \tag{5.17}$$

5.4.2 Time-Bound Back-Propagation

Time-Bound Back-Propagation. The training of a DNN is achieved by the compute-intensive process called back-propagation (Werbos et al., 1990). The goal of back-propagation is to update weight parameter, $\mathbf{W}^l$ of each layer l by computing the gradient (Bachman, 2007) of a loss function, denoted as L, w.r.t. $\mathbf{W}^l$. The back-propagation is repeated with multiple iterations until the loss function, L converges to a particular criterion.

Given sub-output, $\tilde{\mathbf{o}}_j^l$ composed by $\mathbf{a}_i^l$, the gradient of the loss, L w.r.t. $\mathbf{W}^l$ for the j-th training instance, ∇L is:

$$\nabla L = \frac{\partial L}{\partial \mathbf{W}^l} = \frac{\partial L}{\partial \tilde{\mathbf{o}}_j^l} \cdot \mathbf{J}^l = \frac{\partial L}{\partial \left(\mathbf{a}_i^l \circ \sigma\left(\tilde{\mathbf{y}}_j^l\right)\right)} \cdot \mathbf{J}^l \tag{5.18}$$

where $\mathbf{J}^l \equiv \partial\big(\mathbf{a}_i^l \circ \sigma\left(\tilde{\mathbf{y}}_j^l\right)\big)/\partial \mathbf{W}^l$ is the Jacobian matrix (Kaplan, 1984), and $\tilde{\mathbf{y}}_j^l$ is defined as the same in Equation 5.1. By computing ∇L only for the non-zero elements of a sub-output neuron, $\tilde{\mathbf{o}}_j^l$, which is determined by the activation vector, $\mathbf{a}_i^l$, a back-propagation job, J_i is completed within the execution time budget, B_i. We name it as *time-bound back-propagation* since the gradient computation time is bounded by the size and configuration of a sub-network depending on a set of $\mathbf{a}_i^l$.

Since the sparse-neuron-aware gradient is also not supported by the existing back-propagation (Werbos et al., 1990), we propose *sub-convolution-gradient* and *sub-multiplication-gradient* for convolutional and fully-connected layers, respectively.

Sub-Convolution-Gradient. Given sub-input neuron, $\tilde{\mathbf{O}}^{l-1}$ and sub-output neuron, $\tilde{\mathbf{O}}^l$ of a convolutional layer composed by $\mathbf{a}_i^{l-1}$ and $\mathbf{a}_i^l$, respectively, the complexity of computing convolution gradient with filter $\mathbf{W}^l$, $g_c^l(\cdot)$ is given by:

$$g_c^l(\mathbf{a}_i^{l-1}, \mathbf{a}_i^l, \mathbf{W}_i^l) = \mathcal{O}\left(\left\|\mathbf{a}_i^l\right\|_1 \left|\mathbf{W}^l\right| - \max\left(\left\|\mathbf{1} - \mathbf{a}_i^{l-1}\right\|_1 - \left\|\mathbf{a}_i^l\right\|_1 \left|\mathbf{W}^l\right|, 0\right)\right) \tag{5.19}$$

where $\|\cdot\|_1$ is ℓ_1-norm, and $|\cdot|$ is the number of elements. The first term depends on the number of non-zero elements in sub-output, and the second term depends on the number of non-zero elements in sub-input.

We define *sub-convolution-gradient* as the gradient of the convolution of a sub-network whose computational complexity is determined by $\mathbf{a}_i^{l-1}$ and $\mathbf{a}_i^l$ as shown in Equation 5.19. Equation 5.20 is an example of a sub-convolution-gradient for Equation 5.13, which shows only four out of sixteen differentiations are performed for the computation of ∇L.

$$\nabla L = \begin{bmatrix} \frac{\partial L}{\partial w^l_{11}} & \frac{\partial L}{\partial w^l_{12}} & \frac{\partial L}{\partial w^l_{21}} & \frac{\partial L}{\partial w^l_{22}} \end{bmatrix} = \begin{bmatrix} \cancel{\frac{\partial L}{\partial o^l_{11}}} & \frac{\partial L}{\partial o^l_{12}} & \frac{\partial L}{\partial o^l_{21}} & \cancel{\frac{\partial L}{\partial o^l_{22}}} \end{bmatrix} \begin{bmatrix} \xcancel{\frac{\partial o^l_{11}}{\partial w^l_{11}}} & \cancel{\frac{\partial o^l_{11}}{\partial w^l_{12}}} & \cancel{\frac{\partial o^l_{11}}{\partial w^l_{21}}} & \xcancel{\frac{\partial o^l_{11}}{\partial w^l_{22}}} \\ \frac{\partial o^l_{12}}{\partial w^l_{11}} & \frac{\partial o^l_{12}}{\partial w^l_{12}} & \bcancel{\frac{\partial o^l_{12}}{\partial w^l_{21}}} & \bcancel{\frac{\partial o^l_{12}}{\partial w^l_{22}}} \\ \frac{\partial o^l_{21}}{\partial w^l_{11}} & \bcancel{\frac{\partial o^l_{21}}{\partial w^l_{12}}} & \frac{\partial o^l_{21}}{\partial w^l_{21}} & \bcancel{\frac{\partial o^l_{21}}{\partial w^l_{22}}} \\ \cancel{\frac{\partial o^l_{22}}{\partial w^l_{11}}} & \cancel{\frac{\partial o^l_{22}}{\partial w^l_{12}}} & \cancel{\frac{\partial o^l_{22}}{\partial w^l_{21}}} & \cancel{\frac{\partial o^l_{22}}{\partial w^l_{22}}} \end{bmatrix} \tag{5.20}$$

Here, the matrix on the right-hand side is the Jacobian, $\bcancel{\frac{\partial o}{\partial w}} = 0$ and $\cancel{\frac{\partial o}{\partial w}} = 0$ denote that the computation related to $\frac{\partial o}{\partial w}$ is skipped since the corresponding sub-input and sub-output neuron element is zero, respectively. The zero elements of sub-output neuron eliminate the corresponding rows of the Jacobian, e.g., the first and last rows of the Jacobian become unnecessary since o^l_{11} and o^l_{22} are zeros as shown in Equation 5.13. The zero elements of sub-input neuron result in scattered elimination of individual derivatives in the Jacobian, e.g., $\frac{\partial o^l_{12}}{\partial w^l_{21}}$ is not computed since the corresponding sub-input neuron element, o^{l-1}_{22} is zero, i.e., $\frac{\partial o^l_{12}}{\partial w^l_{21}} = o^{l-1}_{22} = 0$.

Sub-Multiplication-Gradient. Given sub-input neuron, $\tilde{\mathbf{O}}^{l-1}$ and sub-output neuron, $\tilde{\mathbf{O}}^l$ of a fully-connected layer composed by $\mathbf{a}^{l-1}_i$ and $\mathbf{a}^l_i$, respectively, the complexity of computing multiplication gradient with weight $\mathbf{W}^l$, $g^l_m(\cdot)$ is:

$$g^l_m(\mathbf{a}^{l-1}_i, \mathbf{a}^l_i, \mathbf{W}^l) = \mathcal{O}\left(\left\|\mathbf{a}^{l-1}_i\right\|_1 \left|\mathbf{a}^l_i\right| + \left\|\mathbf{a}^l_i\right\|_1 \left|\mathbf{W}^l\right| - \left\|\mathbf{a}^l_i\right\|_1 \left\|\mathbf{a}^{l-1}_i\right\|_1\right) \tag{5.21}$$

where $\|\cdot\|_1$ is ℓ_1-norm, and $|\cdot|$ is the number of elements.

We define *sub-multiplication-gradient* as the gradient of matrix multiplication of a sub-network whose computation complexity is determined by $\mathbf{a}^{l-1}_i$ and $\mathbf{a}^l_i$ as shown in Equation 5.21, which is equivalent to the sub-multiplication (Equation 5.15).

5.5 Implementation

We implement SubFlow as an extended version of the TensorFlow library (Abadi et al., 2016), which is fully compatible with the existing TensorFlow operations. The programmers can easily apply SubFlow to their DNNs in the same way they design DNNs without SubFlow. SubFlow is implemented to run on both CPU and GPU, which makes it adaptable to a wide range of platforms. For CPU, it is implemented based on the Eigen library (Guennebaud et al., 2010) that is optimized to perform matrix operations in CPU. For GPU, it is implemented with CUDA library (NVIDIA, 2019b; Cook, 2012) to support the parallel computation of sub-networks like the other GPGPU operations.

In constructing and executing sub-networks, SubFlow does not generate and save multiple versions of sub-networks a priori. Based on a single DNN designed by the programmer, SubFlow is implemented to construct and execute sub-networks at run-time based on time-bound sparse execution.

Figure 5.5 shows the SubFlow framework, along with the TensorFlow, which consists of a sub-network library, Python client operations, and kernel implementations.

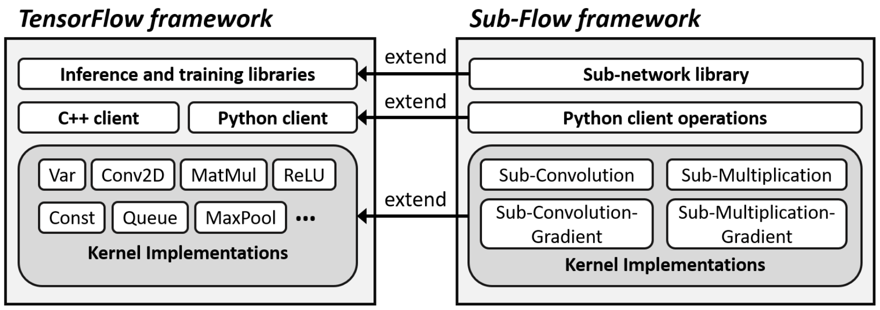


Figure 5.5: SubFlow Framework: The SubFlow framework consists of the sub-network library, python client operations, and kernel implementations. It is fully compatible with TensorFlow and provides all the necessary components of the TensoFlow hierarchy.

Sub-Network Library. It is a high-level module that computes the importance of output neurons in the DNN for the construction of sub-networks. Since the computation of the Hessian matrix in Equation 5.11 is intractable with DNNs of considerable size, an approximation of the Hessian using sample covariance is computed (Hassibi and Stork, 1993) instead. It is also responsible for the online construction of sub-networks. Based on the importance of neurons and time budget of the i-th job, J_i, it produces $\mathbf{a}_i^l$ in Equation 5.1 for all layers at run-time, except the last layer where all the final output neurons of the DNN should be selected.

Python Client Operations. Python client operations provide the programmer with a set of wrapper APIs that help design a DNN model using SubFlow, which is fully compatible with other existing operations of TensorFlow. Each API represents and corresponds to its kernel implementation that is executed with higher efficiency when a sub-network runs.

Kernel Implementations. They are the lower-level implementation of four operations used in SubFlow, i.e., sub-convolution, sub-multiplication, sub-convolution-gradient, and sub-multiplication-gradient. These operations are responsible for executing a sub-network of the DNN designed with Python client operations. Written in C and C++, they are optimized to hardware platforms and able to perform the efficient time-bound execution of sub-network with minimum overhead.

5.6 Experiment

5.6.1 Experimental Setup

Hardware and Software. We conduct experiments on a system consisting of Intel Core i9-9900K CPU, NVIDIA RTX 2080 Ti GPU with 11 GB of memory, and 32 GB of system memory (RAM). We use TensorFlow 1.13.1 with Eigen 3.3.90 and CUDA 10.0 (CUDNN 7.4.2) for implementation.

Datasets and DNN Models. We use three standard machine learning datasets in our evaluation, i.e., MNIST (LeCun et al., 1998) (hand-written digits), CIFAR-10 (Krizhevsky et al., 2009) (image classification), and GSC (Google Speech Commands V2) (Warden, 2018). For each dataset, the state-of-the-art DNN model that provides the best performance for the dataset is designed with SubFlow, i.e., LeNet-5 (LeCun et al., 1998), AlexNet (Krizhevsky et al., 2012), and KWS

(Key-Word Spotting) architecture (Sainath and Parada, 2015). Table 5.1 summarizes the DNN architectures and datasets.

	LeNet-5 (MNIST)	**AlexNet (CIFAR-10)**	**KWS (GSC)**
Layer 1	Input: 28×28×1	Input: 32×32×3	Input: 61×13×1
Layer 2	Conv1: 5×5×1×6	Conv1: 3×3×3×64	Conv1: 12×6×1×64
Layer 3	Conv2: 5×5×6×16	Conv2: 3×3×64×192	Conv2: 6×3×64×64
Layer 4	FC1: 400	Conv3: 3×3×192×384	FC1: 1024
Layer 5	FC2: 84	FC1: 4096	FC2: 512
Layer 6	FC3 (Output): 10	FC2: 2048	FC3 (Output): 35
Layer 7		FC3 (Output): 10	

* KWS: Key-Word Spotting, Conv: Convolution layer, FC: Fully-connected layer

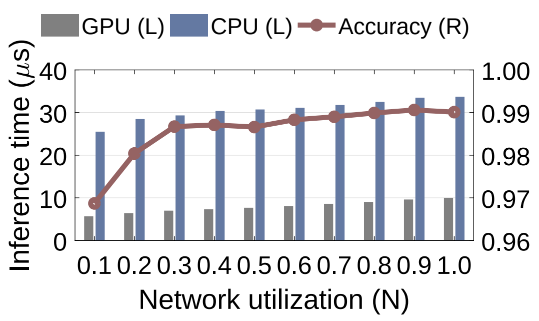
Table 5.1: The DNN models and datasets used in the evaluation.

Time Measurement. The execution time of a sub-network, including individual operations, is measured by using TensorFlow's Timeline tool (Google, 2018b) that traces and records the execution time of all the operations of a DNN in the unit of microseconds. We analyze and compare the execution time of different sub-networks by analyzing their tracing log files saved in the JSON (JavaScript Object Notation) format (Ecma International, 2017).

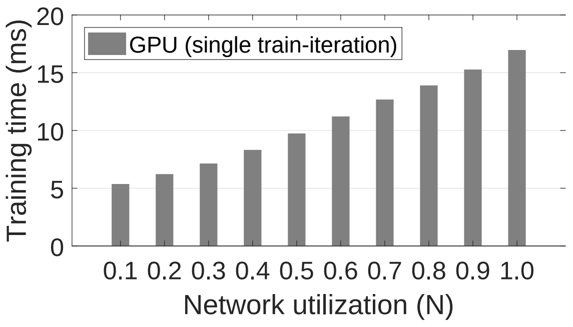
5.6.2 End-to-End Execution Time and Performance

We evaluate the end-to-end execution time and performance (i.e., inference accuracy) of sub-networks of different sizes determined by the network utilization, N. The inference time is measured on both CPU and GPU by calculating the average feed-forward time on the entire test samples as one input batch. The training time is evaluated on GPU by measuring the execution time of one training iteration for both feed-forward and back-propagation with a mini-batch size of 96 samples. We use separate datasets for training and testing.

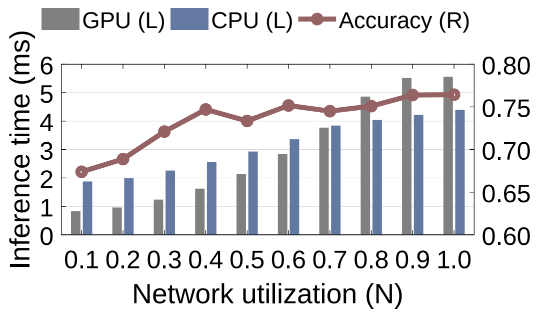
Figure 5.6 shows the end-to-end inference and training time of the three DNNs for different network utilizations, i.e., from $N = 0.1$ to 1.0. All three DNNs show that their inference time decreases as N decreases without significant loss of inference accuracy. For example, the sub-network of AlexNet with $N = 0.1$ achieves 6.7x speedup with only 9% drop of inference accuracy, i.e., from 76% to 67%. The inference accuracy of LeNet-5 stays almost the same, i.e., a



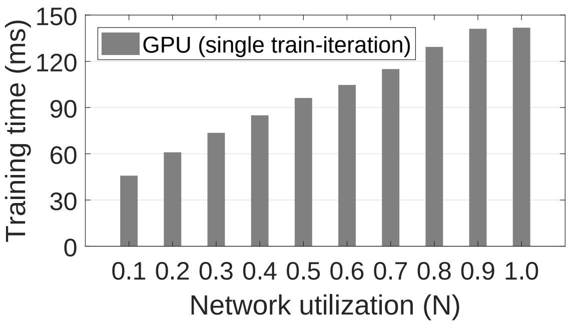
(a) LeNet-5 (MNIST): Inference



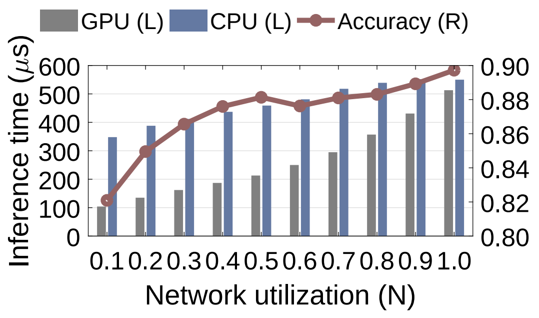
(b) LeNet-5 (MNIST): Training



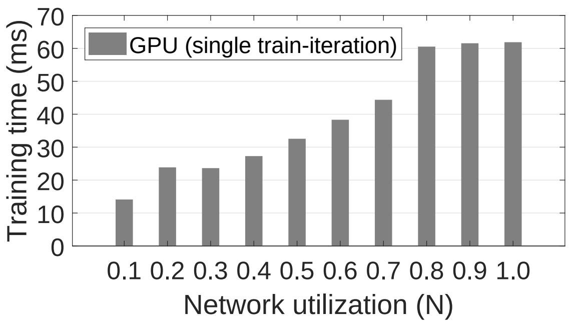
(c) AlexNet (CIFAR-10): Inference



(d) AlexNet (CIFAR-10): Training



(e) KWS (GSC): Inference



(f) KWS (GSC): Training

Figure 5.6: The end-to-end execution time and inference accuracy over the network utilization (N): The inference time is measured on both GPU and CPU, and training time is measured on GPU.

maximum 2% drop while providing 2x speedup when $N = 0.1$. The training time also decreases as N decreases, e.g., the training time of KWS is reduced by 4.4x with a sub-network of $N = 0.1$. However, their speedup is not linear to N since 1) all the neurons in the first and the last layers are activated for all sub-networks in our implementation, and 2) run-time overhead occurs.

5.6.3 Usefulness (Utility) of DNN

We next evaluate how SubFlow improves the usefulness of a DNN given dynamic deadlines against the same original DNNs that run without SubFlow. We measure the usefulness of inference and training tasks based on the inference accuracy and the training ratio that indicates the ratio of the DNN components trained within the deadline, respectively. Figure 5.7 shows the usefulness of the three DNNs, i.e., the inference accuracy (GPU and CPU) and training ratio (GPU) over a range of dynamic deadlines. Unlike the non-SubFlow DNNs, SubFlow makes the best use of the DNNs for a given deadline by flexibly utilizing them, and completing the inference or training task in time. For example, SubFlow AlexNet achieves 74% average inference accuracy, which is 2% lower than the original DNN (76%), for the deadlines ranging between 1800μ and 5700μs, while the non-SubFlow DNN achieves 0% accuracy for the same set of deadlines as shown in Figure 5.7d. As the deadline gets closer to the execution time of the original DNN, the accuracy of SubFlow AlexNet approaches 76% since the full network is executed. On the other hand, the non-SubFlow DNN results in zero usefulness unless the deadline is equivalent to or larger than the execution time. For deadlines smaller than that, its usefulness is zero since they are not even executed, or the execution completed after the deadline.

5.6.4 Run-time Overhead

We measure two types of run-time overheads of SubFlow: 1) the sub-network construction overhead, and 2) the sub-network execution overhead for one single input sample which is the additional computation time required to run a DNN with SubFlow. These two overheads are obtained 1) by measuring the time needed to generate the activation vector, $\mathbf{a}_i^l$ in Equation 5.6 for all layers, which is required to construct a sub-network, and 2) by measuring the execution overhead time and the actual execution time separately during the feed-forward and back-propagation.

Figure 5.8a shows the sub-network construction overhead of the three DNNs for different network utilizations, which stays the same (LeNet-5 and KWS) or increases slightly (AlexNet) with increased network utilization. They do not tend to change significantly with different network

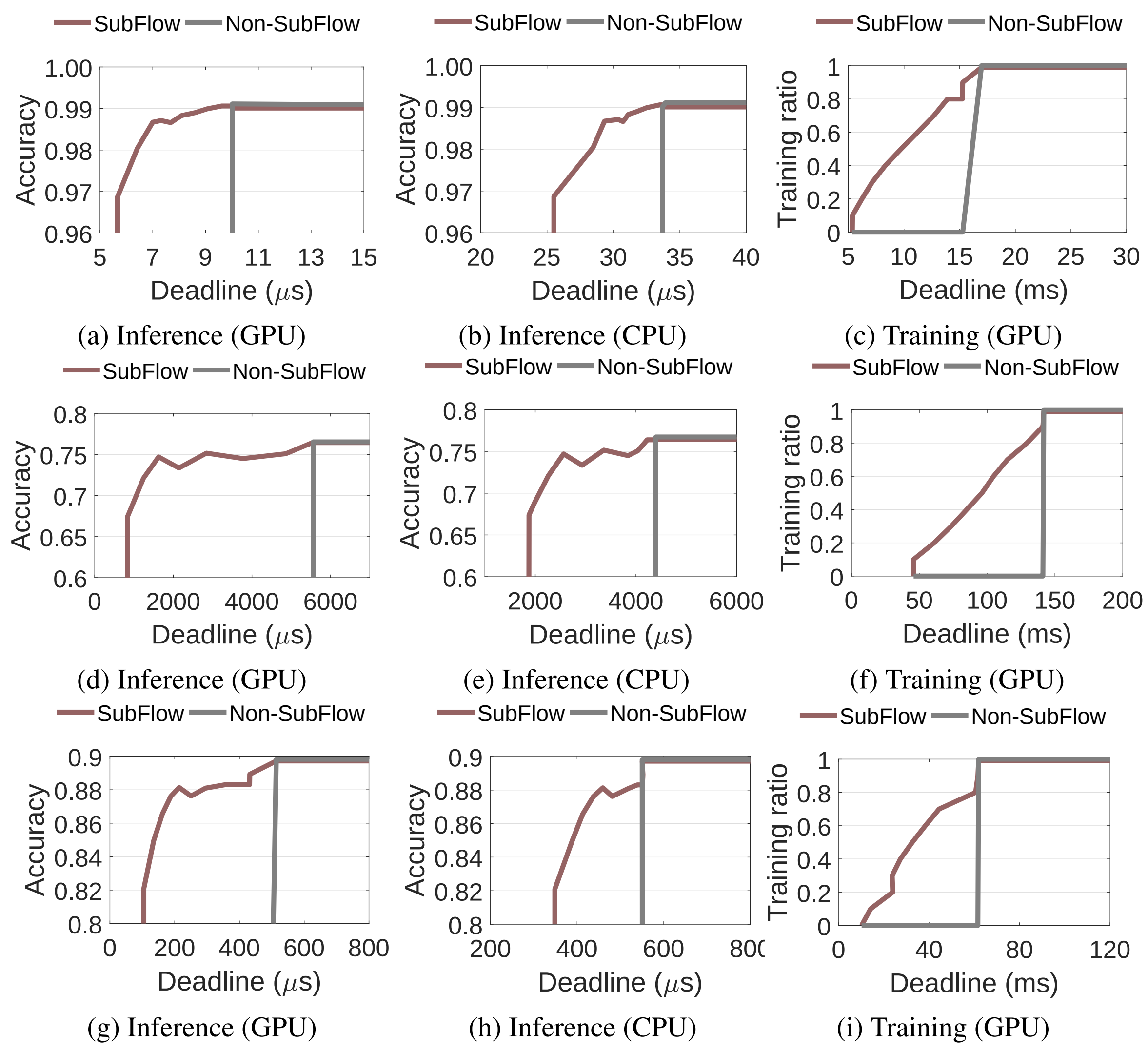
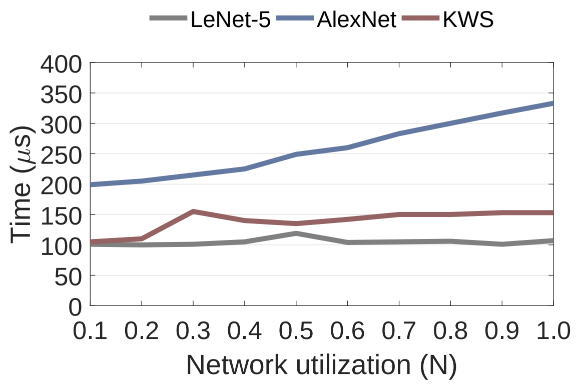
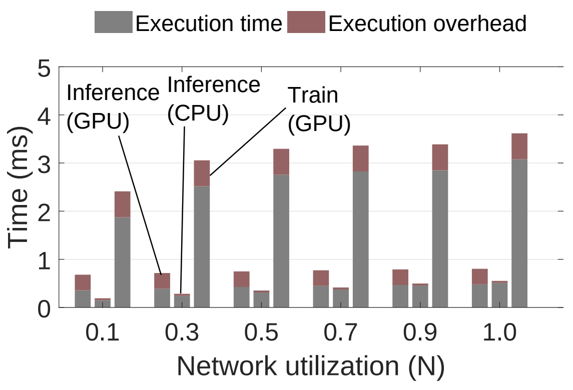


Figure 5.7: The usefulness (inference accuracy and training ratio) over dynamic deadline: SubFlow vs. non-SubFlow. (a)-(c): LeNet-5 (MNIST), (d)-(f): AlexNet (CIFAR-10), and (g)-(i): KWS (GSC).

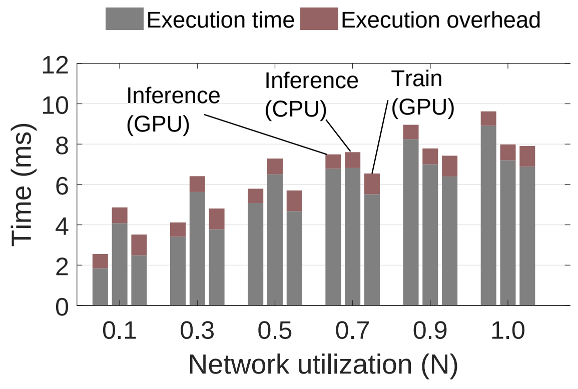
utilization settings since the same length of activation vector is generated for sub-network of any size; the only difference between sub-networks is the composition of ones and zeros. Also, their absolute time costs are low since an activation vector is efficiently generated from the rank of neurons that is pre-computed at compile-time. Figure 5.8b, 5.8c, and 5.8d show the sub-network execution overhead, which is higher than the construction overhead. For example, the inference overhead of AlexNet on GPU, $709\mu s$, which is 7% of the total inference time ($8915\mu s$), is almost twice higher than the construction overhead of the full-size sub-network ($333\mu s$, $N = 1.0$).



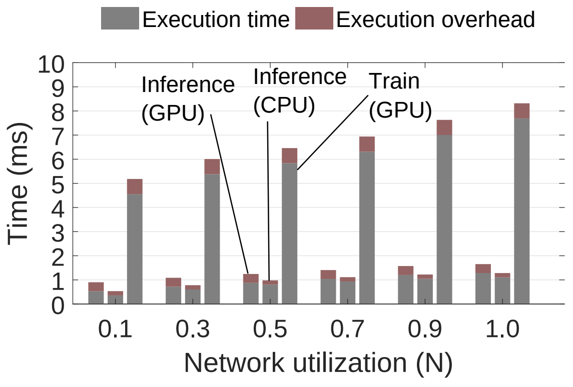
(a) Construction overhead



(b) Execution overhead: LeNet-5



(c) Execution overhead: AlexNet



(d) Execution overhead: KWS

Figure 5.8: The run-time overhead: (a): the construction overhead for all the three DNNs. (b), (c), and (d): the actual execution time (gray) vs. execution overhead (red) of each DNN.

The ratio of execution overhead to the total execution time increases as the sub-network size decreases, e.g., from 7% to 27% in AlexNet with network utilization settings 1.0 and 0.1, respectively. The execution overhead ratio increases for smaller sub-networks since the overhead remains similar for all sizes of sub-network while the actual computation time decreases with the size. It shows that the execution overhead is critical to small sub-networks and should be further decreased so that they can be efficiently executed with tighter time constraints.

5.6.5 Comparison with the State-of-the-Art

We compare SubFlow with two state-of-the-art DNN execution algorithms: 1) BranchyNet (Teerapittayanon et al., 2016) that makes an early exit of the DNN for fast inference and 2) AdapDeep (Liu et al., 2018b) that accelerates a DNN with a combination of compression techniques. Table 5.2

provides their inference and training speeds on CPU and/or GPU, and the inference accuracy of the two DNNs, i.e., LeNet-5 (MNIST) and AlexNet (CIFAR-10). We observe that SubFlow achieves comparable speedup and inference accuracy to the other two. Also, it achieves flexible execution for both inference and training, unlike the other two methods that lack such flexibility (AdaDeep) and training speedup (BranchyNet). For example, SubFlow AlexNet on GPU achieves dynamic speedup for both inference (1.0x–6.7x) and training (1.0x–3.1x), while BranchyNet achieves 1.0x–2.4x speedup only for inference without providing dynamic training speedup. AdaDeep achieves a fixed speedup for inference (2.3x on CPU), but does not achieve training speedup at all.

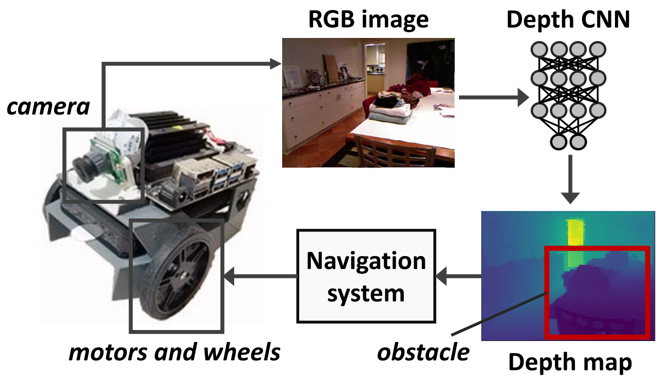
LeNet-5 (MNIST)	**Inference Speed (CPU / GPU)**	**Training Speed (GPU)**	**Inference Accuracy**
SubFlow	**1.0x–1.3x / 1.0x–1.8x**	**1.0x–3.2x**	**0.97–0.99**
BranchyNet (Teerapittayanon et al., 2016)	1.0x–5.4x / 1.0–4.7x	N/A	0.98–0.99
AdaDeep (Liu et al., 2018b)	1.8x / N/A	N/A	0.97
AlextNet (CIFAR-10)	**Inference Speed (CPU / GPU)**	**Training Speed (GPU)**	**Inference Accuracy**
SubFlow	**1.0x–2.4x / 1.0x–6.7x**	**1.0x–3.1x**	**0.67–0.76**
BranchyNet (Teerapittayanon et al., 2016)	1.0–1.5x / 1.0–2.4x	N/A	0.75–0.79
AdaDeep (Liu et al., 2018b)	2.3x / N/A	N/A	0.72

* For SubFlow, the network utilization is set as $N = [0.1, 1.0]$.

Table 5.2: Comparison between SubFlow, BranchNet, and AdaDeep.

5.7 Application

We implement an autonomous mobile robot as an example application of SubFlow, which detects obstacles by generating depth maps from a camera image in real-time (Chakravarty et al., 2017; Chen et al., 2016b; Mancini et al., 2016; Eigen et al., 2014). While driving, a CNN (convolutional neural network) transforms an RGB image into a depth map where the required latency of transformation changes based on the traveling speed of the robot. The faster it runs, the quicker the transformation should be performed to detect an obstacle in time. Figure 5.9a shows our mobile robot that executes a depth-estimation CNN (Chakravarty et al., 2017) with SubFlow on its GPU for obstacle detection. It is implemented using Jetson Nano (NVIDIA, 2019a), an embedded GPU platform having NVIDIA Maxwell GPU, ARM A57 CPU, and 4 GB of RAM. The robot has a camera in the front, and two motors and wheels on both sides installed on the skeleton that we printed on a 3D printer. Table 5.3 shows the architecture of the depth estimation CNN (Chakravarty et al., 2017) executed by the mobile robot. We use NYU depth dataset V2 (Silberman et al., 2012) for training and testing.



(a) Depth-based obstacle detection

(b) The mobile robot

Figure 5.9: SubFlow robot performing depth-based obstacle detection.

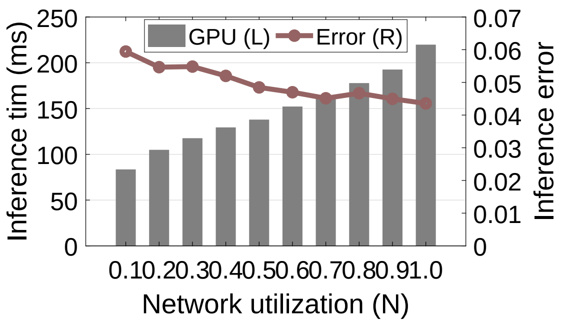


Figure 5.10: Execution time and error over the network utilization.

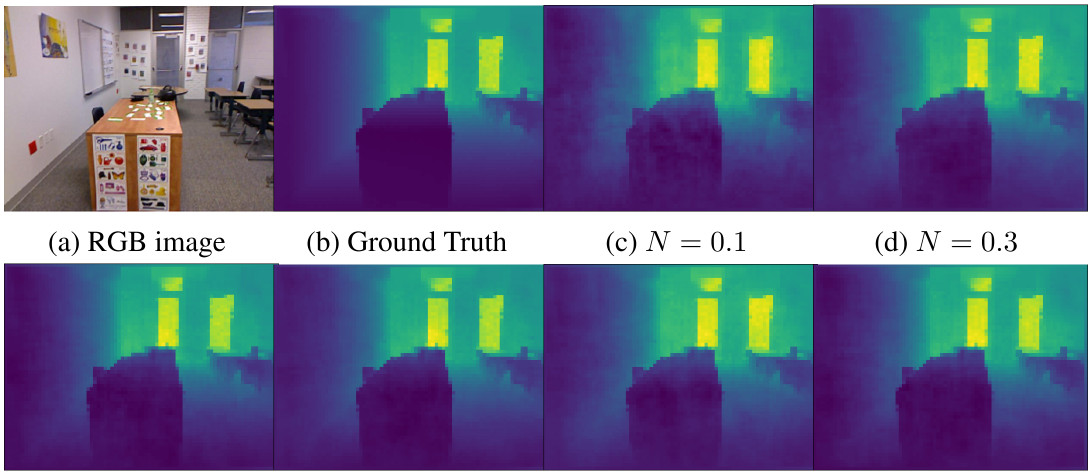
	Depth CNN (NYU2)
Layer 1	Input: 60×80×3
Layer 2	Conv1: 11×11×3×96
Layer 3	Conv2: 5×5×96×256
Layer 4	Conv3: 3×3×256×384
Layer 5	Conv4: 3×3×384×384
Layer 6	FC1: 2048
Layer 7	FC2 (Output): 4800

Table 5.3: The depth estimation CNN architecture (Chakravarty et al., 2017).

5.7.1 End-to-End Execution Time and Performance

Figure 5.10 shows the execution time and depth estimation error of the CNN over the network utilization, N. The execution time is measured the same way as in Section 5.6, and the estimation error is measured with linear RMSE (Mancini et al., 2016), which is calculated by

$\sqrt{\frac{1}{n}\sum_{j=1}^{n}\left\|\tilde{\mathbf{y}}_j - \mathbf{y}_j^*\right\|_2^2}$, where $\tilde{\mathbf{y}}_j$ and $\mathbf{y}_j^*$ is the j-th output of a sub-network and ground truth, respectively. Figure 5.11 shows example depth maps generated from different sub-networks with $N = 0.1,\ 0.3,\ 0.5,\ 0.9$, and 1.0.



(a) RGB image (b) Ground Truth (c) $N = 0.1$ (d) $N = 0.3$

(e) $N = 0.5$ (f) $N = 0.7$ (g) $N = 0.9$ (h) $N = 1.0$

Figure 5.11: Depth map images generated from different settings of N.

5.7.2 Real-World Deployment

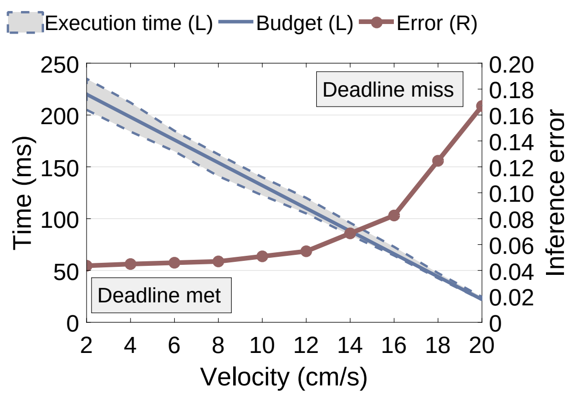
As a real-life experiment, we evaluate the execution time and depth estimation error by running the mobile robot at various speeds that impose different execution time budgets (latency) on the depth CNN. We deploy the robot in the corridor, kitchen, and bedroom of an apartment that has typical furniture such as chairs, desks, and a bed (Figure 5.9b). The robot runs for three hours and executes 50,000 CNN jobs. We randomly change the speed of the robot (2cm/s–20cm/s) to enable dynamic deadlines that we empirically obtain during preliminary experiment. The result, summarized in Table 5.4, shows that the execution of the CNN completes within the time budget with a small variance.

Since obstacle detection is critical to safe driving, the robot may want to execute only the sub-networks generating depth map with an error lower than a threshold, which makes it navigate without a collision. To experiment in this scenario, we limit the execution of sub-networks that cause large errors (0.068). Figure 5.12 shows the execution time and error over velocity with and

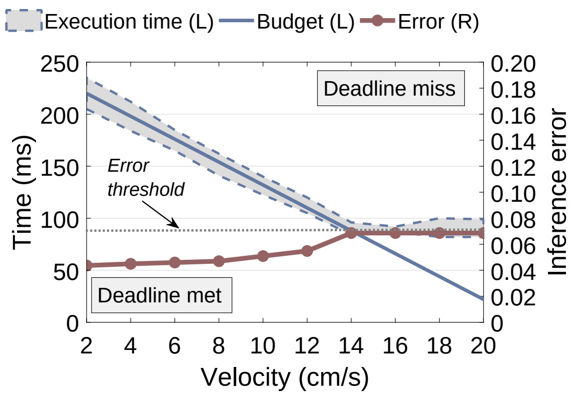
Velocity	**20 cm/s**	**16 cm/s**	**12 cm/s**	**8 cm/s**	**4 cm/s**	**2 cm/s**
Budget	**22 ms**	**66 ms**	**110 ms**	**154 ms**	**198 ms**	**220 ms**
Avg-ET	23.1 ms	68.2 ms	112.6 ms	152.2 ms	197.6 ms	219.8 ms
Min-ET	22.5 ms	66 ms	108 ms	147 ms	192 ms	212.5 ms
Max-ET	23.62 ms	69.7 ms	114 ms	155.7 ms	202 ms	225 ms
N	0.01	0.04	0.25	0.61	0.92	1.00
Error	0.1669438	0.082438	0.054829	0.046986	0.044965	0.04365

* **Velocity**: Traveling speed of the robot (centimeters per second), **Budget**: Execution time budget (milliseconds), **Avg-ET**: Average execution time (milliseconds), **Min-ET**: Minimum execution time (milliseconds), **Max-ET**: Maximum execution time (milliseconds), **N**: Network utilization, **Error**: Depth estimation error

Table 5.4: Execution time budget, actual execution time, network utilization, and depth estimation error of the mobile robot with various running speeds.



(a) Error threshold: None



(b) Error threshold: 0.068

Figure 5.12: The execution time and error over velocity with and without an error threshold: The execution time budget that changes based on the velocity is drawn with a diagonal line. The deadline is met if the execution time is under the diagonal line, missed otherwise.

without the error threshold. The execution time budget is shown as a diagonal line, implying that the execution time above the line is a deadline miss. Without the threshold, Figure 5.12a shows that the robot meets all the deadlines in the entire speed range, but generates a depth map with a high error at high speed. On the other hand, Figure 5.12b shows that the error is limited to 0.068 for all the speeds by executing only the sub-networks with an error below the threshold, which ensures the desired level of performance. In consequence of not performing the sub-networks resulting in error higher than the threshold, the robot misses the deadlines when running faster than 14 cm/s in return for the low error.

5.8 Discussion

Scalability to Larger DNNs. Although the DNNs used in the evaluation, e.g., AlexNet (Krizhevsky et al., 2012) (15M parameters), are smaller than ResNet (He et al., 2016a) (26M parameters), we expect SubFlow to achieve better results with larger networks like ResNet since they have more room for optimization (Wang et al., 2017). An induced sub-graph can be constructed with residual connections, and SubFlow supports both convolution and fully-connected layers. In this chapter, we followed to design our workload based on many recent works (Lee and Nirjon, 2019; Gobieski et al., 2019b; Liu et al., 2018b; Jiang et al., 2018; Yao et al., 2017b) for embedded systems, which demonstrates that results hold for both low-end CPUs and embedded GPUs. We hypothesize that smaller DNNs like LeNet-5 (LeCun et al., 1998) used in the evaluation are harder cases for SubFlow as there is little scope for speedup and/or compression.

Accuracy Requirements. While SubFlow minimizes the loss of inference accuracy when constructing a sub-network for the time-bound execution, the accuracy drop is expected to increase in general as the size of a sub-network decrease. Since the accuracy is critical for many safety-critical applications, SubFlow limits the maximum loss of accuracy above a certain level by controlling the network utilization parameter that limits the construction and execution of sub-networks whose expected accuracy is lower than the desired level. The expected accuracy over network size is obtained by running various sizes of sub-networks offline before the DNN is deployed on the system. For some applications where both accuracy and real-time execution are critical, SubFlow can provide intermediate inference results faster than the full-size network, which serves as preliminary guidance before getting the high-accuracy result from the full-size network.

5.9 Prior Work and Their Limitations

Dynamic Timing Constraints. Some early work argued limitations of standard timing constraints such as fixed deadlines and periods. For example, (Fohler, 1997) stated that only few tasks have ‘natural’ periods and deadlines. As an alternative, they presented dynamic timing constraints having more expressive power, which is similar to the dynamic time budget used in this paper; the time constraints for a single job of a task may be different for each job. (Gerber et al., 1995b) studied relative timing constraints and design based on end-to-end deadlines (Gerber et al., 1995a), and (Cheng and Agrawala, 1995) developed a scheduling algorithm for relative timing constraints. (Schild and Würtz, 2000) proposed using constraints satisfaction methods to schedule relative timing constraints. The slot shifting method for static schedules (Fohler, 1995) is capable of supporting limited dynamic timing constraints.

Real-Time DNNs. There have been several studies on real-time DNNs based on the learning algorithm and architecture of DNNs. RTDNN (Miralles and Bobi, 2016) adapts the parameters and structure of DNN to a dataset in real-time conditions. Although it performs an adaptation without requiring a significant number of samples, it does not support convolutional DNN and relies on competitive learning (Martinetz and Schulten, 1994) that is not widely used in many DNNs. Based on the constructive network model (Huang, 2003), (Huang et al., 2006) proposed a real-time learning algorithm, which can automatically select appropriate values of neural quantizers and determine the parameters of the network. However, their learning is performed without any real-time constraints, which is different from SubFlow having definite time budgets. Above all, none of them provide timing guarantee of DNN execution.

Imprecise Computing. The imprecise computation (Liu et al., 1994, 1991; Lin et al., 1987) divides a time-critical task into two sub-tasks: mandatory and optional. The mandatory sub-task is executed to completion to produce an acceptable result. The optional sub-task refines the result to reduce the error in the result. The milestone, sieve function, and multiple version method (Lin and Natarajan, 1988; Shih et al., 1991; Chung et al., 1990; Kenny and Lin, 1990) are the popular algorithms for it. However, the division of a task is not trivial and increases the complexity of scheduling by adding optional tasks to the system. Also, dividing a task into only two parts does not provide flexible execution. SubFlow does not require an artificial division of a task and automatically executes the proper amount of computation based on flexible construction and execution of sub-networks.

DNN Compression/Prunning. The need to deploy DNNs on resource constrained systems motivated techniques that can reduce the storage and computational costs, including knowledge distillation (Chen et al., 2017a; Hinton et al., 2015; Romero et al., 2014), low-rank factorization (Ioannou et al., 2015; Tai et al., 2015; Sainath et al., 2013), pruning (LeCun et al., 1990; Polyak and Wolf, 2015; Li et al., 2016; Yu et al., 2018; Guo et al., 2016), quantization (Li et al., 2017a; Wu et al., 2016; Han et al., 2015a), compression with structured matrices (Cheng et al., 2015; Sindhwani et al., 2015), network binarization (Li et al., 2017b; Rastegari et al., 2016; Courbariaux et al., 2016), and hashing (Chen et al., 2015b). However, they do not provide real-time guarantee due to their primary focus on size reduction. Also, the significantly compressed DNNs do not run nearly as significantly faster since most parameters are pruned in fully-connected layers while convolutional layers consume most computation time, as shown in (Guo et al., 2016; Park et al., 2016b; Han et al., 2015a). Although some algorithms, such as DeepIoT (Yao et al., 2018a, 2017b) compress DNNs achieving less execution time, the final network is not dynamically changed once it is compressed offline. Moreover, they lack easy-to-follow procedures and require significant effort, e.g., architecture modification, multi-rounds of retraining, fine-tuning. In contrast, SubFlow enables the run-time execution of multiple sub-networks of the DNN instead of compressing the DNN into one single network without requiring such an effort. SubFlow also supports time-bound training, which is missing in most compression works that only focus on the inference.

Improving Inference Speed. To improve the inference speed, parallel techniques such as SIMD (Patterson and Hennessy, 2013) have been used (Vanhoucke et al., 2011), which is also employed in the implementation of SubFlow. Also, faster algorithms specifically for 3x3 convolutional filters have been studied (Lavin and Gray, 2016) for VGGNet (Simonyan and Zisserman, 2014) and ResNet (He et al., 2016a). The early exit is another approach. CDL (Panda et al., 2016) adds classifiers to each layer and monitors the output to decide whether a sample can be exited early.

BranchyNet (Teerapittayanon et al., 2016) enables more general branches with additional layers at each exit point. In contrast, SubFlow executes all the layers without exiting in the middle. Instead, some neurons of each layer are selected and executed for speedup. To speed up sparse convolution, efficient sparse DNNs such as (Liu et al., 2015a), (Li et al., 2016), and (Lebedev and Lempitsky, 2016) have been proposed. Escoin (Chen, 2018) applies the direct sparse convolution (Park et al., 2016a) to GPU in optimizing parallelism and locality. SparseSep (Bhattacharya and Lane, 2016) leverages the sparsification of fully-connected layers and the separation of convolutional kernels for wearable devices. Although SubFlow uses the direct sparse convolution, it does not rely on CSR (compressed sparse row) format that incurs overhead of decoding the sparse format, unlike them.

Improving Training Speed. While the inference latency has been an active area of research, few studies have been conducted to improve the training speed. Dropout (Srivastava et al., 2014) and DropConnect (Wan et al., 2013) can be used not only to increase the performance with reduced overfitting but also to reduce training time by performing back-propagation only for a part of DNN. StochasticDepth (Huang et al., 2016a) starts with deep networks, but during training, randomly drops a subset of layers and bypasses them with the identity function. Highway networks (Srivastava et al., 2015) proposes to modify the architecture of deep feed-forward networks such that information flow across layers becomes easier based on LSTM (long short term memory) (Gers et al., 1999; Hochreiter and Schmidhuber, 1997). In meProp (Sun et al., 2017), only a small subset of the gradient is computed to update the model parameters in back-propagation. MSBP (memorized sparse back-propagation) (Zhang et al., 2019) proposes to store unpropagated gradients in memory for the next learning to remedy the problem of information loss when accelerating propagation through sparseness. They either change the DNN architecture or select weight parameters to be trained based on the magnitude, which may eliminate wrong parameters (Hassibi and Stork, 1993; LeCun et al., 1990), unlike SubFlow that does not modify architecture and use the second-order derivative for selection.

5.10 Summary

We propose SubFlow that enables real-time inference and training of a DNN by dynamically executing an induced sub-graph of the DNN according to varying time budget. We implement SubFlow by extending TensorFlow, which allows a programmer to design time-aware DNNs based on SubFlow. Our empirical evaluation result shows that time-bound inference and training are achieved without experiencing significant performance loss. We implement an autonomous robot as an application of SubFlow, which demonstrates that the object detection task is completed within the time budget that dynamically changes based on the running speed of the robot.

CHAPTER 6: OPPORTUNISTIC ACCELERATED LEARNING

In recent years, deep neural networks (DNN) (Schmidhuber, 2015; LeCun et al., 2015) have shown stellar performance in solving problems in machine learning and related fields (He et al., 2016a; Schroff et al., 2015; Krizhevsky et al., 2012; Young et al., 2018; Goodfellow et al., 2016; Cambria and White, 2014; Deng, 2014; Deng et al., 2013). Following the trend, embedded systems have started to implement lightweight versions of DNNs (Yao et al., 2018b; Wang et al., 2018; Gobieski et al., 2018a), primarily focused on inference or generalization tasks (Lane et al., 2017). The de facto approach to enable deep inference on resource-constrained systems is to obtain a pre-trained model from some other sources and then to compress and/or prune the network until it fits the memory and computing capacity of the embedded platform (Manessi et al., 2018; Zhou et al., 2018; Han et al., 2016, 2015a; Gong et al., 2014). Needless to say, such compression and pruning hacks inevitably degrade the performance, and many large-sized DNNs are quite challenging to port on resource-constrained embedded platforms even after compression and pruning.

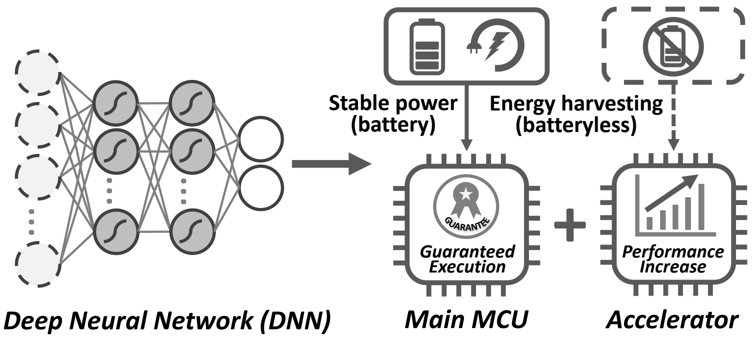


Figure 6.1: Neuro.ZERO: The batteryless accelerator, powered by harvested energy, opportunistically enhances the run-time performance of DNN execution without consuming power from the main system. The main MCU (microcontroller unit) guarantees seamless execution of DNN by using a stable power source such as a battery.

The performance of DNNs running on an embedded system (Chauhan et al., 2018; Gobieski et al., 2018a; Bhattacharya and Lane, 2016) is limited by the platform's CPU, memory, and battery-size; and their scope is limited to inference tasks only. To overcome this, special-purpose co-processors, called *DNN accelerators*, have been proposed and productized (Apple, 2017; Qualcomm, 2017), primarily targeted to smartphone-grade mobile systems. Although especially architected hardware in these accelerators enables faster execution of DNNs, they have some major practical limitations. First, DNN computations are power-hungry. The power consumption of these accelerators remains as a fundamental bottleneck—prohibiting them to be used in battery-powered systems. Second, existing accelerators primarily focus on speeding up the execution of an offline-trained DNN inference task. In general, there is a lack of research on how to facilitate run-time adaptation so that the inference accuracy increases over time as resources become available or newly sampled sensor data can be used to fine-tune the performance. Third, while application-specific hardware accelerators of different types such as FPGAs and ASICs are effective, their lack of standardization, unavailability to system developers, and excessive price are slowing down the development of engineered systems that could leverage DNN acceleration in their embedded sensing and inference applications.

In this chapter, we introduce *Neuro.ZERO*—a novel co-processor architecture consisting of two microcontroller units (MCUs): 1) a battery-powered main MCU that executes a scaled-down[1] DNN inference task, and 2) a batteryless (energy-harvesting) accelerator MCU that enhances the performance of DNN inference that runs on the main MCU. A high-level architectural diagram of Neuro.ZERO is shown in Figure 6.1. Unlike existing DNN accelerators that primarily focus on improving the inference speed (Wang et al., 2016; Gokhale et al., 2014), the accelerator in Neuro.ZERO improves the run-time performance of the DNN on the main MCU by *increasing inference accuracy* or by *enabling on-device training*. Since the accelerator does not draw power from the main system, we call it a *zero-energy* accelerator. By having two MCUs, one powered by a battery and one powered by harvested energy, Neuro.ZERO guarantees sensing and inference for all sensor data while enjoying opportunistic run-time performance gain without spending system's energy. Neuro.ZERO is implemented on off-the-shelf, low-power, low-cost MCUs (TexasInstruments, 2018), and its source code is open (Embedded Intelligence Lab (UNC Chapel Hill), 2019b)–which helps developers build low-power, intelligent sensing, and inference systems faster and at a lower cost.

The architecture of Neuro.ZERO falls into the general category of energy-aware heterogeneous multi-core systems such as ARM's big.LITTLE (Arm, 2013; Kamdar and Kamdar, 2015) and application-specific systems (Naderiparizi et al., 2017; Lu et al., 2011). However, Neuro.ZERO takes this to an extremity where one of the cores runs completely on harvested energy. It flips a common practice of energy-aware heterogeneous multi-core systems where typically a lower-power core remains active, and it controls the sleep/wake cycles of a higher-power core based on the computational demand. Instead, a new execution paradigm is introduced in Neuro.ZERO, where the main MCU executes sensing and basic inference tasks as programmed by a developer to meet its timing and energy constraints, and when the batteryless MCU harvests enough energy to execute a task by itself, it uses up that energy to improve the main MCU's performance in executing its accelerated inference task.

The proposed zero-energy accelerator follows standard practices of intermittently-powered systems. Its core framework is built upon existing work on intermittent computing that address important problems such as atomicity (Maeng et al., 2017; Colin and Lucia, 2016), consistency (Maeng et al., 2017; Colin and Lucia, 2016; Lucia and Ransford, 2015), programmability (Hester et al., 2017), timeliness (Hester et al., 2017), and energy-efficiency (Colin et al., 2018; Hester et al., 2015b; Buettner et al., 2011) to enable efficient code execution of general-purpose tasks. Neuro.ZERO complements existing literature and solves new and higher-level system challenges resulting from the heterogeneous execution pattern of Neuro.ZERO cores as well as fundamental challenges in executing accelerated inference and training on resource-constrained and intermittently-powered systems.

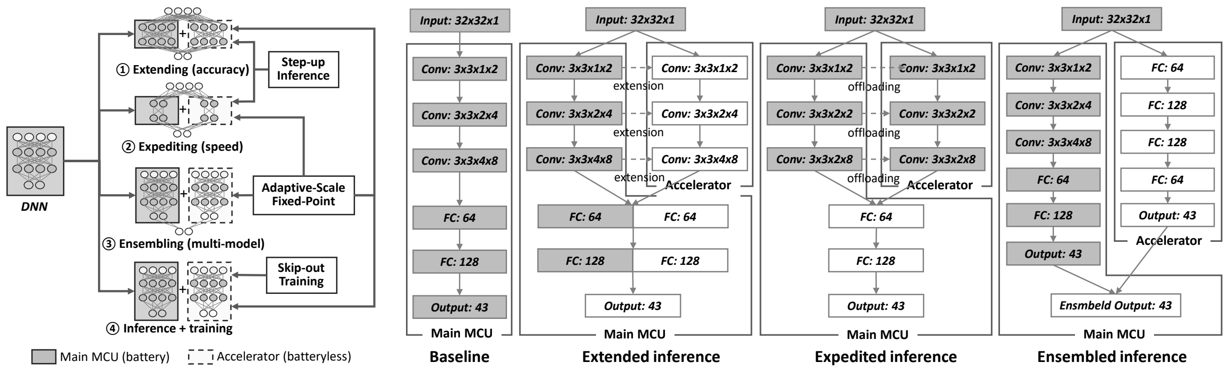
Neuro.ZERO opportunistically accelerates the run-time performance of a DNN via one of its four acceleration modes: *extended inference*, *expedited inference*, *ensemble inference*, and *latent* *training* which facilitates execution of larger sized networks, splits the given DNN for parallel execution, improves confidence of inference via ensembling (Krogh and Vedelsby, 1995), and updates the DNN weights via online training, respectively. To enable these modes, two sets of algorithms have been developed. First, energy and intermittence-aware algorithms have been developed that *steps-up* the DNN inference by scaling up the size of DNN based on the current energy level and *skips-out* back-propagation (Werbos et al., 1990) of some weights during online training as the amount of harvested energy fluctuates at run-time. Second, a fast and high-precision adaptive fixed-point arithmetic has been proposed that beats existing floating-point and fixed-point arithmetic in terms of speedup and precision, respectively, and achieves the best of both. To demonstrate the efficacy of Neuro.ZERO, we implement two applications that use camera and microphone to recognize certain images (i.e., traffic signs) and audio events (i.e., voice commands). These systems have been tested extensively using both standard datasets, i.e., MNIST (LeCun et al., 1998), CIFAR-10 (Krizhevsky et al., 2009), SVHN (Netzer et al., 2011), and Fashion MNIST (Xiao et al., 2017), as well as in real-world experiments.

6.1 Overview

This section describes the architectural design of Neuro.ZERO, the rationale behind the design, and two example applications.

6.1.1 System Design

The goal of Neuro.ZERO is to increase the run-time performance of DNN on resource-constrained, MCU-based systems by having a low-power energy-harvesting MCU as an *accelerator*, that opportunistically improves the accuracy or speed of DNN inference, without drawing any power from the battery. Figure 6.2a shows an architectural diagram of Neuro.ZERO, which depicts how a DNN is converted to one of four different architectures at compile time. At run-time, the shaded part of the generated network run on the main MCU, while the rest run on the accelerator only when it is active. The algorithms enabling these modes are shown on the right.



(a) Neuro.ZERO system design (b) Examples of the three zero-energy inference acceleration: extended, expedited, and ensembled inference

Figure 6.2: Neuro.ZERO's four modes accelerate a DNN in terms of accuracy, speed, multi-model, and training by extending, expediting, ensembling, and training the DNN on the accelerator. They are enabled by energy-aware acceleration (step-up inference and skip-out training) and numerical acceleration (adaptive-scale fixed-point).

Basic Working Principle. Neuro.ZERO comes with a compile-time tool and a run-time system. The compile-time tool takes a baseline DNN architecture, a training dataset, and an acceleration mode as an input. Depending on the chosen acceleration mode, Neuro.ZERO creates two network architectures, trains them using the given training dataset, and generates two DNNs as the output (one for each MCU)—which are ready to be executed on the two-MCU hardware platform designed for Neuro.ZERO. The DNN for the main MCU is generated based on the baseline DNN by appending the necessary architecture for acceleration without changing the baseline DNN. It ensures that the standalone execution of the main MCU is self-sufficient in satisfying the desired application-level performance goals (e.g., achieving the same accuracy and speed of the original baseline DNN). When the accelerating DNN is executed on the accelerator, the two networks combinedly are expected to achieve a better inference accuracy and/or speed. The acceleration does not impose significant overhead on the main MCU since the baseline DNN does not require to be swapped in and out of the memory as the accelerator goes ON and OFF.

The run-time system of Neuro.ZERO is responsible for executing the two DNNs by managing the coordination between the two MCUs. The run-time system also keeps track of the status of the two MCUs and provides APIs to know whether the accelerator is active or involved in the inference as well as APIs to turn ON/OFF the accelerator (e.g., for debugging or experiment purposes). To ensure the consistent execution of DNN, the accelerator executes the DNN only when the energy harvester accrues enough energy to complete a full pass of feed-forward (from input to output layer).

Four Modes of Acceleration. To improve the run-time performance of DNNs, Neuro.ZERO supports four modes of acceleration. Each mode takes advantage of the intermittently-powered accelerator in a unique manner. The *extended inference* mode improves the inference accuracy by extending the DNN's structure and running the extended part on the accelerator. The *expedited inference* mode increases the inference speed by offloading some part of the original DNN to the accelerator. In both extended and expedited modes, Neuro.ZERO ensures that the main MCU runs a self-sufficient DNN when the accelerator is not active. The *ensembled inference* mode runs a different DNN model on the accelerator as a second DNN and combines the output of the two independent DNNs to increase the inference accuracy. The *latent training* mode enables an intermittent on-device training of the baseline DNN for unseen data on the accelerator while allowing the main MCU to keep executing the inference task. The details of these modes are discussed in Section 6.2.

Algorithms Enabling Acceleration. The four acceleration modes of Neuro.ZERO are enabled by a set of algorithms that accelerate the DNN inference and training on an intermittently-powered system and expedite floating-point arithmetic. Since the accelerator runs on sporadically harvested energy, tasks running on it execute intermittently. Such an intermittent execution pattern makes both the inference and the training of a DNN challenging. To address this, we propose two novel algorithms, namely the *step-up inference* and the *skip-out training*, which accelerate the inference and training of DNNs in proportion to the harvested energy (Section 6.3).

Despite these accelerations, we observe that the execution of DNN, in general, is extremely slow on low-power, low-cost MCUs that do not have hardware support for floating-point operations (Anderson et al., 1967). To address this well-known issue, most low-power embedded systems use *fixed-point* arithmetic (Oberstar, 2007), which is computationally efficient but numerically inaccurate than floating-point. In Neuro.ZERO, we rethink the implementation of fixed-point operations and propose *adaptive-scale fixed-point* number representation that provides both the numerical correctness of floating-point arithmetic and the speed-up of fixed-point arithmetic. This is described in Section 6.4.

6.1.2 Design Rationale

We compare three alternative choices of processors for the accelerator in terms of their power consumption, CPU performance (measured in Dhrystone MIPS (Weiss, 2002)), and cost in Table 6.1. Considering the low price and ultra-low power consumption, an MCU is the most suitable choice for an energy harvesting system like Neuro.ZERO as they can be run intermittently on harvested energy and wake-up more frequently due to shorter charge-discharge cycles, and enable large scale deployment due to low-cost. For example, when an RF harvester (Powercast, 2016a,b) (generating 0.2mW–2.0mW) is used, an FPGA or an SoC would take several minutes to hours to harvest enough energy before they can execute any workload. Such a long delay is not suitable for Neuro.ZERO, as the accelerator is more likely to miss sensor data during its long charging time and the value of processing the data may be lost (e.g., in time-sensitive applications) after such long delay. Although large energy harvesters and huge capacitors as energy storage could be a makeshift solution, such systems will be bulky and expensive, and thus are not suitable for most embedded sensing systems.

Accelerator/Processor Type	Power	Performance	Cost
MCU – TI MSP430 (TexasInstruments, 2018)	3.8-6.2mW	13 DMIPS	$3-$5
FPGA – Xilinx Spartan 6 (Shahzad and Oelmann, 2014; Xilinx, 2011)	24-109mW	166 DMIPS	$30-$33
SoC – Qualcomm Snapdragon (Qualcomm, 2018)	2.1-4.8W	13,860 DMIPS	$70-$199

Table 6.1: Comparison of processor choices for the accelerator.

Having an MCU as the choice for the accelerator, Table 6.2 compares four co-processor designs for up to two MCUs. The first two rows show single-MCU systems, and the rest show two-MCU systems. We observe that only when the main MCU is battery-powered, and the accelerator is energy-harvesting, we achieve seamless execution of tasks (on the main MCU) and energy-savings and increased performance (due to the energy-harvesting accelerator).

Main MCU's Power Source	**Accelerator MCU's Power Source**	**Timely/ Seamless Execution**	**Energy Harvesting (Accelerator)**	**Performance Increase by Accelerator**
Battery	-	✔	-	✘
Harvesting	-	✘	-	✘
Battery	Battery	✔	✘	✔
Harvesting	Harvesting	✘	✔	✔
Battery	**Harvesting**	✔	✔	✔

Table 6.2: Comparison of MCU-based architectural choices.

Extensibility and Cost. Neuro.ZERO is developed as a two-MCU system. However, its design principles and algorithms are applicable to many-MCU systems where a subset of MCUs are battery-powered, and the rest are powered by harvested energy. Having additional MCUs in a system adds a one-time cost, but considering their small form-factor and the low cost (<$5 per unit), the benefit of increased accuracy and speedup clearly outweighs the cost.

6.1.3 Example Application Scenarios

We describe two example applications of Neuro.ZERO: 1) a traffic sign recognizer, and 2) a voice command recognizer, which classifies traffic sign images and voice audio data, respectively. In Section 6.7, we describe their implementation and evaluation results.

Wearables for Pedestrian and Biker's Safety. Pedestrians and bikers are often not fully aware of their surroundings, which is causing their lives (Administration, 2013). To augment perception and cognition of pedestrians and bikers, wearable systems have been proposed that recognize imminent dangers on the road, alert the user on time, and help them avoid injury and death (He et al., 2017; Chen et al., 2016a; Singh, 2007). We propose to augment the ability of pedestrians and bikers to see and recognize traffic signs by enabling road-sign recognition on camera-based low-power wearable systems. These battery-powered systems need to process camera images in real-time and produce accurate classification results. Using Neuro.ZERO, we can improve the accuracy and confidence, and lower the execution time of the image recognition applications for such wearable systems. As these systems are expected to be used outdoors, solar energy can be harvested to power the accelerator. In this application, Neuro.ZERO can be operated

1) in the extended inference mode when the user enters an environment that requires higher-resolution images to detect objects, 2) in the expedited mode when the user is in a busy area, 3) in the ensemble mode when there are multiple cameras or a different sensor (e.g., microphone) to independently detect the same event, or 4) in the training mode when environment-specific parameter tuning is necessary to obtain better classification results.

Voice Commands for Smarter Things. Voice-based communication with everyday objects in natural languages is becoming a reality. Today, devices like Amazon Echo acts as a "middle-man" to enable voice communication with smart devices such as home appliances, remote controllers, thermostats, light bulbs, switches, speakers, clocks, and many more. We envision that, in a few years, voice-communication capability will be directly built into every smart object. In order to realize this vision, building low-power, low-cost, MCU-grade systems that recognize voice commands are essential. Neuro.ZERO enables the development of these next-generation smart objects that are able to sense and interpret voice commands on-device and in real-time, and opportunistically improve their inference performance at runtime by leveraging the harvested power from ambient RF energy at indoor environments. In this application, Neuro.ZERO can be operated 1) in the extended inference mode when the environmental noise level is high or the device is far, 2) in the expedited mode when the user interacts with the device more frequently or when there are many users issues commands to the device, 3) in the ensemble mode when there are multiple microphones and each can be specialized on detecting different subset of voice commands, or 4) in the training mode when person or environment-specific parameter tuning is necessary to achieve more accurate classification results.

6.2 Acceleration Modes

In this section, we describe the four modes of zero-energy acceleration in Neuro.ZERO, of which, only one mode is active at a time, as configured by the application developer.

6.2.1 Extended Inference

A larger network having more neurons, in general, is a better classifier (Wang et al., 2005; Lawrence et al., 1998, 1997). Although there are studies showing that the accuracy of a DNN drops when its size grows beyond a certain limit (Glorot and Bengio, 2010; Hochreiter et al., 2001), for resource-constrained embedded systems like Neuro.ZERO, we safely assume that more neurons and connections are likely to improve its inference accuracy. The memory of an MCU being small, a DNN residing in the main MCU of Neuro.ZERO is benefited by additional neurons in the accelerator since some DNNs cannot be stored in a single MCU even after compression. For example, SqueezeNet (470KB) (Iandola et al., 2016) is a compressed version of AlexNet (Krizhevsky et al., 2012), but it is still too large to fit in the main MCU (256KB for MSP430). In such cases, an accelerator becomes necessary for the system to achieve desirable performance.

Based on this assumption, given the baseline DNN, Neuro.ZERO generates an extended version of it by adding additional neurons to each layer. The newly added neurons are identical in numbers and types for each layer. Figure 6.2b shows an example of an extended DNN that has three extended *convolutional* (Conv) and two extended *fully-connected* (FC) layers having the same dimensions as in the baseline DNN. To avoid creating a dependency between the two MCUs which requires extensive communication between them at run-time, we intentionally regularize (remove) the connections between convolutional filters running on the two MCUs and execute all fully connected layers on the main MCU. The benefit of this are two-fold: first, the main MCU independently makes inferences, and second, execution of half of the convolutional filters, which account for 45% of the total energy consumption, are offloaded to the accelerator. For example, scaled-down versions of popular DNNs like ResNet (He et al., 2016a) can be divided into two networks and accelerated by using parallel algorithms for DNNs such as (Günther et al., 2018).

The accelerator executes the convolutional filters in an energy-aware manner by selecting a subset of them for execution based on the current level of harvested energy. We call this *step-up* *inference* since the accelerator's effort toward increasing the inference accuracy increases proportionally with the harvested energy. The details of the algorithm are described in Section 6.3.1.

Exploiting the inherent parallelism in DNN architecture is a common technique to increase the speedup of DNN execution. We leverage this parallelism in Neuro.ZERO by executing a subset of the convolutional filters of the baseline network on the accelerator. Like the extended inference mode, the fully connected layers run on the main MCU to ensure that the main MCU makes inferences without requiring frequent communication with the accelerator. Since executing convolutional layers take as much as 90% of the total execution time, the expedited mode cuts down the inference time approximately by 45%. Figure 6.2b shows an example of expedited DNN having three convolutional layers offloaded from the baseline DNN. Although this mode looks similar to the extended inference, the main difference between the two is that unlike the extended mode, the expedited mode trades off accuracy for speedup.

6.2.2 Expedited Inference

Exploiting the inherent parallelism in DNN architecture is a common technique to increase the speedup of DNN execution. We leverage this parallelism in Neuro.ZERO by executing a subset of the convolutional filters of the baseline DNN on the accelerator. Like the extended inference mode, the fully connected layers run on the main MCU to ensure that the main MCU makes inferences without requiring frequent communication with the accelerator. Since executing convolutional layers take as much as 90% of the total execution time, the expedited mode cuts down the inference time approximately by 45%. Figure 6.2b shows an example of expedited DNN having three convolutional layers offloaded from the baseline DNN. Although this mode looks similar to the extended inference, the main difference between the two is that unlike the extended mode, the expedited mode trades off accuracy for speedup.

6.2.3 Ensembled Inference

Unlike the above two modes, the ensembled inference mode executes an independent DNN on the accelerator, which is given as an additional input to Neuro.ZERO. This mode enables execution of a different DNN that performs the same inference task and provides a second opinion on the inference result. It also allows execution of a different inference task that may complement inference results on the main MCU. Furthermore, the accelerator may choose to use a different sensor than the main MCU to perform the same or a different inference task than the main MCU. Thus, this mode offers the most flexibility, but it does not necessarily improve the speedup. However, by carefully choosing a suitable combination of sensors and inference tasks, novel multi-modal, multi-objective sensing and inference systems can be developed with this mode—which may effectively increase the accuracy and speedup of inference. Figure 6.2b shows an example of an ensembled DNN. Unlike the baseline DNN having a convolutional architecture, the accelerator runs a fully-connected DNN that learns non-spatial features. When the accelerator is available, the output of the accelerator is combined with that of the main MCU to generate the final inference result.

Although Neuro.ZERO is a minimalistic system that has only one main MCU and one accelerator, the design can be extended to support many-MCU systems that run more complicated tasks and ensembles of many networks. Outputs of these networks can be combined using existing techniques such as concatenation and averaging (Tyagi and Mishra, 2014; Hansen and Salamon, 1990).

6.2.4 Latent Training

Real-time training of machine learning classifiers is a desirable feature for many mobile and embedded systems (Foundation, 2019). In recent years, we see a growing trend of online training of embedded classifiers in commercial products such as iPhone's face recognition (Apple, 2017), Google Clip's image capturing (Google, 2018a), and Android smartphone's key-press learning features (Hard et al., 2018). To future-proof Neuro.ZERO, we introduce a fourth acceleration mode that enables retraining of the DNN on the accelerator. We call this *latent training* since the training process gradually progresses over time as the accelerator harvests energy.

Figure 6.3 illustrates the process of latent training. Training happens separately on the accelerator while the main MCU independently executes inference tasks. The two MCUs asynchronously communicate with each other only when the DNN model has been updated via training. Then, the main MCU fetches the newly updated model and uses it from then on.

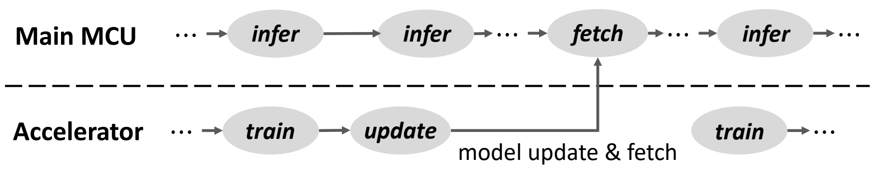


Figure 6.3: The latent training on the accelerator independently (and intermittently) trains and updates a DNN model that is asynchronously fetched by main MCU, which increase the performance.

Unlike the DNNs that have several millions of parameters, requiring thousands of training examples to train, and are meant to run on high-end processors, the DNNs in Neuro.ZERO are much smaller in size and training happens online, i.e., only one example at a time to perform a back-propagation algorithm. However, even a single round of back-propagation is difficult on a small system that is powered intermittently. To solve this challenge, we propose an energy-aware back-propagation algorithm that updates the weight parameters of a DNN in proportion to the amount of harvested energy. The details of the algorithm are in Section 6.3.2.

One caveat of on-device training is that the system requires labeled data. To handle this, we propose several solutions: 1) applying semi-supervised learning principles that do not require labeled data (Peikari et al., 2018; Lee, 2013; Zhu and Goldberg, 2009), 2) relying on an external, high-accuracy inference system to obtain the labels at run-time, and 3) in a distributed sensor network or ensemble scenario, aggregating (e.g., voting) neighboring nodes' inference results and treat it as the label. In our demonstration, we use 2) for the simplicity of implementation.

6.3 Energy-Aware Acceleration

In this section, we introduce energy-aware acceleration algorithms called *step-up inference* and *skip-out training*, which enable intermittent inference and training of DNN based on the energy level.

6.3.1 Step-Up Inference

The *step-up inference* enables flexible inference acceleration of DNNs on the unpredictable harvested-energy. It dynamically adjusts acceleration in proportion to the run-time energy level by stepping up and down the size of DNN executed on the accelerator with multiple steps. Since the network size grows along with steps, e.g., step four has a larger DNN than step three, etc., a higher step is expected to achieve better performance acceleration (i.e., higher accuracy) than a lower one. Every execution of inference, the highest step that can be executed with the current energy level is selected among total n steps and executed on the accelerator.

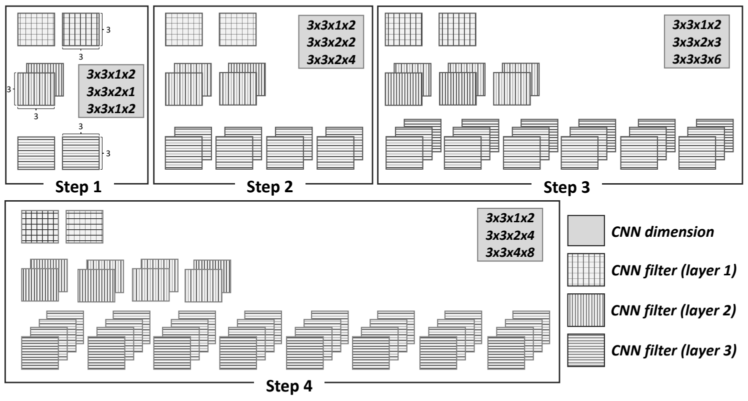


Figure 6.4: An example of step-up inference: The number of CNN filters running on the accelerator incrementally increase along with steps, and only one step is executed based on the energy level.

A set of n steps can be expressed as a set $S = \{S_1, S_2, ..., S_n\}$ and the amount of energy required to execute each step is given by another set $C = \{c_1, c_2, ..., c_n\}$, where S_i is the set of CNN filters of i-th step, and c_i is the energy consumption of the i-th step. Figure 6.4 depicts an example of four steps having different numbers of CNN filters that incrementally increase along with the steps. Starting from the baseline DNN, each step grows by adding a set of new filters to the previous step. Thus, $S_{i+1} = S_i \cup \{\text{new CNN filters}\}$ and $S_i \subset S_{i+1}$. The step S_{i+1} is obtained by training $\{\text{new CNN filters}\}$ step-by-step until n while freezing S_i. In this way, the filters of the previous steps are reused without changes, and they are prevented from learning redundant features. While the total number of steps, n can be arbitrarily set at compile-time based on energy harvesting pattern, the total number of accelerating filters, $\sum_{i=1}^{n} |S_i|$ is limited to the same number of filters as the baseline DNN in the main MCU.

For every inference acceleration, Neuro.ZERO determines a step to be executed by the accelerator based on the currently-available energy at run-time. The step to be executed at the k-th inference execution, s_k is determined by $s_k = \text{argmax}_{i \le n} c_i$ subject to $c_i \le e_k$ where e_k is the current energy level at the k-th inference.

$$s_k = \operatorname*{argmax}_{i \le n} c_i \text{ subject to } c_i < e_k \tag{6.1}$$

where e_k is the current energy level at the k-th inference.

6.3.2 Skip-Out Training

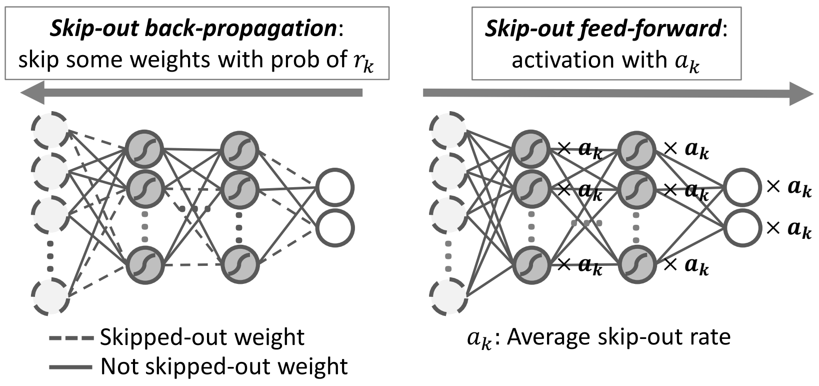
The *skip-out training* enables intermittent training of DNN on the irregular energy harvesting pattern. It accelerates a train by ensuring the completion of one execution of back-propagation regardless of the amount of currently-available energy.

Skip-Out Back-Propagation. Unlike conventional training, the skip-out algorithm skips a back-propagation step for some of the weights with the skip-out rate, r_k at the k-th iteration of training:

$$r_k = 1 - \frac{1}{n} \min\left(\left\lfloor \frac{e_k}{e_f + e_b} \right\rfloor , n \right) \tag{6.2}$$

where n is a total number of weights in a DNN, e_k is the current energy level at the k-th iteration, e_f is the amount of energy needed for feed-forward of one weight, and e_b is the amount of energy needed for back-propagation of one weight. Given the skip-out rate, r_k and a total number of weights, n in a DNN, the number of weights to be trained at the k-th iteration, n_{r_k} is given by $n_{r_k} = \lfloor n(1 - r_k) \rfloor$.

For each k-th iteration, the skip-out rate, r_k is obtained from the current energy level, e_k; and only n_{r_k} weights are trained by back-propagation. Therefore, a different number of weights are trained every iteration, increasing the speed of training by guaranteeing the completion of one back-propagation regardless of the current energy level. Figure 6.5 shows back-propagation with skip-out.



Skip-out back-propagation: skip some weights with prob of r_k
Skip-out feed-forward: activation with a_k
$\times a_k$
Skipped-out weight
Not skipped-out weight
a_k: Average skip-out rate

Figure 6.5: Skip-out back-propagation: Some weights are skipped with skip-out rate r_k for every k-th iteration. Skip-out feed-forward: All neurons are multiplied with the average skip-out rate a_k.

The weights to be skipped are selected using Bernoulli distribution (Uspensky, 1937) with probability, r_k with no other considerations such as their current values. This kind of skipping is effective and is known as *drop-out* (Srivastava et al., 2014) since it not only increases the training accuracy but also mitigates the overfitting problem (Hawkins, 2004). Moreover, the Bernoulli distribution is one of the best choices for our system as any other selection algorithm, e.g., sorting or scoring, consumes more energy, which would not leave enough energy for back-propagation.

Skip-Out vs. Drop-Out (Srivastava et al., 2014) . The difference between skip-out and drop-out is that the skip-out rate changes at every iteration while the drop-out rate stays the same for the entire training (e.g., 0.5). Another difference is that the skip-out algorithm leaves the selected weights as they are without training, so they are used in back-propagation for the survived weights while drop-out completely removes them by setting their values to zero.

Skip-Out Feed-Forward. The skip-out-based feed-forward activation computation at the k-th iteration of training is given by:

$$o_j^{(l)} = \sum_i w_{i,j}^{(l)} \cdot \varphi(o_i^{(l-1)}) \cdot a_k \text{ and } a_k = \frac{1}{k}\sum_{i=1}^{k} r_i \tag{6.3}$$

where $o_j^{(l)}$ is the j-th neuron in the l-th layer, $w_{i,j}$ is the (i,j)-th weight in the l-th layer, $\varphi(\cdot)$ is an activation function, r_i is the skip-out rate at the i-th iteration, and a_k is the average skip-out rate until the k-th iteration. Unlike the skip-out back-propagation that skips some weights, the skip-out feed-forward does not skip any weights for activation computation. Instead, the average skip-out rate, a_k until the k-th iteration in Equation 6.3 is applied to the activation of all neurons. Figure 6.5 shows feed-forward with skip-out.

Skip-out trains a DNN with a different number and combination of weights for each iteration. Since weights are trained with the probability of $1 - r_k$ for k-th iteration in the back-propagation, it is averaged by a_k in the feed-forward. Hence, any weight trained for a specific DNN does not dominate feed-forward. In general, an averaged feed-forward of different DNNs results in a better performance than a feed-forward based on one particular DNN (Srivastava et al., 2014).

Convergence of $\mathbf{a_k}$. The average skip-out rate a_k used for feed-forward converges after a number of training iterations. By using Equation 6.2, a_k in Equation 6.3 can be re-written as:

$$a_k \simeq \frac{1}{k}\sum_{i=1}^{k}\left(1 - \frac{e_i}{n(e_f + e_b)}\right) = \frac{1}{k}\sum_{i=1}^{k} 1 - \frac{\mu_{e_k}}{n(e_f + e_b)} \tag{6.4}$$

where $\mu_{e_k} = \frac{1}{k}\sum_{i=1}^{k} e_i$ is the mean of e_i. Since n, e_f, and e_b are constants and μ_{e_k} tends to converge to a constant as $k \to \infty$, a_k also converges. If we consider e_i as an independent random variable, its distribution tends toward a normal distribution as $e_i \sim \mathcal{N}(\mu_{e_k}, \sigma_{e_k})$ where μ_{e_k} is mean, and σ_{e_k} is variance regardless of its original distribution as $i \to \infty$ based on the *Central Limit theorem* (ROUAUD, 2012).

6.4 Numerical Acceleration

In this section, we present an underlying numerical acceleration of Neuro.ZERO called *Adaptive-Scale Fixed-Point (ASFP)* arithmetic that adjusts the scaling factor of fixed-point (FP) numbers during arithmetic operations. It produces more reliable numerical results than fixed-point while being faster than floating-point operations.

6.4.1 Fixed-Point Numbers

The standard 32-bit IEEE-754 floating-point numbers are either not supported or computationally very slow in embedded systems that do not have an on-board *Floating Point Unit* (FPU) (Anderson et al., 1967). For these reasons, most embedded systems that have no hardware support for floating-point operations, use *fixed-point (FP)* arithmetic (Oberstar, 2007) which is numerically less accurate than floating-point.

Fixed-Point Representation. Given total n bit-width, a number x is represented with FP format using x_f number of fractional bits (Qx_f), i.e., $x = x_b \cdot 2^{-x_f}$ for $1 \leq f \leq n-1$ where x_b is the integer base ranging from -2^{n-1} to $2^{n-1} - 1$ for a signed number. For example, 1.625 is represented as $1664 \cdot 2^{-10}$ with $Q10$.

Increasing the scaling factor increases the range and reduces precision. On the contrary, reducing it reduces the range and increases precision. Hence, FP is a special number format having a unique shared fixed exponent with a trade-off between the range and precision. Since the scaling factor is fixed for every number, overflow and precision loss occurs in compute-intensive DNN tasks.

Given total n bit-width, a number x is represented with FP format using x_f number of fractional bits (Qx_f), i.e., $x = x_b 2^{-x_f}$ for $1 \leq x_f \leq n-1$ where x_b is the integer base ranging from

-2^{n-1} to $2^{n-1} - 1$ for a signed number, e.g., 1.625 is represented as $1664 \cdot 2^{-10}$ with $Q10$. Increasing the scaling factor increases the range and reduces precision. On the contrary, reducing it reduces the range and increases precision. Hence, FP is a number format having a unique shared fixed exponent with a trade-off between the range and precision. Since the scaling factor is fixed for every number, overflow and precision loss occurs in compute-intensive DNN tasks.

6.4.2 Adaptive-Scale Fixed-Point Arithmetic

To overcome the limitations of fixed-point (FP), we propose adaptive-scale fixed-point (ASFP) numbers that adjust the scaling factor when performing the four fundamental arithmetic operations. An ASFP-based DNN can be trained with significantly less error by mitigating the overflow and precision loss problem of FP. Here, we describe adaptive-scale Multiply-Accumulate (MAC) operation that is frequently performed in DNNs. To understand it, the two parts of MAC, i.e., addition and multiplication are first discussed.

ASFP Addition. Addition of two FP number $x = x_b 2^{-x_f}$ and $y = y_b 2^{-y_f}$ $(x_f \geq y_f)$ given total n bit-width is given by:

$$x + y = x_b 2^{-x_f} + y_b 2^{-y_f} = (x_b 2^{y_f - x_f} + y_b) 2^{-k} 2^{-(y_f - k)} \tag{6.5}$$

where $(x_b 2^{y_f - x_f} + y_b) 2^{-k}$ is new integer base and $(y_f - k)$ is new number of fractional bits for the addition result. If $(x_b 2^{y_f - x_f} + y_b) 2^{-k}$ does not fit into the maximum integer base range between -2^{n-1} to $2^{n-1} - 1$, the result will overflow and end up being inaccurate. On the other hand, if it is too small, it will not overflow but end up being too coarse with relatively small fractional bits, which scarifies its precision. Hence, to provide the most fine-grained precision without overflow, new integer base $|x_b 2^{y_f - x_f} + y_b| 2^{-k}$ needs to be maximized by finding minimum k such that:

$$k \geq \lceil \log_2 |x_b 2^{y_f - x_f} + y_b| \rceil - (n-1); \text{ and, } \; k \geq y_f - (n-1) \tag{6.6}$$

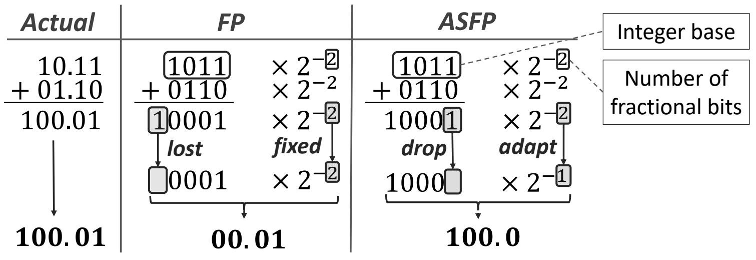


Figure 6.6: For addition of two binary number 10.11 ($Q2$) and 01.10 ($Q2$), FP produces 00.01 using the fixed scaling factor (-2). On the other hand, ASFP produces 100.0 by adapting the integer base (1000) and the scaling factor (-1) which is closer to the actual result 100.01.

$\lceil \log_2(\cdot) \rceil$ is computed by finding the most significant bit (MSB). It is efficiently obtained either using bit-shifting operations, which is extremely fast or using a near constant-time algorithm such as De Bruijn sequence (de Bruijn, 1975). As an example, adaptive-scale addition of two binary FP number 10.11 ($Q2$) and 01.10 ($Q2$) with 4 bit-widths is given in Figure 6.6, which shows ASFP produces more accurate result than FP by preventing overflow.

ASFP Multiplication. Multiplication of two FP number $x = x_b 2^{-x_f}$ and $y = y_b 2^{-y_f}$ given total n bit-width is given by:

$$xy = x_b 2^{-x_f} \cdot y_b 2^{-y_f} = x_b y_b 2^{-k} 2^{-(x_f + y_f - k)} \tag{6.7}$$

where $x_b y_b 2^{-k}$ is new integer base and $(x_f + y_f - k)$ is new number of fractional bits for the multiplication result. The integer base and scaling factor of the multiplication result are adjusted by finding minimum k such that:

$$k \geq \lceil \log_2 |x_b| \rceil + \lceil \log_2 |y_b| \rceil - (n-1); \text{ and, } k \geq x_f + y_f - (n-1) \tag{6.8}$$

Same as the addition, the calculation of $\lceil \log_2(\cdot) \rceil$ and k can be efficiently performed. Once a multiplication is performed, the log value of the multiplication result does not need to be computed any more since it stays as $(n-1)$ from then on, which results in the even faster computation for future multiplications.

ASFP MAC. By combining the adaptive-scale addition and multiplication, the MAC computation is effectively performed with numerical accuracy similar to floating-point, which provides a more reliable result than FP. Also, it is computationally more efficient than floating-point since its basic format is based on FP.

Given two vectors, it first performs element-wise multiplication and updates the scaling factor and the integer base for each multiplication result using Equation 6.8. After all the results are added up, the final summation is obtained by finding the best integer base and the scaling factor based on Equation 6.6. By providing a unified MAC operation, its computational efficiency is further improved when compared to performing individual multiplication and addition.

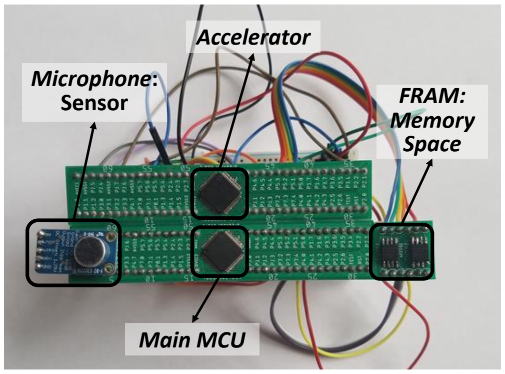
Algorithm 2 describes adaptive-scale MAC computation using ASFP multiplication and addition.

Algorithm 2: Adaptive-Scale MAC operation

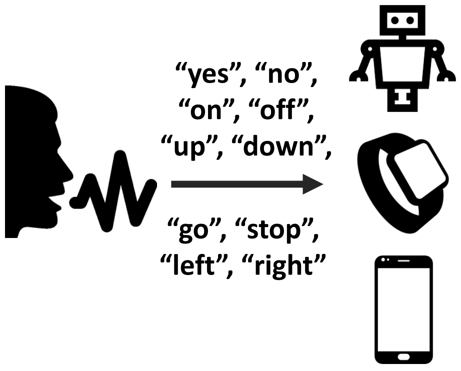
```
Input: Vector $\mathbf{x}$ and $\mathbf{y}$ with length $l$, bit-width $n$
Output: $z_b$, $z_f$ for $\mathbf{x^T y} = z = z_b \cdot 2^{-z_f}$
1  $z_b := 0, z_f := 0, k = 0$, vector $\mathbf{m}$ with length $l$;
2  for $i \leftarrow 1$ to $l$ do
3      $k := x[i]_f + y[i]_f$, $l_x$, $l_y$ := MSB index of $|x[i]_b|$, $|y[i]_b|$;
4      $k := k - \max(k - (l_x + l_y), 0)$;
5      $m[i]_b := x[i]_b y[i]_b >> k$, $m[i]_f := x[i]_f + y[i]_f - k + n - 1$;
6  end
7  $z_f := k := \min(m[1]_f$ to $m[l]_f)$;
8  for $i \leftarrow 1$ to $l$ do
9      $z_b := z_b + (m[i]_b >> (m[i]_f - z_f))$;
10 end
11 $l_z$ := MSB index of $|z_b|$, $z_b = z_b >> \max(l_z - (n-1), 0)$;
12 $k := k - \max(l_z - k, 0)$;
13 $z_f := z_f - k + n - 1$, $z_b := z_b << (n - 1 - k)$;
14 return $z_b$, $z_f$
```

6.5 Implementation

The Neuro.ZERO platform consists of two MCUs, memory space, sensors, and energy storage, which is shown in Figure 6.7a.



(a) Neuro.ZERO prototype platform



(b) Voice command recognition

Figure 6.7: Neuro.ZERO hardware platform: a custom-built dual-MCU prototype for zero-energy acceleration.

The Main MCU and Accelerator. The Neuro.ZERO platform has two microcontrollers (MSP 430FR5994 (TexasInstruments, 2018)), serving as the main MCU and the accelerator, which can operate with low power (118μA–1.8mA) supplied from an energy harvester. Two power connectors dedicated to each MCU allow separate power supplies, i.e., stable power (battery) and energy-harvesting power (batteryless).

Memory Space. A memory module is connected to both the main MCU and the accelerator. It works as a common data storage for the shared data, e.g., intermediary data or sensor readings. A FRAM (Buck, 1952) is chosen to be placed between two MCUs since it is a nonvolatile memory performing read/write operation in nanoseconds with high energy efficiency (Cypress, 2017). These attributes of FRAM minimize data sharing overhead between the main MCU and the accelerator.

Sensors. Sensors are connected to both the main MCU and the accelerator so that the data is accessible by both without any lag. They are powered from the battery for reliable and timely data collection. They are connected through the pin-headers on the below surface, e.g., we connect a camera and microphone for the traffic sign and voice command recognizer, respectively.

Energy Storage. A capacitor charged by an energy harvester works as the energy buffer for the accelerator. When the energy level of the capacitor exceeds the required energy level for acceleration, the system gets accelerated. The amount of energy needed for acceleration is statistically obtained from multiple energy measurements.

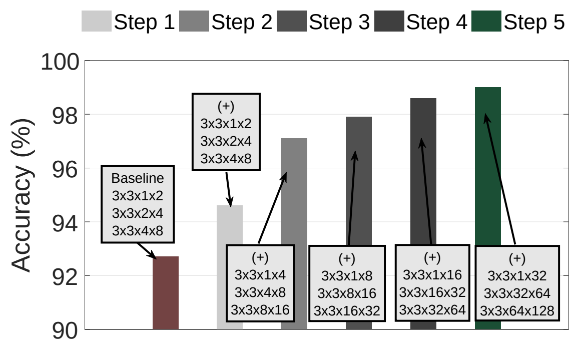
6.6 Algorithm Evaluation

Prior to describing the performance of Neuro.ZERO in real-world scenarios (Section 6.7), we conduct dataset-driven experiments to evaluate the two core algorithms of Neuro.ZERO, i.e., energy-aware acceleration (step-up inference and skip-out training) and adaptive-scale fixed-point (ASFP). The evaluation of the step-up and skip-out algorithms is conducted on a GPU machine (GTX 1080 Ti) using four datasets, i.e., MNIST (LeCun et al., 1998), CIFAR-10 (Krizhevsky et al., 2009), SVHN (Netzer et al., 2011), and Fashion MNIST (Xiao et al., 2017). We use a variation of LeNet architecture (LeCun et al., 1998) as the baseline DNN, which is also used later in the traffic sign recognizer (Section 6.7.1). The performance of ASFP is evaluated on an MCU.

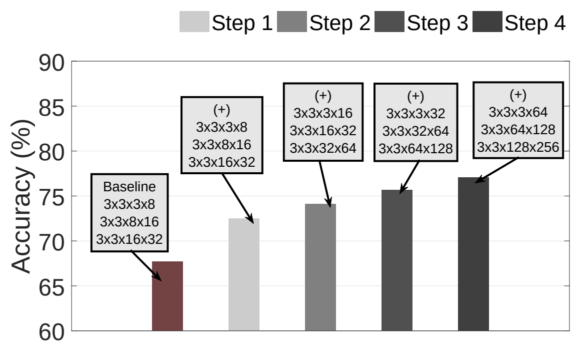
6.6.1 Energy-Aware Acceleration

Evaluation of Step-up Inference. To evaluate the effectiveness of the step-up inference, we measure the inference accuracy of the extended inference mode by varying the *step* from step 1 to step 5 for each of the four datasets. For training, we use a learning rate of 10^{-3} with Adam optimizer (Kingma and Ba, 2014), L2 regularization parameter of 10^{-6}, and a mini-batch size of 96. A separate test dataset (different from the training data) is used to evaluate the accuracy of each network. The test accuracy, along with the size of the DNN and the number of CNN filters for each step are shown in Figure 6.8.

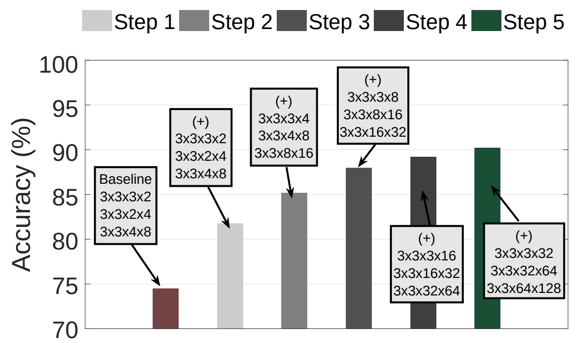
For all datasets, the accuracy increases as steps are increased, i.e., from 92.7% to 99.0%, 67.7% to 77.1%, 74.5% to 90.2%, and 91.7% to 98.9% for MNIST, CIFAR-10, SVHN, and Fashion MNIST dataset, respectively. However, the increment in accuracy is relatively smaller as the network grows. For example, the delta in accuracy for SVHN dataset is initially 7% from step 1 to step 2, but later it drops to 1% from step 4 to step 5. Since the gain in accuracy tends to



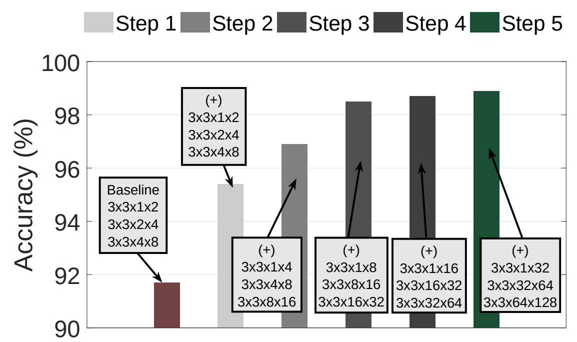
(a) MNIST



(b) CIFAR-10



(c) SVHN



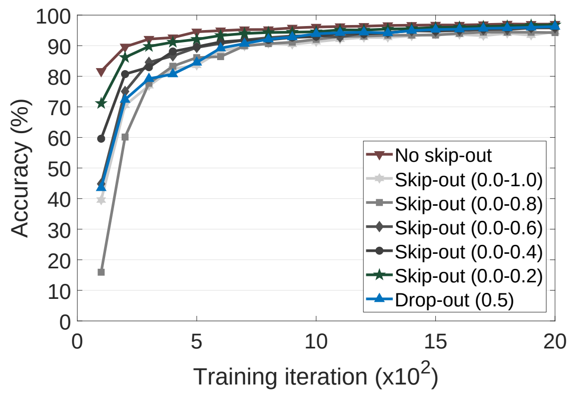
(d) Fashion MNIST

Figure 6.8: Performance of step-up inference for different *steps*.

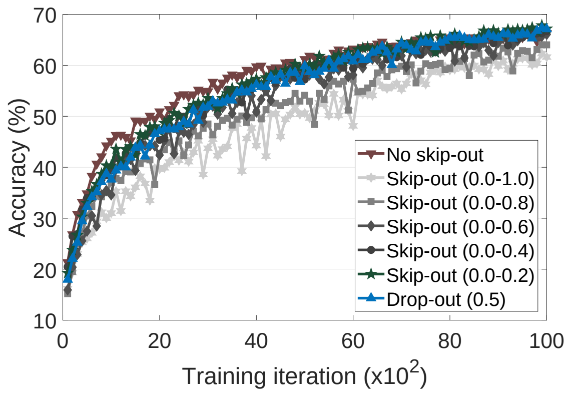
maximize during the first few steps of the step-up algorithm, executing extended inference for smaller *steps* is an effective strategy to improve the inference performance.

Evaluation of Skip-out Training. We compare the inference accuracy of the skip-out algorithm running at different skip-out rates against two baseline solutions: DNNs that do not implement skip-out (no skip-out) and thus it is expected to set the upper limit for skip-out; and DNNs that implement drop-out (sets 50% weights to zero). These DNNs are trained and tested on different, non-overlapping subsets of the dataset. Figure 6.9 shows how the accuracy (evaluated on the test dataset) varies as the number of training iteration (on the training dataset) is increased.

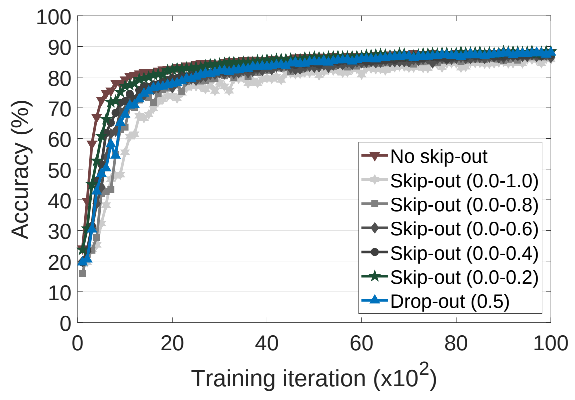
We observe that for every dataset, the accuracy of skip-out converges to no skip-out. For instance, a skip-out rate between 0.0–0.4 results in a similar accuracy to no skip-out with a negligible (0.4%-0.9%) loss in accuracy for any dataset. Furthermore, skip-out yields a similar or higher accuracy to drop-out given similar skip-out rates. In general, the performance of skip-out depends on its rate; as the rate gets closer to zero, its accuracy gets closer to no skip-out.



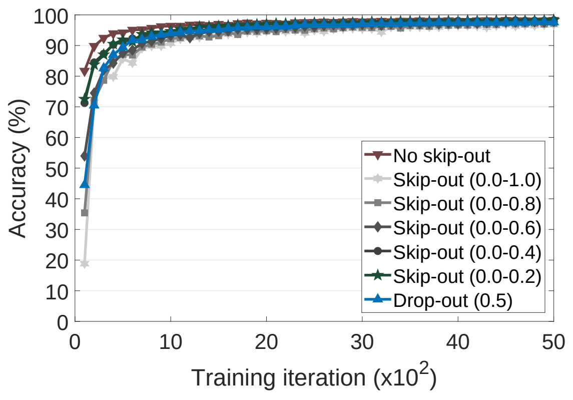
(a) MNIST



(b) CIFAR-10



(c) SVHN



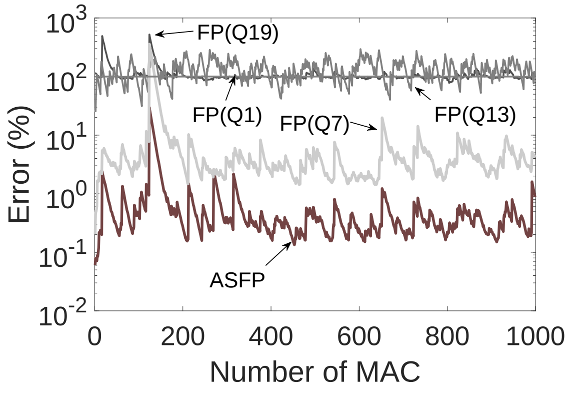
(d) Fashion MNIST

Figure 6.9: Performance of skip-out at different skip-out rates.

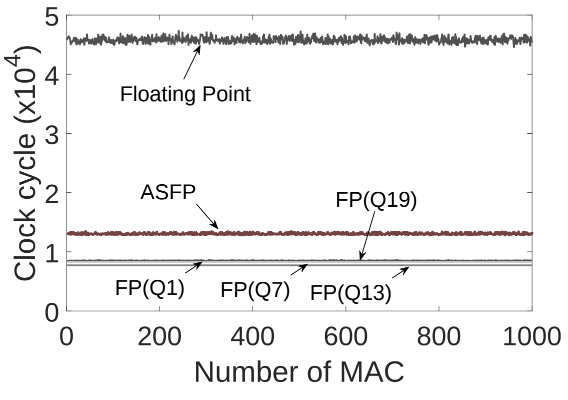
We also observe that skip-out requires a shorter training time to achieve comparable accuracy to no skip-out (not shown in the figure). For instance, skip-out (0.0-0.6) reaches 80% of accuracy about 100 iterations earlier than no skip-out. This is because the number of weights trained at each iteration in skip-out algorithm is flexibly changed based on the energy, which is usually much smaller than the total number of weights. Although the gain in accuracy after each training iteration in skip-out is generally smaller than no skip-out, larger training iterations, given the same time, compensates for the sluggish increase in accuracy, and sometimes it slightly increases the overall accuracy. Skip-out saves training time and energy consumption and guarantees the completion of an iteration regardless of the energy level. This incremental and quick pace of training is more suitable for intermittent online learning.

6.6.2 Numerical Acceleration

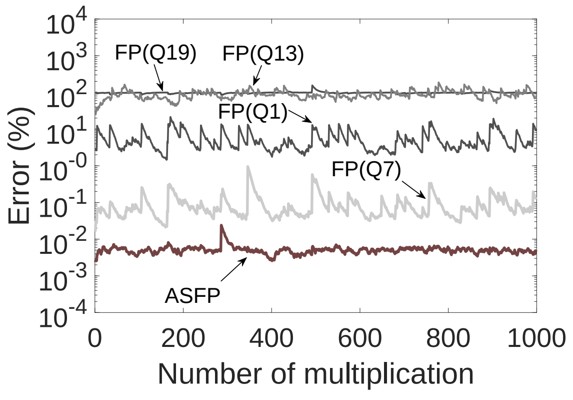
To evaluate the effectiveness of adaptive-scale fixed-point (ASFP), its MAC operation error and execution time are compared against fixed-point (FP). Figure 6.10a shows the MAC operation errors of ASFP, and four FP formats ($Q19$, $Q17$, $Q13$, $Q1$) for two randomly generated 64×1 vectors having 20 bit-widths. Here, Qx denotes that x number of bits are used to represent the fractional part. The error is calculated as $|(f - x)/f| \times 100$, where f is MAC result of floating-point (32bit) and x is the MAC result of ASFP or FP. As shown in the figure, ASFP provided average 0.71% error, which is ten times less than the best-performing FP ($Q7$, 7.32%). The errors of other FPs are numerically intolerable (more than 100%). Although the execution time (measured in clock cycles) of ASFP is 1.5 times slower than fixed-point, it is 3.4 times faster than floating-point with only 0.71% numerical difference, which is shown in Figure 6.10b.



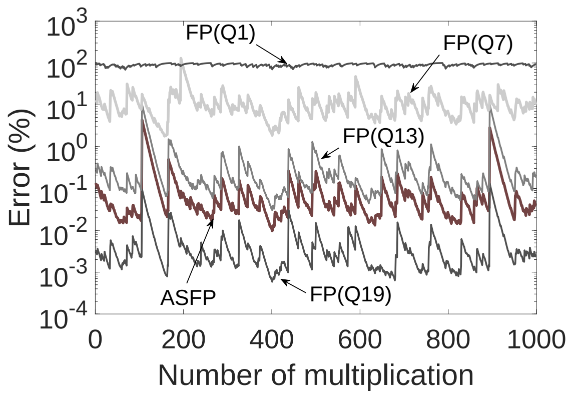
(a) MAC error



(b) MAC execution time



(c) Overflow error



(d) Precision error

Figure 6.10: The MAC error, MAC execution time, overflow error, and precision error of ASFP and four FPs (Q1, Q7, Q13, and Q19).

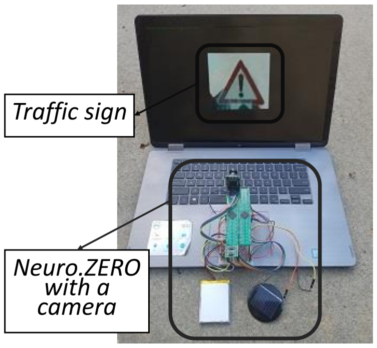
To investigate the results further, we measure the overflow and precision errors separately. The overflow error is measured by multiplying two random numbers ranging from -128 to 128 and the precision error is measured by the multiplication of two random numbers between -2 and 2. Among FPs, Figure 6.10c and 6.10d show that $Q7$ yields the smallest error of 0.009% in the overflow test, while $Q19$ achieves the smallest error of 0.005% in the precision test. The overflow and precision errors of ASFP are 0.005% and 0.123%, respectively. It demonstrates that unlike ASFP, a single fixed-point format can only provide either a small overflow error ($Q7$) or a small precision error ($Q19$), but not both. Hence, ASFP achieves better numerical correctness regarding both overflow and precision at the same time, which a fixed-point format cannot.

6.7 Application Evaluation

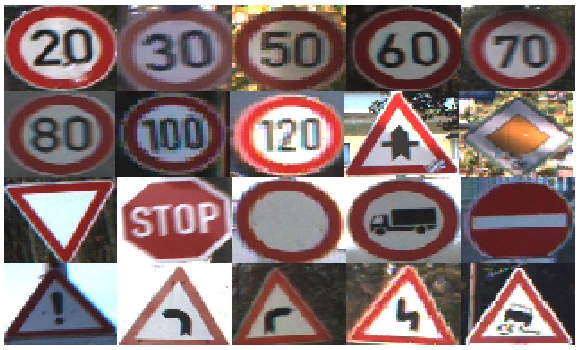
6.7.1 Traffic Sign Recognizer

We implement a traffic sign recognizer, which uses a camera to capture and classifies 43 different types of traffic signs, as shown in Figure 6.11b. The system is powered by a 5V@40mA solar energy harvester. The camera first takes a 64×48 image with RGB565. The MCU converts the image into a grayscale image (32×32) and passes it to the DNN. The baseline DNN running on the main MCU consists of seven layers including the input and the output layers: 32×32×1 (input), 3×3×1×2 (Conv), 3×3×2×4 (Conv), 3×3×4×8 (Conv), 64 (FC), 128 (FC), 43 (output), which is a variant of the LeNet architecture (LeCun et al., 1998) with an additional conv layer. Table 6.3 describes the network architecture during acceleration. We use ASFP with 16 bit-width for all numerical operations.

Performance of the Accelerator. We evaluate the performance of four acceleration modes of Neuro.ZERO. We take photos of traffic signs from the GTSRB dataset (Stallkamp et al., 2011b) using the setup shown in Figure 6.11a. We capture 39,209 training images and 12,630 test images from 43 classes using a camera sensor connected to Neuro.ZERO as the images appear on the screen of a laptop. We also evaluate the performance of the traffic recognizer using the original



(a) Traffic sign photoshoot



(b) Example images taken by the camera

Figure 6.11: Traffic sign recognizer: (a) Traffic signs are captured using a camera. (b) Examples of images taken.

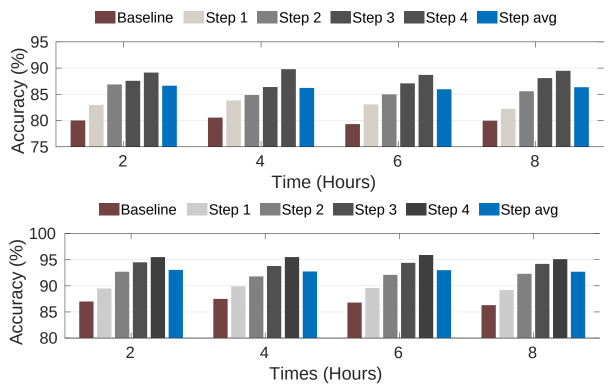
	Main MCU	**Accelerator**
Extended	baseline[1]	step 1: 3×3×1×2, 3×3×2×1, 3×3×1×2 step 2: 3×3×1×2, 3×3×2×2, 3×3×2×4 step 3: 3×3×1×2, 3×3×2×3, 3×3×3×6 step 4: 3×3×1×2, 3×3×2×4, 3×3×4×8
Expedited	step1: baseline[2] step2: baseline[3]	step 1: 3×3×1×2, 3×3×2×1, 3×3×1×8 step 2: 3×3×1×2, 3×3×2×2, 3×3×2×8
Ensembled	baseline	32×32×1, 64, 128, 128, 64, 43 (FC DNN)
Latent Train	baseline	baseline with skip-out rate (0.0–0.4)

* Baseline[1]: 32×32×1, 3×3×1×2, 3×3×2×4, 3×3×4×8, 96, 192, 43
* Baseline[2]: 32×32×1, 3×3×1×2, 3×3×2×3, 3×3×3×8, 64, 128, 43
* Baseline[3]: 32×32×1, 3×3×1×2, 3×3×2×2, 3×3×2×8, 64, 128, 43

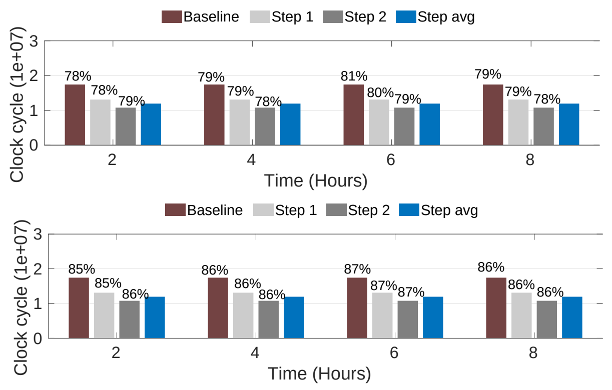
Table 6.3: The DNN architecture of the traffic sign recognizer

traffic sign images from the GTSRB dataset as the input and compare it to the performance of the camera-taken images.

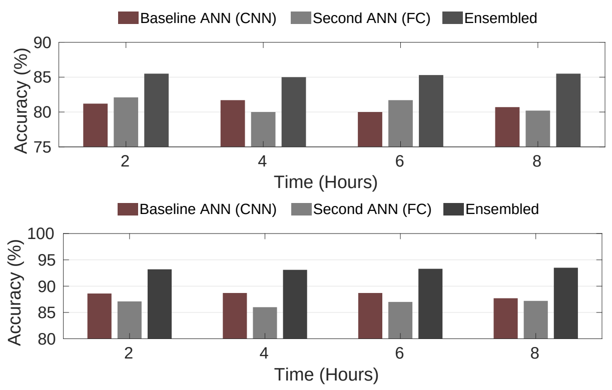
Figure 6.12a shows the recognition accuracy of the extended inference with four steps of incremental extension. Every two hours, a different set of 3,000 traffic signs (43 classes) is classified by the baseline DNN as well as the four steps to measure the accuracy. It shows that higher accuracy is achieved with further steps providing more extension of DNN. For instance, the baseline accuracy is improved from 80% to 83%, 86%, 87%, and 88% on average by each step with the camera-taken traffic sign images. Figure 6.12b shows the execution time of the expedited inference measured by clock cycle of the main MCU and the accuracy given two steps of incremental offloading. Compared to the baseline DNN, the execution time is decreased by 25% and 38%, accelerating the execution speed by 1.3× and 1.6× for step one and two, respectively. Both



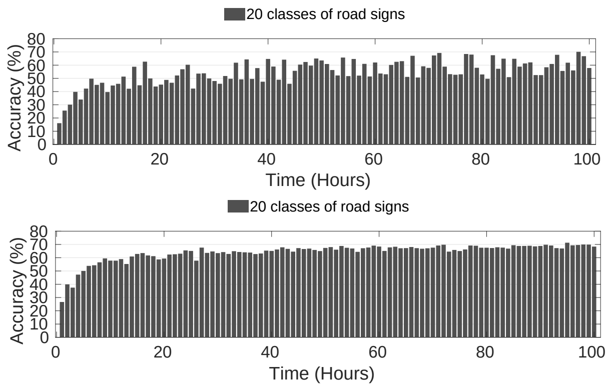
(a) Extended inference with the camera-taken images (up) and the original GTSRB images (down)



(b) Expedited inference with the camera-taken images (up) and the original GTSRB images (down)



(c) Ensembled inference with the camera-taken images (up) and the original GTSRB images (down)

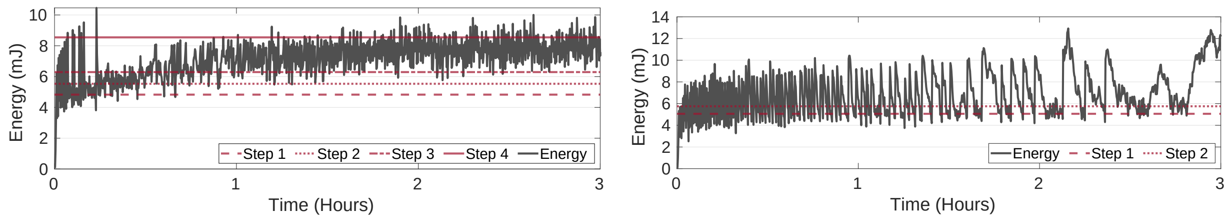


(d) Latent training with the camera-taken images (up) and the original GTSRB images (down)

Figure 6.12: The inference accuracy of the traffic sign recognizer for all four modes of acceleration. Results are shown for both camera-taken images as well as the original GTSRB images.

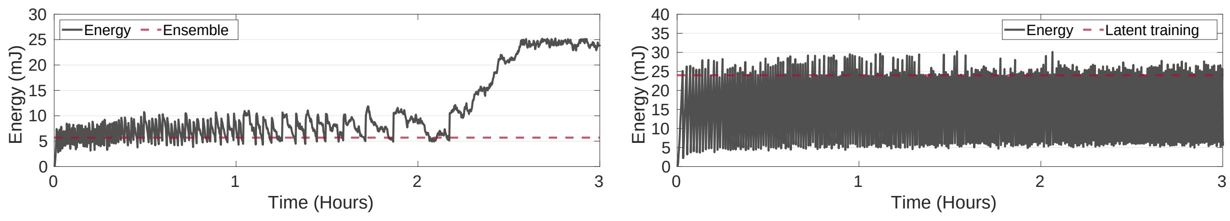
the camera-taken and original images experience only 1% of accuracy degradation by the maximum for the speed acceleration. Figure 6.12c shows the recognition accuracy of the ensembled inference with the second DNN consisting of six FC layers. The output of the two DNNs in the ensemble are combined using a fully-connected layer as done in (Mohammadi and Das, 2016). By ensembling the FC DNN, the accuracy is improved from 80% to 85% and from 87% to 93% on average for the camera-taken and original GTSRB images, respectively.

Figure 6.12d shows the training accuracy over time for 20 classes of traffic signs performed by the latent training on the accelerator. Each single training example is trained online using SGD (Stochastic Gradient Descent (Robbins and Monro, 1985; Kiefer and Wolfowitz, 1952)) with the momentum algorithm (Rumelhart et al., 1988). As shown in the figure, the latent training



(a) Extended inference

(b) Expedited inference



(c) Ensembled inference

(d) Latent training

Figure 6.13: Energy level of the capacitor during the execution of traffic sign recognizer.

keeps improving the accuracy over time up to 65% and 70% for the camera-taken and original GTSRB images, respectively. However, their accuracy is about 15% lower than the offline-trained DNNs (80% and 85%) on average since it uses SGD and ASFP instead of mini-batch and floating-point which usually provide better training performance.

Execution Pattern of the Accelerator. We evaluate the execution pattern of four accelerations regarding the available energy of the accelerator. We measured the run-time energy level of the capacitor charged from a solar-harvesting panel (5V@40mA) for three hours (9 am - 12 pm) while executing each mode of acceleration every ten seconds which consume the energy in the capacitor.

Figure 6.13 shows the remaining energy level of the capacitor (harvesting minus consuming) over time and the amount of energy required by each step of four accelerations, i.e., the extended, expedited, ensembled inference, and the latent training. The horizontal lines on the figures indicate the minimum energy threshold required for executing the acceleration with each step. When the current energy level is above one of the thresholds, the corresponding step is executed accordingly. For instance, as shown in Figure 6.13a, the extended inference with step 3 is executed at hour one since the energy level (7.35mJ) is larger than the step 3 threshold (6.29mJ) but smaller than the step 4 threshold (8.54mJ). Unlike the extended and expedited inference, the ensemble inference has only one energy threshold since the second DNN running on the accelerator is executed not in an energy-aware manner. Without the step-up inference algorithm, it is either executed or not. The latent train also has one energy threshold (train or not), but it runs with the skip-out that provides better utilization of energy. When the energy level is above the training threshold, it spends all the energy by training a portion of DNN proportionate to the current energy level.

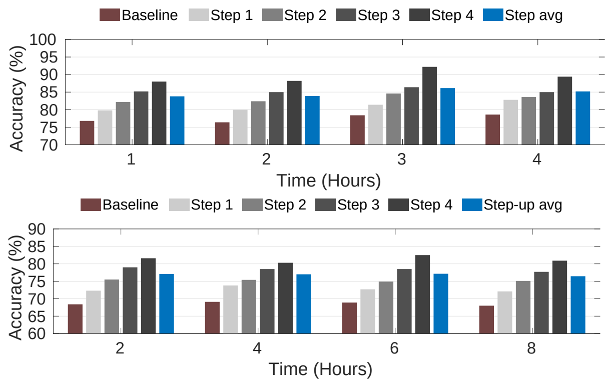
The execution pattern of accelerations depends on the available energy (harvesting) and the energy required for acceleration (consuming). Table 6.4 shows the execution pattern of accelerations with each step out of total 1,080 executions. The baseline column indicates the standalone execution of main MCU (no acceleration) whereas the rests indicate the step-up accelerations on the accelerator. For the ensembled inference and latent training, we put their accelerations (ensembling/training) in the step 1 column. The execution patterns are different from each other since each acceleration with a different step consumes a different amount of energy.

Mode	Baseline	Step 1	Step 2	Step 3	Step 4
Extended	42 (3.8%)	56 (5.1%)	140 (12.9%)	729 (67.5%)	113 (10.4%)
Expedited	195 (18.0%)	169 (15.6%)	716 (66.2%)	-	-
Ensembled	172 (15.9%)	908 (84.0%)	-	-	-
Latent Training	862 (79.8%)	218 (20.1%)	-	-	-

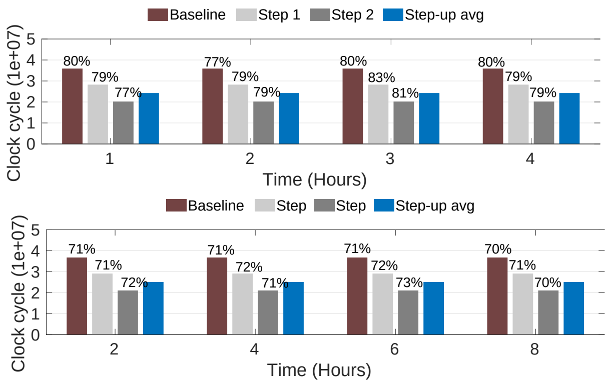
Table 6.4: The execution pattern of accelerations (traffic sign)

6.7.2 Voice Command Recognizer

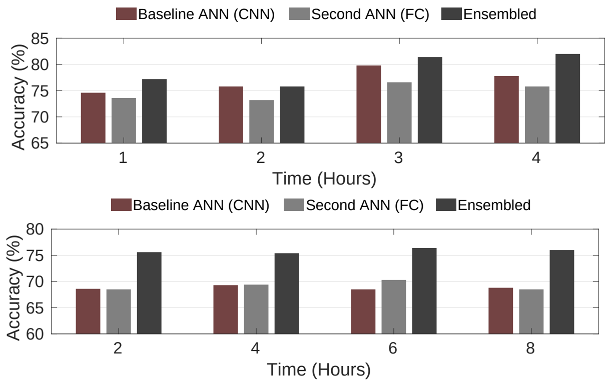
We implement a limited-vocabulary speech recognition system that recognizes ten voice commands sensed through a microphone (Figure 6.7): {yes, no, on, off, up, down, go, stop, left, right} by using an RF energy harvester (Powercast, 2016a,b). To generate input for the DNN, the microphone first samples voice data at 8kHz. Then, it is divided into small frames consisting of 256 samples having an overlap of 128 samples between two frames. Frequency information is obtained for each frame using FFT with the help of the DSP module (TexasInstruments, 2018) in the MCU. Finally, MFCCs are generated as input data for the DNN by filling Mel-filter banks. The baseline DNN consists of total six layers including input and output: $61\times13\times1$ (input),



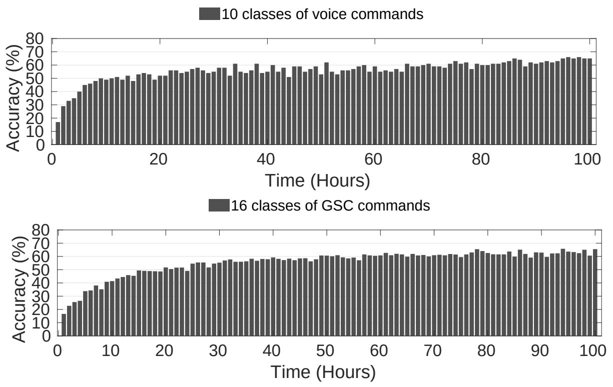
(a) Extended inference with the microphone-captured words (up) and the GSC dataset (down)



(b) Expedited inference with the microphone-captured words (up) and the GSC dataset (down)



(c) Ensembled inference with the microphone-captured words (up) and the GSC dataset (down)

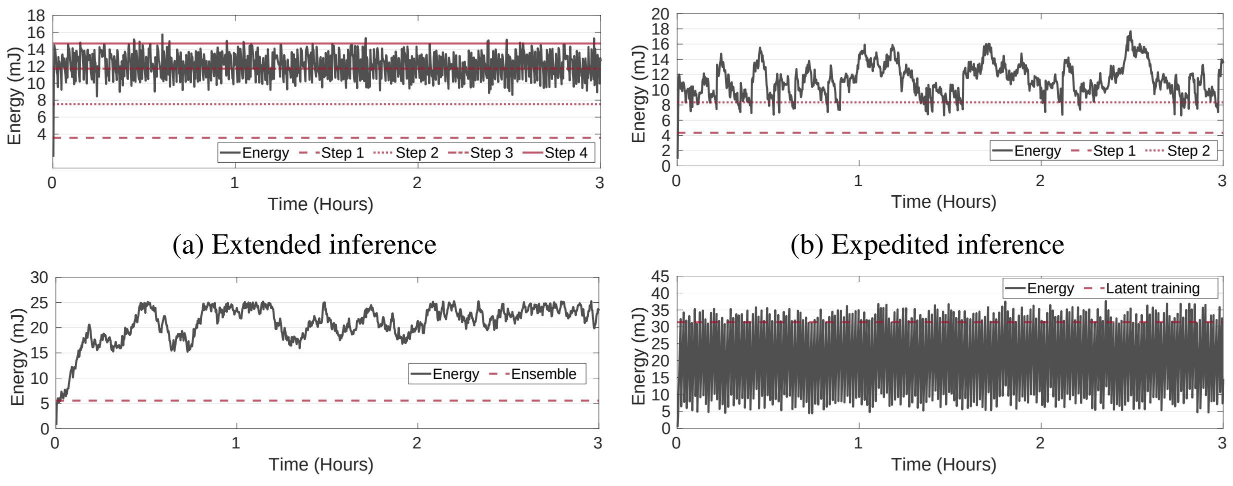


(d) Latent training with the microphone-captured words (up) and the GSC dataset (down)

Figure 6.14: The inference accuracy of the voice command recognizer for all four modes of acceleration. Results are shown for both microphone-captured utterances as well as the GSC dataset.

12×6×1×4 (Conv), 6×3×4×8 (Conv), 64 (FC), 96 (FC), 10 (output), which is based on the small-footprint keyword spotting architecture (Sainath and Parada, 2015). Table 6.5 describes the detailed network architecture for acceleration. We apply the proposed adaptive-scale fixed point (ASFP) with 16 bit-width for all numerical operations.

Performance of the Accelerator. We evaluate the performance of four accelerations by collecting voice commands from four people. In total, 10,000 commands (8,000 for train and 2,000 for test) from ten-word classes were captured through the microphone on our voice command recognizer. We also evaluate the performance with GSC dataset (Google Speech Command) (Warden,



(a) Extended inference
(b) Expedited inference
(c) Ensembled inference
(d) Latent training

Figure 6.15: Energy level of the capacitor during voice command recognition.

Mode	Main MCU	Accelerator
Extended	baseline[1]	step 1: 12×6×1×1, 6×3×1×2 step 2: 12×6×1×2, 6×3×2×4 step 3: 12×6×1×3, 6×3×3×6 step 4: 12×6×1×4, 6×3×4×8
Expedited	step1: baseline[2] step2: baseline[3]	step 1: 12×6×1×1, 6×3×1×8 step 2: 12×6×1×2, 6×3×2×8
Ensembled	baseline	61×13×1, 64, 128, 128, 64, 10 (FC DNN)
Latent Training	baseline	baseline with skip-out rate (0.0–0.4)

* Baseline[1]: 61×13×1, 12×6×1×4, 6×3×4×8, 96, 144, 10
* Baseline[2]: 61×13×1, 12×6×1×3, 6×3×3×8, 64, 96, 10
* Baseline[3]: 61×13×1, 12×6×1×2, 6×3×2×8, 64, 96, 10

Table 6.5: The DNN architecture of the voice command recognizer

2018) (84,843 training words in 35 classes and 11,005 test words) with the same experimental setup and compare it with the microphone-captured commands.

Figure 6.14a shows the recognition accuracy of the extended inference with four steps of incremental extension. For the microphone-captured commands, a different set of 500 commands (10 classes) is classified by the baseline DNN as well as the four steps to measure the accuracy every hour. For GSC dataset, a different set of 2,500 commands (35 classes) is classified every two hours. For both datasets, the accuracy is improved along with the steps from 76% to 92% (microphone) and from 68% to 77% (GSC) by the maximum. Figure 6.14b shows the execution time of the expedited inference measured by clock cycle of the main MCU and the accuracy given two steps of incremental offloading. Compared to the baseline DNN, the execution time is decreased by 21% and 43%, accelerating the execution speed by 1.2× and 1.7× for step one and two, respectively. Both the microphone-captured and GSC commands experience only 1.3% of accuracy degradation for the speed acceleration. Figure 6.14c shows the recognition accuracy of the ensembled inference with the second DNN consisting of six FC layers. By ensembling, the accuracy is improved from 77% to 80% and from 68% to 75% on average for the microphone-captured and GSC commands, respectively.

Figure 6.14d shows the training accuracy over time performed by the latent training. Based on SGD and momentum algorithm, 10 and 16 classes of voice command are trained for the microphone-captured and GSC commands system, respectively. The accuracy keeps improving over time and converges to 65% and 60% for the microphone-captured and GSC commands, which are about 11% and 8% lower than the offline-trained DNNs (76% and 68%).

Execution Pattern of the Accelerator. We evaluate the execution pattern of four accelerations regarding the available energy of the accelerator. We measured the run-time energy level of the capacitor charged from an RF energy harvester (Powercast, 2016a,b) for three hours while executing each mode of acceleration every ten seconds which consume the energy in the capacitor. Figure 6.15 show the remaining energy level of the capacitor (harvesting minus consuming) over time and the amount of energy required by each step of four accelerations. For instance, as shown in Figure 6.15b, the expedited inference with step 2 is executed at hour two since the energy level (10.27mJ) is larger than the step 2 threshold (8.34mJ). Table 6.6 shows the execution pattern out of total 1,080 executions.

Mode	**Baseline**	**Step 1**	**Step 2**	**Step 3**	**Step 4**
Extended	0 (0%)	0 (0%)	448 (41.4%)	615 (56.9%)	17 (1.5%)
Expedited	0 (0%)	81 (7.5%)	999 (92.5%)	-	-
Ensembled	5 (0.4%)	1075 (99.5%)	-	-	-
Latent Training	902 (83.5%)	178 (16.4%)	-	-	-

Table 6.6: The execution pattern of accelerations (voice command)

6.7.3 Overhead of Acceleration

Although the accelerator runs only on harvested energy, the main MCU needs to process the data from the accelerator, which causes an overhead on the main MCU. We evaluate this overhead by measuring the additional power consumption and clock cycles required for acceleration, compared to standalone execution of the main MCU. Table 6.7 shows the overhead of the main MCU for four modes of acceleration for the traffic sign recognizer. The percentages indicate their relative amount compared to standalone execution of the main MCU without the accelerator. For latent training, the overhead refers to the cost of fetching a trained model from the accelerator to the main MCU. We observe that the three inferences require less than 1% extra energy and clock cycles for acceleration, while the overhead of latent training is relatively higher. This is because the amount of data moved between two MCUs are different for inference and latent training. During inference, the two MCUs exchange relatively small chunks of data for input/output and intermediate results, whereas latent training requires movement of relatively larger sized classifier models. However, the frequency of fetching models is much lower than the frequency of interaction between the two MCUs in other three modes since a model is fetched occasionally, only when it has been improved by completing a predefined number of training iterations on the accelerator.

Mode	Energy Overhead	Clock Cycle Overhead
Extended Inference	0.065 mJ (0.7%)	256×10^3 (0.9%)
Expedited Inference	0.058 mJ (0.9%)	240×10^3 (1.4%)
Ensembled Inference	0.055 mJ (0.6%)	240×10^3 (0.9%)
Latent Train (fetching)	3.4 mJ (-%)	$15,344 \times 10^3$ (-%)

Table 6.7: The overhead of the main MCU due to acceleration.

6.8 Discussion

Unpredictable Harvested Energy. Due to the unpredictable nature of harvested energy, the accelerator may not be available at desired instants. To enable timely wake-up, specially designed energy management unit, along with scheduling algorithms for energy harvesting systems (Luo and Nirjon, 2019; Chetto and El Ghor, 2019; Baknina and Ulukus, 2017; Audet et al., 2011; Moser et al., 2007) should be implemented alongside Neuro.ZERO. To meet the varying energy demands of the running application, reconfiguration energy storage (Colin et al., 2018) and/or multi-capacitor systems (Hester et al., 2015b) should be implemented to scale and/or partition harvested energy for efficient and timely use.

Caveats to On-Device Training. There are certain caveats to on-device online training on Neuro.ZERO. First, a mini-batch size of one is used in our implementation of the latent training mode of Neuro.ZERO, which might increase noise and cause abrupt changes in the training process (Masters and Luschi, 2018; Li et al., 2014a; Bengio, 2012). To mitigate this, multiple examples should be stored and trained together as a batch instead of training only one. Second, a large learning rate may never converge to an optimal solution but to a sub-optimal one (Goodfellow et al., 2016; Attoh-Okine, 1999). To handle this, the learning rate should be decayed after a number of training iterations. Alternatively, transfer learning techniques (Torrey and Shavlik, 2010; Pan and Yang, 2009) such as retraining only the last few layers as opposed to training the whole network could be employed.

Using Battery as a Backup. Besides the energy harvester, the accelerator could use a battery of its own as a backup source. Although such a design allows waking-up the accelerator in times of need, eventually, the battery will die, and the design will fall back to our current implementation of Neuro.ZERO. Another alternative design is to design a single-MCU system that has both a battery and a harvester. Although such a design increases the battery-life of the MCU, the performance-gain in DNN acceleration on a single-MCU system is never going to be as high as a multi-MCU system like Neuro.ZERO, which has more computational capacity.

6.9 Prior Work and Their Limitations

Embedded DNN Accelerator. DNN inference accelerators using FPGAs such as (Qiu et al., 2016; Wang et al., 2016; Zhang et al., 2015; Farabet et al., 2011) have been widely studied due to their high performance and reconfigurability. Also, there are some accelerators based on different platforms, e.g., embedded GPU (Han et al., 2016; Cavigelli et al., 2015) or ARM microprocessors (Gokhale et al., 2014). However, they only focus on speeding up DNN inference given a pre-trained model, unlike the proposed accelerator that enables accuracy improvement as well as training. Although some work introduced trainable accelerators, they depend on a specific platform such as NVIDIA GPU (Dundar et al., 2017) or specific hardware (Chen et al., 2014b; Kim et al., 2014; Merolla et al., 2014). None of these works utilize energy harvesters as their power source in an energy-aware manner.

DNN with Fixed-Point. Fixed-point arithmetic for DNNs has been explored in earlier works ranging from the theoretical analysis (Draghici, 2002; Holi and Hwang, 1993) to implementations (Gupta et al., 2015; Courbariaux et al., 2014). Recently, (Lin et al., 2015) showed that DNNs could be effectively trained using only fixed-point arithmetic and many approaches have been proposed to increase accuracy and efficiency. Most popular approaches are based on compression of a pre-trained model, including quantization of weights (Hubara et al., 2017; Anwar et al., 2015; Gong et al., 2014; Vanhoucke et al., 2011) and extremely low-precision (1-3 bits) (Courbariaux et al., 2015; Hwang and Sung, 2014). However, the proposed adaptive-scale fixed-point is applicable to general DNNs without requiring any compression. Although a pre-trained DNN model can be effectively reduced for fixed-point by compressing, (Rastegari et al., 2016; Lin et al., 2015) showed that training DNN models with fixed-point results in better accuracy. In accordance with these results, we train DNNs from scratch using adaptive-scale arithmetic, i.e., MAC, multiplication, and addition. Similar to our work, (Gupta et al., 2015) trained DNNs by taking a maximum integer base using stochastic and near-rounding. However, unlike ours, their scaling factor is fixed for the entire training.

Intermittent Computing. Existing work on intermittent computing address some important system-level problems, such as atomicity (Maeng et al., 2017; Colin and Lucia, 2016), consistency (Maeng et al., 2017; Colin and Lucia, 2016; Lucia and Ransford, 2015), programmability (Hester et al., 2017), timeliness (Hester et al., 2017), and energy efficiency (Colin et al., 2018;

Hester et al., 2015b; Buettner et al., 2011). Although they enable efficient code execution of general-purpose tasks on batteryless systems, none of them considers the learning aspects of a DNN such as their accuracy or training. Recently, (Gobieski et al., 2018a) implemented intermittent inference on harvested energy using a microcontroller. However, its execution of inference is not guaranteed unlike the proposed system since it entirely depends on harvested energy. For DNN training, (Nirjon, 2018) proposed layer-by-layer training approach with the concept of lifelong learning, which repeatedly trains a fixed number of weights without skipping out.

6.10 Summary

We introduce Neuro.ZERO, an intermittently-powered accelerator that draws no system energy and opportunistically accelerates the performance of a DNN based on the four modes of acceleration. To enable zero-energy acceleration, energy-aware acceleration algorithms, and adaptive-scale fixed-point are proposed. A traffic sign and a voice command recognizer are implemented, and they have been demonstrated that the inference accuracy and speed increase.

CHAPTER 7: ENERGY-AWARE INTERMITTENT LEARNING

We envision a future where batteryless embedded platforms will be an effective alternative to battery-powered systems. Being batteryless will reduce environmental hazard caused by billions of batteries containing toxic and corrosive materials that are dumped in the environment every year (Zeng et al., 2012). The prolonged life of batteryless systems will eliminate the cost and effort of recharging and replacing batteries and make IoT scalable (Gartner, Inc., 2016). In the absence of batteries, electronic devices will be lightweight and miniature. We will be able to develop batteryless implantables and wearables that monitor and control a person's health vitals throughout their entire lifetime (Mosa et al., 2017). With this vision in mind, batteryless computing platforms have been proposed in recent years.

With the emergence of batteryless computing platforms, we are now able to execute computer programs on embedded systems that do not require a dedicated energy source. These platforms are typically used in sensing applications (Yerva et al., 2012; Sudevalayam and Kulkarni, 2011; Seah et al., 2009; Kansal and Srivastava, 2003; Gorlatova et al., 2010), and their hardware architecture consists primarily of a sensor-enabled microcontroller that is powered by some form of harvested energy such as solar, RF or piezoelectric (Priya and Inman, 2009). Programs that run on these platforms follow the so-called *intermittent computing* paradigm (Maeng et al., 2017; Van Der Woude and Hicks, 2016; Xie et al., 2016; Lucia et al., 2017) where a system pauses and resumes its code execution based on the availability of harvested energy. Over the past decade, the efficiency of batteryless computing platforms has been improved by reducing their energy waste through hardware provisioning, through check-pointing (Ransford et al., 2012) to avoid restarting code execution from the beginning at each power-up (Balsamo et al., 2015), and through discarding stale sensor data (Hester et al., 2017) which are no longer useful. Despite these ad-

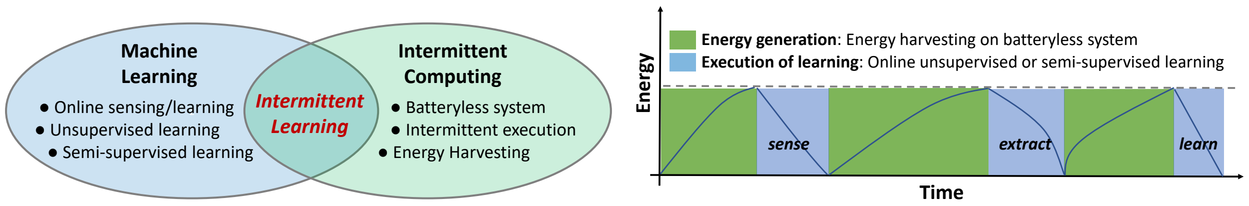

Machine Learning
• Online sensing/learning
• Unsupervised learning
• Semi-supervised learning
Intermittent Learning
Intermittent Computing
• Batteryless system
• Intermittent execution
• Energy Harvesting
Energy generation: Energy harvesting on batteryless system
Execution of learning: Online unsupervised or semi-supervised learning
Energy
sense
extract
learn
Time

Figure 7.1: An *intermittent learner* intermittently executes on-device online machine learning algorithms using harvested energy.

vancements, the capability of batteryless computing platforms has remained limited to simple sensing applications only.

In this study, we introduce the concept of *intermittent learning* (Figure 7.1), which makes energy harvested embedded systems capable of executing lightweight machine learning tasks. Their ability to run machine learning tasks inside energy harvesting microcontrollers pushes the boundary of batteryless computing as these devices are able to sense, learn, infer, and evolve over a prolonged lifetime. The proposed intermittent learning paradigm enables a true lifelong learning experience in mobile and embedded systems and advances sensor systems from being smart to smarter. Once deployed in the field, an intermittent learner classifies sensor data as well as learns from them to update the classifier at run-time—without requiring any help from any external system. Such on-device learning capability makes an intermittent learner privacy-aware, secure, autonomous, untethered, responsive, adaptive, and evolving forever.

The notion of intermittent learning is similar to the intermittent computing paradigm with the primary difference that the program that runs on the microcontroller executes a machine learning task—involving both *training* and *inferring*. Although it may appear to be that all machine learning tasks are merely pieces of codes that could very well be run on platforms that support intermittent computing, for several reasons, a machine learning task in an intermittent computing setup is quite different. The fundamental difference between a machine learning task and a typical task on a batteryless system (e.g., sensing and executing an offline-trained classifier) lies in the data and application semantics, which requires special treatment for effective learning under an extreme energy budget. Existing works on intermittent computing address important problems, such as ensuring atomicity (Maeng et al., 2017; Colin and Lucia, 2016), consistency (Maeng et al., 2017; Colin and Lucia, 2016; Lucia and Ransford, 2015), programmability (Hester et al., 2017), timeliness (Hester et al., 2017), and energy-efficiency (Colin et al., 2018; Hester et al., 2015b; Buettner et al., 2011), which enable efficient code execution of general-purpose tasks. Our work complements existing literature and specializes a batteryless system on efficient and effective on-device learning by explicitly considering the *utility of sensor data* and *the execution order of different modules* of a machine learning task.

Three key properties of intermittent learning make it unique and a harder problem to solve. First, when energy is scarce, an intermittent learning system needs to decide the best action (e.g., learn vs. infer) for that moment so that its overall learning objective (e.g., the completion of learning a desired number and types of examples) is achieved. Second, since not all training examples are equally important to learning, an intermittent learning system should smartly decide to keep or discard examples at run-time, and thus be able to eliminate a large number of unnecessary and energy-wasting training actions. Third, a system that pauses and resumes its executing based on the state of its energy harvester runs a greater risk of missing real-world events that it wants to detect or learn. When both the generation of energy and the generation of training/inferable sensor data are intermittent and uncertain, the problem of learning becomes an extremely challenging feat. None of the existing intermittent computing platforms consider these issues, and thus they are not effective in learning when we execute machine learning tasks on them.

In this chapter, we address these aforementioned challenges and propose the first intermittent learning framework for intermittently powered systems. The framework is targeted to a class of learning problems where the presence of energy implies the presence of data—which means either the cause of energy and data are the same, or they are highly correlated, or data is always available for best-effort sensing and inference (e.g., sporadic classification of air quality). Furthermore, we focus on *long-term* and *online machine learning* tasks where a batteryless system is expected to run for an extended period in time, and its learning performance is expected to improve over time. In our proposed framework, the availability of labeled data is not an absolute necessity. In other words, we study *unsupervised* (Russell and Norvig, 2016) and *semi-supervised* (Chapelle et al., 2009) machine learning problems in this chapter, although the framework can be easily extended to incorporate supervised (Russell and Norvig, 2016) and reinforcement learning (Russell and Norvig, 2016) tasks by enabling real-time feedback from the environment or humans.

We provide a programming model that allows a programmer to develop an intermittent learning application that executes correctly when a system is intermittently powered. Based on the action-based (task-based) intermittent programming model (Colin and Lucia, 2018; Yıldırım et al., 2018; Maeng et al., 2017; Hester et al., 2017; Colin and Lucia, 2016; Lucia and Ransford, 2015), the proposed framework provides application programmers with an energy pre-inspection tool that helps them split an existing application code into sub-modules called *action* that can be executed to completion with intermittently harvested energy. The user study of the intermittent learning programming model shows that the concept of action-based intermittent learning is intuitive and applicable to a variety of applications, and the intermittent learning framework provides the necessary components to write on-device machine learning programs on intermittently powered systems.

We envision a wide variety of applications where the proposed intermittent learning paradigm applies. Three such applications are implemented and evaluated in this chapter to demonstrate the efficacy of the proposed intermittent learning framework. The first one is an air quality learning system where sunlight and air-quality sensitive environmental sensors are powered by harvesting solar energy to detect an anomaly in the air quality. This batteryless learner has been monitoring, classifying, and learning air-quality indices continuously since September 2018. We have developed a webpage showing its real-time learning status[1]. The second application is an RF energy-based human presence learning system which learns to detect humans passing by it in indoor spaces from the variation in RSSI patterns. The last application is a vibration monitoring scenario (applicable to human health and machine monitoring applications) where an accelerometer-based sensing system is powered by harvesting piezoelectric energy. To demonstrate that the proposed

[1]Intermittent air quality learning system: `https://www.cs.unc.edu/~seulki/intermittent-learning/air-quality-learning.html`

framework is portable to different platforms, we have used an AVR, a PIC, and an MSP430-based microcontroller to implement these three applications, respectively. The framework is implemented in C and has been open-sourced (Embedded Intelligence Lab (UNC Chapel Hill), 2019a).

The main contributions of this chapter are the following:

• This is the first work that introduces the *intermittent learning* concept and proposes an intermittent learning framework that enables energy harvested computing platforms to perform on-device machine learning *training*.

• We define a set of *action primitives* for intermittent learners and devise an algorithm to determine a sequence of actions to achieve the desired learning objective while maximizing energy efficiency.

• We propose three learning-example selection heuristics that enable an intermittent learner to decide whether to learn or to discard examples—which increase the efficiency in learning under tight energy constraints.

• We provide a programming model and development tool of intermittent learning, which allows a programmer to implement an intermittent learning application based on the action-based intermittent execution.

• We implement and evaluate three intermittent learning applications: an air quality, a human presence, and a vibration learning system. We demonstrate that the proposed framework improves the energy efficiency of a learning task by up to 100% and cuts down the learning time by 50%.

• We have open-sourced the software framework to the community to facilitate the widespread use of the proposed intermittent learning framework. The anonymized code repository can be accessed here (Embedded Intelligence Lab (UNC Chapel Hill), 2019a).

7.1 Overview

The goal of intermittent learning is to enable efficient and effective execution of a class of machine learning tasks on embedded systems that are powered intermittently from harvested energy. Throughout the lifetime, an intermittent learner sporadically senses, infers, learns (trains), and thus evolves its classier and model parameters over time, and get better at detecting and inferring events of interest. Like existing intermittent computing systems, an intermittent learner also pauses its execution when the system runs out of energy and resumes its execution when the system has harvested enough energy to carry out its next action. However, due to the nature of the data and application semantics of a machine learning task, an intermittent learner has to do a much better job in deciding *what actions to perform* and *what data to learn*—so that it can ensure its progress toward learning and inferring events of interests, while making the best use of sporadically available harvested energy.

7.1.1 Motivation Behind Intermittent Learning

On-device machine learning on embedded systems is an emerging research area (Li et al., 2018a; Chauhan et al., 2018; Yao et al., 2017b). Batteryless systems have also joined this revolution. Recent literature on intermittent computing routinely uses *on-device inference* as one of many example applications (Li et al., 2018b; Li and Zhou, 2018; Truong et al., 2018; Gobieski et al., 2018a; Hester et al., 2017; Ransford et al., 2012). For example, (Li et al., 2018b; Li and Zhou, 2018) harvests energy from the ambient light to power up a gesture recognition system that implements *Constant False Alarm Rate* (CFAR) algorithm (Scharf and Demeure, 1991), CapBand (Truong et al., 2018) implements a *Convolutional Neural Network* (CNN) to classify hand gestures on a batteryless system that is powered by a combination of solar and RF harvesters, (Gobieski et al., 2018a) implements a deep network compression algorithm (Han et al., 2015a) to fit a *Deep Neural Network* (DNN) into a resource-constrained microcontroller (MSP430) which runs on energy harvested from RF sources. While these application-specific systems have inspired our work, we observe that these systems are capable of only making *on-device inferences* using an *offline-trained classifier*. These systems treat machine learning tasks the same way as any other computational load, and thus, they are not able to optimize the execution of machine learning-specific tasks. Furthermore, the pre-trained classifiers running on these systems are fixed and non-adaptive, which does not allow these applications to adapt automatically at run-time to improve the accuracy of the classifier.

To complement and advance the state-of-the-art of the batteryless machine learning systems, we propose the intermittent learning framework which explicitly takes into account the dynamics of a machine learning task, in order to improve the energy and learning efficiency of an intermittent learner in a systemic fashion. The fundamental difference between the proposed framework and the existing literature is that, besides *improving the efficiency of on-device inference*, the intermittent learning framework enables *on-device training* to improve the effectiveness and accuracy of the learner over time.

7.1.2 Alternatives to Intermittent Learning

An alternative to on-device learning on batteryless systems would be to sense and transmit raw or semi-processed sensor data from a batteryless system to a base station that executes the inference and/or training tasks. In fact, such *offloading* solutions were popular back in the days when *Wireless Sensor Networks* (WSNs) were deployed to collect data from the sensor nodes, only to be analyzed later on a remote base station (Shaikh and Zeadally, 2016; Akhtar and Rehmani, 2015; Shaikh and Zeadally, 2016; Lu et al., 2015). Compared to the sensor motes of those WSNs, today's microcontroller-based systems are far more advanced in terms of CPU and memory, and their energy efficiency has improved by several orders of magnitude. For instance, the latest mixed-signal microcontrollers from Texas Instruments (i.e., TI MSP430 series) comes with up to 16-bit/25 MHz CPU, 512 KB flash memory, 66 KB RAM, and 256 KB non-volatile FRAM—which are comparable to the 16-bit Intel x86 microprocessors of the early 80s which ran MS-DOS. These devices are quite capable of executing simple machine learning workloads that perform on-device classification of sensor data (Gobieski et al., 2018b). In general, there are several advantages of on-device intermittent learning over relaying data to a base station:

Data Transmission Cost and Latency. Data communication between a device and a base station introduces delays and increases energy cost per bit transmission. Using back-scatter communication (Lu et al., 2018) apparently lower the energy cost, but the dependency on an external entity and the unpredictable delay in wireless communication still remain, which we want to avoid by design.

Privacy and Security. Private and confidential data, such as health vitals from a wearable device, can be safely learned on-device – without exposing them to external entities. Security problems caused by side-channel and man-in-the-middle attacks (Aziz and Hamilton, 2009; Kügler, 2003) are avoided by design when we adopt on-device processing of sensitive data.

Precision Learning and Resource Management. Many human-in-the-loop machine learning applications running on wearable and implantable systems benefit from run-time adaptation as different persons have different preferences and different expectations from the same application. On-device learning helps a system adjust itself at run-time to satisfy each individual's need and to optimize its own resource management.

Adaptability and Lifelong Learning. Lifelong learning (Chen and Liu, 2016) is an emerging concept in robotics and autonomous systems where the vision is to create intelligent machines that learn and adapt throughout their lifetime. Intermittent learning enables true lifelong learning by liberating these devices from being stationary and connected to power sources, to mobile, ubiquitous, and autonomous.

We acknowledge that some of the pitfalls of offloading machine learning tasks to base stations can be avoided via alternative methods. For instance, on-device data encryption arguably can ensure security and privacy, backscatter techniques can reduce communication energy cost, and over-the-air code updates could make the classifier adaptive. However, each of these comes with their limitations and overheads, and none are maintenance-free. Hence, considering the autonomy and maintenance-free nature of intermittent learners, combined with the full package of benefits mentioned earlier, we opt for batteryless on-device learners as our design choice.

7.1.3 The Scope of Intermittent Learning

We limit the scope of this paper to specific types of machine learning problems and study the corresponding research challenges.

Online Unsupervised and Semi-Supervised Learning. Based on the availability and use of labeled ground-truth data, a machine learning problem can be categorized into supervised, semi-supervised, and unsupervised types (Russell and Norvig, 2016). Since batteryless computers are meant to last long and operate unattended, we exclude purely supervised learning (where labeled data is a must) from the scope of this work. Instead, we focus on the other two types, where either labeled data are unnecessary (unsupervised) or some labeled data are available for use (semi-supervised). For instance, a motion-activated intermittent learner can observe sensor readings over time and look for statistical anomalies (e.g., using an outlier detection or a cluster analysis algorithm) in its data stream. In many applications, these statistical anomalies are the ones that correspond to events of interests such as fall detection, aggressive behavior recognition, and intruder detection. Furthermore, we consider online machine learning problems where examples (i.e., a vector of sensor readings that we want to classify or learn) come one at a time, and the classifier is incrementally trained and updated as they arrive.

Selection of Training Data. In an online learning task, a learner's model parameters are updated as new training examples arrive. A typical learning algorithm takes hundreds of iterations of model updating – one iteration for each training example – before the learner attains a reasonable classification accuracy. However, in a real-world online learning scenario, a system might continue to receive too many similar examples and use them all to update the learner's model parameters. In such cases, the learner wastes a significant amount of energy and compute cycles in repeatedly learning the same example where learning only one representation example would have been sufficient. In summary, since not all training examples are equally important to learning, it is beneficial to discard examples that do not contribute to a learner's gain in accuracy.

Choice of Actions at Run-time. A machine learning task includes several sub-tasks (or, *actions*) such as sampling the sensors, assessing the utility of sensor values in learning, saving sensor values for later use, updating the classifier model upon sensing a new data point, classifying the sensor data, and sending alerts to external systems. When the system harvests enough energy to take one or more of these actions, it must determine the best action for that moment so that its overall learning objective (e.g., the completion of its learning task and/or learning a desired number and types of examples) can be fulfilled.

For instance, suppose, a system has harvested just enough energy to either update the current model parameters by training the learner with recently sampled data or to classify the new data using the current model. Based on the learner's performance of that moment, either action can be a valid choice. If the learner is under-performing, retraining is a more sensible action. On the other hand, if the learner is performing at its best, it makes more sense to do frequent classifications than training. Hence, dynamically choosing a proper action is an important aspect of intermittent learning, which is not considered by existing intermittent computing systems.

If we employ existing intermittent computing frameworks like MayFly (Hester et al., 2017) to execute machine learning tasks, the system would blindly use every incoming training example to update the model parameters and thus drain the harvested energy much faster than needed. Although it considers the *staleness* of data to increase the system lifetime, it does not help a learner as the data can be fresh, yet their utility toward an application's high-level goal can be null. Likewise, data can be stale, yet their utility in a learning algorithm can be high. Hence, we need to devise a mechanism to smartly choose or discard examples at run-time, and thus be able to eliminate a large number of unnecessary and energy-wasting training actions.

Occurrence of Sample Data and Energy Harvesting Cycles. An intermittent learner learns and infers physical world events. Occurrences of these events are, in general, unpredictable. Energy harvesting cycles also depend on physical world phenomena such as motion, sunlight, or radio signals, and thus, the time and amount of harvested energy are unpredictable as well. Hence, an intermittent learner has to learn through these dual uncertainties. We identify two cases when intermittent learning is suitable.

In some applications, the physical phenomena behind the event of interest and energy harvesting are either the same or strongly correlated. For instance, piezoelectric harvesters that generate energy from motion are used in many people-centric machine learning applications, such as vibration-related health condition monitoring, sleep motion detection, and fall detection, where the core learning task is to classify human motions. In this class of applications, data and energy are available at the same time, and they are correlated. Intermittent learning framework applies to these applications with greater certainty of learning.

There is another class of sensing and inference applications where the data are either always available, or the rate of change in data is so low that an intermittently powered system can gather sufficient data during its operating cycles. Examples include environmental monitoring applications such as detecting pollutants or gaseous anomaly in the air (e.g., excessive carbon dioxide concentration), and sound pollution monitoring. An intermittent learner in these scenarios learns and infers in a best-effort manner.

7.2 Framework

7.2.1 Intermittent Learning Framework Overview

We propose an intermittent learning framework for intermittently powered systems that want to execute an end-to-end machine learning task which involves data acquisition, learning, and inferring. Figure 7.2 shows a high-level architectural diagram of the proposed framework. The three main modules of the proposed framework corresponding to energy management, machine learning, and task planning are briefly discussed as follows:

Energy Harvester. Batteryless computing platforms consist of one or more energy harvesters such as piezoelectric, RF, or solar panels that harvest energy from various types of sources such as the sunlight, motion, RF, vibration, wind, heat, and chemical. This subsystem monitors the

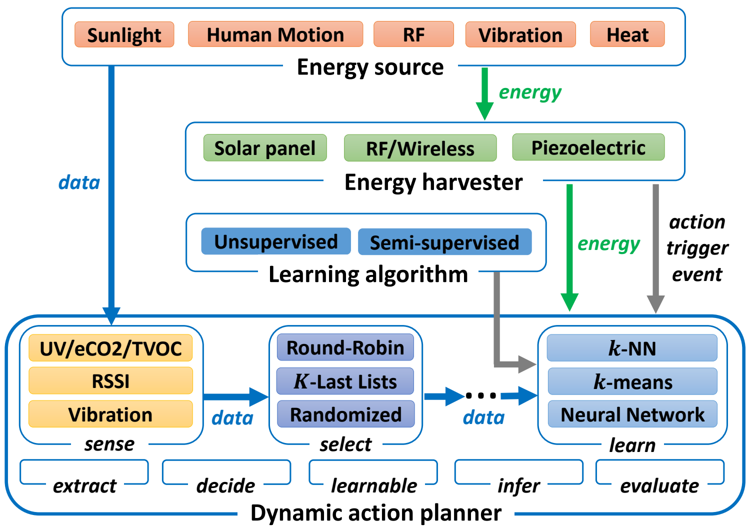


Figure 7.2: The intermittent learning framework showing energy sources, energy harvesters, learning algorithms, and a dynamic action planner.

energy generated by the energy harvester and generates an interrupt that triggers an intermittent execution of learning tasks whenever a sufficient amount of energy is generated. In certain systems, such as (Truong et al., 2018), where multiple energy harvesters are used to guarantee continuous energy supply, e.g., RF for indoors and solar for outdoors, the energy harvester subsystem takes care of selecting and switching to the preferred harvester transparently.

Library of Learning Algorithms. We have developed a library of machine learning algorithms which contains specialized implementations of commonly used unsupervised or semi-supervised algorithms for an intermittently powered system. These algorithms are split into small pieces of code so that they are suitable for executing the intermittently powered system. Currently, the library contains the implementation of three common machine learning algorithms as templates: k-nearest neighbors, k-means, and a neural network (described later in this section). While these are able to solve many practical learning problems, if a new learning algorithm needs to be implemented for an intermittent execution, a developer can follow the modular implementation of these classifiers to get inspired on how to implement a custom algorithm in an intermittent fashion.

Dynamic Action Planner. This module is the heart of the framework, which is responsible for selecting the right action at the right moment in order to advance the learning task toward achieving its desired learning objectives. It contains implementations of intermittently executable methods and algorithms to schedule actions, to select what to learn, and to evaluate the progress of an intermittent learner toward task completion. This module is described in detail in Section 7.3.

7.2.2 Action Primitives

We identify eight basic operations—which we refer to as *actions*—that an intermittent learner *may* execute in its lifetime. A complete list of actions and their brief description are presented in Table 7.1. Breaking a task into pieces is similar to existing task-based intermittent computing frameworks (Colin and Lucia, 2018; Yıldırım et al., 2018; Maeng et al., 2017; Hester et al., 2017; Colin and Lucia, 2016; Lucia and Ransford, 2015) with the difference that each action in an intermittent learning framework is associated with a semantic meaning, and the set of actions being exhaustive, we are able to optimize their execution better than a general-purpose program.

Some of these actions such as *sense*, *extract*, *learn*, and *infer* are self-explanatory. The action *decide* makes a decision to execute either a *learn* or an *infer* action based on the learning objective (desired goal states) of a learner described in Section 7.3.2. *Select* is related to choosing a suitable training example for learning. Heuristics for choosing training examples are described in Section 7.4. *Learnable* is used to enforce preconditions of a learning algorithm, e.g., clustering algorithms require a minimum number of examples so that they can form clusters. The action *evaluate* is related to the performance of the current learning model and action planning, which is described in Sections 7.3.

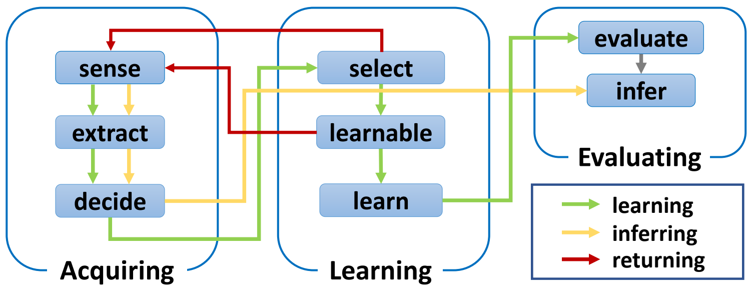


Figure 7.3: Action state diagram showing all actions and how they interact with each other.

Action	Description
sense	Sense and convert data to an example.
extract	Extract features from an example.
decide	Decide to *learn* or *infer*.
select	Determine whether a training example increases the learning performance.
learnable	Check prerequisites of a *learn* action.
learn	Execute a learning algorithm intermittently.
evaluate	Evaluate the learning performance.
infer	Make an inference using the current model.

Table 7.1: List of Action Primitives.

7.2.3 Action State Diagram

A learning task involves a subset of the actions that must be executed in a certain order. An intermittent learner has to enforce this ordering of actions when executing them at run-time. For instance, *sense* precedes all actions as this is where raw sensor readings are converted into an object, which we call an *example*, that is processed further. Similarly, *learn* or *infer* cannot be executed until we execute *extract* to extract features from an example to represent them in terms of feature vectors. Figure 7.3 shows a state diagram consisting of all eight actions along with the direction of data flow between two consecutive actions in an execution order. For ease of understanding, we categorize them into groups of acquiring, learning, and evaluating actions.

7.2.4 Intermittent Action Execution

Several of the actions in Table 7.1 are larger than what it takes to execute them at one shot by an intermittent learner. The limit comes from the size of the energy storage, i.e., the size of the capacitor that stores harvested energy, that can keep the system awake for a limited period in time. The size of the capacitor cannot be made arbitrarily large as that increases the charging time, and a longer charging time will result in excessive delays in sensing and processing of new data. In general, an intermittent learner sleeps and wakes up multiple times during the execution of an action. In this section, we describe how an action is implemented to make it suitable for intermittent execution by the proposed framework.

How to program actions for an intermittent execution? An application developer implements or overrides all or a subset of the action primitives. Corresponding to each action, there is an ordered list of functions, where each function executes a part of the action that is small enough for running to completion at one shot (i.e., without interruptions). Actions can be bypassed (not programmed) if a learning algorithm does not require them. Listing 7.1 shows an example of four user-programmed actions (*sense*, *extract*, *select* and *learn*). The *learn* action being large, it has been split into three smaller functions. An array of function pointers is implemented in the framework to facilitate an orderly execution of these parts of an action.

How to determine if an action requires splitting? Application programmers are provided with a *battery-powered* development tool that guides the action splitting process. The tool checks if each action written by the programmer can be completed using a certain amount of energy, which is also specified by the programmer. We call this *energy pre-inspection*– which is an automated tool that identifies and warns if an action requires more energy than the target. This tool helps a programmer interactively split implemented modules until they fit into the target energy. The details of action decomposition are described in Section 7.2.5.

Listing 7.1: User-programmed action example.

```
/* actions.c */
/* learning actions programmed by user */
int sense() { /* user-defined code of sense */ }
int extract() { /* user-defined code of extract */ }
int select() { /* user-defined code of select */ }
int learn_1() { /* user-defined 1st part of learn */ }
int learn_2() { /* user-defined 2nd part of learn */ }
int learn_3() { /* user-defined 3rd part of learn */ }

```

```
/* list of each action */
int (*sense_[]) () = { sense };
int (* extract_ []) () = { extract };
int (* select_ []) () = { select };
int (* learn_ []) () = { learn_1 , learn_2 , learn_3 };
```

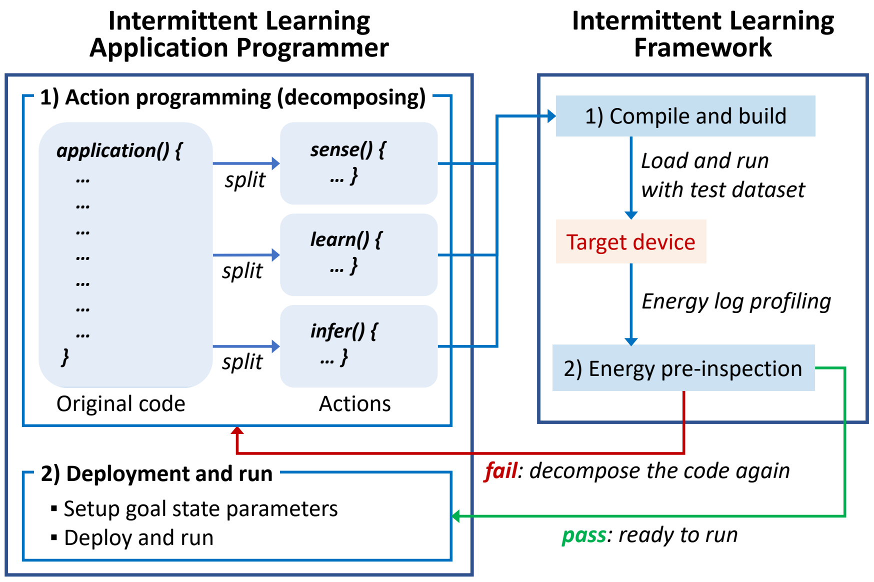
Listing 7.2: Brief workflow of intermittent learning.

```
/* intermittent_learning .c */
int (*dynamic_action_planner () ) () {
    // code for selecting next action
    return next_action ;
}
void action_trigger () { // action-trigger event ISR
    action = dynamic_action_planner () ; // next action
    action () ; // execute selected next action
}
void main() {
     init_actions () ; // executed only once
     set_interrupt () ; // setup action-trigger event
    sleep () ; // enter low-power mode
}
```

Who invokes these actions? At each wake-up, the dynamic action planner routine is called upon by the framework to select an action to execute. Listing 7.2 shows a code snippet showing three functions, including the `main()`. The function `action_trigger()` is executed at each wake up and it calls `dynamic_action_planner()` to get a pointer to an action to execute.

7.2.5 Intermittent Learning Programming Model

We provide a programming interface that allows a programmer to develop an intermittent learning application that executes correctly when a system is intermittently powered.



Intermittent Learning Application Programmer
Intermittent Learning Framework
1) Action programming (decomposing)
application() { … … … … … … … }
split
split
split
sense() { … }
learn() { … }
infer() { … }
Original code
Actions
1) Compile and build
Load and run with test dataset
Target device
Energy log profiling
2) Energy pre-inspection
fail: decompose the code again
pass: ready to run
2) Deployment and run
▪ Setup goal state parameters
▪ Deploy and run

Figure 7.4: Illustration of the programming model.

Action-based Programming. Similar to the task-based intermittent computing platforms (Colin and Lucia, 2018; Yıldırım et al., 2018; Maeng et al., 2017; Hester et al., 2017; Colin and Lucia, 2016; Lucia and Ransford, 2015), an action in the proposed intermittent learning framework is a user-defined block of code. An action, given sufficient energy to execute to completion, is guaranteed to have memory-consistency and control-flow that can be equivalently achieved with a continuously-powered execution. If power fails during an action's execution, the intermittent learning framework discards the intermediate results, and the action starts over from the beginning when power becomes available again by keeping track of the completion status of each action. Actions that consume more energy than the maximum energy budget that the hardware can support need to be decomposed into smaller actions.

Memory Model. Similar to task-based intermittent computing platforms (Maeng et al., 2017), atomicity of actions is guaranteed by maintaining two types of data — *global* data that are shared between actions and *local* data that reside in a single action. Different actions can share global data by using *action-shared variables*, which are named in the global scope and allocated in the non-volatile memory. Once an action completes writing a value to an action-shared variable, the value can be read by any action by referencing the variable name. Local data are scoped only to a single action like ordinary local variables in a function and are allocated in the volatile memory.

Application Development. Figure 7.4 depicts the development process of an intermittent learning application. To develop a new application, the programmer decomposes the application code into actions by implementing or overriding all or a subset of action primitives which are executed in the order defined by the state diagram. Once actions are implemented, *energy pre-inspection* is performed to make sure that no action consumes more energy than the hardware can support. The energy pre-inspection is performed by a custom tool that we developed by extending TI's EnergyTrace++ (Instrument, 2018), which comes with the intermittent learning framework. The tool first loads and runs the compiled binary on the battery-powered target device and measures the energy consumption of each action using EnergyTrace. In order to obtain the worst-case energy consumption of an action at reasonably high confidence, the target device runs all test cases from all datasets as the input. This is done to maximize the chances of the system to execute different control flows and data-based branches. The tool analyzes the log file of energy measurements and lists all actions that consumed more energy than the maximum allowed and prompts the programmer to split those actions further until all actions pass the test. Finally, the binary that passes the energy pre-inspection is pushed to the target batteryless device.

User Study. We conduct a user study to understand 1) whether the concept of action-based intermittent learning is intuitive and applicable to applications, and 2) the intermittent learning framework provides the necessary components to write on-device machine learning programs on intermittently powered systems.

The study involved 35 undergrad computer science students (15 female and 20 male) who were provided with an application code having three large functions (actions), and were asked to decompose and reprogram it into actions having a certain energy budget. Prior to the study, a short introduction to the concept of intermittently-powered systems and the energy constraints associated with programming such systems was provided. After the 30 minute experiment, the participants assessed the difficulty and intuitiveness of the programming model by answering the questions shown in Table 7.2. On average, the participants assessed that the difficulty level of decomposition is moderate (5.4 and 5.6), and they spent 14 minutes to split the code. We acknowledge that the study is limited due to small sample size and difficulty in testing multiple applications. Nevertheless, the user study shows that the developers with basic programming knowledge can easily program an intermittent learning application without any significant trouble.

	Avg	Min	Max
Q1. In a scale 1-to-10, how easy did you find to understand the concept of action-based intermittent learning?	5.4	2	10
Q2. In a scale 1-to-10, how easy did you find to split the code?	5.6	2	10
Q3. In a scale 1-to-10, how easy did you find to calculate the total energy consumption of the code?	3.7	1	10
Q4. How much time did you spend to split the code (in minutes)?	14	3	30

Table 7.2: The result of the user study. The scale for questions Q1-Q3: 1 = the easiest, and 10 = the most difficult.

7.2.6 Example: An Intermittent Neural Network

Among all the actions in Table 7.1, in general, the *learn* action has a higher complexity than most others. Hence, we discuss an intermittent execution of it as an illustration. In particular, we illustrate how a feed-forward neural network learner is executed intermittently (Figure 7.5). We choose an execution strategy where each layer of the neural network is processed at a time. This is the same network which is later used in the neural network-based k-means algorithm in the vibration learning application in Section 7.5.3.

Figure 7.5 shows that when the dynamic action planner decides to launch a *learn* action, each of the m layers of the original neural network $\{l_1, l_2, \ldots, l_m\}$ gets executed sequentially in the forward direction (feed-forward) and then in the backward direction (back-propagation) to com-

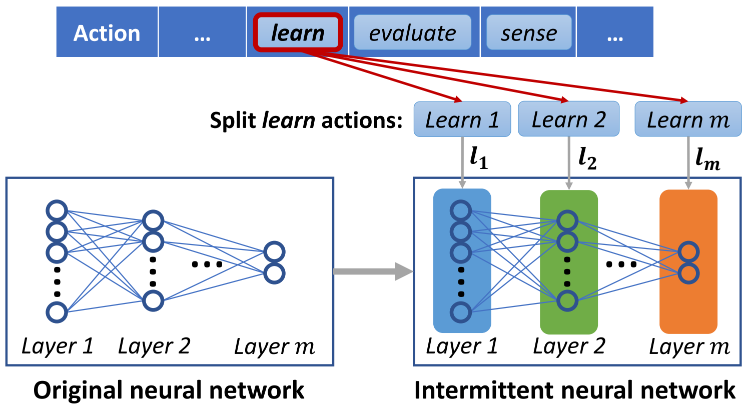


Figure 7.5: An example of intermittent execution of back-propagation algorithm to train a neural network. The original network is segmented into layers and each layer is intermittently executed. Both feed-forward and back-propagation are performed layer by layer in an intermittent manner.

plete one cycle of learning. The system continues to execute each layer l_i as long as the current energy level is higher than required. Once a cycle is completed, the dynamic action planner gets back the control and chooses the next action.

7.3 Dynamic Action Planner

In this section, we describe the *dynamic action planner* which determines a sequence of actions in an online manner. Whenever a sufficient amount of energy is harvested to execute at least one action, the planner dynamically selects the best action that should be performed next, considering the current energy level and the performance of the learner over a short time horizon in the future.

7.3.1 System State and Transitions

We define the state of the proposed system in terms of the examples that are currently in the system and their execution status. Note that the *state of the system* is different from the *action state diagram* (Figure 7.3) which does not involve the execution status of the examples.

For instance, at the beginning of the system, there is no example inside the system. The first time the system harvests enough energy to act, it *senses* new data x_i and then waits for the next action. We denote this state as $\{(x_i, sense)\}$. The next time the system harvests energy, it has more options depending upon the amount of harvested energy, e.g., it can either *sense* a new data x_{i+1}, or execute the next action *extract* on x_i. This results in two possible next states: $\{(x_i, sense), (x_{i+1}, sense)\}$ and $\{(x_i, extract)\}$. In general, given a set of examples in the system, $X = \{x_1, \ldots, x_N\}$ and the supported actions by the system, $A = \{a_1, \ldots, a_K\}$, the state of the system, S is defined by a set of two-tuples $\{(x_i, a_j)\} \subset X \times A$, which denotes that the most recent action performed on x_i is a_j.

A transition from one state S to another state S' happens in one of the following two ways:

• The dynamic action planner may choose to sense new data. In this case, a new example x_{N+1} enters the system, resulting in an addition of a new tuple of the form $(x_{N+1}, sense)$ to the system state. Hence, $S' = S \cup \{(x_{N+1}, sense)\}$.

• A tuple $(u, v) \in S$ is chosen by the dynamic action planner. The system determines the next action v', for example, u in accordance with the action state diagram of Figure 7.3, and either takes action v' on u, or u leaves the system if there are no next actions. Hence, the new state S' is either $\{S - (u, v)\} \cup \{(u, v')\}$, or just $S - \{(u, v)\}$.

7.3.2 Desirable Goal States

The goal of the dynamic action planner is to advance the current system state toward a desirable goal state via a series of state transition decisions. The goal state of an online learning system, especially in the absence of labeled ground truth data, is defined in terms of *the rate of examples learned*, *the rate of inferences performed*, or *a combination of these two rates*. For instance, a common strategy is to maintain a desirable learning rate, ρ_l (i.e., learned examples in L energy harvesting cycles) in the beginning, and once the system has learned a desirable number of example n_l, the goal is reset to maintaining a desirable inference rate, ρ_c (i.e., inferring the desired number of examples in L energy harvesting cycles). Parameters such as ρ_l, n_l, ρ_c, L are application dependent and are determined via empirical studies and from domain expertise.

However, for some applications, the empirical parameters may not bring the desired behavior as the learning environment (e.g., distribution of input examples) changes over time. To overcome this, intermittent learning systems should learn and update the goal state parameters. For example, by evaluating the need for further learning (e.g., via human feedback or obtaining inference results from more capable externals systems) the parameters can be readjusted at run-time. The system can also continue to build statistics on the frequency of learning based on the utility of learning examples obtained from the example selection methods discussed in Section 7.4. In our current implementation of the framework, we use empirically determined parameters. We leave the research on automatic parameter adaptation strategy as future work.

7.3.3 Selecting an Action

Action Selection. For a learner that learns and evolves throughout its lifetime, the process of selecting the best action at every decision point is a never-ending search process as the decision horizon consisting of all future steps is open-ended and infinite. Furthermore, since each state has more than one possible next states, the state-space of the system grows exponentially. Hence, if we aim at selecting a globally best sequence of decisions, depending on the nature of the desired goal state, the optimization algorithm may take forever to find a solution.

To handle this state explosion problem, we consider a *finite decision horizon* on which we search for a locally best solution. In other words, at each decision point, the action planner looks ahead at all possible resultant states due to the next L transitions to find a sequence of state transitions that take the system closest to a goal state. From our experience, L should be in the order of the longest path on the action state diagram. Once the sequence is obtained, only the first action corresponding to the first state transition is selected for execution.

Increasing Planning Efficiency. Even within a finite horizon of length L, the planner has to consider a large number of states. For instance, assuming N examples currently in the system and a horizon of length L, there are $\mathcal{O}(N^L)$ states for the planner to explore. To improve the efficiency of the search, we take additional measures during state-space unfolding, i.e., limiting the number of admitted examples, limiting the value of L, bypassing some boolean actions like *select* and *learnable* at random (with a low probability) and using their default return value instead, and combining lightweight actions with succeeding actions. The last two refinements reduce the dwell time of an example in the system, and thus reduces the average number of active examples within the decision horizon.

7.4 Selecting Examples to Learn

An intermittent computing system must be very keen on exploiting every opportunity to save energy. In an intermittent learning scenario, a substantial amount of energy is saved when a learner selects a minimal subset of training examples that yield a comparable learning performance to using the full training set. This section describes how the framework decides whether an example should be used to retrain the current classifier. At first, we describe four well-known example selection criteria in machine learning (Kabkab et al., 2016; Brown and Mues, 2012). Then we describe three heuristics that meet one or more of these criteria and are currently implemented in the proposed framework.

7.4.1 Desired Criteria for Selecting Examples

Before proposing metrics to quantify the utility of an example toward learning performance, we list a set of desired criteria for the chosen subset, B of a given training set, T.

Uncertainty. The current learning model, θ should be less certain about an example $x \in B$ belonging to any class, y. Otherwise, x does not bring new information to the current learner. This can be expressed as:

$$x = \underset{x \in B}{\operatorname{argmax}} \Big(-\sum_{y} P(y|x,\theta) \log P(y|x,\theta) \Big) \tag{7.1}$$

Balance. The set of chosen examples B should have a balanced selection from all classes. Otherwise, the learner will be biased toward the class that has more training examples.

Diversity. The chosen examples $x \in B$ should be diverse within themselves. Otherwise, the set of chosen examples will have redundancy. Therefore, given a dissimilarity metric $d(x_i, x_j)$, we maximize the mean distance between all pairs of selected examples:

$$\underset{B \subset T}{\operatorname{argmax}} \frac{1}{|B|^2} \sum_{x_i \in B} \sum_{x_j \in B} d(x_i, x_j) \tag{7.2}$$

Representation. The left-out examples should have representatives in the chosen set, B. Otherwise, a learner will miss important information that may be left out in the non-selected set. Therefore, we should minimize the average distance between selected and non-selected examples:

$$\underset{B \subset T}{\operatorname{argmin}} \frac{1}{|B| \times |T - B|} \sum_{x_i \in B} \sum_{x_j \in T - B} d(x_i, x_j) \tag{7.3}$$

The balance criterion has been analytically proven by the machine learning community to increase the convergence rate of gradient-based iterative learning algorithms (Brown and Mues, 2012). Likewise, the other three criteria, i.e., uncertainty, diversity, and representation have been also proven to increase learning performance (Kabkab et al., 2016).

7.4.2 Proposed Online Example Selection Heuristics

Selecting a subset of the training set that satisfies all or most of the above criteria are computationally expensive. Furthermore, in an online learning scenario, the full training set is not readily available as the learner observes examples one at a time over its lifetime. Hence, in order to determine if an example should be learned by an intermittent learner, we devise three simple yet effective heuristics that are incorporated into the framework:

Round-Robin . To ensure *balance*, selected examples fall into k clusters in a round-robin fashion. Assuming n examples have so far been used to obtain clusters with centroids $\mu_1, \ldots, \mu_k$, example x_{n+1} is selected if the following condition is true:

$$1 + n \bmod k = \operatorname*{argmin}_{1 \leq j \leq k} d(x_{n+1}, \mu_j) \tag{7.4}$$

k-Last Lists. To ensure *diversity* and *representation*, we maintain two k-element lists B and B' that keep track of the last $2k$ examples that were selected and not selected, respectively. The *diversity* and *representation* scores (as described in the previous subsection) are calculated using the lists B and B'. A new example x_i is selected if both of the following conditions are met:

$$\begin{gathered} diversity\ (B \cup \{x_i\}) > diversity\ (B) \\ representation\ (B \cup \{x_i\}, B') < representation\ (B, B') \end{gathered} \tag{7.5}$$

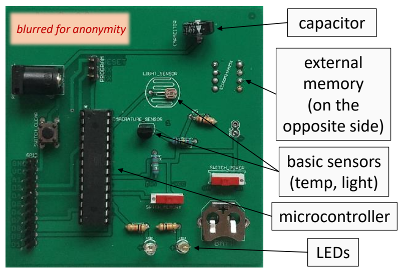
Randomized Choice. To ensure *uncertainty*, we select an example x_i with a probability of p_i. Here, the value of p_i can be used as a threshold for entropy to meet the uncertainty criterion (mentioned in the previous subsection) or can simply be a value to control the selection rate of examples.

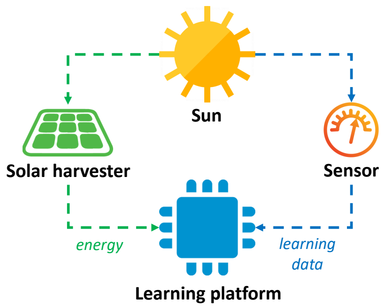
Note that none of these above heuristics require the knowledge of the complete training set. These are applicable to unsupervised and semi-supervised learners as they do not require the class labels. The effectiveness of these heuristics largely depends on the nature of the online learning problem. A comparison of these is presented in the evaluation section.

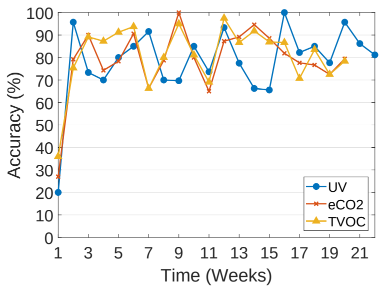
7.5 Application Implementation

We implement three intermittent learning applications that monitor, learn, and classify air quality indices, human presence, and vibration pattern. These systems are powered by solar, RF, and piezoelectric harvesters, respectively. To demonstrate the portability of the proposed framework, these systems are implemented on three different microcontroller platforms, i.e., an AVR, a PIC, and an MSP430-based microcontroller, respectively. This section describes the implementation of these systems along with their end-to-end classification performance, deferring the in-depth evaluation to Section 7.6.

7.5.1 Air Quality Learning (Solar)


blurred for anonymity
capacitor
external memory (on the opposite side)
basic sensors (temp, light)
microcontroller
LEDs





(a) *Custom learning platform PCB* (b) Air quality learning system (c) Detection accuracy

Figure 7.6: Air quality learning system uses a custom-built platform and is powered by solar energy.

Overview. The air quality learning system detects and notifies abnormalities in air quality indices such as the *ultraviolet radiation* (UV), *equivalent carbon dioxide* (eCO2), and *total volatile organic compound* (TVOC) by learning their normal levels on harvested solar energy. Unlike sensing systems that just report the absolute sensing values, it learns the evolving status of air quality and provides environmental context-based notifications, which is smarter than reporting simple index values.

The system has been deployed in the real-world (near a window of an apartment), and it is active since September 21, 2018. We have an anonymous website showing the real-time status of the learner, which is updated every 10 minutes[1]. For demonstration purpose, we use an additional gateway device that reads the classification results from the batteryless learner and sends them to the web.

System. As the experimental platform, we develop a custom printed circuit board (PCB) which is shown in Figure 7.6(a). The board consists of an ATmega328p microcontroller having a 1KB internal EEPROM, light and temperature sensors, a 32KB external non-volatile EEPROM, a

0.2F supercapacitor as the energy reservoir, output indicator LEDs, and energy harvester circuitry. Although more advanced energy management hardware such as multiple capacitors (Colin et al., 2018; Hester and Sorber, 2017; Hester et al., 2015a) can be used for more efficient use of harvested energy, we keep our hardware design simple to focus on the feasibility, behavior, and performance of the learning framework. The air-quality sensors measuring UV, eCO2, and TVOC are externally connected to the PCB (not shown in the figure). The board harvests solar energy and executes machine learning algorithms following the proposed intermittent learning framework. As shown in Figure 7.6(b), the air quality learning system utilizes the custom PCB as the learning platform and a small solar panel for energy harvesting. When the sunlight is available, the solar panel charges the supercapacitor and powers up the circuitry to wake up the learner. Upon wake up, the system collects data from sensors and executes the learning actions. Note that although the sunlight is present for the most of the day, as the system is powered through a limited sized capacitor that drains quickly when the system runs, the input power to the system is intermittent, and thus requiring the framework to save/restore the intermediate system states into/from the non-volatile memory.

Learning Algorithm. The k-nearest neighbor algorithm is used to learn and detect an anomaly in the ambient air quality. We choose the k-nearest neighbor algorithm for clustering among other alternatives such as *autoencoders* since the application does not deal with high dimensional data and the carefully-designed features (described next) are more compute- and energy-efficient than autoencoders. Following the proposed framework, we implement the *sense* action that reads three sensor values (UV, eCO2, and TVOC) every 32 seconds. For every 60 sensor readings, the *extract* action generates five features– mean, standard deviation, median, root mean square (RMS), and peak-to-peak amplitude (P2P). The five features generated by the *extract* action constitute an example which is used for learning (i.e., the *learn* action) or detecting an anomaly (i.e., the *infer* action).

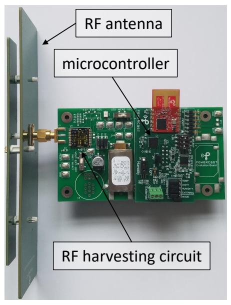
Prior to learning, the *select* action determines whether the newly-obtained example should be learned or discarded using the example selection heuristic. If the example is selected for learning, the *learn* action updates the threshold score for anomaly detection by learning the latest set of examples, including the newly-obtained one. The anomaly score AS_i for the i^{th} example e_i in an example set is calculated as $AS_i = \sum_{j=1}^{k} d(e_i, e_j)$, where e_j is the j^{th} nearest neighbor example of e_i, k is the number of nearest neighbors in the set, and $d(\cdot)$ is the feature distance function (Cola et al., 2015). The feature distance between two examples e_i and e_j is defined as $d(e_i, e_j) = \sqrt{\sum_{m=1}^{n}(f_m^{e_i} - f_m^{e_j})^2}$, where $f_m^{e_i}$ is the m^{th} feature of the example e_i, $f_m^{e_j}$ is the m^{th} feature of the example e_j, and n is dimension of the feature vector. After computing the anomaly score for all examples in the set, an anomaly threshold AS_{TH} is determined by taking the 90th percentile of the anomaly score.

To detect an anomaly (i.e., the *infer* action), the system calculates the anomaly score AS_{new} for the newly-obtained example. It is classified as abnormal, if $AS_{new} > AS_{TH}$, and normal, otherwise. Note that the anomaly threshold AS_{TH} evolves over time as new examples are learned at run-time.

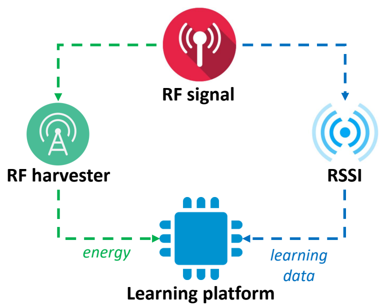
Figure 7.6(c) shows the anomaly detection accuracy of the system for the three indicators, i.e., UV, eCO2, TVOC for 20 weeks. The anomalies are detected with 81%–83% average accuracy for the air quality indicators. To calculate the accuracy of the learners, we download the classification results as well as the raw data from the device once every week. The raw data is visualized and inspected by human experts to obtain the ground truth labeling, which is compared with the classification results of the learner to calculate the accuracy.

7.5.2 Mobile Human Presence Learning (RF)

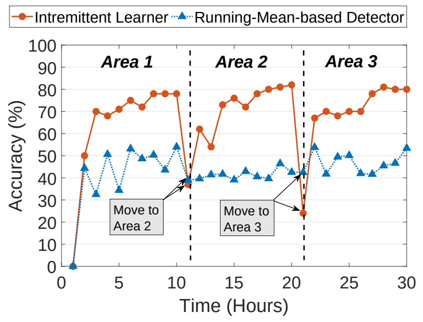
Overview. We implement a mobile human presence learning system that is powered by harvesting RF energy. It detects the presence of a person in indoor space by observing the short-term variation in the received signal strength indicator (RSSI) values and by learning a dynamic threshold that helps it determine if a person is present or not. This is different from an RSSI threshold-based human presence detection system which does not generalize across different physical world environments or when the RF properties of the same environment change. Using the pro-



(a) RF energy learning platform



(b) Human presence learning system



(c) Detection accuracy

Figure 7.7: Mobile human presence learning on RF energy.

posed intermittent learning framework, the human presence learner continuously learns the RF pattern and thus it is able to learn and adapt its model parameters to accurately detect the presence of humans—even when the system is moved from one place to another. Using this learner, a mobile social robot (Lemaignan et al., 2017) can perceive the presence of humans when other types of sensors are ineffective (e.g., cameras in the dark).

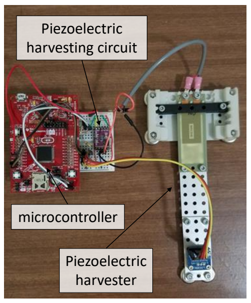
System. The system consists of three major parts that are shown in Figure 7.7(a) – an RF antenna (850-950 MHz) (Powercast, 2016a), an RF harvesting circuit (P2110) (Powercast, 2016b) and a PIC24F16KA102 microcontroller. Additionally, a 50mF capacitor and a 512-byte EEPROM (built-in the microcontroller) are used as the energy reservoir and non-volatile data storage, respectively. Figure 7.7(b) shows that both energy and data come from the RF signal. When the capacitor is charged by harvesting energy from the RF power source, the system starts to measure RSSI and learns to detect human presence or absence. The learning examples consist of the features obtained from RSSI values, and the learning model is saved in the non-volatile memory so that when the power goes off, the system does not lose its state.

Learning Algorithm. Similar to the air quality learning system, a k-nearest neighbor learner is used for anomaly detection. First, the RSSI power levels received at the antenna (ranging from 0.04mW to 50mW) are measured and calculated by the *sense* action to collect a set of 10 to 30 values. The number and rate of RSSI readings constituting the set depends on the strength and the power of the signal. Four features (i.e., mean, standard deviation, median, and root mean square (RMS) of RSSI values) are extracted by the *extract* action from a set of RSSI values. The extracted features constitute an example which is used either for learning (*learn*) or for human presence detection (*infer*) as dictated by the dynamic action planner. Since the learning and inferring algorithms in this application are similar to the air quality learning system, their details are omitted. The main difference between these two systems is that the human presence learner learns and updates its model more frequently and more intermittently (between tens of milliseconds and seconds) than the air quality learner (between minutes and hours) since RF signals change much faster than air quality sensor values.

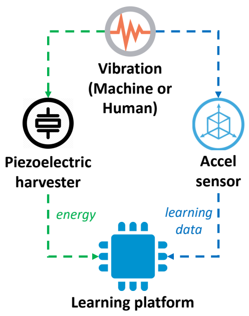
In order to evaluate the performance of the system and its ability to adapt in a new environment, we deploy and measure its accuracy at three different areas by moving it from one place to another. The accuracy is compared against a baseline system that uses a threshold changing over time based on the run-time mean of the RSSI values to detect human presence. Figure 7.7(c) shows the accuracy of the system at three different locations as the system is moved. The accuracy is tested every hour using 30 test cases of human presence and absence. As shown in the figure, when the intermittent learning system is moved to a new area, it recovers its detection accuracy within a few hours by adapting its model parameters to the new RF environment which is very different from the previous one. For instance, the accuracy drops to 38% at hour 11 after moving to area 2, but it rises back to 76% at hour 15 and increases to 82% at hour 20. The baseline system's accuracy stays below 50% for all areas.

7.5.3 Vibration Learning (Piezoelectric)

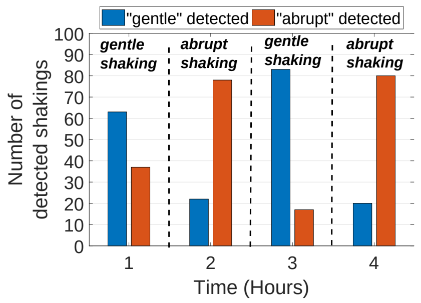
Overview. The vibration of machines such as industrial machinery, HVAC equipment, vehicles, and household appliances carries the signature of their state of operation and health status. By observing and learning their regular vibration pattern, we can predict their impending failure when there is a deviation or irregularity in their vibration pattern. Vibration anomaly detection systems can also be used in human health and wellness applications. For example, a gait anomaly detector can give a warning sign of walking abnormalities such as the freezing of gait (Giladi and



(a) Piezoelectric energy learning platform



(b) Vibration learning system



(c) Detection accuracy

Figure 7.8: Vibration learning on piezoelectric energy.

Fahn, 1998) or a sudden fall by learning and classifying a user's walking pattern. Early detection of Parkinson's disease is possible by noticing tremors (hand or foot shaking) (Zimmermann et al., 1994), and detecting leg shaking (SPINDLES) (Xia et al., 2017) are examples of people-centric vibration sensing and inference application.

We develop a vibration learning system that is powered by harvesting piezoelectric energy. It detects a potential malfunction of a vibrating object or a human limb by monitoring and learning the regular vibration pattern using an accelerometer sensor, and then detects and reports anomalies. The system is shown in Figure 7.8(b). The system can be attached to a target to learn the level of vibration that may relate to an impending breakdown or an anomaly.

System. As shown in Figure 7.8(a), a piezoelectric harvester (PPA-2014) (Corporation, 2017), generating power between 1.8mW and 36.5mW, is connected to an MSP430FR5994 microcontroller via a piezoelectric harvesting circuit (LTC3588). A 6mF capacitor stores the harvested energy. We use the microcontroller's built-in 256KB FRAM as the non-volatile storage to save the system state. A low-power accelerometer sensor (LIS3DH) attached to the tip of the piezoelectric harvester senses the three-dimensional vibration at the sampling rate of 50Hz.

Learning Algorithm. We implement a *cluster-then-label* (Goldberg and Zhu, 2010; Zhu, 2005) learner that utilizes both labeled and unlabeled data where the training examples first go through a clustering step, and then the clusters are labeled. The learner classifies new examples by finding the cluster it belongs to and then uses the label of the cluster to classify the example. This approach falls under the general category of semi-supervised learning but is different from alternatives such as label propagation (Xiaojin and Zoubin, 2002).

For clustering, we implement a two-layer neural network-based k-means algorithm (Marsland, 2015) where the input and output layers correspond to the feature vector of an example and the two clusters (normal and abnormal vibration), respectively. Unlike typical k-means algorithms that have all examples in its batch learning setup, only one example (at a time) is available to our online learner, and the cluster means are unknown. Hence, we feed one example to the neural network at a time and approximate the cluster means by moving the neuron closer to the current input example—making that center even more likely to be the best match next time that input is seen.

The *learn* action implements the clustering algorithm which uses feature vectors extracted by the *extract* action consisting of the mean, standard deviation, median, root mean square (RMS), peak-to-peak amplitude (P2P), zero-crossing rate (ZCR), and average absolute acceleration variations (AAV). Two output neurons corresponding to the two clusters (normal and abnormal vibration) are fully connected to the input layer neurons. An activation value, a_j for each neuron is calculated by $a_j = \sum_{i=1}^{n} w_{ij}x_i$, where w_{ij} is the weight between the ith element of the input vector and the jth neuron, x_i is the ith element of the input vector, and n is the length of the input vector, $\mathbf{x}$. We implement competitive learning where only the neuron with the largest activation value wins and only the weights connected to the winner are updated at each iteration since the winner neuron corresponds to the cluster that is the closest to the current input. The weights of the winner neuron, w_{ij} are updated by $\Delta w_{ij} = \eta(x_i - w_{ij})$, where η is the learning rate. To classify new data (i.e., *infer* action), features of new example are extracted and fed into the neural network as the input. The output neuron with the highest activation value is chosen as the predicted class.

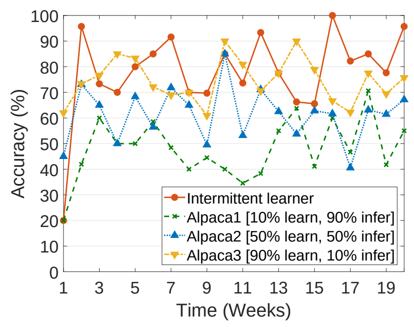
We conduct a set of controlled experiments with the vibration anomaly detector. We attach the system to an arm of a person and let the system learn to cluster the arm shaking into two categories: gentle vs. abrupt shaking. Gentle and abrupt arm movements are performed by shaking the arm less than five times and more than ten times in five seconds, respectively. Figure 7.8(c) shows the classification accuracy for four hours of the experiment. 100 gentle shaking gestures are performed during the first and the third hour, while 100 abrupt shaking gestures are performed during the second and the fourth hour. As shown in the figure, the system learns and classifies the two movements with 76% average accuracy using the kinetic energy generated by the arm shaking gestures.

7.6 Evaluation

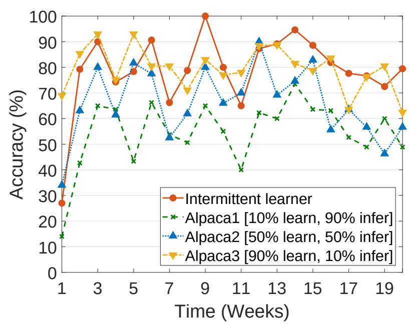
We conduct in-depth experiments to evaluate various aspects of the three applications described in the previous section. First, their performance is compared with 1) state-of-the-art intermittent computing systems that execute learning and inference steps periodically, and implements neither the dynamic action planner nor the example selection heuristics (Section 7.6.1), and 2) three popular offline machine learning algorithms for anomaly detection (Section 7.6.2). Second, we evaluate the effect of example selection heuristics (Section 7.6.3) and energy harvesting patterns (Section 7.6.4) on the performance of the learner. Third, we measure the energy consumption and execution time of each action and quantify the overhead of the system (Section 7.6.5).

7.6.1 Comparison with the State-of-the-Art Intermittent Computing Systems

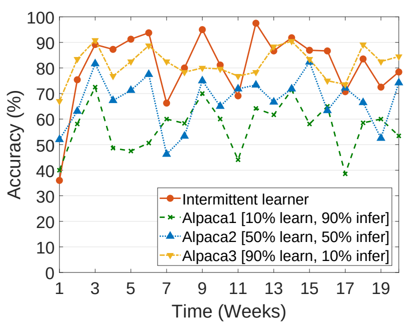
We compare the accuracy of the three intermittent learners (air-quality, human presence, and vibration learning) against two state-of-the-art task-based intermittent computing systems: Alpaca (Maeng et al., 2017) and Mayfly (Hester et al., 2017). Both of the baseline systems execute the same learning algorithm as ours, but they do not implement the proposed framework. Instead, the two baseline systems repeat a fixed sequence of actions periodically, and they *duty-cycle* the execution of *learn* and *infer* actions according to a predefined schedule. For example, Alpaca with a duty-cycle parameter of [90% *learn*, 10% *infer*] executes the *learn* action 90% of the time and the *infer* action 10% of the time, after executing the *sense* and *extract* actions. Mayfly works



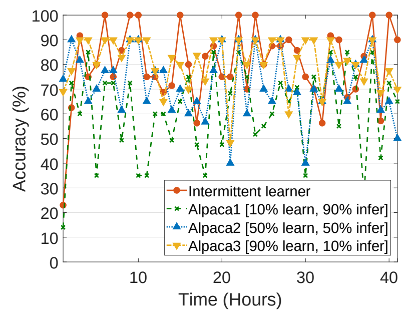
(a) Air quality 1 (UV)



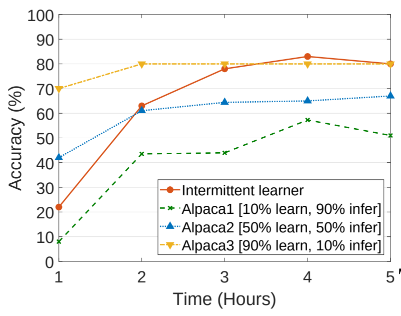
(b) Air quality 2 (eCO2)



(c) Air quality 3 (TVOC)



(d) Human presence



(e) Vibration

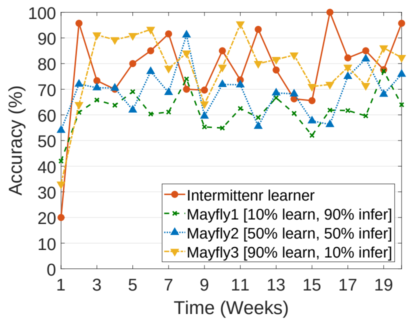
	Inter. Lean	Alpaca Duty 10/90	Alpaca Duty 50/50	Alpaca Duty 90/10
UV	**81%**	48%	61%	74%
eCO2	**81%**	54%	66%	79%
TVOC	**83%**	57%	61%	81%
Human Presence	**82%**	60%	71%	81%
Vibration	**76%**	40%	59%	78%

Table 7.3: Average detection accuracy (%): Intermittent learner vs. Alpaca.

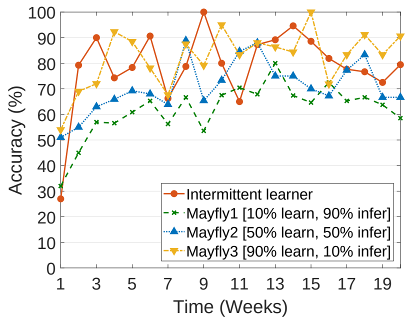
Figure 7.10: Accuracy comparison with Alpaca (no dynamic action planner and example selection)

the same way as Alpaca with the exception that it discards stale examples by setting a data expiration interval. None of the baseline solutions implement example selection heuristics.

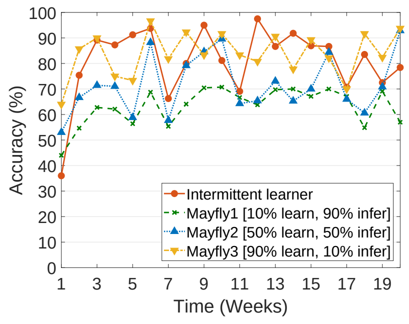
Figures 7.10(a)-(e) and 7.12(a)-(e) compare the accuracy of the intermittent learners against Alpaca and Mayfly-based implementation of the same applications. We use three duty-cycle parameters for the baseline solutions: [10% learn, 90% infer], [50% learn, 50% infer], and [90% learn, 10% infer]. Table 7.3 and 7.4 summarize the results. Overall, the intermittent learning systems achieve 80% average accuracy while Alpaca and Mayfly-based implementations achieve 54%–79% and 59%–78% average accuracy, respectively, depending on the duty-cycle parameters. For both Alpaca and Mayfly, as the amount of *learn* action increases from 10% to 90%, the accuracy increases, and finally, it becomes comparable to the accuracy of the intermittent learning systems when the duty-cycle has 90% *learn* actions. However, the intermittent learning systems achieve 80% accuracy by executing 50% less number of *learn* actions compared to Alpaca and Mayfly for [90% learn, 10% infer] duty-cycle. As a result, the intermittent learners increase the



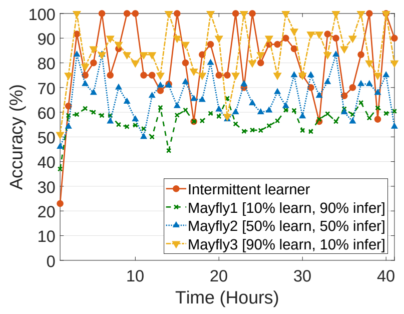
(a) Air quality 1 (UV)



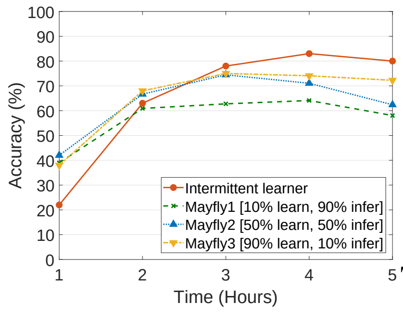
(b) Air quality 2 (eCO2)



(c) Air quality 3 (TVOC)



(d) Human presence



(e) Vibration

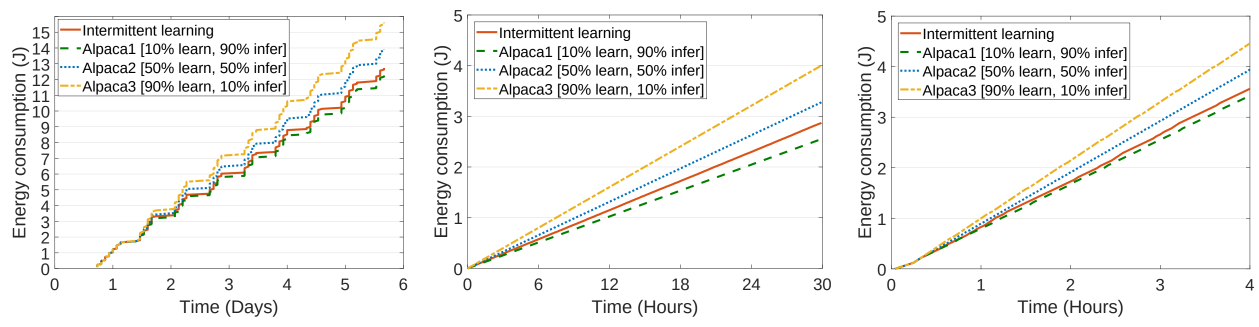
	Inter. Lean	**Mayfly Duty 10/90**	**Mayfly Duty 50/50**	**Mayfly Duty 90/10**
UV	**81%**	61%	69%	79%
eCO2	**81%**	61%	71%	81%
TVOC	**83%**	63%	71%	83%
Human Presence	**82%**	56%	66%	84%
Vibration	**76%**	56%	63%	65%

Table 7.4: Average detection accuracy (%): Intermittent learner vs. Mayfly.

Figure 7.12: Accuracy comparison with Mayfly (no dynamic action planner and example selection)

inference throughput by performing more *infer* actions than the baseline intermittent computing systems that waste time and energy in performing unproductive *learn* actions. We also observe that different actions are chosen by the dynamic action planner at run-time based on the state of the system, while the baseline systems follow a repeated fixed-sequence of actions, e.g., 90% of the time [*sense, extract, learn*] and 10% of the time [*sense, extract, infer*] sequence without caring for the learning performance.

Figures 7.13(a)-(c) compare the total energy consumption of the intermittent leaning framework and Alpaca-based implementation of the three applications over time. For all three applications, the intermittent learning system consumes less energy than Alpaca with [90% learn, 10% infer] and [50% learn, 50% infer] duty-cycle parameters, but consume slightly more energy than Alpaca with [10% learn, 90% infer] duty-cycle parameters. For instance, the proposed system consumes 37% less energy than Alpaca with [90% learn, 10% infer] duty-cycle at hour 30 for the human presence learning experiment in Figure 7.13(b), but still achieves similar average accu-



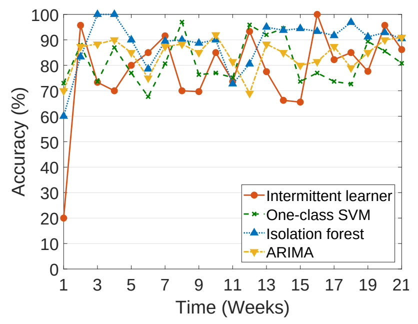
(a) Air quality (b) Human presence (c) Vibration

Figure 7.13: Energy consumption comparison with Alpaca (no dynamic action planner and example selection)

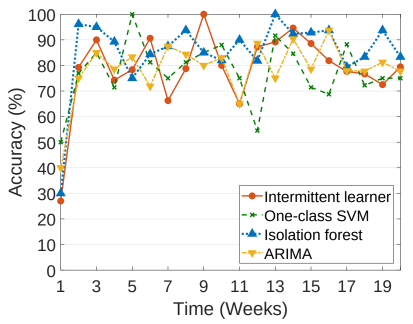
racy to Alpaca with [90% learn, 10% infer] duty-cycle. In other words, the intermittent learning system achieves at least 1.6× higher accuracy than Alpaca when both the systems consume the same amount of energy. This is because the dynamic action planner intelligently selects actions at run-time, which leads the system to spend less energy and time. Furthermore, the data selection module trains the system with examples that are likely to improve its learning performance and prevents the system from wasting energy in learning examples that do not.

7.6.2 Comparison with Offline Machine Learning Algorithms

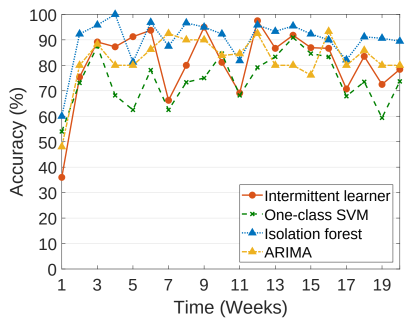
We compare the accuracy of anomaly detection of the three intermittent learners against three widely used offline anomaly detectors that are based on: 1) one-class SVM (Support Vector Machine) (Manevitz and Yousef, 2001) with RBF (Radial Basis Function) kernel, 2) isolation forest (Liu et al., 2012, 2008), and 3) Auto-Regressive Integrated Moving Average (ARIMA)-based clustering. Unlike the proposed framework which selects examples to learn at run-time, these offline detectors use all the examples for anomaly detection at once. Figures 7.15(a)-(e) compare the accuracy of the intermittent learners against the offline anomaly detectors. The average accuracy of these detectors are summarized in Table 7.5. We observe that the intermittent learners achieve a comparable accuracy (80%) to the three offline detectors (78%, 86% and 83% for the one-class SVM, isolation forest and ARIMA, respectively) while selecting and learning



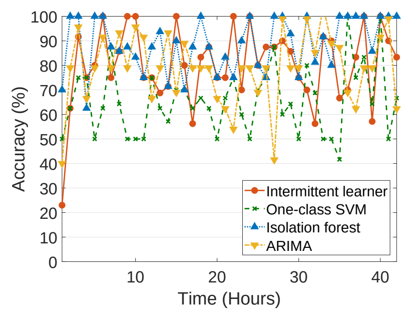
(a) Air quality 1 (UV)



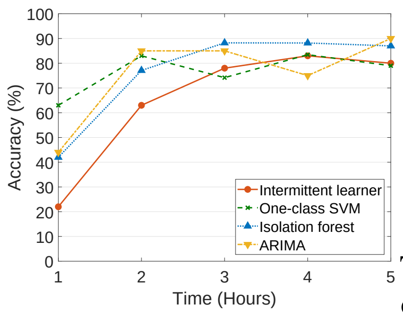
(b) Air quality 2 (eCO2)



(c) Air quality 3 (TVOC)



(d) Human Presence



(e) Vibration

	Inter. Learn	**One-class SVM**	**Isolation Forest**	**ARIMA**
UV	**81%**	81%	88%	84%
eCO2	**81%**	78%	88%	80%
TVOC	**83%**	75%	89%	80%
Human Presence	**82%**	70%	85%	79%
Vibration	**76%**	79%	85%	83%

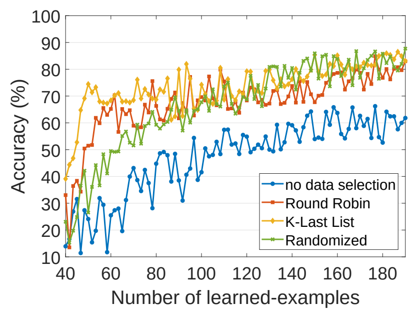
Table 7.5: Average detection accuracy (%): Intermittent learner vs. offline machine learning anomaly detectors.

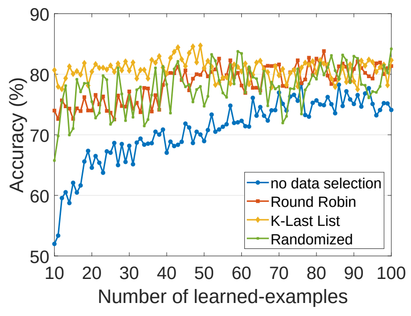
Figure 7.15: Accuracy comparison with offline machine learning anomaly detectors (one-class SVM, isolation forest and ARIMA).

only 44% of the input examples and judiciously discarding 56% of the examples that are unlikely to increase the accuracy of the learner by using the round-robin selection method.

7.6.3 Effect of Example Selection Heuristics

To evaluate the effect of example selection heuristics, we compare the three proposed training example selection heuristics, i.e., round-robin, k-last lists, and randomized selection against no data selection strategy, i.e., every example is used for training. Figures 7.16(a)-(c) plot the detection accuracy over the number of learned-examples for each heuristic. We observe that all three heuristics consistently demonstrate higher accuracy than the no data selection policy. This may seem counter-intuitive at first, but the reason for a higher accuracy by any of the heuristics than the no data selection policy is that in Figures 7.16(a)-(c), we report the actual number of examples learned by the four strategies, which is not generally the same as the number of examples


(a) Air quality (UV)


(b) Human presence

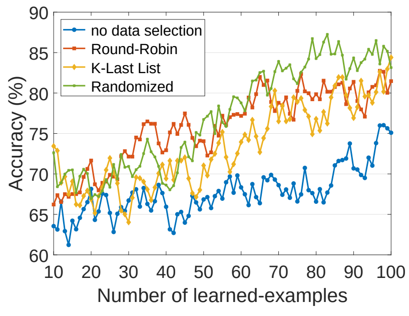
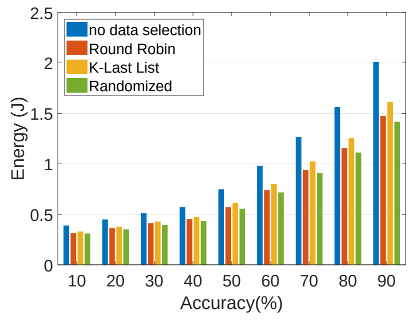
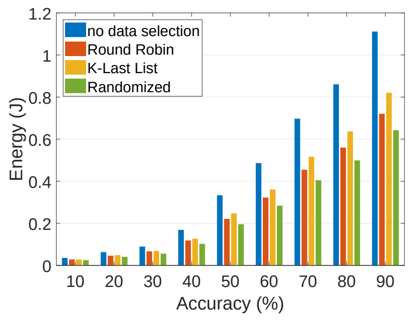

(c) Vibration

Figure 7.16: Effect of example selection heuristics: accuracy vs. number of learned-examples


(a) Air quality (UV)


(b) Human presence

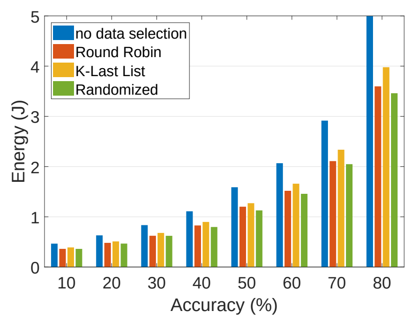

(c) Vibration

Figure 7.17: Effect of example selection heuristics: accuracy vs. energy

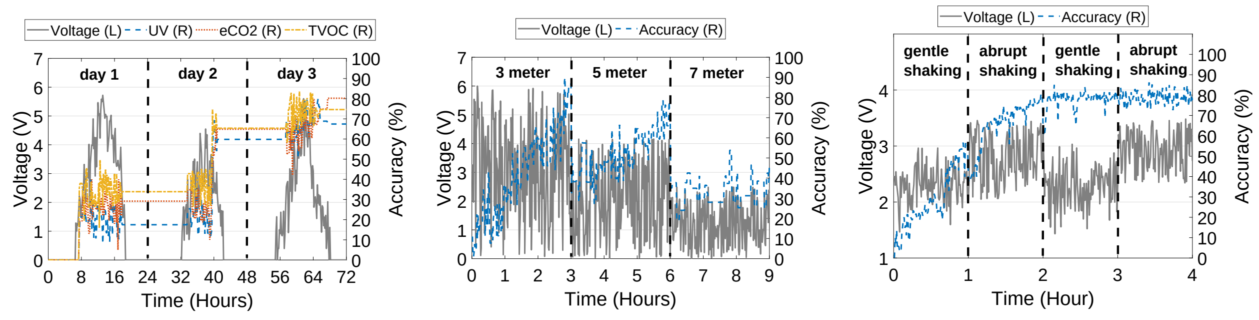
entering the system. For instance, the no data selection policy learns all of the 180 examples it encounters and achieves 60%; whereas the round-robin heuristic achieves 80% accuracy after learning 180 examples, but it has encountered much more than 180 examples and chose to learn only the best 180 ones. By skipping examples that are unlikely to improve the accuracy, the intermittent learning systems with these selection heuristics achieve the same level of accuracy with less energy, which is evident from Figures 7.17(a)-(c).

In both air quality and human presence learning systems, the k-last lists selection increases the accuracy rapidly in the beginning as shown in Figure 7.16(a) and 7.16(b). The round-robin and randomized heuristic catch up with the accuracy of k-last lists as more examples are seen and finally the accuracy converges to 82% and 80% for air quality and human presence learning systems, respectively. For the vibration learning system in Figure 7.16(c), the k-last lists and the randomized selection reach a similar level of accuracy (83–84%) after learning about 100 examples, but the randomized selection heuristic reaches the highest accuracy (87%) earlier than the k-last lists. The round-robin heuristic shows a better performance for a smaller number of examples (20–45 examples) in the beginning, but gets caught up by the other two after learning 50–70 examples. Considering the high computational complexity and energy cost of k-last lists, we conclude that randomization or round-robin heuristics are reasonable choices for systems having tighter energy constraints.

7.6.4 Effect of Energy Harvesting Pattern

In Figure 7.18(a)-(c), we plot the energy harvesting patterns (voltage level) of the energy harvesters (solar, RF and piezoelectric harvester) for the three systems and evaluate their accuracy over time. In order to assess the effect of energy harvesting pattern on the detection accuracy, the time period is divided into segments that are expected to have different energy harvesting patterns.

Figure 7.18(a) shows the solar energy harvesting pattern, along with the accuracy of the air quality learning system for three consecutive days. As shown in the figure, the detection accuracy improves during the daytime (8 am–5 pm) as the system learns new examples using the harvested energy. At night, the system is essentially off, and in the next morning, the system resumes learning new examples, and its accuracy improves over time. We also occasionally observe interruptions in the otherwise continuous energy harvesting pattern during the daytime due to inadequate sunlight. During these periods of inadequate energy supply when a full cycle of learning is not possible, the system senses, selects, and saves the examples that have the potential to improve accuracy. When sufficient energy is harvested again, the system resumes learning the saved examples. Thus, the intermittent learner does not require sensor data to be acquired and processed simultaneously in real-time. Acquired data are buffered by the system in the non-volatile memory, and the CPU processes it when energy is available. This cannot be achieved by the state-of-the-art intermittent computing system (Hester et al., 2017) that collects sensor data without considering their utility towards learning and discards them when they are stale—which leaves no data to learn when energy becomes available.



(a) Air quality (b) Human presence (c) Vibration

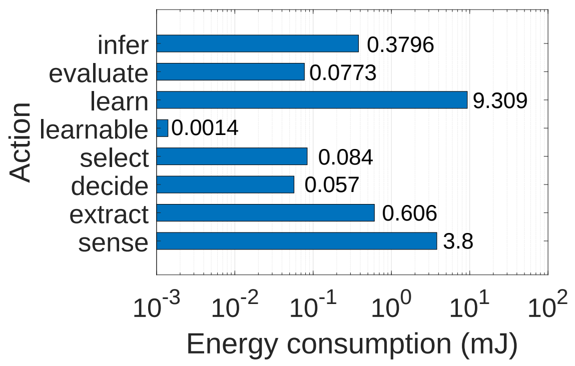
Figure 7.18: Effect of Energy Harvesting Pattern

Figure 7.18(b) shows the RF energy harvesting pattern and the detection accuracy of the human presence learning system for nine hours. Every three hours, the system is placed at different distances (3, 5, and 7 meters) from the RF energy source, and the amount of energy harvested at each distance is measured. As expected, less amount of energy gets harvested as the distance increases with an average of 3.1V, 2.2V, and 0.9V at 3, 5, and 7 meters, respectively—which results in a decrease in detection accuracy with the distance, i.e., 86%, 74%, and 46% at hour 3, 6, and 9, respectively. Since a change of location causes changes in the RSSI pattern, the system needs to learn a new RSSI pattern whenever it relocates. However, due to the less harvested energy at a longer distance, it takes more time to harvest energy to execute the *learn* and the *infer* actions, which slows down the execution rate of both learning and inference. The difficulty in learning RSSI patterns from weaker signals at a longer distance is another reason for the decrease in accuracy.

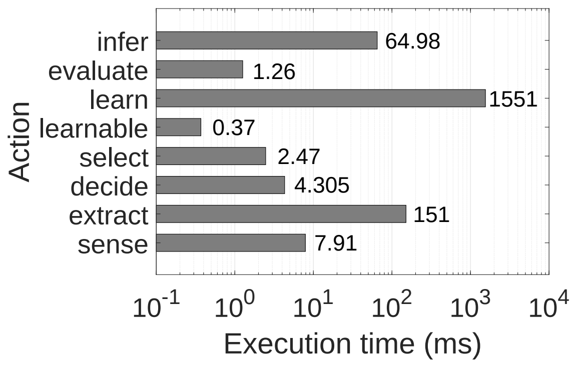
Figure 7.18(c) shows the piezoelectric energy harvesting pattern and the detection accuracy of the intermittent vibration learning system. The time period is divided into four one-hour segments. To capture different harvesting patterns, the harvester is shaken gently during the first and third hour and abruptly during the second and the fourth hour. The accuracy of the learner increases over time and converges to 80% at hour 4, irrespective of the shaking type and consequent energy harvesting pattern (2.29V, 2.81V 2.27V, and 2.92V on average at hour 1, 2, 3, and 4, respectively). The energy harvesting pattern, in this case, does not seem to affect the accuracy much since the amount of energy harvested from both gentle and abrupt shakes are above the minimum operation voltage (2V) of the system which allows it to select learning data and execute *learn* action stably.

7.6.5 Time and Energy Overhead

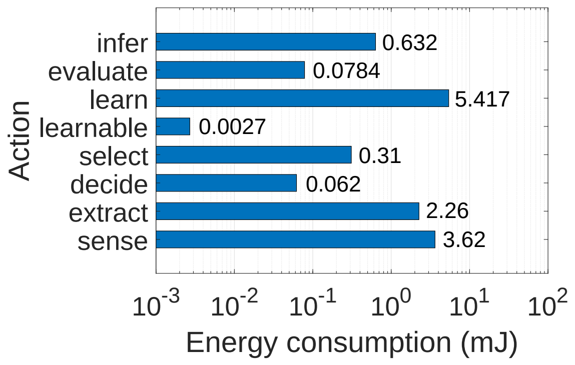
We measure the energy consumption and execution time of all the action primitives, the dynamic action planner, and the three example selection heuristics to quantify the overhead of the proposed framework. We use an MSP430FR5994 as the experimental platform and measure the energy consumption of each module using the EnergyTrace tool (Instrument, 2018).



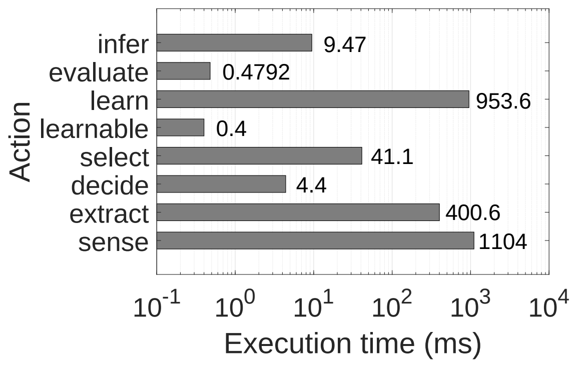
(a) Energy consumption (k-NN)



(b) Execution time (k-NN)



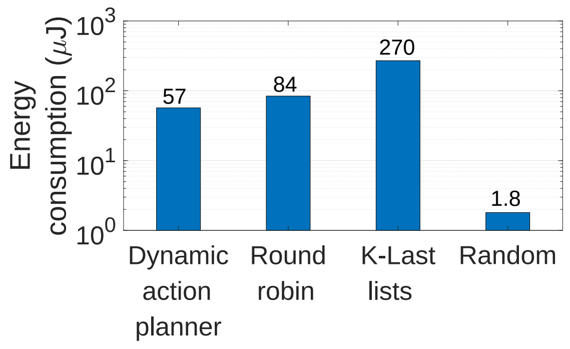
(c) Energy consumption (k-means)



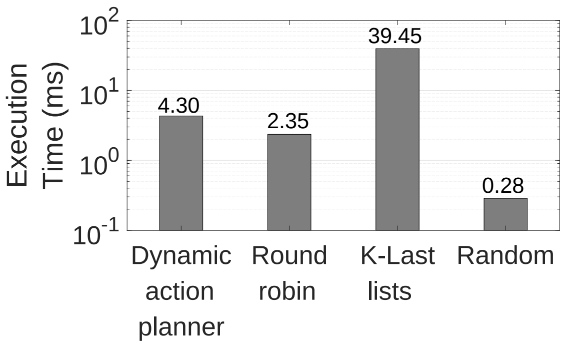
(d) Execution time (k-means)

Figure 7.19: Energy consumption and execution time of actions in two different learning algorithms. (a) and (b): k-nearest neighbors (k-NN). (c) and (d): neural network-based k-means (k-means). All plots are in log-scale.

Figures 7.19(a) and 7.19(b) show the energy and time required by each action of the k-NN algorithm used in the air quality learning system. As expected, *learn* consumes the highest 9.309mJ energy, which is decomposed into three sub-actions for intermittent execution. The



(a) Energy consumption



(b) Execution time

Figure 7.20: Overhead (energy and execution time) of the dynamic action planner and three example selection algorithms. All plots are in log-scale.

energy consumption of *sense* is relatively large (3.8mJ) since it acquires raw data from three sensors (UV, eCO2, and TVOC). Similar to energy consumption, *learn* takes the longest time (1551ms) to execute, followed by *extract* (151ms) and *infer* (64.98ms).

Figures 7.19(c) and 7.19(d) show the energy and time required by each action of the neural network-based k-means algorithm used in the vibration learning system (Marsland, 2015). The *sense* and *extract* actions consume the second (3.62mJ) and third (2.26mJ) largest energy after the *learn* (5.417mJ) since they process acceleration sensor data at a high sampling rate. The *learn* and *infer* use the same neural network, but their overheads are different. The overhead of *learn* (5.417mJ and 953.6ms) is about 100X higher than *infer* (0.0632mJ and 9.47ms) since *learn* involves several orders of magnitude more arithmetic operations and more iterations than *infer*.

Figure 7.20 shows the energy and time overhead of the dynamic action planner and the three example selection heuristics of the vibration learning system. We set the maximum number of admitted examples to two for the dynamic action planner, and the k-last lists uses three examples. As shown in Figures 7.20(a) and 7.20(b), the dynamic action planner has an energy and time overhead of 57μJ and 4.3ms, respectively. Although the action planner is executed more frequently than any of the actions (once after each action), its total overhead is below 3.5% compared to end-to-end processing of an example. In more detail, 2.9% of energy and 1.4% of time overhead is imposed for learning, while 4.2% of energy and 4.3% of time overhead is imposed for inference of an example, compared to the same system that does not run the dynamic action planner.

Figure 7.20 also compares the example selection heuristics. Among the three heuristics, the k-last lists consumes the highest 270μJ energy, whereas the randomized selection consumes the lowest 1.8μJ. This is because the k-last lists computes the diversity and the representation scores for $2k$ examples ($\mathcal{O}(k^2)$) while the random heuristic only needs to generate a random number without looking into the acquired data.

7.7 Limitations

This work proposes the first step towards the intermittent machine learning on embedded devices, which enables them to adapt their learning capability over a prolonged period of time without a battery. Despite the promising results, our work has several limitations that need to be further studied in future work.

Usability. The type and scope of intermittent learning applications can be limited by intermittent energy sources and their relationship to the data of interests to learn and infer. First, the desired learning algorithm or system setup (e.g., high-resolution sensor) cannot be employed if the expected amount of energy intermittently harvested from available energy sources, e.g., environment or human is not sufficient for it. Next, the occurrence of sensing data and the energy-harvesting source might be uncorrelated or independent in many cases, which results in low performance in learning. Unless the data is always available like the air-quality monitoring application implemented in this paper, where UV data is available all daytime, the events of interest can be missed due to the timing mismatch between energy-harvesting and data-occurrence pattern.

Programmability. Decomposing a source code into actions associated with energy constraints is a challenging problem. Although the intermittent learning framework provides an energy pre-inspection tool that helps the programmer write actions, the tool might fail to estimate the exact amount of energy required by an action. Our approach that measures the worst-case energy consumption of action by iteratively feeding all the available data during development is based on the assumption that machine learning modules usually follow a standard control-flow for learning. However, this iterative and statistical approach does not guarantee that the system experiences all the possible execution scenarios (e.g., different control-flow or data-based branching). Also, the dynamic changes at run-time introduce a variation in energy consumption of an action, e.g., system failure, re-configuration, deterioration of hardware, which makes the execution of action incomplete.

Learning Algorithm. The intermittent learning framework is designed to perform supervised and unsupervised learning. It does not fully support supervised learning since it requires labeled data that is not usually available in online. Relying on an external, high-accuracy inference system to obtain the labels at run-time can be an option; however, it causes energy cost for data transmission and increases latency, which is not desirable to an intermittently powered system. Alternatively, reinforcement learning (Russell and Norvig, 2016) can be used to enable real-time feedback from the environment or humans in trying to maximize the reward. Also, deep neural networks, the state-of-the-art learning algorithm, are still challenging to be executed on embedded devices with intermittent power. Although we demonstrate that a simple neural network can be intermittently executed in our application, large sizes of deep neural networks are not easily applied to an intermittent learning system in practice due to their small amount of available energy and limited computing power.

7.8 Prior Work and Their Limitations

Intermittent Computing Platform. Several application-specific energy harvesting systems have been proposed that run on harvested RF (Philipose et al., 2005; Sample et al., 2008; Buettner et al., 2009; Naderiparizi et al., 2015; Zhang et al., 2011) or piezoelectric (kinetic) energy (Huang et al., 2016b; Rodriguez et al., 2017). In general, the goal of general-purpose intermittent computing platforms is to overcome the challenges due to the irregular and scarce power-supply. Mementos (Ransford et al., 2012) transforms general-purpose programs into interruptible computations that are protected from frequent power losses by automatic, energy-aware checkpointing. Ratchet (Van Der Woude and Hicks, 2016) proposes a compiler-based technique that adds lightweight checkpoints to unmodified programs that allow existing programs to execute across power cycles correctly. To eliminate the need for checkpoint placement heuristics, Hibernus (Balsamo et al., 2015, 2016) puts the system to hibernation by monitoring the voltage and saving the system state when power is about to be lost. Chinchilla (Maeng and Lucia, 2018) runs unmodified C programs efficiently by overprovisioning the program with checkpoints to assure that the system makes progress, even with scarce energy. Approaches that are not based on the checkpointing technique have also been proposed. Chain (Colin and Lucia, 2016) utilizes a set of programmer-defined tasks that compute and exchange data through channels. Alpaca (Maeng et al., 2017) preserves execution progress at the granularity of a task by privatizing the shared data between tasks that are detected using idempotence analysis. Clank (Hicks, 2017) proposes a set of hardware buffers and memory access monitors that dynamically maintain idempotency. Several studies (Hester et al., 2015b; Colin et al., 2018) focus on the power management of batteryless systems.

Unlike the proposed intermittent learning system, none of these work considers how on-device machine learning can be performed effectively on harvested energy by considering the semantics of machine learning tasks. Several work propose sensing systems (Yerva et al., 2012; Sudevalayam and Kulkarni, 2011; Seah et al., 2009; Kansal and Srivastava, 2003) whose role is to sense data and forward them to other systems for further processing, but they do not perform on-device learning. Furthermore, these systems neither consider the utility of data nor provide any analysis of a system's expected task completion based on energy. Mayfly (Hester et al., 2017) considers the timeliness of data, but neither takes into account the usefulness of data nor provides any energy analysis. Some energy prediction models such as (Kansal et al., 2007) based on *Exponentially Weighted Moving-Average* filter (Cox, 1961) or *Weather-Conditioned Moving*

Average algorithm (Piorno et al., 2009) require complex models designed for specific energy (solar) harvesters. Both use conventional time-domain energy analysis and prediction techniques, which is difficult to make in practice while the proposed framework performs energy event-based analysis.

Embedded Machine Learning. Machine learning algorithms that run on low-performance processors have been studied. Bonsai (Kumar et al., 2017) develops a tree-based algorithm for efficient inference on IoT devices having limited resources (e.g., 2KB RAM and 32KB read-only flash). ProtoNN (Gupta et al., 2017) proposes compressed and accurate k-nearest neighbors algorithm for devices with limited storage. Deep neural networks have been implemented to run on embedded devices by reducing redundancy in their network model (Denil et al., 2013). Neural network compression techniques such as quantization and encoding (Han et al., 2015a), fixed-point number or binary representation (Courbariaux et al., 2015), HashedNets (Chen et al., 2015b), Sparse Neural Networks (Bourely et al., 2017), multiplications using shift and addition (Ding et al., 2017), vector quantization (Gong et al., 2014), circulant weight matrix (Kotagiri, 2014), and structured transform (Sindhwani et al., 2015) significantly reduce the size of neural network and run them on some high-performance embedded systems such as mobile devices.

Several hardware architectures have been introduced to surmount the computational limitation of embedded machine learning. (Lee and Verma, 2013) proposes a custom processor integrating a CPU with configurable accelerators for discriminative machine-learning functions. Mixed-signal circuits such as (Murmann et al., 2015) explore a variety of design techniques that are leveraged in the design of embedded ConvNet ASICs. In computer vision domain, a number of accelerators have been proposed for embedded systems, e.g., NeuFlow (a bio-inspired vision SoC) (Pham et al., 2012), ShiDianNao (Convolutional Neural Network within an SRAM) (Du et al., 2015), and a scalable non-von Neumann architecture (Merolla et al., 2014).

Machine Learning on Harvested Energy. Recently, an intermittent neural network *inference* system (Gobieski et al., 2018a,b) has been proposed. But these work are quite different from the proposed framework and is limited in several ways. For instance, they only execute an inference task (i.e., no on-device training), the task pipeline is fixed at compile time (i.e., no dynamic task adjustment), and the evaluation reads pre-loaded in-memory processed data (i.e., no real sensing). Whereas the proposed intermittent learning systems consider all aspects of a machine learning task (including on-device training), portions of these learning tasks (i.e., actions) are dynamically scheduled at run-time by the dynamic action planner, and our evaluation has multiple end-to-end real systems. There exist batteryless systems that are designed for specific applications, such as eye-tracking (Li and Zhou, 2018) and gesture recognition (Li et al., 2018b), that use a simple threshold-based CFAR algorithm. CapBand (Truong et al., 2018) combines two energy harvesters (solar and RF) to recognize hand gestures using a Convolutional Neural Network (CNN). Although these systems intermittently classify sensor data, their implementation is application-specific, and they neither consider the data and application level semantics of machine learning algorithms nor implement on-device training and adaptation.

7.9 Summary

A new paradigm called the intermittent learning for embedded systems that are powered by harvested energy is introduced. To learn and build up intelligence from harvested energy, a learning task is divided into actions such as sensing, selecting, learning, or inferring, and they are dynamically executed based on an algorithm which chooses the best action to execute that maximizes learning performance under the energy constraints. The proposed system not only optimizes the sequence of actions but also makes a decision which examples should be learned while considering their potential to improve the learning performance as well as the energy level. A programming model and development tool of intermittent learning are proposed based on the action-based model with which three example applications are implemented, i.e., air-quality monitoring, human presence detecting, and vibration learning systems. The evaluation results show that the learning tasks of the three applications are intermittently executed with both energy- and data-efficiency based on the dynamic action plan and the example selection heuristics.

CHAPTER 8: CONCLUSION

8.1 Summary of Results

Focusing on enabling deep intelligence on resource-constrained embedded systems, the results presented in this dissertation can be summarized as follows.

Weight Virtualization Algorithm for Fast and Scalable Deep Multitask Learning. In chapter 4, we proposed *in-memory multitask learning* based on the concept of *Neural Weight Virtualization* (Lee and Nirjon, 2020a) – which enables fast and scalable in-memory multitask deep learning on memory-constrained embedded intelligent systems. The goal of neural weight virtualization is two-fold: 1) packing multiple DNNs into a fixed-sized main memory whose combined memory requirement is larger than the main memory, and 2) enabling fast in-memory execution of the DNNs. To this end, we proposed a two-phase approach: 1) *virtualization of weight parameters* for fine-grained parameter sharing at the level of weights that scales up to multiple heterogeneous DNNs of arbitrary network architectures, and 2) in-memory data structure and run-time execution framework for *in-memory execution and context-switching* of DNN tasks.

We implemented two multitask learning systems: 1) an embedded GPU-based mobile robot, and 2) a microcontroller-based IoT device. We thoroughly evaluate the proposed algorithms as well as the two systems that involve ten state-of-the-art DNNs. Our evaluation showed that weight virtualization improves memory efficiency, execution time, and energy efficiency of the multitask learning systems by 4.1x, 36.9x, and 4.2x, respectively.

Real-Time Dynamic Sub-Network Construction and Execution. In chapter 5, we proposed *adaptive real-time learning* based on the concept of *SubFlow* (Lee and Nirjon, 2020c)—a dynamic adaptation and execution strategy for a deep neural network (DNN), which enables real-time DNN inference and training on embedded systems of limited computing resources. The goal of SubFlow is to complete the execution of a DNN task within a timing constraint, which may dynamically change while ensuring comparable performance to executing the full network by executing a subset of the DNN at run-time. To this end, we proposed two online algorithms that enable SubFlow: 1) *dynamic construction* of a sub-network which constructs the best sub-network of the DNN in terms of size and configuration, and 2) *time-bound execution* which executes the sub-network within a given time budget for both inference and training.

We implemented and open-sourced SubFlow by extending TensorFlow with full compatibility by adding SubFlow operations for convolutional and fully-connected layers of a DNN. We evaluated SubFlow with three popular DNN models (LeNet-5, AlexNet, and KWS), which shows that it provides flexible run-time execution and increases the utility of a DNN under dynamic timing constraints, e.g., 1x–6.7x range of execution times with average -3% of performance (inference accuracy) difference. We also implemented an autonomous robot as an example system that uses SubFlow and demonstrate that its obstacle detection DNN is flexibly executed to meet a range of deadlines that varies depending on its running speed.

Run-Time Performance Improvement with Zero Energy. In chapter 6, we proposed *opportunistic accelerated learning* based on the concept of *Neuro.ZERO* (Lee and Nirjon, 2019)—a co-processor architecture consisting of a main microcontroller (MCU) that executes scaled-down versions of a deep neural network[1] (DNN) inference task, and an accelerator microcontroller that is powered by harvested energy and follows the intermittent computing paradigm Lucia et al. (2017). The goal of the accelerator is to enhance the inference performance of the DNN that is running on the main microcontroller. Neuro.ZERO opportunistically accelerates the run-time performance of a DNN via one of its four acceleration modes: *extended inference*, *expedited inference*, *ensemble inference*, and *latent training*. To enable these modes, we proposed two sets

[1]The Deep Neural Network (DNN), by definition, refers to neural networks having more than one hidden layers Hanin (2017); Lu et al. (2017); Hornik (1991); Lee and Nirjon (2019). Thus, a wide variety of networks qualify as a DNN in the existing literature. DNNs considered in this study have up to 10^5 neurons and weights combined. They fit into 256KB memory of an MCU; have convolutional, ReLU, pooling, and fully-connected structures as regular DNNs; and perform on-device inference Gobieski et al. (2019a, 2018c).

of algorithms: 1) *energy and intermittence-aware DNN inference and training algorithms*, and 2) *a fast and high-precision adaptive fixed-point arithmetic* that beats existing floating-point and fixed-point arithmetic in terms of speed and precision, respectively, and achieves the best of both.

We implemented low-power image and audio recognition applications and demonstrate that their inference speedup increases by 1.6× and 1.7×, respectively, and the inference accuracy increases by 10% and 16%, respectively, when compared to battery-powered single-MCU systems.

On-Device Machine Learning on Intermittently Powered Systems. In chapter 7, we proposed *energy-aware intermittent learning* (Lee et al., 2019) that makes energy-harvested batteryless systems capable of executing lightweight machine learning tasks intermittently based on the availability of harvested energy. The notion of intermittent learning is similar to the intermittent computing paradigm with the primary difference that the program that runs on the microcontroller executes a machine learning task—involving both *training* and *inferring*.

To complement and advance the state-of-the-art of the batteryless machine learning systems, we proposed the intermittent learning framework which explicitly takes into account the dynamics of a machine learning task, in order to improve the energy and learning efficiency of an intermittent learner in a systemic fashion. The fundamental difference between the proposed framework and the existing literature is that, besides *improving the efficiency of on-device inference*, the intermittent learning framework enables *on-device training* to improve the effectiveness and accuracy of the learner over time.

8.2 Looking into the Future

This dissertation has just made the first step towards Embedded Artificial Intelligence (EAI) by tackling a few of the challenges of employing artificial intelligence technologies on resource-constrained embedded systems. Based on that, the dimensionality of state-of-the-art EAI will be extended by 1) enabling intelligent systems to adapt to the new learning environment on the device, 2) exploring new learning paradigm better suited to resource-constrained embedded systems beyond deep learning, and 3) performing smarter learning via physical interaction as cyber-physical intelligent systems. The followings provide an overview of each research plan.

8.2.1 On-Device Adaptation to Non-Stationary Environment

Problem Statement. In the real world, *the learning conditions in which we use intelligent systems will differ from the conditions in which they were trained.* Especially, environments are non-stationary, and sometimes the difficulties of matching the development scenario to the use are too significant or too costly. This learning environment mismatch is called *dataset shift*. An example of a dataset shift is a face recognition algorithm trained predominantly on younger faces, yet the real dataset has a much larger proportion of older faces. Unfortunately, mainstream machine learning methods work by ignoring these differences and presuming that the real environment and training environment match.

In this research, the following research questions will be asked about *how to adapt an embedded intelligent system on the device to an ever-evolving learning environment* when the possibility of dataset shift is allowed (Figure 8.1), which happens to many practical, intelligent systems deployed in the wild.

• *Dataset Shift Detection*: When to perform the on-device adaptation to the environment?

• *Efficient Adaptation*: How to enable the on-device adaptation using limited resources of the system?

• *Learning Example Selection*: What (which) online learning examples to learn for on-device adaptation?

Research Direction. This problem can be formulated as *an on-device dataset adaptation problem* and answer those three research questions by updating the learning model of the system as described in the following.

• *Dataset Shift Detection*: The model adaptation is initiated *when the feature distribution of real examples becomes different from that of train examples*, and the difference is greater than the level where re-training of the current model is required due to the potential performance decrease

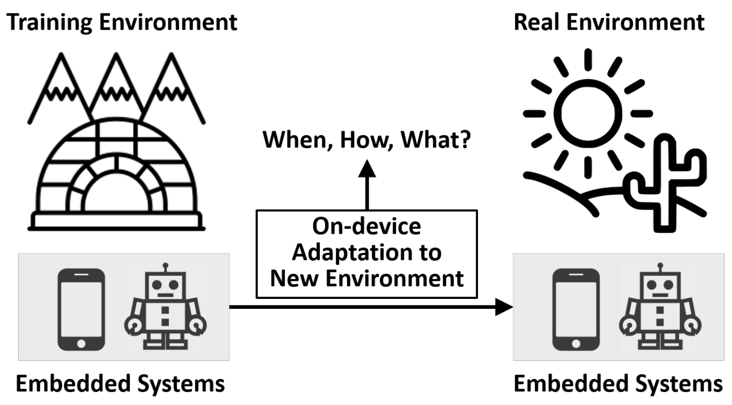


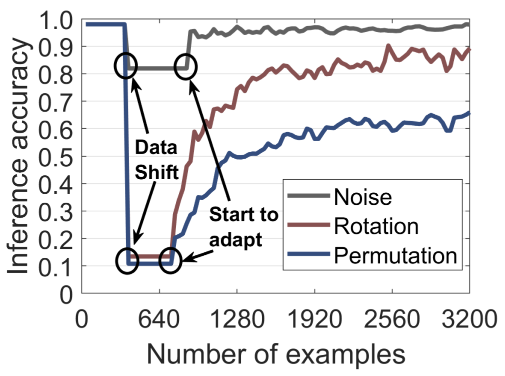
Figure 8.1: Embedded systems adapt to the new environment via on-device adaptation.

caused by a dataset shift. After a few iterations of re-training, the adaptation is terminated when the distribution becomes stabilized.

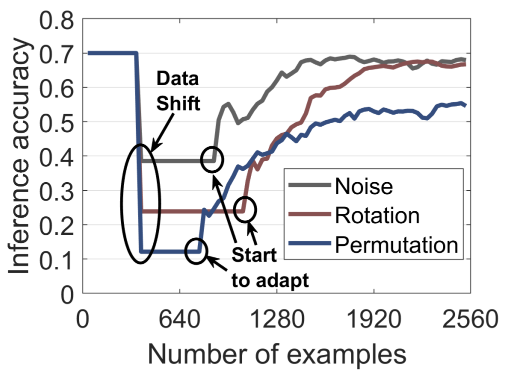
• *Efficient Adaptation*: Only the last layer's weight parameters are updated via re-training of the model, *similar to the transfer learning, but without back-propagation* that is computationally too expensive for many embedded systems (Ng, 2016; Pratt, 1993). The lightweight on-device dataset adaptation (re-training of the model) becomes possible by utilizing the computational outputs of feed-forward execution for inference. Also, the re-training effect of the model for each example is maximized by weighting the importance of the loss for every single example constituting a re-training batch, which works as a dynamic learning rate.

• *Learning Example Selection*: Learning examples are selected in two steps. First, each incoming example's learning utility is examined based on their uncertainty, which is measure by their gradient norm without requiring ground-truth labels. Then, only the labels of input examples with high learning utility are obtained as active learning based on diversity criteria to compose a re-training batch.

Preliminary Result. Figure 8.2 shows the on-device adaptation of MNIST and CIFAR-10 datasets on MSP430 microcontroller, where examples of different datasets come to the system, causing three dataset shifts (noise, rotation, and permutation). It shows how inference accuracy changes over the data examples by the proposed adaptation. When the data is shifted, the accuracy drops and stays the same until the adaptation is initiated. Once the shift is detected, the deep



(a) MNIST



(b) CIFAR-10

Figure 8.2: The end-to-end on-device adaptation (inference accuracy) on MNIST and CIFAR-10.

model is successfully adapted to the new data to recover the inference accuracy with examples of new dataset (Lee and Nirjon, 2020b).

8.2.2 Next Learning Model Beyond Deep Learning

Problem Statement. While Deep Neural Networks (DNNs) have started to run on embedded systems, their lack of decomposability into understandable components makes them hard to interpret. Although many works such as model distillation and feature importance tried to interpret DNNs, they still remain as black boxes in most cases, limiting their deployment on real systems. *Their inexplicability is impeding the deployment of today's high-performing DNNs on embedded systems of limited resources*, which usually requires to customize a large size of DNNs until they become executable on embedded systems. However, since the knowledge of how DNNs work is missing, such DNN manipulation is usually conducted with trial-and-error approaches that involve multiple iterations of re-training, e.g., compression or pruning techniques. Moreover, such heuristic methods not based on the analytical understanding of DNNs are prone to deform DNNs inadvertently, resulting in unpredictable behavior of the DNNs.

In this research, the following research questions will be asked about *a new learning model that can approximate a black-box DNN to an understandable computer program* consisting of functions written in high-level programming languages, which is optimized to run on resource-constrained embedded systems.

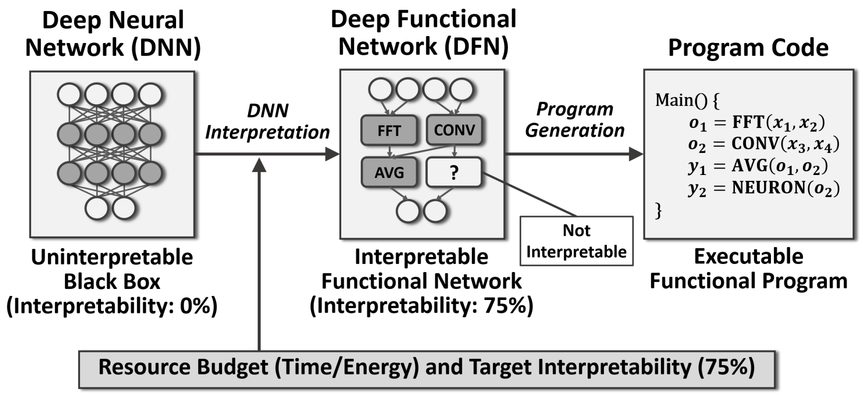

Deep Neural Network (DNN)
Uninterpretable Black Box (Interpretability: 0%)
DNN Interpretation
Deep Functional Network (DFN)
FFT
CONV
AVG
?
Not Interpretable
Interpretable Functional Network (Interpretability: 75%)
Program Generation
Program Code
Main() {
o_1 = FFT(x_1, x_2)
o_2 = CONV(x_3, x_4)
y_1 = AVG(o_1, o_2)
y_2 = NEURON(o_2)
}
Executable Functional Program
Resource Budget (Time/Energy) and Target Interpretability (75%)

Figure 8.3: A DFN approximates a black-box DNN into human-understandable functional program easy to deploy on resource-constrained embedded systems.

• *Learning Model*: What kind of learning model should be an alternative to today's DNNs?

• *Representation*: In what form, the new learning model should be represented?

• *Training (Learning)*: How to efficiently as well as effectively train (or learn) the new learning model?

Research Direction. We propose *Deep Functional Network (DFN)* that approximates a DNN to a *human-understandable computer code consisting of programming functions* written in high-level languages. It would be an alternative to many inexplicable, massive, and rigid DNNs that are onerous to run on resource-constrained embedded systems. Based on the explainability of the problem-solving process, which allows a precise description of the solution, *a well-interpreted DFN will achieve the same goal of the DNN with better resource efficiency*. Also, modular functions and their data flow presented in a DFN will enable flexible and analytical optimization based on the system requirements, e.g., execution time, energy, or memory.

• *Learning Model*: Our new learning model takes two steps to generate a DFN from a DNN. First, it searches for a set of functions representing independent algorithms (e.g., AVG, FFT), which are expected to be required by the final program via the process called *function estimation*. Second, it finds the data flow between the functions and builds a network architecture of DFN in the form of a DAG, which achieves the task of DNN, under the run-time constraints of the system via the process called *network formation*.

• *Representation*: A DFN is represented in the form of a *directed acyclic graph (DAG)* by using *functional programming*, which interprets the DNN as a tree consisting of a set of functions (vertices) and their connections (edges). The functional programming of the DNN makes it easier to understand by allowing function definitions to be trees of expressions that each return an output based on the declarative programming paradigm. Figure 8.3 shows a diagram of the proposed DFN in which a DNN is first interpreted into a DFN with four functions (e.g., FFT, CONV, AVG) and then converted into an executable program under the resource budget of the target system.

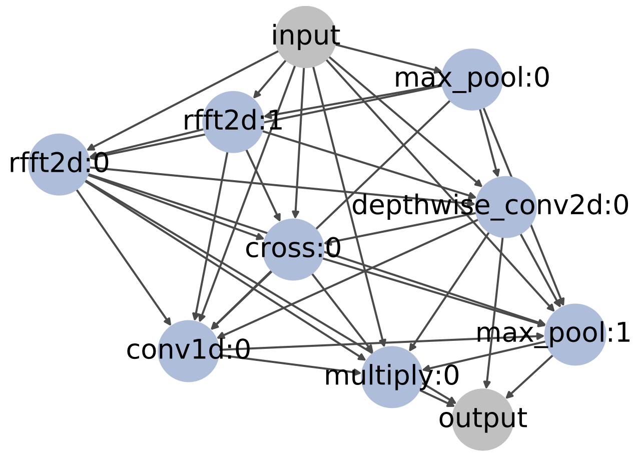


Figure 8.4: An example of DFN (ResNet/MNIST)

• *Training (Learning)*: A DFN is generated (trained) by approximating the DNN such that it provides the same output through the similar inner workings of the DNN. The internal behavior of the DFN becomes similar to that of the DNN by *enforcing the derivatives of the DFN output w.r.t. the input to be identical to that of the DNN* during the solution search. It is different from the curve-fitting that finds the best fit to input/output data points without taking into account how the solution gets to the answer.

Preliminary Result. Figure 8.4 shows the DFN interpreted from the ResNet-152 DNN trained on MNIST. It shows a promising possibility that many state-of-the-art DNNs performing different tasks can be successfully interpreted into DFNs. The DFN achieves comparable classification accuracy on MNIST to ResNet-152, i.e., 97% vs. 99%, with 1,235x memory efficiency.

8.2.3 Smarter Learning via Physical Interaction

Problem Statement. In general, *AI algorithms are applied to the data only obtained from the given circumstances due to their inability to involve with the physical environment*. Since they usually do not directly engage with or influence the physical world, they cannot actively seek or create a favorable learning environment where their learning objective can be maximized. Although reinforcement learning considers the consequences of an agent's action to the environment, it assumes that the environment is an external factor that the agent cannot change and thus does not try to create or find a better learning environment. In contrast, *many embedded systems are capable of sensing the physical world and taking physical actions via actuation* (Lee and Nirjon, 2018), e.g., a mobile robot can sense through a camera and use its wheels to move around. Thus, such sensing- and actuation-capable intelligent systems can proactively seek and change its surroundings to better achieve their learning objective, instead of learning in a given environment passively.

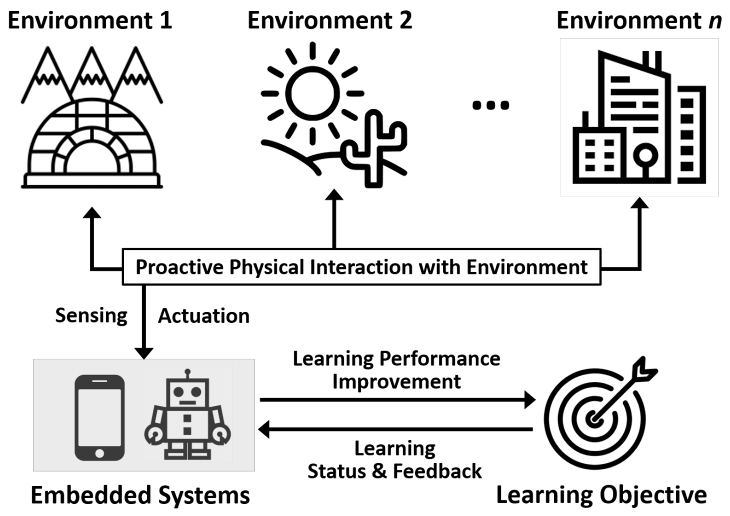

Environment 1
Environment 2
Environment *n*
...
Proactive Physical Interaction with Environment
Sensing
Actuation
Learning Performance Improvement
Learning Status & Feedback
Embedded Systems
Learning Objective

Figure 8.5: Smart learning of embedded intelligent systems via physical interaction (i.e., sensing and actuation) given heterogeneous environments.

In this research, the following research questions will be asked about if intelligent systems can improve their learning performance by actively sensing the physical world and taking actions via actuation based on the concept called *Cyber-Physical Intelligent Systems (CPIS)*, unlike the conventional AI approaches that do not involve with physical processes.

• *Finding Best Condition*: What is the optimal physical condition maximizing the learning objective?

• *Optimal Action Planning*: What actions should be taken to promote the best physical condition?

• *Balance between Activeness and Passiveness*: How much and when the system should perform physical interaction with the environment?

Research Direction. We propose smarter learning of intelligent systems via *physical interaction (i.e., sensing and actuation) given heterogeneous environments, which enhances the system's learning objective* (Figure 8.5). By proactively finding the best learning environment via active sensing and promoting such an environment to the system via actuation, the system's learning performance is improved. To this end, we propose *learning condition exploration, a dynamic physical-intelligent action planner, and a conditional action trigger*, as described in the following.

• *Finding Best Condition*: The best physical environment is defined as *the learning condition that provides useful learning data of high utility, improving the system's learning performance*. To find such an environment, the system first senses various environments of different learning conditions and then performs actuation to bring the best one to the system. we propose *learning condition exploration* based on reinforcement learning, which enables efficient exploration of the environments of various learning conditions by building statistics of learning conditions. For example, a mobile robot that has fully learned a particular type of data in one place can search for other sites and move to the next best place where a different kind of data that can enhance its learning performance is available.

• *Optimal Action Planning*: Since an intelligent system capable of physical actuation performs multiple heterogeneous actions such as sensing, feature extracting, inferring, rotating motors, rolling the wheels, etc., the search space of optimal actions not only increases exponentially but also involves with both physical and learning-related actions. To solve this problem, we propose a *dynamic physical-intelligent action planner* that determines a set of actions the system needs to take at run-time by taking account of two orthogonal types of actions simultaneously, i.e., physical and learning-related actions, similar to (Lee et al., 2019). To make a feasible action plan, the decision horizon is limited to finite next steps in which the locally optimal solution is searched instead of the globally optimal solution.

• *Balance between Activeness and Passiveness*: Since there is a risk that physical actions fail to promote the desired environment due to various reasons such as the insufficient physical capability of the system or changing environment, we propose to invoke active actions only when it is expected to promote the desired learning condition, namely *the conditional action trigger*. It also considers the trade-off between physical action-induced active learning and passive learning with no physical involvement, such as increased energy consumption of actuation or miss of important learning data due to the environment change.

REFERENCES

Abadi, M., Barham, P., Chen, J., Chen, Z., Davis, A., Dean, J., Devin, M., Ghemawat, S., Irving, G., Isard, M., et al. (2016). Tensorflow: A system for large-scale machine learning. In *12th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 16)*, pages 265–283.

Abbasi-Asl, R. and Yu, B. (2017). Structural compression of convolutional neural networks based on greedy filter pruning. *arXiv preprint arXiv:1705.07356*.

Administration, N. H. T. S. (2013). Traffic safety facts. `https://crashstats.nhtsa.dot.gov/Api/Public/ViewPublication/812124`.

Agarwal, A., Gerber, S., and Daume, H. (2010). Learning multiple tasks using manifold regularization. In *Advances in neural information processing systems*, pages 46–54.

Akhtar, F. and Rehmani, M. H. (2015). Energy replenishment using renewable and traditional energy resources for sustainable wireless sensor networks: A review. *Renewable and Sustainable Energy Reviews*, 45:769–784.

Al-Qizwini, M., Barjasteh, I., Al-Qassab, H., and Radha, H. (2017). Deep learning algorithm for autonomous driving using googlenet. In *2017 IEEE Intelligent Vehicles Symposium (IV)*, pages 89–96. IEEE.

Amari, S.-i. (1997). Neural learning in structured parameter spaces-natural riemannian gradient. In *Advances in neural information processing systems*, pages 127–133.

Amari, S.-I. (1998). Natural gradient works efficiently in learning. *Neural computation*, 10(2):251–276.

Amari, S.-i., Kurata, K., and Nagaoka, H. (1992). Information geometry of boltzmann machines. *IEEE Transactions on neural networks*, 3(2):260–271.

Amazon (2019). Introducing alexa voice service integration for aws iot core, a new way to cost-effectively bring alexa voice to any type of connected device. `https://aws.amazon.com/blogs/iot/introducing-alexa-voice-service-integration-for-aws-iot-core/`.

Anderson, S., Earle, J., Goldschmidt, R. E., and Powers, D. (1967). The ibm system/360 model 91: floating-point execution unit. *IBM Journal of research and development*, 11(1):34–53.

Anwar, S., Hwang, K., and Sung, W. (2015). Fixed point optimization of deep convolutional neural networks for object recognition. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, pages 1131–1135. IEEE.

Apple (2017). Neural engine. `https://www.apple.com/iphone-xs/a12-bionic/`.

Arfken, G. B. and Weber, H. J. (1999). Mathematical methods for physicists.

Arm (2013). big.little technology. `https://www.arm.com/files/pdf/big_LITTLE_Technology_the_Futue_of_Mobile.pdf`.

Attoh-Okine, N. O. (1999). Analysis of learning rate and momentum term in backpropagation neural network algorithm trained to predict pavement performance. *Advances in Engineering Software*, 30(4):291–302.

Audet, D., De Oliveira, L. C., MacMillan, N., Marinakis, D., and Wu, K. (2011). Scheduling recurring tasks in energy harvesting sensors. In *2011 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*, pages 277–282. IEEE.

Augonnet, C., Thibault, S., and Namyst, R. (2010). *StarPU: a runtime system for scheduling tasks over accelerator-based multicore machines*. PhD thesis, INRIA.

Aytar, Y., Vondrick, C., and Torralba, A. (2017). See, hear, and read: Deep aligned representations. *arXiv preprint arXiv:1706.00932*.

Aziz, B. and Hamilton, G. (2009). Detecting man-in-the-middle attacks by precise timing. In *Emerging Security Information, Systems and Technologies, 2009. SECURWARE'09. Third International Conference on*, pages 81–86. Ieee.

Bachman, D. (2007). *Advanced calculus demystified*. McGrawHill.

Badreddin, E. (1992). Obstacle avoidance using tactile sensing for an autonomous mobile robot. *IFAC Proceedings Volumes*, 25(29):325–329.

Baknina, A. and Ulukus, S. (2017). Online scheduling for energy harvesting channels with processing costs. *IEEE Transactions on Green Communications and Networking*, 1(3):281–293.

Balog, H., Salmon, D., DeVries, P., Saks, M., Jansen, B., and Arnison, S. (2002). Dynamic protocol selection and routing of content to mobile devices. US Patent App. 09/823,654.

Balsamo, D., Weddell, A. S., Das, A., Arreola, A. R., Brunelli, D., Al-Hashimi, B. M., Merrett, G. V., and Benini, L. (2016). Hibernus++: a self-calibrating and adaptive system for transiently-powered embedded devices. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(12):1968–1980.

Balsamo, D., Weddell, A. S., Merrett, G. V., Al-Hashimi, B. M., Brunelli, D., and Benini, L. (2015). Hibernus: Sustaining computation during intermittent supply for energy-harvesting systems. *IEEE Embedded Systems Letters*, 7(1):15–18.

Bariya, M., Nyein, H. Y. Y., and Javey, A. (2018). Wearable sweat sensors. *Nature Electronics*, 1(3):160–171.

Bartel, J. (2011). Non-preemptive multitasking. *The Computer Journal*, 30:37–39.

Barzilai, A. and Crammer, K. (2015). Convex multi-task learning by clustering. In *Artificial Intelligence and Statistics*, pages 65–73.

Baxter, J. (1997). A bayesian/information theoretic model of learning to learn via multiple task sampling. *Machine learning*, 28(1):7–39.

Bengio, Y. (2012). Practical recommendations for gradient-based training of deep architectures. In *Neural networks: Tricks of the trade*, pages 437–478. Springer.

Berger, A. (2001). *Embedded systems design: an introduction to processes, tools, and techniques*. CRC Press.

Bhattacharya, S. and Lane, N. D. (2016). 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*, pages 176–189. ACM.

Billinghurst, M. and Starner, T. (1999). Wearable devices: new ways to manage information. *Computer*, 32(1):57–64.

Bohren, J., Rusu, R. B., Jones, E. G., Marder-Eppstein, E., Pantofaru, C., Wise, M., Mösenlechner, L., Meeussen, W., and Holzer, S. (2011). Towards autonomous robotic butlers: Lessons learned with the pr2. In *2011 IEEE International Conference on Robotics and Automation*, pages 5568–5575. IEEE.

Bojarski, M., Del Testa, D., Dworakowski, D., Firner, B., Flepp, B., Goyal, P., Jackel, L. D., Monfort, M., Muller, U., Zhang, J., et al. (2016). End to end learning for self-driving cars. *arXiv preprint arXiv:1604.07316*.

Bojarski, M., Yeres, P., Choromanska, A., Choromanski, K., Firner, B., Jackel, L., and Muller, U. (2017). Explaining how a deep neural network trained with end-to-end learning steers a car. *arXiv preprint arXiv:1704.07911*.

Bourely, A., Boueri, J. P., and Choromonski, K. (2017). Sparse neural networks topologies. *arXiv preprint arXiv:1706.05683*.

Brown, I. and Mues, C. (2012). An experimental comparison of classification algorithms for imbalanced credit scoring data sets. *Expert Systems with Applications*, 39(3):3446–3453.

Buck, D. A. (1952). Ferroelectrics for digital information storage and switching. Technical report, MASSACHUSETTS INST OF TECH CAMBRIDGE DIGITAL COMPUTER LAB.

Buettner, M., Greenstein, B., and Wetherall, D. (2011). Dewdrop: an energy-aware runtime for computational rfid. In *Proc. USENIX NSDI*, pages 197–210.

Buettner, M., Prasad, R., Philipose, M., and Wetherall, D. (2009). Recognizing daily activities with rfid-based sensors. In *Proceedings of the 11th international conference on Ubiquitous computing*, pages 51–60. ACM.

Cambria, E. and White, B. (2014). Jumping nlp curves: A review of natural language processing research. *IEEE Computational intelligence magazine*, 9(2):48–57.

Cao, Q., Shen, L., Xie, W., Parkhi, O. M., and Zisserman, A. (2018). Vggface2: A dataset for recognising faces across pose and age. In *2018 13th IEEE International Conference on Automatic Face & Gesture Recognition (FG 2018)*, pages 67–74. IEEE.

Caruana, R. (1997). Multitask learning. *Machine learning*, 28(1):41–75.

Cavigelli, L., Magno, M., and Benini, L. (2015). Accelerating real-time embedded scene labeling with convolutional networks. In *Proceedings of the 52nd Annual Design Automation Conference*, page 108. ACM.

Chakravarty, P., Kelchtermans, K., Roussel, T., Wellens, S., Tuytelaars, T., and Van Eycken, L. (2017). Cnn-based single image obstacle avoidance on a quadrotor. In *2017 IEEE International Conference on Robotics and Automation (ICRA)*, pages 6369–6374. IEEE.

Chapelle, O., Scholkopf, B., and Zien, A. (2009). Semi-supervised learning (chapelle, o. et al., eds.; 2006)[book reviews]. *IEEE Transactions on Neural Networks*, 20(3):542–542.

Chauhan, J., Seneviratne, S., Hu, Y., Misra, A., Seneviratne, A., and Lee, Y. (2018). Breathing-based authentication on resource-constrained iot devices using recurrent neural networks. *Computer*, 51(5):60–67.

Chen, C., Seff, A., Kornhauser, A., and Xiao, J. (2015a). Deepdriving: Learning affordance for direct perception in autonomous driving. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2722–2730.

Chen, G., Choi, W., Yu, X., Han, T., and Chandraker, M. (2017a). Learning efficient object detection models with knowledge distillation. In *Advances in Neural Information Processing Systems*, pages 742–751.

Chen, G., Parada, C., and Heigold, G. (2014a). Small-footprint keyword spotting using deep neural networks. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 4087–4091. IEEE.

Chen, J., Tang, L., Liu, J., and Ye, J. (2009). A convex formulation for learning shared structures from multiple tasks. In *Proceedings of the 26th Annual International Conference on Machine Learning*, pages 137–144. ACM.

Chen, L.-B., Chang, W.-J., Su, J.-P., Ciou, J.-Y., Ciou, Y.-J., Kuo, C.-C., and Li, K. S.-M. (2016a). A wearable-glasses-based drowsiness-fatigue-detection system for improving road safety. In *2016 IEEE 5th Global Conference on Consumer Electronics*, pages 1–2. IEEE.

Chen, L.-H. and Lu, H.-W. (2007). An extended assignment problem considering multiple inputs and outputs. *Applied Mathematical Modelling*, 31(10):2239–2248.

Chen, P., Dang, Y., Liang, R., Zhu, W., and He, X. (2017b). Real-time object tracking on a drone with multi-inertial sensing data. *IEEE Transactions on Intelligent Transportation Systems*, 19(1):131–139.

Chen, T., Du, Z., Sun, N., Wang, J., Wu, C., Chen, Y., and Temam, O. (2014b). Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning. *ACM Sigplan Notices*, 49(4):269–284.

Chen, W., Wilson, J., Tyree, S., Weinberger, K., and Chen, Y. (2015b). Compressing neural networks with the hashing trick. In *International Conference on Machine Learning*, pages 2285–2294.

Chen, X. (2018). Escoin: Efficient Sparse Convolutional Neural Network Inference on GPUs. *arXiv e-prints*, page arXiv:1802.10280.

Chen, X., Kundu, K., Zhang, Z., Ma, H., Fidler, S., and Urtasun, R. (2016b). Monocular 3d object detection for autonomous driving. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2147–2156.

Chen, X., Ma, H., Wan, J., Li, B., and Xia, T. (2017c). Multi-view 3d object detection network for autonomous driving. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1907–1915.

Chen, Z. and Liu, B. (2016). Lifelong machine learning. *Synthesis Lectures on Artificial Intelligence and Machine Learning*, 10(3):1–145.

Cheng, S.-T. and Agrawala, A. K. (1995). Allocation and scheduling of real-time periodic tasks with relative timing constraints. In *Proceedings Second International Workshop on Real-Time Computing Systems and Applications*, pages 210–217. IEEE.

Cheng, Y., Wang, D., Zhou, P., and Zhang, T. (2017). A survey of model compression and acceleration for deep neural networks. *arXiv preprint arXiv:1710.09282*.

Cheng, Y., Yu, F. X., Feris, R. S., Kumar, S., Choudhary, A., and Chang, S.-F. (2015). An exploration of parameter redundancy in deep networks with circulant projections. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2857–2865.

Chetto, M. and El Ghor, H. (2019). Scheduling and power management in energy harvesting computing systems with real-time constraints. *Journal of Systems Architecture*.

Chou, Y.-M., Chan, Y.-M., Lee, J.-H., Chiu, C.-Y., and Chen, C.-S. (2018a). Merging deep neural networks for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 1686–1694.

Chou, Y.-M., Chan, Y.-M., Lee, J.-H., Chiu, C.-Y., and Chen, C.-S. (2018b). Unifying and merging well-trained deep neural networks for inference stage. *arXiv preprint arXiv:1805.04980*.

Chowdhury, N. M. K. and Boutaba, R. (2010). A survey of network virtualization. *Computer Networks*, 54(5):862–876.

Chung, J.-Y., Liu, J. W.-S., and Lin, K.-J. (1990). Scheduling periodic jobs that allow imprecise results. *IEEE transactions on computers*, 39(9):1156–1174.

Cola, G., Avvenuti, M., Vecchio, A., Yang, G.-Z., and Lo, B. (2015). An on-node processing approach for anomaly detection in gait. *IEEE Sensors Journal*, 15(11):6640–6649.

Colin, A. and Lucia, B. (2016). Chain: tasks and channels for reliable intermittent programs. *ACM SIGPLAN Notices*, 51(10):514–530.

Colin, A. and Lucia, B. (2018). Termination checking and task decomposition for task-based intermittent programs. In *Proceedings of the 27th International Conference on Compiler Construction*, pages 116–127. ACM.

Colin, A., Ruppel, E., and Lucia, B. (2018). A reconfigurable energy storage architecture for energy-harvesting devices. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 767–781. ACM.

Cook, S. (2012). *CUDA programming: a developer's guide to parallel computing with GPUs*. Newnes.

Corporation, M. T. (2017). Ppa products datasheet & user manual.

Cortex, A. (2004). M3 processor. *Cortex-M Series*.

Courbariaux, M., Bengio, Y., and David, J.-P. (2014). Training deep neural networks with low precision multiplications. *arXiv preprint arXiv:1412.7024*.

Courbariaux, M., Bengio, Y., and David, J.-P. (2015). Binaryconnect: Training deep neural networks with binary weights during propagations. In *Advances in neural information processing systems*, pages 3123–3131.

Courbariaux, M., Hubara, I., Soudry, D., El-Yaniv, R., and Bengio, Y. (2016). Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1. *arXiv preprint arXiv:1602.02830*.

Cox, D. R. (1961). Prediction by exponentially weighted moving averages and related methods. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 414–422.

Cypress (2017). Cy15b104q. `http://www.cypress.com/file/209146/download`.

Davis, C. (1962). The norm of the schur product operation. *Numerische Mathematik*, 4(1):343–344.

de Bruijn, N. G. (1975). *Acknowledgement of priority to C. Flye Sainte-Marie on the counting of circular arrangements of 2n zeros and ones that show each n-letter word exactly once*. Department of Mathematics, Technological University.

Deng, J., Dong, W., Socher, R., Li, L.-J., Li, K., and Fei-Fei, L. (2009). ImageNet: A Large-Scale Hierarchical Image Database. In *CVPR09*.

Deng, L. (2014). A tutorial survey of architectures, algorithms, and applications for deep learning. *APSIPA Transactions on Signal and Information Processing*, 3.

Deng, L., Hinton, G., and Kingsbury, B. (2013). New types of deep neural network learning for speech recognition and related applications: An overview. In *Acoustics, Speech and Signal Processing (ICASSP), 2013 IEEE International Conference on*, pages 8599–8603. IEEE.

Deng, L. and Liu, Y. (2018). *Deep Learning in Natural Language Processing*. Springer.

Deng, S., Zhao, H., Yin, J., Dustdar, S., and Zomaya, A. Y. (2019). Edge intelligence: the confluence of edge computing and artificial intelligence. *arXiv preprint arXiv:1909.00560*.

Denil, M., Shakibi, B., Dinh, L., Ranzato, M., and De Freitas, N. (2013). Predicting parameters in deep learning. In *Advances in neural information processing systems*, pages 2148–2156.

Diestel, R. (2006). Graph theory (graduate texts in mathematics). 3rd. *Ed Springer*, pages 17–18.

Ding, R., Liu, Z., Shi, R., Marculescu, D., and Blanton, R. (2017). Lightnn: Filling the gap between conventional deep neural networks and binarized networks. In *Proceedings of the on Great Lakes Symposium on VLSI 2017*, pages 35–40. ACM.

Dong, X., Chen, S., and Pan, S. (2017a). Learning to prune deep neural networks via layer-wise optimal brain surgeon. In *Advances in Neural Information Processing Systems*, pages 4857–4867.

Dong, Z., Liu, Y., Zhou, H., Xiao, X., Gu, Y., Zhang, L., and Liu, C. (2017b). An energy-efficient offloading framework with predictable temporal correctness. In *Proceedings of the Second ACM/IEEE Symposium on Edge Computing*, page 19. ACM.

Draghici, S. (2002). On the capabilities of neural networks using limited precision weights. *Neural networks*, 15(3):395–414.

Drews, P., Williams, G., Goldfain, B., Theodorou, E. A., and Rehg, J. M. (2017). Aggressive deep driving: Model predictive control with a cnn cost model. *arXiv preprint arXiv:1707.05303*.

Du, Z., Fasthuber, R., Chen, T., Ienne, P., Li, L., Luo, T., Feng, X., Chen, Y., and Temam, O. (2015). Shidiannao: Shifting vision processing closer to the sensor. In *ACM SIGARCH Computer Architecture News*, volume 43, pages 92–104. ACM.

Dundar, A., Jin, J., Martini, B., and Culurciello, E. (2017). Embedded streaming deep neural networks accelerator with applications. *IEEE transactions on neural networks and learning systems*, 28(7):1572–1583.

Duong, L., Cohn, T., Bird, S., and Cook, P. (2015). Low resource dependency parsing: Cross-lingual parameter sharing in a neural network parser. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers)*, pages 845–850.

Ecma International (2017). Standard ecma-404 the json data interchange syntax. `https://www.ecma-international.org/publications/standards/Ecma-404.htm`.

Eigen, D., Puhrsch, C., and Fergus, R. (2014). Depth map prediction from a single image using a multi-scale deep network. In *Advances in neural information processing systems*, pages 2366–2374.

Elliott, G. A. and Anderson, J. H. (2012). Robust real-time multiprocessor interrupt handling motivated by gpus. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 267–276. IEEE.

Elliott, G. A. and Anderson, J. H. (2013). An optimal k-exclusion real-time locking protocol motivated by multi-gpu systems. *Real-Time Systems*, 49(2):140–170.

Elliott, G. A. and Anderson, J. H. (2014). Exploring the multitude of real-time multi-gpu configurations. In *2014 IEEE Real-Time Systems Symposium*, pages 260–271. IEEE.

Elliott, G. A., Ward, B. C., and Anderson, J. H. (2013). Gpusync: A framework for real-time gpu management. In *2013 IEEE 34th Real-Time Systems Symposium*, pages 33–44. IEEE.

Embedded Intelligence Lab (UNC Chapel Hill) (2019a). Intermittent learning open source project.

Embedded Intelligence Lab (UNC Chapel Hill) (2019b). Neuro.zero open source project. `https://github.com/learning1234embed/Neuro.ZERO`.

Ericsson (2019). Iot connections outlook: Nb-iot and cat-m technologies will account for close to 45 percent of cellular iot connections in 2024.

Eskin, E., Smola, A. J., and Vishwanathan, S. (2004). Laplace propagation. In *Advances in Neural Information Processing Systems*, pages 441–448.

Fang, B., Zeng, X., and Zhang, M. (2018). Nestdnn: Resource-aware multi-tenant on-device deep learning for continuous mobile vision. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 115–127.

Farabet, C., Martini, B., Corda, B., Akselrod, P., Culurciello, E., and LeCun, Y. (2011). Neuflow: A runtime reconfigurable dataflow processor for vision. In *Computer Vision and Pattern Recognition Workshops (CVPRW), 2011 IEEE Computer Society Conference on*, pages 109–116. IEEE.

Fohler, G. (1995). Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. In *Proceedings 16th IEEE Real-Time Systems Symposium*, pages 152–161. IEEE.

Fohler, G. (1997). Dynamic timing constraints-relaxing over-constraining specifications of real-time systems. In *Proceedings of Work-in-Progress Session, 18th IEEE Real-Time Systems Symposium, December*.

Foundation, N. S. (2019). Real-time machine learning (rtml). `https://www.nsf.gov/pubs/2019/nsf19566/nsf19566.htm?WT.mc_id=USNSF_25&WT.mc_ev=click`.

French, R. M. (1999). Catastrophic forgetting in connectionist networks. *Trends in cognitive sciences*, 3(4):128–135.

Gao, H., Cheng, B., Wang, J., Li, K., Zhao, J., and Li, D. (2018). Object classification using cnn-based fusion of vision and lidar in autonomous vehicle environment. *IEEE Transactions on Industrial Informatics*, 14(9):4224–4231.

Gartner, Inc. (2016). Forecast: Internet of things – endpoints and associated services.

Gates, B. (2007). A robot in every home. *Scientific American*, 296(1):58–65.

Gerber, R., Hong, S., and Saksena, M. (1995a). Guaranteeing real-time requirements with resource-based calibration of periodic processes. *IEEE Transactions on Software Engineering*, 21(7):579–592.

Gerber, R., Pugh, W., and Saksena, M. (1995b). Parametric dispatching of hard real-time tasks. *IEEE transactions on computers*, 44(3):471–479.

Gers, F. A., Schmidhuber, J., and Cummins, F. (1999). Learning to forget: Continual prediction with lstm.

Giladi, N. and Fahn, S. (1998). Freezing phenomenon, the fifth cardinal sign of parkinsonism. In *Progress in Alzheimer's and Parkinson's Diseases*, pages 329–335. Springer.

Giusti, A., Guzzi, J., Cireşan, D. C., He, F.-L., Rodríguez, J. P., Fontana, F., Faessler, M., Forster, C., Schmidhuber, J., Di Caro, G., et al. (2015). A machine learning approach to visual perception of forest trails for mobile robots. *IEEE Robotics and Automation Letters*, 1(2):661–667.

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256.

Glorot, X., Bordes, A., and Bengio, Y. (2011). Deep sparse rectifier neural networks. In *Proceedings of the fourteenth international conference on artificial intelligence and statistics*, pages 315–323.

Gobieski, G., Beckmann, N., and Lucia, B. (2018a). Intelligence beyond the edge: Inference on intermittent embedded systems. *arXiv preprint arXiv:1810.07751*.

Gobieski, G., Beckmann, N., and Lucia, B. (2018b). Intermittent deep neural network inference. *SysML*.

Gobieski, G., Beckmann, N., and Lucia, B. (2018c). Intermittent deep neural network inference. *SysML*.

Gobieski, G., Beckmann, N., and Lucia, B. (2019a). Intelligence Beyond the Edge: Inference on Intermittent Embedded Systems. In *ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

Gobieski, G., Lucia, B., and Beckmann, N. (2019b). Intelligence beyond the edge: Inference on intermittent embedded systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 199–213. ACM.

Gokhale, V., Jin, J., Dundar, A., Martini, B., and Culurciello, E. (2014). A 240 g-ops/s mobile coprocessor for deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pages 682–687.

Goldberg, A. B. and Zhu, X. (2010). *New directions in semi-supervised learning*. PhD thesis, University of Wisconsin–Madison.

Gong, Y., Liu, L., Yang, M., and Bourdev, L. (2014). Compressing deep convolutional networks using vector quantization. *arXiv preprint arXiv:1412.6115*.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep learning*. MIT press.

Goodfellow, I. J., Mirza, M., Xiao, D., Courville, A., and Bengio, Y. (2013). An empirical investigation of catastrophic forgetting in gradient-based neural networks. *arXiv preprint arXiv:1312.6211*.

Google (2018a). Google clips. `https://store.google.com/us/product/google_clips?hl=en-US`.

Google (2018b). Timeline visualization for tensorflow using chrome trace format. `https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/client/timeline.py`.

Google (2019a). Face match on google nest hub max. `https://support.google.com/googlenest/answer/9320885?hl=en`.

Google (2019b). Faqs on camera sensing: Google nest hub max. `https://support.google.com/googlenest/answer/9449279?hl=en`.

Google (2019c). Introducing google nest hub. `https://support.google.com/googlenest/answer/9136909?hl=en`.

Google (2019d). Learn about familiar face detection and how to manage your library. `https://support.google.com/googlenest/answer/9268625?co=GENIE.Platform%3DAndroid&hl=en`.

Google (2019e). Tensorflow lite for microcontrollers. `https://www.tensorflow.org/lite/microcontrollers`.

Google (2019f). Voice match and media on google nest and google home speakers and displays. `https://support.google.com/googlenest/answer/7342711?hl=en`.

Gorlatova, M., Kinget, P., Kymissis, I., Rubenstein, D., Wang, X., and Zussman, G. (2010). Energy harvesting active networked tags (enhants) for ubiquitous object networking. *IEEE Wireless Communications*, 17(6).

Guennebaud, G., Jacob, B., et al. (2010). Eigen v3. `http://eigen.tuxfamily.org`.

Günther, S., Ruthotto, L., Schroder, J. B., Cyr, E., and Gauger, N. R. (2018). Layer-parallel training of deep residual neural networks. *arXiv preprint arXiv:1812.04352*.

Guo, Y., Yao, A., and Chen, Y. (2016). Dynamic network surgery for efficient dnns. In *Advances In Neural Information Processing Systems*, pages 1379–1387.

Gupta, C., Suggala, A. S., Goyal, A., Simhadri, H. V., Paranjape, B., Kumar, A., Goyal, S., Udupa, R., Varma, M., and Jain, P. (2017). Protonn: Compressed and accurate knn for resource-scarce devices. In *International Conference on Machine Learning*, pages 1331–1340.

Gupta, S., Agrawal, A., Gopalakrishnan, K., and Narayanan, P. (2015). Deep learning with limited numerical precision. In *International Conference on Machine Learning*, pages 1737–1746.

Han, J. and Moraga, C. (1995). The influence of the sigmoid function parameters on the speed of backpropagation learning. In *International Workshop on Artificial Neural Networks*, pages 195–201. Springer.

Han, L. and Zhang, Y. (2015). Learning multi-level task groups in multi-task learning. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*.

Han, L. and Zhang, Y. (2016). Multi-stage multi-task learning with reduced rank. In *Thirtieth AAAI Conference on Artificial Intelligence*.

Han, S., Liu, X., Mao, H., Pu, J., Pedram, A., Horowitz, M. A., and Dally, W. J. (2016). Eie: efficient inference engine on compressed deep neural network. In *Computer Architecture (ISCA), 2016 ACM/IEEE 43rd Annual International Symposium on*, pages 243–254. IEEE.

Han, S., Mao, H., and Dally, W. J. (2015a). A deep neural network compression pipeline: Pruning, quantization, huffman encoding. *arXiv preprint arXiv:1510.00149*, 10.

Han, S., Pool, J., Tran, J., and Dally, W. (2015b). Learning both weights and connections for efficient neural network. In *Advances in neural information processing systems*, pages 1135–1143.

Hanin, B. (2017). Universal function approximation by deep neural nets with bounded width and relu activations. *arXiv preprint arXiv:1708.02691*.

Hansen, L. K. and Salamon, P. (1990). Neural network ensembles. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, (10):993–1001.

Hard, A., Rao, K., Mathews, R., Beaufays, F., Augenstein, S., Eichner, H., Kiddon, C., and Ramage, D. (2018). Federated learning for mobile keyboard prediction. *arXiv preprint arXiv:1811.03604*.

Hassibi, B. and Stork, D. G. (1993). Second order derivatives for network pruning: Optimal brain surgeon. In *Advances in neural information processing systems*, pages 164–171.

Hawkins, D. M. (2004). The problem of overfitting. *Journal of chemical information and computer sciences*, 44(1):1–12.

He, J., Choi, W., Yang, Y., Lu, J., Wu, X., and Peng, K. (2017). Detection of driver drowsiness using wearable devices: A feasibility study of the proximity sensor. *Applied ergonomics*, 65:473–480.

He, K., Zhang, X., Ren, S., and Sun, J. (2016a). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778.

He, Q., Segee, B., and Weaver, V. (2016b). Raspberry pi 2 b+ gpu power, performance, and energy implications. In *2016 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 163–167. IEEE.

He, S., Liu, Y., and Zhou, H. (2015). Optimizing smartphone power consumption through dynamic resolution scaling. In *Proceedings of the 21st Annual International Conference on Mobile Computing and Networking*, pages 27–39. ACM.

He, T., Fan, Y., Qian, Y., Tan, T., and Yu, K. (2014). Reshaping deep neural network for fast decoding by node-pruning. In *2014 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 245–249. IEEE.

He, X., Zhou, Z., and Thiele, L. (2018a). Multi-task zipping via layer-wise neuron sharing. In *Advances in Neural Information Processing Systems*, pages 6016–6026.

He, Y., Lin, J., Liu, Z., Wang, H., Li, L.-J., and Han, S. (2018b). Amc: Automl for model compression and acceleration on mobile devices. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 784–800.

Hecht-Nielsen, R. (1992). Theory of the backpropagation neural network. In *Neural networks for perception*, pages 65–93. Elsevier.

Henzinger, T. A. and Sifakis, J. (2006). The embedded systems design challenge. In *International Symposium on Formal Methods*, pages 1–15. Springer.

Hester, J., Sitanayah, L., and Sorber, J. (2015a). A hardware platform for separating energy concerns in tiny, intermittently-powered sensors. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, pages 447–448. ACM.

Hester, J., Sitanayah, L., and Sorber, J. (2015b). Tragedy of the coulombs: Federating energy storage for tiny, intermittently-powered sensors. In *Proceedings of the 13th ACM Conference on Embedded Networked Sensor Systems*, pages 5–16. ACM.

Hester, J. and Sorber, J. (2017). Flicker: Rapid prototyping for the batteryless internet-of-things. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems*, page 19. ACM.

Hester, J., Storer, K., and Sorber, J. (2017). Timely execution on intermi! ently powered ba! eryless sensors. In *Proceedings of the 15th ACM Conference on Embedded Network Sensor Systems, SenSys*, volume 17.

Hicks, M. (2017). Clank: Architectural support for intermittent computation. In *Computer Architecture (ISCA), 2017 ACM/IEEE 44th Annual International Symposium on*, pages 228–240. IEEE.

Hinton, G., Vinyals, O., and Dean, J. (2015). Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*.

Hochreiter, S., Bengio, Y., Frasconi, P., and Schmidhuber, J. (2001). Gradient flow in recurrent nets: the difficulty of learning long-term dependencies.

Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural computation*, 9(8):1735–1780.

Holi, J. L. and Hwang, J.-N. (1993). Finite precision error analysis of neural network hardware implementations. *IEEE Transactions on Computers*, (3):281–290.

Holton, J. and Fratangelo, T. (2012). Raspberry pi architecture. *Raspberry Pi Foundation, London, UK*.

Hornik, K. (1991). Approximation capabilities of multilayer feedforward networks. *Neural networks*, 4(2):251–257.

Howard, A. G., Zhu, M., Chen, B., Kalenichenko, D., Wang, W., Weyand, T., Andreetto, M., and Adam, H. (2017). Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*.

Hu, H., Peng, R., Tai, Y.-W., and Tang, C.-K. (2016). Network trimming: A data-driven neuron pruning approach towards efficient deep architectures. *arXiv preprint arXiv:1607.03250*.

Huang, G., Sun, Y., Liu, Z., Sedra, D., and Weinberger, K. Q. (2016a). Deep networks with stochastic depth. In *European conference on computer vision*, pages 646–661. Springer.

Huang, G.-B. (2003). Learning capability and storage capacity of two-hidden-layer feedforward networks. *IEEE Transactions on Neural Networks*, 14(2):274–281.

Huang, G. B., Mattar, M., Berg, T., and Learned-Miller, E. (2008). Labeled faces in the wild: A database forstudying face recognition in unconstrained environments. In *Workshop on faces in'Real-Life'Images: detection, alignment, and recognition*.

Huang, G.-B., Zhu, Q.-Y., and Siew, C. K. (2006). Real-time learning capability of neural networks. *IEEE Trans. Neural Networks*, 17(4):863–878.

Huang, Q., Mei, Y., Wang, W., and Zhang, Q. (2016b). Battery-free sensing platform for wearable devices: The synergy between two feet. In *Computer Communications, IEEE INFOCOM 2016-The 35th Annual IEEE International Conference on*, pages 1–9. IEEE.

Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., and Bengio, Y. (2017). Quantized neural networks: Training neural networks with low precision weights and activations. *The Journal of Machine Learning Research*, 18(1):6869–6898.

Hwang, K. and Sung, W. (2014). Fixed-point feedforward deep neural network design using weights+ 1, 0, and- 1. In *Signal Processing Systems (SiPS), 2014 IEEE Workshop on*, pages 1–6. IEEE.

Iandola, F. N., Han, S., Moskewicz, M. W., Ashraf, K., Dally, W. J., and Keutzer, K. (2016). Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*.

Ingle, S. and Phute, M. (2016). Tesla autopilot: semi autonomous driving, an uptick for future autonomy. *International Research Journal of Engineering and Technology*, 3(9).

Instrument, T. (2018). Msp energytrace technology.

Ioannou, Y., Robertson, D., Shotton, J., Cipolla, R., and Criminisi, A. (2015). Training cnns with low-rank filters for efficient image classification. *arXiv preprint arXiv:1511.06744*.

Jiang, C., Li, G., Qian, C., and Tang, K. (2018). Efficient dnn neuron pruning by minimizing layer-wise nonlinear reconstruction error. In *IJCAI*, volume 2018, pages 2–2.

Jiang, F., Jiang, Y., Zhi, H., Dong, Y., Li, H., Ma, S., Wang, Y., Dong, Q., Shen, H., and Wang, Y. (2017). Artificial intelligence in healthcare: past, present and future. *Stroke and vascular neurology*, 2(4):230–243.

Jiménez, M., Palomera, R., and Couvertier, I. (2013). *Introduction to embedded systems*. Springer.

Justus, D., Brennan, J., Bonner, S., and McGough, A. S. (2018). Predicting the computational cost of deep learning models. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 3873–3882. IEEE.

Kabkab, M., Alavi, A., and Chellappa, R. (2016). Dcnns on a diet: Sampling strategies for reducing the training set size. *arXiv preprint arXiv:1606.04232*.

Kaiser, L., Gomez, A. N., Shazeer, N., Vaswani, A., Parmar, N., Jones, L., and Uszkoreit, J. (2017). One model to learn them all. *arXiv preprint arXiv:1706.05137*.

Kamdar, S. and Kamdar, N. (2015). big. little architecture: Heterogeneous multicore processing. *International Journal of Computer Applications*, 119(1).

Kang, Z., Grauman, K., and Sha, F. (2011). Learning with whom to share in multi-task feature learning. In *ICML*.

Kansal, A., Hsu, J., Zahedi, S., and Srivastava, M. B. (2007). Power management in energy harvesting sensor networks. *ACM Transactions on Embedded Computing Systems (TECS)*, 6(4):32.

Kansal, A. and Srivastava, M. B. (2003). An environmental energy harvesting framework for sensor networks. In *Low Power Electronics and Design, 2003. ISLPED'03. Proceedings of the 2003 International Symposium on*, pages 481–486. IEEE.

Kaplan, W. (1984). Advanced calculus. redwood city.

Kappassov, Z., Corrales, J.-A., and Perdereau, V. (2015). Tactile sensing in dexterous robot hands. *Robotics and Autonomous Systems*, 74:195–220.

Kato, S., Lakshmanan, K., Rajkumar, R., and Ishikawa, Y. (2011). Timegraph: Gpu scheduling for real-time multi-tasking environments. In *Proc. USENIX ATC*, pages 17–30.

Kato, S., McThrow, M., Maltzahn, C., and Brandt, S. (2012). Gdev: First-class {GPU} resource management in the operating system. In *Presented as part of the 2012 {USENIX} Annual Technical Conference ({USENIX}{ATC} 12)*, pages 401–412.

Kawsar, F., Min, C., Mathur, A., Montanari, A., Acer, U. G., and Van den Broeck, M. (2018). esense: Open earable platform for human sensing. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, pages 371–372. ACM.

Kenny, K. B. and Lin, K.-J. (1990). Structuring large real-time systems with performance polymorphism. In *[1990] Proceedings 11th Real-Time Systems Symposium*, pages 238–246. IEEE.

Khan, W., Daud, A., Nasir, J. A., and Amjad, T. (2016). A survey on the state-of-the-art machine learning models in the context of nlp. *Kuwait journal of Science*, 43(4).

Kiefer, J. and Wolfowitz, J. (1952). Stochastic estimation of the maximum of a regression function. *The Annals of Mathematical Statistics*, 23(3):462–466.

Kim, C., Jeong, K.-Y., Choi, S.-G., Lee, K., and Sohn, C. (2017). A surveys based on cloud internet of things frameworks for smart home service. In *2017 Manila International Conference on" Trends in Engineering and Technology"(MTET-17)*.

Kim, J., Hwang, K., and Sung, W. (2014). X1000 real-time phoneme recognition vlsi using feed-forward deep neural networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2014 IEEE International Conference on*, pages 7510–7514. IEEE.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Kirkpatrick, J., Pascanu, R., Rabinowitz, N., Veness, J., Desjardins, G., Rusu, A. A., Milan, K., Quan, J., Ramalho, T., Grabska-Barwinska, A., et al. (2017). Overcoming catastrophic forgetting in neural networks. *Proceedings of the national academy of sciences*, 114(13):3521–3526.

Köksal, F., Alpaydyn, E., and Dündar, G. (2001). Weight quantization for multi-layer perceptrons using soft weight sharing. In *International Conference on Artificial Neural Networks*, pages 211–216. Springer.

Koopman, P. (2016). Cooperative and preemptive context switching: 18-348 embedded system engineering.

Kotagiri, V. S. (2014). Memory capacity of neural networks using a circulant weight matrix. *arXiv preprint arXiv:1403.3115*.

Krizhevsky, A., Hinton, G., et al. (2009). Learning multiple layers of features from tiny images. Technical report, Citeseer.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105.

Krogh, A. and Vedelsby, J. (1995). Neural network ensembles, cross validation, and active learning. In *Advances in neural information processing systems*, pages 231–238.

Kügler, D. (2003). "man in the middle" attacks on bluetooth. In *International Conference on Financial Cryptography*, pages 149–161. Springer.

Kumar, A. and Daume III, H. (2012). Learning task grouping and overlap in multi-task learning. *arXiv preprint arXiv:1206.6417*.

Kumar, A., Goyal, S., and Varma, M. (2017). Resource-efficient machine learning in 2 kb ram for the internet of things. In *International Conference on Machine Learning*, pages 1935–1944.

Lane, N. D., Bhattacharya, S., Mathur, A., Georgiev, P., Forlivesi, C., and Kawsar, F. (2017). Squeezing deep learning into mobile and embedded devices. *IEEE Pervasive Computing*, 16(3):82–88.

Lavin, A. and Gray, S. (2016). Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4013–4021.

Lawrence, S., Giles, C. L., and Tsoi, A. C. (1997). Lessons in neural network training: Overfitting may be harder than expected. In *AAAI/IAAI*, pages 540–545. Citeseer.

Lawrence, S., Giles, C. L., and Tsoi, A. C. (1998). What size neural network gives optimal generalization? convergence properties of backpropagation. Technical report.

Lebedev, V. and Lempitsky, V. (2016). Fast convnets using group-wise brain damage. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 2554–2564.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *nature*, 521(7553):436.

LeCun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

LeCun, Y., Denker, J. S., and Solla, S. A. (1990). Optimal brain damage. In *Advances in neural information processing systems*, pages 598–605.

LeCun, Y., Jackel, L., Bottou, L., Brunot, A., Cortes, C., Denker, J., Drucker, H., Guyon, I., Muller, U., Sackinger, E., et al. (1995). Comparison of learning algorithms for handwritten digit recognition. In *International conference on artificial neural networks*, volume 60, pages 53–60. Perth, Australia.

Lee, D.-H. (2013). Pseudo-label: The simple and efficient semi-supervised learning method for deep neural networks. In *Workshop on Challenges in Representation Learning, ICML*, volume 3, page 2.

Lee, E. A., Seshia, S. A., et al. (2011). Introduction to embedded systems. *A Cyber-Physical Systems Approach*, 499.

Lee, G., Yang, E., and Hwang, S. (2016). Asymmetric multi-task learning based on task relatedness and loss. In *International Conference on Machine Learning*, pages 230–238.

Lee, K. H. and Verma, N. (2013). A low-power processor with configurable embedded machine-learning accelerators for high-order and adaptive analysis of medical-sensor signals. *IEEE Journal of Solid-State Circuits*, 48(7):1625–1637.

Lee, S., Islam, B., Luo, Y., and Nirjon, S. (2019). Intermittent learning: On-device machine learning on intermittently powered system. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 3(4):1–30.

Lee, S. and Nirjon, S. (2018). Knowledge transfer between embedded controllers. In *2018 14th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pages 61–68. IEEE.

Lee, S. and Nirjon, S. (2019). Neuro. zero: a zero-energy neural network accelerator for embedded sensing and inference systems. In *Proceedings of the 17th Conference on Embedded Networked Sensor Systems*, pages 138–152. ACM.

Lee, S. and Nirjon, S. (2020a). Fast and scalable in-memory deep multitask learning via neural weight virtualization. In *Proceedings of the 18th International Conference on Mobile Systems, Applications, and Services*, pages 175–190.

Lee, S. and Nirjon, S. (2020b). Learning in the wild: When, how, and what to learn for on-device dataset adaptation. In *Proceedings of the 2nd International Workshop on Challenges in Artificial Intelligence and Machine Learning for Internet of Things*, pages 34–40.

Lee, S. and Nirjon, S. (2020c). Subflow: A dynamic induced-subgraph strategy toward real-time dnn inference and training. In *2020 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 15–29. IEEE.

Lehmann, E. L. and Casella, G. (2006). *Theory of point estimation*. Springer Science & Business Media.

Lemaignan, S., Warnier, M., Sisbot, E. A., Clodic, A., and Alami, R. (2017). Artificial cognition for social human–robot interaction: An implementation. *Artificial Intelligence*, 247:45–69.

Levi, G. and Hassner, T. (2015). Age and gender classification using convolutional neural networks. In *Proceedings of the iEEE conference on computer vision and pattern recognition workshops*, pages 34–42.

Li, H., De, S., Xu, Z., Studer, C., Samet, H., and Goldstein, T. (2017a). Training quantized nets: A deeper understanding. In *Advances in Neural Information Processing Systems*, pages 5811–5821.

Li, H., Kadav, A., Durdanovic, I., Samet, H., and Graf, H. P. (2016). Pruning filters for efficient convnets. *arXiv preprint arXiv:1608.08710*.

Li, H., Ota, K., and Dong, M. (2018a). Learning iot in edge: deep learning for the internet of things with edge computing. *IEEE Network*, 32(1):96–101.

Li, M., Zhang, T., Chen, Y., and Smola, A. J. (2014a). Efficient mini-batch training for stochastic optimization. In *Proceedings of the 20th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 661–670. ACM.

Li, P., Chen, X., and Shen, S. (2019). Stereo r-cnn based 3d object detection for autonomous driving. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7644–7652.

Li, S., Liu, Z.-Q., and Chan, A. B. (2014b). Heterogeneous multi-task learning for human pose estimation with deep convolutional neural network. In *Proceedings of the IEEE conference on computer vision and pattern recognition workshops*, pages 482–489.

Li, T. and Zhou, X. (2018). Battery-free eye tracker on glasses. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking*, pages 67–82. ACM.

Li, X. and Zhou, Z. (2017). Speech command recognition with convolutional neural network. *CS229 Stanford education*.

Li, Y., Li, T., Patel, R. A., Yang, X.-D., and Zhou, X. (2018b). Self-powered gesture recognition with ambient light. In *The 31st Annual ACM Symposium on User Interface Software and Technology*, pages 595–608. ACM.

Li, Z., Wang, X., Lv, X., and Yang, T. (2017b). Sep-nets: Small and effective pattern networks. *arXiv preprint arXiv:1706.03912*.

Liao, Y., Kodagoda, S., Wang, Y., Shi, L., and Liu, Y. (2016). Understand scene categories by objects: A semantic regularized scene classifier using convolutional neural networks. In *2016 IEEE international conference on robotics and automation (ICRA)*, pages 2318–2325. IEEE.

Lin, K.-J. and Natarajan, S. (1988). Expressing and maintaining timing constraints in flex. In *Proceedings. Real-Time Systems Symposium*, pages 96–105. IEEE.

Lin, K.-J., Natarajan, S., and Liu, J. W.-S. (1987). Imprecise results: Utilizing partial computations in real-time systems.

Lin, T.-Y., Maire, M., Belongie, S., Hays, J., Perona, P., Ramanan, D., Dollár, P., and Zitnick, C. L. (2014). Microsoft coco: Common objects in context. In *European conference on computer vision*, pages 740–755. Springer.

Lin, Z., Courbariaux, M., Memisevic, R., and Bengio, Y. (2015). Neural networks with few multiplications. *arXiv preprint arXiv:1510.03009*.

Litjens, G., Kooi, T., Bejnordi, B. E., Setio, A. A. A., Ciompi, F., Ghafoorian, M., Van Der Laak, J. A., Van Ginneken, B., and Sánchez, C. I. (2017). A survey on deep learning in medical image analysis. *Medical image analysis*, 42:60–88.

Liu, B., Wang, M., Foroosh, H., Tappen, M., and Pensky, M. (2015a). Sparse convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 806–814.

Liu, C., Zoph, B., Neumann, M., Shlens, J., Hua, W., Li, L.-J., Fei-Fei, L., Yuille, A., Huang, J., and Murphy, K. (2018a). Progressive neural architecture search. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 19–34.

Liu, F. T., Ting, K. M., and Zhou, Z.-H. (2008). Isolation forest. In *2008 Eighth IEEE International Conference on Data Mining*, pages 413–422. IEEE.

Liu, F. T., Ting, K. M., and Zhou, Z.-H. (2012). Isolation-based anomaly detection. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 6(1):3.

Liu, J. W., Shih, W.-K., Lin, K.-J., Bettati, R., and Chung, J.-Y. (1994). Imprecise computations. *Proceedings of the IEEE*, 82(1):83–94.

Liu, J. W.-S., Lin, K.-J., Shih, W. K., Yu, A. C.-s., Chung, J.-Y., and Zhao, W. (1991). Algorithms for scheduling imprecise computations. In *Foundations of Real-Time Computing: Scheduling and Resource Management*, pages 203–249. Springer.

Liu, P., Qiu, X., and Huang, X. (2017). Adversarial multi-task learning for text classification. *arXiv preprint arXiv:1704.05742*.

Liu, S., Lin, Y., Zhou, Z., Nan, K., Liu, H., and Du, J. (2018b). On-demand deep model compression for mobile devices: A usage-driven model selection framework. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services*, pages 389–400. ACM.

Liu, W., Mei, T., Zhang, Y., Che, C., and Luo, J. (2015b). Multi-task deep visual-semantic embedding for video thumbnail selection. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3707–3715.

Long, M., Cao, Z., Wang, J., and Philip, S. Y. (2017). Learning multiple tasks with multilinear relationship networks. In *Advances in Neural Information Processing Systems*, pages 1594–1603.

Lu, H., Brush, A. B., Priyantha, B., Karlson, A. K., and Liu, J. (2011). Speakersense: Energy efficient unobtrusive speaker identification on mobile phones. In *International conference on pervasive computing*, pages 188–205. Springer.

Lu, X., Niyato, D., Jiang, H., Kim, D. I., Xiao, Y., and Han, Z. (2018). Ambient backscatter assisted wireless powered communications. *IEEE Wireless Communications*, 25(2):170–177.

Lu, X., Wang, P., Niyato, D., Kim, D. I., and Han, Z. (2015). Wireless networks with rf energy harvesting: A contemporary survey. *IEEE Communications Surveys & Tutorials*, 17(2):757–789.

Lu, Z., Pu, H., Wang, F., Hu, Z., and Wang, L. (2017). The expressive power of neural networks: A view from the width. In *Advances in Neural Information Processing Systems*, pages 6231–6239.

Lucia, B., Balaji, V., Colin, A., Maeng, K., and Ruppel, E. (2017). Intermittent computing: Challenges and opportunities. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 71. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

Lucia, B. and Ransford, B. (2015). A simpler, safer programming and execution model for intermittent systems. *ACM SIGPLAN Notices*, 50(6):575–585.

Luk, C.-K., Hong, S., and Kim, H. (2009). Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 45–55. IEEE.

Luo, Y. and Nirjon, S. (2019). Spoton: Just-in-time active event detection on energy autonomous sensing systems. In *Proceedings of the 25th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS WIP Session)*, Montreal, Canada. IEEE.

Ma, J., Zhao, Z., Chen, J., Li, A., Hong, L., and Chi, E. H. (2019). Snr: Sub-network routing for flexible parameter sharing in multi-task learning. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 33, pages 216–223.

Ma, J., Zhao, Z., Yi, X., Chen, J., Hong, L., and Chi, E. H. (2018). Modeling task relationships in multi-task learning with multi-gate mixture-of-experts. In *Proceedings of the 24th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, pages 1930–1939. ACM.

Maeng, K., Colin, A., and Lucia, B. (2017). Alpaca: intermittent execution without checkpoints. *Proceedings of the ACM on Programming Languages*, 1(OOPSLA):96.

Maeng, K. and Lucia, B. (2018). Adaptive dynamic checkpointing for safe efficient intermittent computing. In *13th {USENIX} Symposium on Operating Systems Design and Implementation ({OSDI} 18)*, pages 129–144. USENIX Association.

Majumder, S., Mondal, T., and Deen, M. J. (2017). Wearable sensors for remote health monitoring. *Sensors*, 17(1):130.

Mallya, A. and Lazebnik, S. (2018). Packnet: Adding multiple tasks to a single network by iterative pruning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 7765–7773.

Mancini, M., Costante, G., Valigi, P., and Ciarfuglia, T. A. (2016). Fast robust monocular depth estimation for obstacle detection with fully convolutional networks. In *2016 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4296–4303. IEEE.

Manessi, F., Rozza, A., Bianco, S., Napoletano, P., and Schettini, R. (2018). Automated pruning for deep neural network compression. In *2018 24th International Conference on Pattern Recognition (ICPR)*, pages 657–664. IEEE.

Manevitz, L. M. and Yousef, M. (2001). One-class svms for document classification. *Journal of machine Learning research*, 2(Dec):139–154.

Marsland, S. (2015). *Machine learning: an algorithmic perspective*. CRC press.

Martinetz, T. and Schulten, K. (1994). Topology representing networks. *Neural Networks*, 7(3):507–522.

Masters, D. and Luschi, C. (2018). Revisiting small batch training for deep neural networks. *arXiv preprint arXiv:1804.07612*.

Maturana, D. and Scherer, S. (2015). Voxnet: A 3d convolutional neural network for real-time object recognition. In *2015 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 922–928. IEEE.

McDonald, A. M., Pontil, M., and Stamos, D. (2014). Spectral k-support norm regularization. In *Advances in neural information processing systems*, pages 3644–3652.

Merolla, P. A., Arthur, J. V., Alvarez-Icaza, R., Cassidy, A. S., Sawada, J., Akopyan, F., Jackson, B. L., Imam, N., Guo, C., Nakamura, Y., et al. (2014). A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345(6197):668–673.

Miotto, R., Wang, F., Wang, S., Jiang, X., and Dudley, J. T. (2017). Deep learning for healthcare: review, opportunities and challenges. *Briefings in bioinformatics*, 19(6):1236–1246.

Miralles, Á. S. and Bobi, M. Á. S. (2016). Real time dynamic neural network (rtdnn).

Misra, I., Shrivastava, A., Gupta, A., and Hebert, M. (2016). Cross-stitch networks for multi-task learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 3994–4003.

Mohammadi, M. and Das, S. (2016). Snn: stacked neural networks. *arXiv preprint arXiv:1605.08512*.

Mosa, I. M., Pattammattel, A., Kadimisetty, K., Pande, P., El-Kady, M. F., Bishop, G. W., Novak, M., Kaner, R. B., Basu, A. K., Kumar, C. V., et al. (2017). Ultrathin graphene–protein supercapacitors for miniaturized bioelectronics. *Advanced energy materials*, 7(17):1700358.

Moser, C., Brunelli, D., Thiele, L., and Benini, L. (2007). Real-time scheduling for energy harvesting sensor nodes. *Real-Time Systems*, 37(3):233–260.

Mrkšić, N., Séaghdha, D. O., Thomson, B., Gašić, M., Su, P.-H., Vandyke, D., Wen, T.-H., and Young, S. (2015). Multi-domain dialog state tracking using recurrent neural networks. *arXiv preprint arXiv:1506.07190*.

Mudrakarta, P. K., Sandler, M., Zhmoginov, A., and Howard, A. (2018). K for the price of 1: Parameter-efficient multi-task and transfer learning. *arXiv preprint arXiv:1810.10703*.

Munkres, J. (1957). Algorithms for the assignment and transportation problems. *Journal of the society for industrial and applied mathematics*, 5(1):32–38.

Murmann, B., Bankman, D., Chai, E., Miyashita, D., and Yang, L. (2015). Mixed-signal circuits for embedded machine-learning applications. In *Signals, Systems and Computers, 2015 49th Asilomar Conference on*, pages 1341–1345. IEEE.

Naderiparizi, S., Parks, A. N., Kapetanovic, Z., Ransford, B., and Smith, J. R. (2015). Wispcam: A battery-free rfid camera. In *RFID (RFID), 2015 IEEE International Conference on*, pages 166–173. IEEE.

Naderiparizi, S., Zhang, P., Philipose, M., Priyantha, B., Liu, J., and Ganesan, D. (2017). Glimpse: A programmable early-discard camera architecture for continuous mobile vision. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*, pages 292–305. ACM.

Nägeli, T., Meier, L., Domahidi, A., Alonso-Mora, J., and Hilliges, O. (2017). Real-time planning for automated multi-view drone cinematography. *ACM Transactions on Graphics (TOG)*, 36(4):132.

Netzer, Y., Wang, T., Coates, A., Bissacco, A., Wu, B., and Ng, A. Y. (2011). Reading digits in natural images with unsupervised feature learning.

Ng, A. (2016). Nuts and bolts of building ai applications using deep learning. *NIPS Keynote Talk*.

Nirjon, S. (2018). Lifelong Learning on Harvested Energy. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services (MOBISYS 2018)*. ACM.

Nowlan, S. J. and Hinton, G. E. (1992). Simplifying neural networks by soft weight-sharing. *Neural computation*, 4(4):473–493.

NVIDIA (2019a). Jetson nano. `https://developer.nvidia.com/embedded/jetson-nano-developer-kit`.

NVIDIA (2019b). Nvidia cuda home page. `https://developer.nvidia.com/cuda-zone`.

Oberstar, E. L. (2007). Fixed-point representation & fractional math. *Oberstar Consulting*, page 9.

Ollivier, Y., Arnold, L., Auger, A., and Hansen, N. (2017). Information-geometric optimization algorithms: A unifying picture via invariance principles. *The Journal of Machine Learning Research*, 18(1):564–628.

O'Shea, K. and Nash, R. (2015). An introduction to convolutional neural networks. *arXiv preprint arXiv:1511.08458*.

Ota, H., Chao, M., Gao, Y., Wu, E., Tai, L.-C., Chen, K., Matsuoka, Y., Iwai, K., Fahad, H. M., Gao, W., et al. (2017). 3d printed "earable" smart devices for real-time detection of core body temperature. *ACS sensors*, 2(7):990–997.

Pan, S. J. and Yang, Q. (2009). A survey on transfer learning. *IEEE Transactions on knowledge and data engineering*, 22(10):1345–1359.

Pan, X., Shi, J., Luo, P., Wang, X., and Tang, X. (2018). Spatial as deep: Spatial cnn for traffic scene understanding. In *Thirty-Second AAAI Conference on Artificial Intelligence*.

Panda, P., Sengupta, A., and Roy, K. (2016). Conditional deep learning for energy-efficient and enhanced pattern recognition. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 475–480. IEEE.

Parashar, A., Rhu, M., Mukkara, A., Puglielli, A., Venkatesan, R., Khailany, B., Emer, J., Keckler, S. W., and Dally, W. J. (2017). Scnn: An accelerator for compressed-sparse convolutional neural networks. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 27–40. IEEE.

Parisi, G. I., Kemker, R., Part, J. L., Kanan, C., and Wermter, S. (2019). Continual lifelong learning with neural networks: A review. *Neural Networks*.

Park, H., Amari, S.-I., and Fukumizu, K. (2000). Adaptive natural gradient learning algorithms for various stochastic models. *Neural Networks*, 13(7):755–764.

Park, J., Li, S., Wen, W., Tang, P. T. P., Li, H., Chen, Y., and Dubey, P. (2016a). Faster cnns with direct sparse convolutions and guided pruning. *arXiv preprint arXiv:1608.01409*.

Park, J., Li, S. R., Wen, W., Li, H., Chen, Y., and Dubey, P. (2016b). Holistic sparsecnn: Forging the trident of accuracy, speed, and size. *arXiv preprint arXiv:1608.01409*, 1(2).

Pascanu, R. and Bengio, Y. (2013). Revisiting natural gradient for deep networks. *arXiv preprint arXiv:1301.3584*.

Patterson, D. A. and Hennessy, J. L. (2013). *Computer organization and design MIPS edition: the hardware/software interface*. Newnes.

Peikari, M., Salama, S., Nofech-Mozes, S., and Martel, A. L. (2018). A cluster-then-label semi-supervised learning approach for pathology image classification. *Scientific reports*, 8(1):7193.

Pham, P.-H., Jelaca, D., Farabet, C., Martini, B., LeCun, Y., and Culurciello, E. (2012). Neuflow: Dataflow vision processing system-on-a-chip. In *Circuits and Systems (MWSCAS), 2012 IEEE 55th International Midwest Symposium on*, pages 1044–1047. IEEE.

Philipose, M., Smith, J. R., Jiang, B., Mamishev, A., Roy, S., and Sundara-Rajan, K. (2005). Battery-free wireless identification and sensing. *IEEE Pervasive computing*, 4(1):37–45.

Piorno, J. R., Bergonzini, C., Atienza, D., and Rosing, T. S. (2009). Prediction and management in energy harvested wireless sensor nodes. In *Wireless Communication, Vehicular Technology, Information Theory and Aerospace & Electronic Systems Technology, 2009. Wireless VITAE 2009. 1st International Conference on*, pages 6–10. IEEE.

Plessl, C. and Platzner, M. (2004). Virtualization of hardware-introduction and survey. In *ERSA*, pages 63–69. Citeseer.

Polyak, A. and Wolf, L. (2015). Channel-level acceleration of deep face representations. *IEEE Access*, 3:2163–2175.

Pongpunwattana, A. and Rysdyk, R. (2004). Real-time planning for multiple autonomous vehicles in dynamic uncertain environments. *Journal of Aerospace Computing, Information, and Communication*, 1(12):580–604.

Powercast (2016a). Powercast p2110b. `http://www.powercastco.com/wp-content/uploads/2016/12/P2110B-Datasheet-Rev-3.pdf`.

Powercast (2016b). Powercaster transmitter. `http://www.powercastco.com/wp-content/uploads/2016/11/User-Manual-TX-915-01-Rev-A-4.pdf`.

Prassler, E. and Kosuge, K. (2008). Domestic robotics. *Springer handbook of robotics*, pages 1253–1281.

Pratt, L. Y. (1993). Discriminability-based transfer between neural networks. In *Advances in neural information processing systems*, pages 204–211.

Priya, S. and Inman, D. J. (2009). *Energy harvesting technologies*, volume 21. Springer.

Qiu, J., Wang, J., Yao, S., Guo, K., Li, B., Zhou, E., Yu, J., Tang, T., Xu, N., and Song, S. (2016). Going deeper with embedded fpga platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 26–35. ACM.

Qualcomm (2017). Snapdragon 845 mobile platform. `https://www.qualcomm.com/media/documents/files/snapdragon-845-mobile-platform-product-brief.pdf`.

Qualcomm (2018). Qualcomm snapdragon 820e processor (apq8096sge). `https://developer.qualcomm.com/download/sd820e/qualcomm-snapdragon-820e-processor-apq8096sge-device-specification.pdf`.

Ransford, B., Sorber, J., and Fu, K. (2012). Mementos: System support for long-running computation on rfid-scale devices. *Acm Sigplan Notices*, 47(4):159–170.

Rastegari, M., Ordonez, V., Redmon, J., and Farhadi, A. (2016). Xnor-net: Imagenet classification using binary convolutional neural networks. In *European Conference on Computer Vision*, pages 525–542. Springer.

Redmon, J. and Angelova, A. (2015). Real-time grasp detection using convolutional neural networks. In *2015 IEEE International Conference on Robotics and Automation (ICRA)*, pages 1316–1322. IEEE.

Robbins, H. and Monro, S. (1985). A stochastic approximation method. In *Herbert Robbins Selected Papers*, pages 102–109. Springer.

Rodriguez, A., Balsamo, D., Luo, Z., Beeby, S. P., Merrett, G. V., and Weddel, A. S. (2017). Intermittently-powered energy harvesting step counter for fitness tracking. In *Sensors Applications Symposium (SAS), 2017 IEEE*, pages 1–6. IEEE.

Romero, A., Ballas, N., Kahou, S. E., Chassang, A., Gatta, C., and Bengio, Y. (2014). Fitnets: Hints for thin deep nets. *arXiv preprint arXiv:1412.6550*.

Rossbach, C. J., Currey, J., Silberstein, M., Ray, B., and Witchel, E. (2011). Ptask: operating system abstractions to manage gpus as compute devices. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, pages 233–248. ACM.

Roth, W. and Pernkopf, F. (2018). Bayesian neural networks with weight sharing using dirichlet processes. *IEEE transactions on pattern analysis and machine intelligence*.

ROUAUD, M. (2012). Probabilités, statistiques et analyses multicritères.

Ruder, S. (2017). An overview of multi-task learning in deep neural networks. *arXiv preprint arXiv:1706.05098*.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1985). Learning internal representations by error propagation. Technical report, California Univ San Diego La Jolla Inst for Cognitive Science.

Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1988). Learning representations by back-propagating errors. *Cognitive modeling*, 5(3):1.

Russell, S. J. and Norvig, P. (2016). *Artificial intelligence: a modern approach*. Malaysia; Pearson Education Limited,.

Sainath, T. and Parada, C. (2015). Convolutional neural networks for small-footprint keyword spotting.

Sainath, T. N., Kingsbury, B., Sindhwani, V., Arisoy, E., and Ramabhadran, B. (2013). Low-rank matrix factorization for deep neural network training with high-dimensional output targets. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6655–6659. IEEE.

Salamon, J., Jacoby, C., and Bello, J. P. (2014). A dataset and taxonomy for urban sound research. In *Proceedings of the 22nd ACM international conference on Multimedia*, pages 1041–1044. ACM.

Sample, A. P., Yeager, D. J., Powledge, P. S., Mamishev, A. V., and Smith, J. R. (2008). Design of an rfid-based battery-free programmable sensing platform. *IEEE transactions on instrumentation and measurement*, 57(11):2608–2615.

Sandler, M., Howard, A., Zhu, M., Zhmoginov, A., and Chen, L.-C. (2018). Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4510–4520.

Sarikaya, D., Corso, J. J., and Guru, K. A. (2017). Detection and localization of robotic tools in robot-assisted surgery videos using deep neural networks for region proposal and detection. *IEEE transactions on medical imaging*, 36(7):1542–1549.

Scharf, L. L. and Demeure, C. (1991). *Statistical signal processing: detection, estimation, and time series analysis*, volume 63. Addison-Wesley Reading, MA.

Schild, K. and Würtz, J. (2000). Scheduling of time-triggered real-time systems. *Constraints*, 5(4):335–357.

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural networks*, 61:85–117.

Schrijver, A. (2003). *Combinatorial optimization: polyhedra and efficiency*, volume 24. Springer Science & Business Media.

Schroff, F., Kalenichenko, D., and Philbin, J. (2015). Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 815–823.

Seah, W. K., Eu, Z. A., and Tan, H.-P. (2009). Wireless sensor networks powered by ambient energy harvesting (wsn-heap)-survey and challenges. In *Wireless Communication, Vehicular Technology, Information Theory and Aerospace & Electronic Systems Technology, 2009. Wireless VITAE 2009. 1st International Conference on*, pages 1–5. Ieee.

Shahzad, K. and Oelmann, B. (2014). Investigating energy consumption of an sram-based fpga for duty-cycle applications. In *International Conference on Parallel Computing-ParCo 2013, 10-13 Sept, Munich*, pages 548–559.

Shaikh, F. K. and Zeadally, S. (2016). Energy harvesting in wireless sensor networks: A comprehensive review. *Renewable and Sustainable Energy Reviews*, 55:1041–1054.

Sharma, A., Wolfe, N., and Raj, B. (2017). The incredible shrinking neural network: New perspectives on learning representations through the lens of pruning. *arXiv preprint arXiv:1701.04465*.

Shih, W.-K., Liu, J. W., and Chung, J.-Y. (1991). Algorithms for scheduling imprecise computations with timing constraints. *SIAM Journal on Computing*, 20(3):537–552.

Shiller, Z., Gwo, Y.-R., et al. (1991). Dynamic motion planning of autonomous vehicles. *IEEE Transactions on Robotics and Automation*, 7(2):241–249.

Siegwart, R., Arras, K. O., Bouabdallah, S., Burnier, D., Froidevaux, G., Greppin, X., Jensen, B., Lorotte, A., Mayor, L., Meisser, M., et al. (2003). Robox at expo. 02: A large-scale installation of personal robots. *Robotics and Autonomous Systems*, 42(3-4):203–222.

Silberman, N., Hoiem, D., Kohli, P., and Fergus, R. (2012). Indoor segmentation and support inference from rgbd images. In *European Conference on Computer Vision*, pages 746–760. Springer.

Silberschatz, A., Gagne, G., and Galvin, P. B. (2018). *Operating system concepts*. Wiley.

Silberschatz, A., Galvin, P. B., and Gagne, G. (2012). Operating system concepts. 9th.

Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

Sindhwani, V., Sainath, T., and Kumar, S. (2015). Structured transforms for small-footprint deep learning. In *Advances in Neural Information Processing Systems*, pages 3088–3096.

Singh, R. R. (2007). Preventing road accidents with wearable biosensors and innovative architectural design. In *2nd ISSS National Conference on MEMS, Pilani, India*, pages 1–8.

Socher, R., Bengio, Y., and Manning, C. (2012). Deep learning for nlp. *Tutorial at Association of Computational Logistics (ACL)*.

Son, S., Nah, S., and Mu Lee, K. (2018). Clustering convolutional kernels to compress deep neural networks. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pages 216–232.

Song, W. and Hai, J. (2003). A survey of storage virtualization. *Mini-micro Systems*, 4.

Soto, M., Nava, P., and Alvarado, L. (2007). Drone formation control system real-time path planning. In *AIAA Infotech@ Aerospace 2007 Conference and Exhibit*, page 2770.

Spieksma, F. C. (2000). Multi index assignment problems: complexity, approximation, applications. In *Nonlinear assignment problems*, pages 1–12. Springer.

Spyridon, M. G. and Eleftheria, M. (2012). Classification of domestic robots. *Proceedings in ARSA-Advanced Research in Scientific Areas*, 1(7):1693.

Sridhar, D. V., Bartlett, E. B., and Seagrave, R. C. (1999). An information theoretic approach for combining neural network process models. *Neural Networks*, 12(6):915–926.

Sridhar, D. V., Seagrave, R. C., and Bartlett, E. B. (1996). Process modeling using stacked neural networks. *AIChE Journal*, 42(9):2529–2539.

Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958.

Srivastava, R. K., Greff, K., and Schmidhuber, J. (2015). Highway networks. *arXiv preprint arXiv:1505.00387*.

Stallkamp, J., Schlipsing, M., Salmen, J., and Igel, C. (2011a). The german traffic sign recognition benchmark: A multi-class classification competition. In *IJCNN*, volume 6, page 7.

Stallkamp, J., Schlipsing, M., Salmen, J., and Igel, C. (2011b). The German Traffic Sign Recognition Benchmark: A multi-class classification competition. In *IEEE International Joint Conference on Neural Networks*, pages 1453–1460.

Stewart, D. B. and Khosla, P. K. (1991). Real-time scheduling of sensor-based control systems. *IFAC Proceedings Volumes*, 24(2):139–144.

Su, Y., Zhang, K., Wang, J., and Madani, K. (2019). Environment sound classification using a two-stream cnn based on decision-level fusion. *Sensors*, 19(7):1733.

Sudevalayam, S. and Kulkarni, P. (2011). Energy harvesting sensor nodes: Survey and implications. *IEEE Communications Surveys & Tutorials*, 13(3):443–461.

Sun, X., Ren, X., Ma, S., and Wang, H. (2017). meprop: Sparsified back propagation for accelerated deep learning with reduced overfitting. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 3299–3308. JMLR. org.

Sussmann, H. J. (1992). Uniqueness of the weights for minimal feedforward nets with a given input-output map. *Neural networks*, 5(4):589–593.

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

Szegedy, C., Ioffe, S., Vanhoucke, V., and Alemi, A. A. (2017). Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-First AAAI Conference on Artificial Intelligence*.

Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S., Anguelov, D., Erhan, D., Vanhoucke, V., and Rabinovich, A. (2015). Going deeper with convolutions. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1–9.

Tai, C., Xiao, T., Zhang, Y., Wang, X., et al. (2015). Convolutional neural networks with low-rank regularization. *arXiv preprint arXiv:1511.06067*.

Taniguchi, K., Kondo, H., Kurosawa, M., and Nishikawa, A. (2018). Earable tempo: a novel, hands-free input device that uses the movement of the tongue measured with a wearable ear sensor. *Sensors*, 18(3):733.

Taş, Ö. Ş., Kuhnt, F., Zöllner, J. M., and Stiller, C. (2016). Functional system architectures towards fully automated driving. In *2016 IEEE Intelligent vehicles symposium (IV)*, pages 304–309. IEEE.

Teerapittayanon, S., McDanel, B., and Kung, H.-T. (2016). Branchynet: Fast inference via early exiting from deep neural networks. In *2016 23rd International Conference on Pattern Recognition (ICPR)*, pages 2464–2469. IEEE.

TexasInstruments (2018). Msp430fr5994. `http://www.ti.com/product/MSP430FR5994`.

Thrun, S. and O'Sullivan, J. (1996). Discovering structure in multiple learning tasks: The tc algorithm. In *ICML*, volume 96, pages 489–497.

Torrey, L. and Shavlik, J. (2010). Transfer learning. In *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*, pages 242–264. IGI Global.

Toy, W. N. and Zee, B. (1986). *Computer Hardware-Software Architecture*. Prentice Hall Professional Technical Reference.

Truong, H., Zhang, S., Muncuk, U., Nguyen, P., Bui, N., Nguyen, A., Lv, Q., Chowdhury, K., Dinh, T., and Vu, T. (2018). Capband: Battery-free successive capacitance sensing wristband for hand gesture recognition. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, pages 54–67. ACM.

Tu, M., Berisha, V., Woolf, M., Seo, J.-s., and Cao, Y. (2016). Ranking the parameters of deep neural networks using the fisher information. In *2016 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2647–2651. IEEE.

Tyagi, V. and Mishra, A. (2014). A survey on ensemble combination schemes of neural network. *International Journal of Computer Applications*, 95(16).

Ullrich, K., Meeds, E., and Welling, M. (2017). Soft weight-sharing for neural network compression. *arXiv preprint arXiv:1702.04008*.

Upton, G. and Cook, I. (2014). *A dictionary of statistics 3e*. Oxford university press.

Uspensky, J. V. (1937). Introduction to mathematical probability.

Uyeda, F. (2009). Lecture 7: Memory management, cse 120: Principles of operating systems.

Van Der Woude, J. and Hicks, M. (2016). Intermittent computation without hardware support or programmer intervention. In *OSDI*, pages 17–32.

Vanhoucke, V., Senior, A., and Mao, M. Z. (2011). Improving the speed of neural networks on cpus.

Vinyals, O., Toshev, A., Bengio, S., and Erhan, D. (2015). Show and tell: A neural image caption generator. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 3156–3164.

Wan, L., Zeiler, M., Zhang, S., Le Cun, Y., and Fergus, R. (2013). Regularization of neural networks using dropconnect. In *International conference on machine learning*, pages 1058–1066.

Wang, C., Yu, Q., Gong, L., Li, X., Xie, Y., and Zhou, X. (2016). Dlau: A scalable deep learning accelerator unit on fpga. *arXiv preprint arXiv:1605.06894*.

Wang, L., Guo, S., Huang, W., and Qiao, Y. (2015). Places205-vggnet models for scene recognition. *arXiv preprint arXiv:1508.01667*.

Wang, L., Quek, H. C., Tee, K. H., Zhou, N., and Wan, C. (2005). Optimal size of a feedforward neural network: How much does it matter? In *Joint International Conference on Autonomic and Autonomous Systems and International Conference on Networking and Services-(icas-isns' 05)*, pages 69–69. IEEE.

Wang, Y., Chao, W.-L., Garg, D., Hariharan, B., Campbell, M., and Weinberger, K. Q. (2019). Pseudo-lidar from visual depth estimation: Bridging the gap in 3d object detection for autonomous driving. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 8445–8453.

Wang, Y., Nguyen, T., Zhao, Y., Wang, Z., Lin, Y., and Baraniuk, R. (2018). Energynet: Energy-efficient dynamic inference.

Wang, Y., Xu, C., Xu, C., and Tao, D. (2017). Beyond filters: Compact feature map for portable deep model. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 3703–3711. JMLR. org.

Wanpeng, C. and Wei, B. (2014). Adaptive and dynamic mobile phone data encryption method. *China Communications*, 11(1):103–109.

Warden, P. (2018). Speech commands: A dataset for limited-vocabulary speech recognition. *arXiv preprint arXiv:1804.03209*.

Warden, P. (2018). Speech Commands: A Dataset for Limited-Vocabulary Speech Recognition. *ArXiv e-prints*.

Weiss, A. R. (2002). Dhrystone benchmark: History, analysis, scores and recommendations.

Werbos, P. J. et al. (1990). Backpropagation through time: what it does and how to do it. *Proceedings of the IEEE*, 78(10):1550–1560.

West, D. B. et al. (1996). *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, NJ.

Wu, C., Wen, W., Afzal, T., Zhang, Y., Chen, Y., et al. (2017). A compact dnn: approaching googlenet-level accuracy of classification and domain adaptation. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 5668–5677.

Wu, J., Leng, C., Wang, Y., Hu, Q., and Cheng, J. (2016). Quantized convolutional neural networks for mobile devices. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4820–4828.

Xia, S., Lu, Y., Wei, P., and Jiang, X. (2017). Spindles: a smartphone platform for intelligent detection and notification of leg shaking. In *Proceedings of the 2017 ACM International Joint Conference on Pervasive and Ubiquitous Computing and Proceedings of the 2017 ACM International Symposium on Wearable Computers*, pages 607–612. ACM.

Xiao, H., Rasul, K., and Vollgraf, R. (2017). Fashion-mnist: a novel image dataset for benchmarking machine learning algorithms.

Xiaojin, Z. and Zoubin, G. (2002). Learning from labeled and unlabeled data with label propagation. *Tech. Rep., Technical Report CMU-CALD-02–107, Carnegie Mellon University*.

Xie, L., Wang, S., Markham, A., and Trigoni, N. (2017). Towards monocular vision based obstacle avoidance through deep reinforcement learning. *arXiv preprint arXiv:1706.09829*.

Xie, M., Zhao, M., Pan, C., Li, H., Liu, Y., Zhang, Y., Xue, C. J., and Hu, J. (2016). Checkpoint aware hybrid cache architecture for nv processor in energy harvesting powered systems. In *Proceedings of the Eleventh IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, page 22. ACM.

Xilinx (2011). Spartan-6 family overview. `https://www.xilinx.com/support/documentation/data_sheets/ds160.pdf`.

Yang, Y. and Hospedales, T. (2016a). Deep multi-task representation learning: A tensor factorisation approach. *arXiv preprint arXiv:1605.06391*.

Yang, Y. and Hospedales, T. M. (2016b). Trace norm regularised deep multi-task learning. *arXiv preprint arXiv:1606.04038*.

Yao, S., Hu, S., Zhao, Y., Zhang, A., and Abdelzaher, T. (2017a). Deepsense: A unified deep learning framework for time-series mobile sensing data processing. In *Proceedings of the 26th International Conference on World Wide Web*, pages 351–360. International World Wide Web Conferences Steering Committee.

Yao, S., Zhao, Y., Shao, H., Liu, S., Liu, D., Su, L., and Abdelzaher, T. (2018a). Fastdeepiot: Towards understanding and optimizing neural network execution time on mobile and embedded devices. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, pages 278–291. ACM.

Yao, S., Zhao, Y., Zhang, A., Hu, S., Shao, H., Zhang, C., Su, L., and Abdelzaher, T. (2018b). Deep learning for the internet of things. *Computer*, 51(5):32–41.

Yao, S., Zhao, Y., Zhang, A., Su, L., and Abdelzaher, T. (2017b). 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*, page 4. ACM.

Yerva, L., Campbell, B., Bansal, A., Schmid, T., and Dutta, P. (2012). Grafting energy-harvesting leaves onto the sensornet tree. In *Proceedings of the 11th international conference on Information Processing in Sensor Networks*, pages 197–208. ACM.

Yıldırım, K. S., Majid, A. Y., Patoukas, D., Schaper, K., Pawelczak, P., and Hester, J. (2018). Ink: Reactive kernel for tiny batteryless sensors. In *Proceedings of the 16th ACM Conference on Embedded Networked Sensor Systems*, pages 41–53. ACM.

Young, T., Hazarika, D., Poria, S., and Cambria, E. (2018). Recent trends in deep learning based natural language processing. *ieee Computational intelligenCe magazine*, 13(3):55–75.

Yu, R., Li, A., Chen, C.-F., Lai, J.-H., Morariu, V. I., Han, X., Gao, M., Lin, C.-Y., and Davis, L. S. (2018). Nisp: Pruning networks using neuron importance score propagation. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 9194–9203.

Zeng, X., Li, J., and Ren, Y. (2012). Prediction of various discarded lithium batteries in china. In *2012 IEEE International Symposium on Sustainable Systems and Technology (ISSST)*, pages 1–4. IEEE.

Zenke, F., Poole, B., and Ganguli, S. (2017). Continual learning through synaptic intelligence. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 3987–3995. JMLR. org.

Zhang, C., Li, P., Sun, G., Guan, Y., Xiao, B., and Cong, J. (2015). Optimizing fpga-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pages 161–170. ACM.

Zhang, H., Gummeson, J., Ransford, B., and Fu, K. (2011). Moo: A batteryless computational rfid and sensing platform. *University of Massachusetts Computer Science Technical Report UM-CS-2011-020*.

Zhang, J., Ghahramani, Z., and Yang, Y. (2008). Flexible latent variable models for multi-task learning. *Machine Learning*, 73(3):221–242.

Zhang, J., Wang, X., Li, D., and Wang, Y. (2018a). Dynamically hierarchy revolution: dirnet for compressing recurrent neural network on mobile devices. *arXiv preprint arXiv:1806.01248*.

Zhang, W., Li, R., Zeng, T., Sun, Q., Kumar, S., Ye, J., and Ji, S. (2016). Deep model based transfer and multi-task learning for biological image analysis. *IEEE transactions on Big Data*.

Zhang, Y. and Yang, Q. (2017a). An overview of multi-task learning. *National Science Review*, 5(1):30–43.

Zhang, Y. and Yang, Q. (2017b). A survey on multi-task learning. *arXiv preprint arXiv:1707.08114*.

Zhang, Y. and Yeung, D.-Y. (2013a). Learning high-order task relationships in multi-task learning. In *Twenty-Third International Joint Conference on Artificial Intelligence*.

Zhang, Y. and Yeung, D.-Y. (2013b). Multilabel relationship learning. *ACM Transactions on Knowledge Discovery from Data (TKDD)*, 7(2):7.

Zhang, Z., Luo, P., Loy, C. C., and Tang, X. (2014). Facial landmark detection by deep multi-task learning. In *European conference on computer vision*, pages 94–108. Springer.

Zhang, Z., Xu, S., Cao, S., and Zhang, S. (2018b). Deep convolutional neural network with mixup for environmental sound classification. In *Chinese Conference on Pattern Recognition and Computer Vision (PRCV)*, pages 356–367. Springer.

Zhang, Z., Yang, P., Ren, X., and Sun, X. (2019). Memorized sparse backpropagation. *arXiv preprint arXiv:1905.10194*.

Zhou, A., Yao, A., Wang, K., and Chen, Y. (2018). Explicit loss-error-aware quantization for low-bit deep neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 9426–9435.

Zhou, B., Khosla, A., Lapedriza, A., Oliva, A., and Torralba, A. (2014a). Object detectors emerge in deep scene cnns. *arXiv preprint arXiv:1412.6856*.

Zhou, B., Khosla, A., Lapedriza, A., Oliva, A., and Torralba, A. (2016). Learning deep features for discriminative localization. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2921–2929.

Zhou, B., Lapedriza, A., Xiao, J., Torralba, A., and Oliva, A. (2014b). Learning deep features for scene recognition using places database. In *Advances in neural information processing systems*, pages 487–495.

Zhou, H. and Liu, C. (2014). Task mapping in heterogeneous embedded systems for fast completion time. In *2014 International Conference on Embedded Software (EMSOFT)*, pages 1–10. IEEE.

Zhou, Q. and Zhao, Q. (2015). Flexible clustered multi-task learning by learning representative tasks. *IEEE transactions on pattern analysis and machine intelligence*, 38(2):266–278.

Zhou, Z., Chen, X., Li, E., Zeng, L., Luo, K., and Zhang, J. (2019). Edge intelligence: Paving the last mile of artificial intelligence with edge computing. *arXiv preprint arXiv:1905.10083*.

Zhu, J.-Y., Park, T., Isola, P., and Efros, A. A. (2017). Unpaired image-to-image translation using cycle-consistent adversarial networks. In *Proceedings of the IEEE international conference on computer vision*, pages 2223–2232.

Zhu, X. and Goldberg, A. B. (2009). Introduction to semi-supervised learning. *Synthesis lectures on artificial intelligence and machine learning*, 3(1):1–130.

Zhu, X. J. (2005). Semi-supervised learning literature survey. Technical report, University of Wisconsin-Madison Department of Computer Sciences.

Zimmermann, R., Deuschl, G., Hornig, A., Schulte-Mönting, J., Fuchs, G., and Lücking, C. (1994). Tremors in parkinson's disease: symptom analysis and rating. *Clinical neuropharmacology*.