Title
Learning-Based Techniques for Energy-Efficient and Secure Computation on the Edge

Author
Guo, Jia

Publication Date
2019

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA

Los Angeles

Learning-Based Techniques for Energy-Efficient and Secure Computation on the Edge

A dissertation submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Jia Guo

2019

ABSTRACT OF THE DISSERTATION

Learning-Based Techniques
for Energy-Efficient and Secure
Computation on the Edge

by

Jia Guo
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 2019
Professor Miodrag Potkonjak, Chair

In the paradigm of *Internet-of-Things* (IoT), smart devices will proliferate our living and working spaces. The recent decade has already witnessed an explosive growth of smartphones and wearable devices. A plethora of newer and even more powerful systems are emerging. IoT will enable more fluid human-computer interaction and immersive experiences in smart homes. IoT will facilitate rich sensing and actuating in intelligent warehousing and manufacturing. IoT will also empower fast and accurate perception and decision making in autonomous vehicles. The paradigm has elevated the role of the devices that constitute the edge of the network. Because of the sensitive nature and the sheer volume of the data generated by those devices, *edge computing* becomes a more effective and efficient option. While it brings better privacy protection and latency reduction in applications, edge computing is associated with various constraints. For the sizable list of devices that are operating on batteries, their sustainable operation usually calls for extremely efficient and judicious use of energy. Further, the inherent vulnerability accompanying the deployment in unsafe environments requires extra layers of security.

In this dissertation, we study the energy and security problems of edge computing in the context of machine learning. We present various learning-based techniques for improving energy efficiency. In contrast to the traditional resource allocation mechanisms that typically adopt handcrafted rules and heuristics, we adopt a framework where we use machine learning learn to create

online resource allocation strategies from optimal offline solutions. We demonstrate the effectiveness of the framework in applications and scenarios including DVFS, computation offloading and sensor networks. Our machine learning enabled strategies have approximated optimal solutions with an average of 2% error and achieved 40% in energy savings. In an increasing number of edge computing applications, machine learning algorithms themselves constitute the core and the major workload. Many of those applications have high energy consumption and are vulnerable to security issues such as intellectual property theft. To solve the problems, we derive techniques directly from the machine learning processes. We present computer vision-oriented adaptive subsampling strategies for image sensors, deep neural network pruning and customization, and deep neural network watermarking for intellectual property protection. With little to no loss in the performance, we improve the energy efficiency and security machine learning systems.

The dissertation of Jia Guo is approved.

Miloš D. Ercegovac

Jens Palsberg

Gregory J. Pottie

Miodrag Potkonjak, Committee Chair

University of California, Los Angeles

2019

To my parents.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

First and foremost, I would like to thank my adviser and committee chair, Professor Miodrag Potkonjak, for his support and guidance through all of my years at UCLA. I would not have become a graduate student in computer science without his help in the first place. I would not have grasped the art and craft of scientific research without his instructions. Not only have I benefited from the invaluable insights and advice on that he offered on my career and life, but I have also been inspired tremendously by his perseverance and optimism in the face of unimaginable hardship.

To my doctoral committee members, Professors Miloš D. Ercegovac, Jens Palsberg, Gregory J. Pottie, and the late Professor Mario Gerla, thank you for your insightful comments and advice on my prospectus, dissertation, and final defense, and for aiding me in the completion of my graduate studies.

I would like to thank my professors in UCLA, my professors at Zhejiang University and my mentors at MSRA for not only providing me with a solid foundation in science and engineering but also giving me the passion to learn, research and build. I would also like to thank my hosts and mentors at Baidu, InferVision, and Google for offering me the opportunities to be exposed to the latest technological trends and gain valuable industry experiences.

Many thanks to my collaborators and colleagues at UCLA: James B. Wendt, Teng Xu, Hongxiang Gu, and Moustafa Alzantot, who discussed research ideas with me, worked together with me on research projects, and offered me a great deal of help in carrying out experiments and in writing. I would like to thank James B. Wendt for helping me kick off my research at UCLA with the work in Chapter 2. I would like to thank Teng Xu for being my Ph.D. mentor and supporting the work in Chapter 4. I would like to extend a huge thank you to Hongxiang Gu not only for assisting me with the work in Chapter 5, but also for being a generous sharer of knowledge and ideas, an articulate critic, and the only lab mate for an extended period of time. Special thanks go to Professor Potkonjak who oversaw and co-authored all of the research work that constituted the chapters in this thesis.

And finally, I would like to thank my family and friends who have supported me throughout my academic journey. A special thank you to my parents for giving me unconditional love and support,

motivating my academic pursuits, and providing me with the means and confidence to dream big. And to Daxiao for always being there to give her unwavering support and encouragement, despite the distance between us. Thank you.

VITA

2010–2014 B.Eng. (Electronic and Information Engineering)
Zhejiang University.

2014–2018 M.S. (Computer Science)
University of California, Los Angeles.

PUBLICATIONS

J. Guo, and M. Potkonjak, Evolutionary Trigger Set Generation for DNN Watermarking, *submitted*, Feb. 2019.

J. Guo, and M. Potkonjak, Watermarking Deep Neural Networks for Embedded Systems, *IEEE/ACM International Conference on Computer Aided Design (ICCAD)*, pp. 133, Nov. 2018.

J. Guo, H. Gu, and M. Potkonjak, Efficient Image Sensor Subsampling for DNN-Based Image Classification, *IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, Jul. 2018.

J. Guo, and M. Potkonjak, Pruning ConvNets Online for Efficient Specialist Models, *IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pp. 430-437, Jul. 2017.

J. Guo, and M. Potkonjak, Pruning Filters and Classes: Towards On-Device Customization of Convolutional Neural Networks, *International Workshop on Deep Learning for Mobile Systems and Applications (EMDL)*, Jun. 2017.

J. Guo, and M. Potkonjak, Coarse-Grained Learning-Based Dynamic Voltage Frequency Scaling for Video Decoding, *International Symposium on Power And Timing Modeling, Optimization and Simulation (PATMOS)*, Sep. 2016.

J. Guo, T. Xu, T. Stavrinos, and M. Potkonjak, Enabling environmentally-powered indoor sensor networks with dynamic routing and operation, *International Symposium on Power And Timing Modeling, Optimization and Simulation (PATMOS)*, Sep. 2016.

J. Guo, J.B. Wendt, and M. Potkonjak, Learning-Based Localized Offloading with Resource-Constrained Data Centers, *IEEE International Conference on Cloud and Autonomic Computing (ICCAC)*, Sep. 2015.

CHAPTER 1

Introduction

1.1 Internet of Things and Edge Computing

Since its conception at the turn of the century [6], Internet of Things (IoT) has gained tremendous traction in both the academia and the industry. As its name suggests, the paradigm of IoT is established upon the prevalence and interconnectedness of smart agents or "things". Through communication, sensing and actuating, they collectively undertake ambitious tasks. The concept has been adapted and applied to a wide range of industries, including manufacturing, transportation, surveillance, healthcare, environment, and home automation. It is estimated that there will be over 75 billion connected devices worldwide by 2025 [7], and by then the market is expected to reach $ 1.1 trillion [8]. IoT constitutes an even larger ecosystem if we take into consideration the trillion-dollar PC and smartphone industries, both of which are crucial parts of the paradigm. The paradigm of IoT promises to enable applications that transform business and people's lives.

IoT systems run a diverse range of workloads. To begin with, communication is the fundamental need of IoT systems. The ability to seamlessly integrate with smartphones over low power Bluetooth make wearable devices best-selling gadgets. The ad hoc communication channels allow massive sensor networks to be deployed in the field [9]. Secondly, sensing is one of the most important components. Systems that require sensing range from surveillance cameras in smart homes, to automated production line in smart factories. The sensing task can be as simple as reading a temperature sensor. It can also be as complex as encoding and decoding videos [10]. Lastly, recent advances in artificial intelligence have also pushed the boundaries of IoT systems. Machine learning (ML), in particular, stands out as a practical and effective method to supply IoT devices with autonomy and machine intelligence. Machine learning has been applied to infer activity from

smartphone sensor data [11] [12]. ML-based speech recognition has allowed smooth and natural conversations with smart home devices [13]. Autonomous vehicles heavily rely on computer vision algorithms that are based on deep convolutional neural networks [14]. Executing machine learning models have become a unique type of workload on modern IoT devices.

To accommodate the workload, edge computing has emerged as the preferred computation model in the paradigm of IoT [15]. As opposed to cloud computing, where computation occurs in the centralized cloud servers, edge computing defines a framework where computation is carried out on the edge of the network. The edge can refer to a gateway or a cloudlet with computing capabilities. More often, it refers to the "things" themselves. The motivations for adopting edge computing are manifold. Most importantly, with their growing sensing capabilities, the edge will generate an increasingly large amount of data. In many applications, edge devices can process data with much better energy efficiency. Fitted with engines that carry 5000 sensors, a single Bombardier C Series jetliner generates 844 TB of data during a 12-hour flight [16]. Consumer electronics are also equipped with ever-evolving sensing capabilities. One second of a raw 30 FPS 4K video, which many of today's smartphone cameras are capable of producing [17], amounts to over 600 MB of data. Uploading such a large amount of data to the cloud for processing is extremely inefficient, if not infeasible. Secondly, edge computing safeguards users' privacy. Smart home devices such as the Amazon Echo series [18] and the Google Home/Nest series [19], have gradually found their way into households. Due to their ability to collect data through microphones and cameras, they have been under strict public scrutiny [20]. Companies have established rules to ban the compromising act of transferring sensitive data to the cloud. As a result, edge computing remains the only viable option for data processing. Finally, edge computing induces much lower latency compared to cloud-based solutions. Applications such autonomous vehicles require data to be processed in real time. Today's mobile network on average takes over 100 ms for a round trip from the device to the cloud [21], and it is far from satisfying latency requirements of those applications. Only computing on the edge will be able to satisfy the stringent requirements in latency. It has become clear that IoT devices are transforming from data consumers to data producers. Edge computing is going to play a more important role as the devices undergo the transition.

1.2 Challenges and Motivation

The realization of the IoT and edge computing vision is subject to a different set of constraints comparing to those from the PC and internet era. In particular, energy and security are among the most important constraints.

Many IoT devices are battery-powered and face energy constraints. The most relatable example might be the various wearable devices that people today carry in their daily lives [22]. Smartphones are also an integral part of the loop by serving as a central information hub connecting the devices. Without an exception, the devices mentioned above are all powered by batteries. Although various energy-saving features have been built in, the battery life is still extremely limited, in some cases as low as an hour [23]. Autonomous vehicles are also powerful moving IoT nodes that are powered by batteries. While vehicles are using the batteries of a much higher caliber than those of wearable devices, energy efficiency remains an important issue for them. The intensive computation required for the perception and control components makes the autonomous vehicles extremely power hungry. It is reported that sensing and computation consume around 2500 watts of power [24], at which rate the commercial electric car batteries will be drained in hours even without the user driving it [25]. In addition, IoT has brought automation to what used to be labor-intensive industries such as farming and environmental monitoring. For example, agriculture technology companies are using battery-powered drones to monitor crop growth [26]. Even in remote areas where no infrastructure exists to provide with electric power, IoT is playing a role. Researchers have long envisioned the large scale deployment of environmentally powered sensor networks [27]. The initiative is slowly coming into fruition with the commercialization efforts such as the deployment of solar-powered devices in the ocean [28]. We are convinced that energy efficiency is a crucial requirement of the IoT systems. Reducing the energy consumption of such systems is thus widely recognized as an active research field.

Another important desideratum of IoT systems is security. Tasked with connecting the physical world, such systems are bound to encounter diverse operating environments. Unlike PCs and servers, which are safely harbored in offices or data centers, edge devices are exposed to attacks that range from the physical layer, computation, and communication layer, to the semantic layer. In

one incident, an attacker was able to control other connected devices through a hack that involves physical access to a Nest thermostat [29]. Many other incidents involve cameras being attacked remotely over the internet [30]. The complexity of the security issues with IoT is not only reflected by the diverse ways of attack, but also the target of the attack. Information from both the user and the developer may be compromised. One critical type of attacks that target developers is intellectual property (IP) theft. With physical access to compromised devices, it is easier for attackers to steal proprietary data and conduct software piracy than in the cloud or web-based setting. IP is usually considered one of the most important competitive advantage of a tech company. IP theft will incur a considerable loss to the developers and harm the health of the ecosystem.

To improve the energy efficiency and security of different types of workloads in IoT systems, we adopt different strategies. The communication and sensing workload in IoT systems sometimes depend on the external factors. Congestion on the Internet, like many other things in the world that adhere to our society's schedule, follows a periodic pattern. It is only wise for IoT systems to avoid busy periods in order to save resources. As another example, the amount of computation needed to process data, e.g. decode a video, can be dependent on the data itself. It is unclear how much computational power to allocate in order to process the data timely and efficiently. All of the examples above more or less involve a prediction problem. We need to predict how much resources we need before we allocate them efficiently. Simple heuristics often fail to capture the complex nature of the situations. This is when we need the help of machine learning. There are challenges associated with using machine learning to solve the prediction problem. To begin with, we usually don't have the labeled data which is crucial to training a machine learning model. In addition, it is never easy to design lightweight models that are able to predict with high accuracy. But if we are able to create models with high performance, the potential benefits we can bring to the systems are huge.

The third type of workload is machine learning. While we could treat the machine learning models as black boxes as any other workloads, it is not the most effective approach. Many machine learning models are complex non-linear functions. Methods that improve the energy efficiency and security of machine learning applications should align with, if not be part of, the optimization process of models. Deep neural networks, for instance, are typically trained with gradient descent

	Non-ML Applications	ML Applications
Energy Efficiency	• Code Offloading with Resource-Constrained Cloud (Ch. 2) • Learning-Based DVFS (Ch. 3) • Sustainable Operation of Environmentally Powered WSN (Ch. 4)	• Adaptive Image Sensor Subsampling for Computer Vision (Ch. 5) • Customized On-Device DNN Pruning (Ch. 6)
Security		• Secure DNN Watermarking for IP Protection (Ch. 7) • DNN Watermarking with Improved Robustness (Ch. 7)

Table 1.1: Major contributions and organization of the dissertation.

algorithms. The neurons in the model are the result of a fragile co-adaptation [31]. As a result, any subtle changes in the weights may have huge effects on the inference results. If we should like to reduce the model size or modify the model to incorporate security features, we will need a method to negate such effects. Finding a proper method is a challenge, especially given the unique constraints posed by different scenarios. In addition, many peripheral components that are interdependent with machine learning models should also be treated in a similar manner. For instance, any changes to the sensor that provides data to machine learning models should be made in such a way that minimizes the impact on the models. In summary, we need to design a method that can be integrated with the machine learning process in order to make it more energy efficient and secure.

1.3 Contributions and Organization

This thesis addresses the energy and security issues of edge computing systems by examining a handful of popular applications. Table 1.1 summarizes the contributions and the organization of the thesis. As different workloads require different methods, we categorize the applications into ML-based and non-ML-based ones. The set of methods we present in Chapter 5, Chapter 6, and

Chapter 7 fall into the ML-based category. Chapter 2, Chapter 4, and Chapter 3 describe learning-based techniques designed for various non-ML applications.

We begin the thesis by presenting methods to improve energy consumption for mobile offloading, dynamic voltage and frequency scaling (DVFS), and environmentally powered sensor networks. Code offloading refers to the frameworks where parts of the program are migrated from the IoT devices to the cloud in order to reduce energy consumption. When the cloud does not have an infinite amount of resources, it should prioritize programs that are most effective in bringing energy savings. In Chapter 2, we propose scheduling techniques and a machine learning-based offloading algorithm to further to address the issue. In Chapter 3, we focus on the energy management technique called dynamic voltage and frequency scaling (DVFS). The power consumption of a circuit is proportional to the square of the operation voltage. DVFS saves energy consumption by assigning a lower CPU voltage and lowering the frequency and slowing the execution when there is a smaller workload. Without application-level semantic information, it is inherently hard to find a suitable CPU frequency. Using video decoding as an example, we propose to use machine learning for CPU frequency prediction and improve the effectiveness of DVFS. Chapter 4 offers a case study of an environmentally powered sensor network. We analyze the energy harvested by each sensing and message forwarding node and propose a routing and operation schedule that maximizes the lifetime of the entire sensor network. The chapters follow a unified procedure. We first design offline algorithms that find optimal or near-optimal solutions. Then we learn to approximate such solutions and derive our online learning-based strategies.

Next, we introduce a set of techniques to improve the energy efficiency of programs that center around machine learning. In Chapter 5, we aim to bridge the gap between the ultra-high resolution of commercial image sensors and the low resolution needed for computer vision algorithms. Instead of sampling the entire pixel array of the image sensor, which is extremely inefficient in terms of energy, we propose an adaptive subsampling strategy that reduces the number of pixels needed to create an image. The strategy is designed to maintain the performance of computer vision algorithms when the sampled images are used as inputs. Modern computer vision pipelines usually include stages that range from image sensing to inference using machine learning. Following the work on image sensing, we present the energy consumption optimization of machine learning

models in Chapter 6. In particular, we prune channels from convolution neural networks to create faster, energy efficient and customized models. The approaches we introduce are closely coupled with the intrinsic characteristics of the machine learning models and provide unique insights into their respective problems.

Finally, we discuss techniques to improve the security of IoT systems. In Chapter 7, we focus on the problem of IP protection in the wake of massive commercialization of deep neural networks. The problem is especially prominent when attackers have complete access to the IoT devices and may be able to steal the intellectual property with various attacks. In order to protect the IP right of the model owners, we propose to embed watermarks in the deep neural network models to enable identification and proof of ownership. We first layout a method where we place the owner's signature onto regular inputs to create the "trigger set", a type of inputs used to trigger a specific behavior of the neural network. By showing the disproportionally low probability of such behavior occurring naturally, the owner identifies and proves his ownership. After that, we describe an evolutionary algorithm-based technique to improve the robustness of the method against fine-tuning attacks. We integrate the proposed framework in the training processing of the neural network in order to make the watermark effective.

CHAPTER 2

Learning-Based Code Offloading under Cloud Resource Constraints

Energy is one of the biggest constraints for the development of edge computing. In recent years, researchers have proposed various frameworks that enable offloading applications to the cloud as a way of saving energy. Cuervo *et al.* [32] introduced MAUI in 2010, a code offloading framework that makes fine-grained offloading decisions at run time. Chun *et al.* [33] also approach the same problem with a VM based offloading framework, where an image of the mobile system is set up in the cloud so that minimal effort is needed to modify the original program. Kosta *et al.* [34] takes the same approach a step further by enabling scalable parallel execution in the cloud. Even though these systems are in the early stages of development, we believe that offloading can be a viable solution to the energy problem.

Though researchers have used *mobile cloud computing* to denote the new paradigm [35], there are key differences between offloading in edge computing systems and traditional cloud computing systems. First, the location of execution(i.e. locally or in the cloud) is flexible. A task can be executed in the cloud if in such way energy can be saved. Or the task can be executed locally if the offloading overhead outweighs the benefit. Traditional cloud computing systems put heavy emphasis on reliability and availability because of the all-or-nothing style service they offer. When a web server is down, the website cannot be accessed until the outage is over. Similarly, users of EC2 would not expect the service to be unavailable and prepare a ready-to-run local version. In the offloading settings, however, tasks always have the option to be executed locally. Therefore, a better definition for offloading is a *cloud-based augmentation* [36]. It is intrinsically a bonus feature that enhances user experience. Second, offloading frameworks provide accurate profiling of tasks. The profiling is usually done as part of the run-time optimization engine. In MAUI,

for example, program profiles consist of software states and data, CPU cycles and execution time [32]. The MAUI solver takes into account network condition and program profile and solves for a program partitioning strategy that minimizes the smart phone's energy consumption under latency constraints. Such fine-grained profiling of programs provides prior knowledge of heterogeneous tasks, which is not accessible in the traditional cloud computing environment. This allows for fine-grained task scheduling in the data center. It also enables the evaluation of tasks in terms of their efficiency in saving energy.

Those characteristics provide new opportunities for data centers as well. The data center nowadays is characterized by over-provisioning. A typical data center uses only 10-50% of its computing power for most of the time [37]. Given poor energy-proportionality [38], a large cost is incurred not only by the one-time purchase of hardware but also by the energy consumption of the under-utilized node. Sophisticated load prediction techniques [39] [40] may be able to alleviate the overhead. However, such cost is intrinsically caused by the availability requirement of the cloud, or requirement that cloud service being *on-demand self-service* [41]. The problem can only be alleviated rather than solved. A data center targeting offloading applications, on the other hand, has the potential to change the picture. We envision a cost-aware data center that provides constrained resources for offloading. In the face of resource shortage, the data center becomes selective in scheduling tasks such that the maximum amount of energy is saved globally. The fine-grained task profiles enable the selection of tasks. The flexibility of offloading frameworks allows some tasks to be executed locally. The efficiency of energy saving and the low cost will justify the system design as a service. Overall, we believe such data center design is well justified and suited for offloading scenarios.

The difficulty in designing such systems mainly involves solving an optimization problem: given data center resource constraints, how can we schedule tasks such that the global energy savings are maximized. The key is to design a solution that solves the optimization problem with low overhead. In particular, the system is desired to be localized, i.e., an optimal offloading decision should be made by mobile devices without communicating with the data center to avoid unnecessary energy overhead. Another aspect to consider is the fairness of the shared system. There is a trade-off between the maximization of energy savings and a fair share of the resources.

Only a few frameworks have been proposed that seek to optimally allocate resources when multiple users offload simultaneously. Shu *et al.* modeled the system as a queuing network, and studied the global energy consumption and delay tradeoff [42]. Yet the system is not localized as they assume tasks are from a single mobile client. Barbarossa *et al.* targeted multiple users collocated at a small cell base station, and proposed an algorithm that jointly optimizes the computation resources and radio power through the formulation of a convex function [43]. The algorithm again requires a centralized system, and wireless channel contention in their problem might not be valid in a more general setting. In the same collocated mobile clients problem setting, Chen proposed localized game theory based solution under wireless communication channel resource constraints [44]. In conclusion, existing works on resource allocation either a) target specific scenarios and lack generality, or b) rely on centralized schemes, which increase communication overhead. In addition, no work has considered the fairness issue yet.

Our solution is a framework where the system learns online localized decisions from offline centralized solutions. In the offline phase, all task information is given as prior knowledge, and the data center needs to obtain a near-optimal solution that maximizes energy savings. We formulate the offline problem as a scheduling problem and obtain upper bounds through Linear Programming (LP). We propose a probability-based algorithm that is able to obtain an approximate solution in polynomial time. The results are then used to train a logistic regression classifier. In the online phase, mobile clients use the classifier to make localized real-time decisions on offloading, thereby avoiding sending out inefficient tasks. When considering fairness, the users allows a probability for the tasks to be offloaded without going through decision making by the classifier. The data center uses a proportional fair scheduler and achieves fairness among offloaded tasks while maintaining relatively high performance. The system workflow is shown in Figure 2.1. Based on simulation using real-life workload traces, we observe that our offline algorithm produces a near-optimal solution, and the classifier-based online approach yields result close to the offline while outperforming baselines by a large margin.

Our contributions are multi-fold. First, to the best of our knowledge, we are the first to propose a learning-based localized solution to the resource allocation problem in offloading systems. Second, we propose a novel time-efficient offline scheduling algorithm that, compared with the

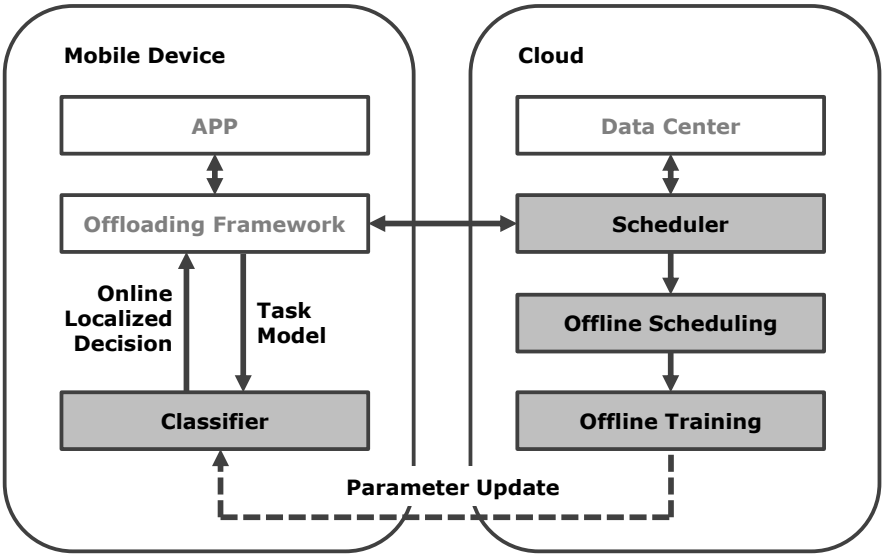


Figure 2.1: System workflow of code offloading

derived upper bounds, achieves near-optimal results. Third, we are the first to consider the fairness of the data center resource allocation problem in the offloading scenario. Lastly, we use real-life work load traces to build realistic simulation of the offloading system.

The rest of the chapter is organized as follows. Section 2.1 analyzes real-life traces, introduces the models we use for simulation and formulates the problem. Section 2.2 introduces the details of each parts of the framework: Section 2.2.1 covers offline scheduling problem and our approximation algorithm, Section 2.2.2 describes classifier and feature selection, Section 2.2.3 introduces fairness and proposes mechanisms for fairness in the framework, and Section 2.2.4 shows how the system work in simulation. Section 2.3 evaluates the system: Section 2.3.1 introduces the simulation setup and analyzes the synthetic workload to provide background knowledge, Section 2.3.2 evaluates the performance of the offline scheduling algorithm, classification, as well as overall performance in energy saving, Section 2.3.3 shows the effect of various system parameters have on performance and Section 2.3.4 shows the trade-off between performance and fairness.

2.1 Preliminaries

Consider a scenario where various mobile clients offload tasks to the cloud for remote execution. The mobile clients neither communicate nor interfere with each other. The tasks have a release time as well as a deadline and are also characterized by a set of parameters derived from computation, network, and energy models. The tasks can either run locally or be executed remotely in the cloud.

2.1.1 System Model

In this subsection, we define the model we use for the chapter. We describe various parameters of the tasks in terms of computation and network, from which we derive per task-based energy models.

2.1.1.1 Computation

Let f^l represent the CPU frequency of the mobile device, and f^c the CPU frequency of the data center[1]. If the task runs in the data center, there's an additional speedup ratio of r due to additional memory [45]. Consider a task T that takes c clock cycles to run, requiring $\tau^l = c/f^l$ seconds to execute locally. The time required to execute the task in the cloud is $\tau^c = \tau^l \cdot f^l/(rf^c)$. We assume the data center has a fixed number of machines, represented by the capacity, C, each can only run one task at a time. We assume that tasks can be executed immediately after it arrives. In addition, the scheduling of tasks is *non-preemptive*, i.e. once the data center makes the commitment to schedule a task, the execution of the task cannot be interrupted and resume later.

2.1.1.2 Network

The mobile network is intrinsically unpredictable and hard to model. We resort to a simple model where the mobile client knows the average bandwidth of n^b of its network. Assume that task T has a data size of n^d bytes, then it takes $\tau^t = n^d/n^b$ seconds to transmit the data over to the data center.

[1]We use superscripts to denote the "type" of a variable so that subscripts can be reserved for the numbering of individuals

We assume that the data center has sufficient network bandwidth. Thus the offloading processes will not experience any bottleneck in data transmission.

2.1.1.3 Energy

We adopt a simple energy model where the mobile device operates in one of the three states: computing state with power P^c, transmitting state with power P^{tr}, and idle state with power P^i [45]. The energy for local execution is $P^c\tau^l$, the energy for transmission is $P^{tr}\tau^t$, and the energy spent waiting for remote execution results is $P^i\tau^w$, where $\tau^w \geq \tau^c$ is the waiting time. Notice that the waiting time can be longer than the time of execution.

One interesting fact reported by the authors of [33] is that the benefit from offloading grows more than linearly as computation size increases. It is due to the fact that larger energy savings will amortize the fixed "start-up" cost of various components triggered by the code migrating process. For example, there are costs associated to power up the radio, to save program states, etc., which are a roughly fixed cost and do not grow with program sizes. We accommodate this aspect in our model by adopting a fixed term in the calculation of migration energy cost.

Tasks will be in one of the three states: successfully offloaded and executed in the cloud, offloaded yet not scheduled due to resource limits, and locally executed (without being offloaded). We denote the states by `remote`, `offloaded`, and `local`, respectively. The energy savings ΔE of different states are:

$$\Delta E = \begin{cases} E^l - E^c & \text{if } \texttt{remote} \\ -E^c & \text{if } \texttt{offloaded} \\ 0 & \text{if } \texttt{local} \end{cases} \tag{2.1}$$

2.1.1.4 Task

The final task model is listed in Table 2.1. We choose to consolidate the task parameters into a simpler abstraction consisting of two categories, time and energy. The table also shows how the parameters are derived, where the additional parameters involved (as we show in the following

Parameter	Explanation	Derivation
τ^l	Local exec. time	c/f^l
τ^c	Cloud exec. time	$\tau^l f^l/(rf^c)$
τ^t	Transmission time	n^d/n^b
E^l	Local exec. energy	$P^c\tau^l$
E^c	Offload energy overhead	$P^{tr}\tau^t$ + $P^i\tau^w$
t^r	Release time	
t^d	Deadline	

Table 2.1: Code offloading task parameters

section) are generated with reference to real-life models. Notice that for convenience we define t^r to be the time when tasks can start execution in the cloud. The actual offload decision is made earlier. We further define a few variables that are useful in our algorithms and discussion. *Benefit*, b refers to the efficiency of energy saving of a task, i.e. $b = \Delta E'/\tau^c$. Since ΔE is not known in advance, we use the estimated energy savings, $\Delta E' = E^l - E^c$. *Slack* defined by $t^d - t^r - \tau^c$, is the period that a task can afford to wait before its deadline is reached. We test our model in the simulation and it shows accordance with experimental results from the offloading frameworks [33] [34].

In building the system model, we carefully seek a balance between simplicity and fidelity of the model. The scheduling problem lies on an extremely simple abstraction, where only a single type of resources is studied. On the other hand, the model for data center often involves sophisticated modeling of CPU, memory, I/O, network, etc. [46]. To resolve the conflict, we rely on the following observations. First, tasks in offloading are more computation intensive than data-intensive, as demonstrated in the typical applications like a chess game, face recognition, etc [32]. Given the limited network bandwidth of cellular networks and energy constraints, only a relatively small amount of data can be offloaded (as compared to generic cloud applications). Thus CPU is more likely to be the single bottleneck in resource allocation. Second, task execution is not distributed among multiple machines. Researchers have shown that parallel execution does not improve the performance linearly with an increasing number of machines used [34]. Thus we

assume tasks will be running only in a single container (physical machine or VM, denoted by *machine* for the rest of the chapter). Thus unlike the situation for modeling MapReduce tasks, the I/O bandwidth of machines in our case is not of much concern. Those observations justify our approach to attribute execution time cost only to computation and using the abstraction that each task runs on a single machine.

2.1.2 Generating System Parameters

Since no offloading systems exist in full production, there is currently limited available real usage data. We can only assume the similarity between some aspects of existing systems that target mobile computing systems and the offloading scenario. The answer to the following three questions are of particular importance to our study:

1. what is the distribution of computation resource usage of offloading tasks;
2. how can we characterize the arrival of requests;
3. how the usage is distributed among users.

To answer the first question, we examined the traces of Google computing clusters [47], with the assumption that the distribution of computational resource usage of tasks in the generic cloud computing environment is similar to that in the mobile cloud computing environment. To answer the second question, we use the cellular network access traces of a cellular network provider in the US, with the assumption that the network access data reflects the behavior of mobile users, especially in terms of usage as a function of time. We found that the CPU usage follows a power law distribution while the disk usage follows a log-normal distribution. The generation of the rest of the parameters is described in Section 2.3.1.1. To answer the third question, which is critical for the study of fairness, we use mobile network usage data. The network usage reflects different users' frequency and habits of using mobile internet. We assume that it bears a similarity with the frequency of offloading with different users.

2.1.2.1 Distribution of CPU and Data Size

The Google cluster traces provide a month-long workloads traces of 12,000 machines. Researchers have studied the characteristic traces [48] and some even proposed ways to synthesize data center workload based on these traces [49]. In our project, we are only interested in the high-level distribution of task sizes. Typically the tasks we use are MapReduce type tasks. We aggregated the CPU usage overall jobs that belong to one task to obtain per-task total CPU usage. The distribution of CPU usage is shown in Figure 2.2(a). The linearity in the logarithmic scale implies that distribution follows a power law distribution. The distribution of disk usage as a function of CPU usage is shown in Figure 2.2(b). Figure 2.2(c) extracts tasks with CPU usage in a small range and plot their disk usage. We observe that the distribution of disk usage is log-normal to the corresponding CPU usage of the task.

Along with the observation, we generate the number of CPU cycles c from a power-law distribution, and data size n^d in a corresponding log-normal distribution.

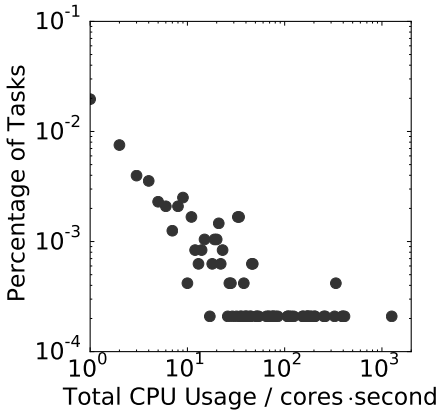
2.1.2.2 Arrival Model

Workload prediction of data centers has been a difficult research problem. In this chapter, we hold the assumption that the workload of offloading bears similarity with that of mobile phone users visiting the Internet. We believe the variance and unpredictability of the offloading workload can be characterized by the fluctuation of cellular network record.

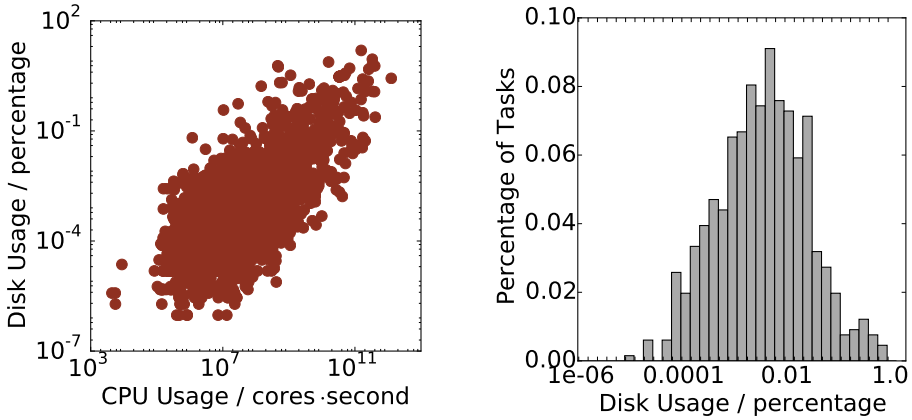
It's long been shown that Internet traffic is "bursty" at all time scales or self-similar [50]. Recently, Han *et al.* also showed that the requests arrival process in the cloud is self-similar [40]. Self-similarity is characterized by a series X having the same correlation function with its "aggregated" version $X^{(m)}$. The aggregated series is defined by the following equation

$$X^{(m)}(t) = \frac{1}{m} \sum_{i=(t-1)m+1}^{tm} X(i)$$

The autocorrelation function over the series, $r(k) = E((X(t) - \mu)(X(t+k) - \mu))/\sigma^2$, exhibits a power law decay over a long range: $r(k)\ k^{-\beta}$ as $k \to \infty$. This is called *long-range dependence*. The *Hurst* parameter $H = 1 - \beta/2$, measures the decay of such dependence over range. Thus it's



(a)



(b) (c)

Figure 2.2: Characteristics of Google cluster traces: (a) shows the distribution of CPU usage; (b) shows the distribution of disk usage against the distribution of CPU usage; (c) shows distribution of disk usage of tasks with CPU usage between a small range

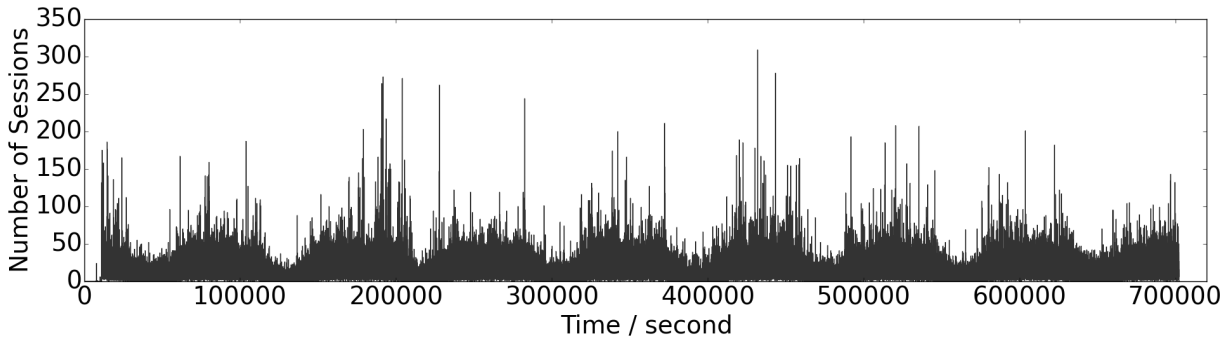


Figure 2.3: Sample of cellular network usage traces

also used as an indicator of the degree of self-independence. Basically, self-similarity implies the fact that bursts and spikes on the second's scale also exist in tens of seconds or minutes scale. Thus such an arrival process is intrinsically hard to predict.

Self-similarity alone is not enough for the arrival model. In addition to burstiness, traffic also contains a cyclic element that characterizes the change in people's activities. Many researchers have observed workloads daily, weekly, and seasonal cyclic patterns in data center [37] [39].

We conclude that it is both easier and more convincing to use realistic traces. We thereby turn to real life traces of cellular network, which contains both cyclic and bursty behavior, as shown in Figure 2.3. Notice usage in the sample is measured in sessions, which is defined as a period of semi-continuous network activity of a single user over a span of time. To utilize the traces, we define our model in discrete time slots (e.g. 10 seconds). The number of tasks that arrive at a particular time slot will be scaled from the number of sessions in the corresponding time slot in the traces.

2.1.2.3 Distribution of Usage

Figure 2.4 shows the distribution of network traffic of different users in bytes. Both axes are in log scale. It's clear that the distribution follows a power law.

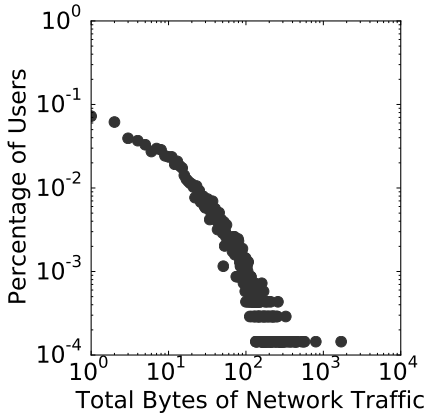


Figure 2.4: Distribution of network traffic of users

2.1.3 Problem Formulation

Now that the system model has been described, we proceed to formulate our problem. Define $D_i^l(t) \in \{1, 0\}$ to be the local decision on whether task T_i is offloaded to the cloud at time slot t, and $D_i^c(t)$ to be whether the data center executes T_i at time slot t. Under data center resources constraints, some of the tasks may be offloaded and others may not be. Among the offloaded tasks, only part of them may be executed in the cloud. The energy savings of T_i, ΔE_i, will then be either of the three cases described in Section 2.1.1.3 depending on whether the task is offloaded and whether it is executed in the data center. Our goal is to find an online localized decision process for $D_i^l(t)$ and $D_i^c(t)$ such that the overall energy savings $\sum_i \Delta E_i$ is near-maximal.

2.2 Method

To solve the optimization problem, we first examine it in an offline environment. We formulated our problem as a scheduling problem and showed upper bounds of the solution to the problem. We then proposed a probabilistic algorithm that is able to achieve near-optimal results. The information from offline scheduling is extracted by classification, where features are selected through regularization. We introduce the classification to local mobile clients, which enable them to make a localized real-time online decision for offloading. The algorithms for fairness scheduling is also

introduced in this section. An overview of the algorithms we used is listed in Table 2.2.

2.2.1 Offline Scheduling

The offline phase aims at finding a near-optimal solution to the assignment of tasks, by assuming centralized control and the knowledge of all the tasks. Despite the complicated models and procedures, the core task of the offline phase is equivalent of the following: we want to decide which tasks to be scheduled locally and which in the cloud, such that the energy is maximized. Thus offline problem can be simplified to a scheduling problem described in the following sections. The solution of the scheduling problem represents an ideal state of the execution of all the tasks. The tasks should all be in state `local` or state `remote`, and it should not matter what $D_i^l(t)$ and $D_i^c(t)$ is as long as there's one final conclusion for each task on whether and when it's executed in the data center, denoted by $D_i(t)$. For simplicity, we also ignore the cost of waiting. It remains an open question on how to solve the scheduling problem when the weight is varying.

Offline Alg.	**Summary**
Upper bound	Relax the problem to a linear program and obtain an optimal solution
Baseline: deadline	Schedule the task with earlies deadline first
Baseline: greedy	Schedule the task with highest benefit (of all tasks) first
Probabilistic	The proposed approximation algorithm
Online Alg.	**Summary**
Baseline: FCFS	The First-Come-First-Serve scheduling
Baseline: greedy	Schedule tasks with highest benefit (in the queue) first
Proportional fair	Schedule the user with the highest task benefit divided cumulative energy savings

Table 2.2: Overview of the scheduling algorithms for offloaded tasks

2.2.1.1 The Scheduling Problem

Scheduling problems have been exhaustively studied in the past. According to the survey [51], our problem falls in the category of minimizing the number of weighted tardy jobs, $P||\sum wU$. In the problem representation, P stands for *identical parallel machines*, where the number of machines corresponds to data center resources C in our model; w stands for the *weight*, which corresponds to the energy savings ΔE ; U stands for the *unit penalty*, where $U = 1$ if the task is not scheduled within deadline (referred to as *tardy job*), and 0 otherwise. By minimizing the energy savings "lost" with unscheduled tasks, i.e. $\sum wU$, it is effectively achieving our goal to maximize energy savings of scheduled tasks.

The aforementioned scheduling problem is NP-hard [51]. There exists a dynamic programming based pseudo polynomial solution [52] [53], but it takes $O(mn(max_j\{d_j\})^m$ to solve, where n is the number of tasks, d_j is the deadline and m is the number of machines [51]. It is apparently not feasible to obtain the optimal results even in the offline settings. Bar-Noy et al. proposed an approximation algorithm using relaxation from a linear programming upper bound [54]. Yet the linear programming problem is time-consuming to solve. Therefore, we propose a polynomial near-optimal algorithm to tackle the problem.

2.2.1.2 Bounds and Baselines

Bar-Noy et al. provided an non-preemptive upper bound which was claimed to be the best possible upper bound achievable [54]. Yet in our problem, their approach proved too time-consuming. We thus provide a running-time optimized linear programming (LP) based upper bound. The LP relaxes the problem by a) allowing preemptive scheduling; and b) allowing a fraction of a task to be scheduled in such time slots. For any task, the sum of the fractions should be no more than the original execution time. At any time slot, the sum of all fractions should not exceed the number of machines. We thereby define x_{it} to be the fraction for each task T_i at each time slot t. Note that x_{it} is 0 for $t \notin [t_i^r, t_i^d)$. The formulation of LP is as follows:

$$\underset{x_{it}}{\text{maximize}} \quad \sum_{i=1}^{N} \sum_{t=t_i^r}^{t_i^d-1} b_i \cdot x_{it}$$

subject to

$$\sum_{t=t_i^r}^{t_i^d-1} x_{it} \leq \tau_i^c, \text{ for each } i$$

$$\sum_{i=0}^{N} x_{it} \leq C, \text{ for each } t$$

$$0 \leq x_{it} \leq 1$$

We compare the two upper bounds in Table 2.5.

We use two baseline algorithms: schedule by the earliest deadline and a greedy algorithm. The earliest deadline algorithm is proved to be optimal in unweighted scheduling problems. The greedy algorithm schedules orders tasks according to their benefit b and schedules each task at the earliest time slot available. The results of algorithm performance are shown in Table 2.7.

2.2.1.3 The Probabilistic Approach

The proposed offline algorithm builds upon the greedy algorithm, which seeks to schedule as many high benefit tasks as possible, but it takes a step further by first "shifting" the tasks to the less congested area so as to schedule as many tasks as possible. The difficulty encountered when shifting a task, however, is that we have to consider the probability of another task being shifted to the same slot later. We thus come up with the method of shifting based on a probabilistic profile, where each task has a probability to be in all the time slots available to it. With the probabilistic profile, we can achieve the goal of reducing congestion by shifting each task just once, thus achieving linear complexity.

Now we define our approach in more details. Let n be the number of possible schedules for a task T, where $n = t^d - t^r - \tau^c + 1$, thus each schedule takes probability $1/n$. The probability of the task at a particular time slot is the sum of the probabilities of all the schedules that cover that slot. Figure 2.5 shows the probabilistic distribution of two example tasks. Both tasks have $t^r = 0$

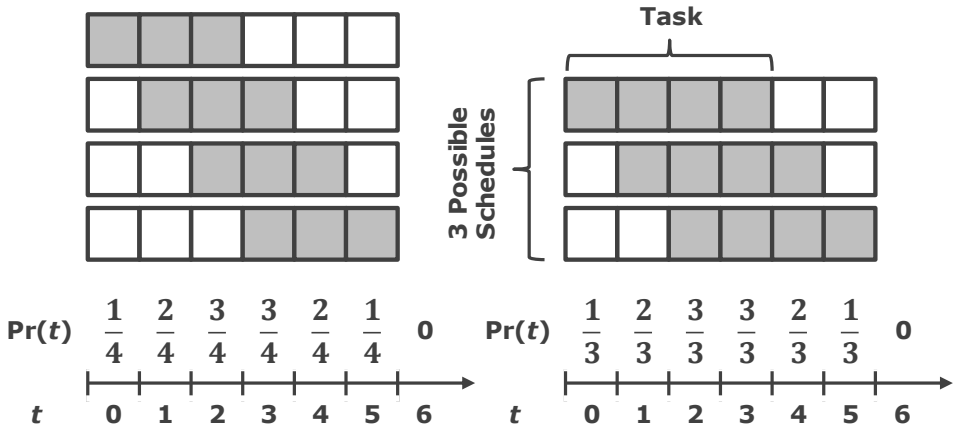


Figure 2.5: The probabilistic distribution of two example offloading tasks

and $t^d = 6$, while τ^c equals 3 and 4 respectively. Consider the first task, where one schedule covers slot 0, thus yielding a probability of 1/4; two schedules cover slot 2, thus yielding a probability of 2/4, etc. Let $t^t = \min(t^r + \tau^c - 1, t^d - \tau^c)$. The probability can be defined by Equation 2.2.

$$\Pr(t) = \begin{cases} \frac{t - t^r + 1}{n} & \text{if } t^r \leq t < t^t \\ \frac{min(n, \tau^c)}{n} & \text{if } t^t \leq t < t^d - \tau^c \\ \frac{t^d - t}{n} & \text{if } t^d - \tau^c \leq t < t^d \\ 0 & \text{otherwise} \end{cases} \tag{2.2}$$

Algorithm 1 describes the procedure of probabilistic shifting. The algorithm iteratively finds the least congested slot and shifts a task in the slot to a position with the minimum average probabilistic congestion. With bounded $t_i^d - t_r^d$, the algorithm execute in $O(N)$ time. After the shifting, we then apply the greedy scheduling algorithm to obtain the final offline results. Figure 2.6 shows a demo of our algorithm. In the figure, the load at the middle congested region is successfully shifted to the side.

2.2.2 Classification

In the classification phase, we train local offload decision $D_i^l(t)$ based on the offline solution $D_i(t)$. The idea is to enable local mobile clients to be able to assess the quality of tasks, and thus avoid

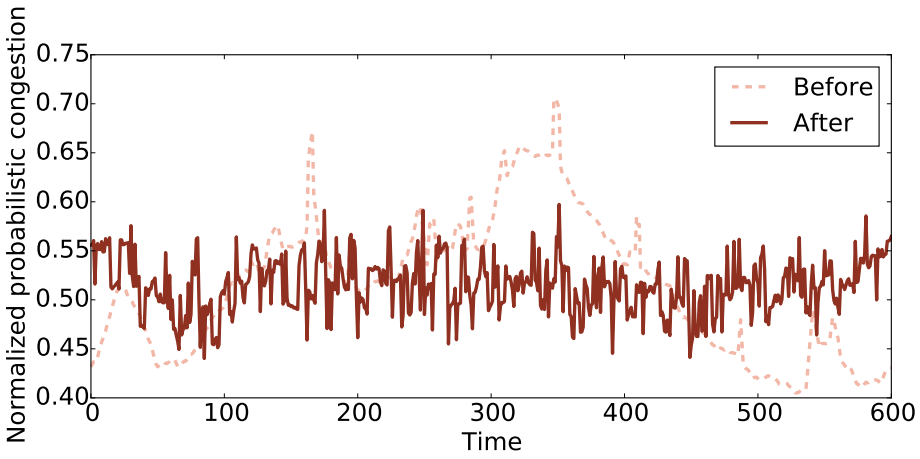


Figure 2.6: Demonstration of the probabilistic scheduling algorithm

Algorithm 1 Probabilistic Shifting Algorithm

$pc(t) \leftarrow$ probabilistic congestion at time slot t
while not all t marked explored **do**
    $t_{min} \leftarrow \underset{t}{\text{argmin}}\ pc(t)$
    $\mathbb{T} = \{T_i | t_i^r \leq t_{min} < t_i^d, T_i \text{ is not explored } \}$
    if $\mathbb{T} == \emptyset$ **then**
        mark t_{min} as explored
    else
        select $T_i \in \mathbb{T}$
        remove T_i from pc
        $t_i^{r\prime} \leftarrow \underset{t_i^{r\prime}}{\text{argmin}}\ \frac{1}{t_i^d - t_i^{r\prime}} \left(\sum_{t=t_i^{r\prime}}^{t_i^d} pc(t) + \tau_i^c \cdot 1 \right)$
        mark T_i as explored
        update $pc(t)$ where $t \in [t_i^{r\prime}, t_i^d)$
    end if
end while

offloading inefficient tasks that are unlikely to be scheduled by the data center. In short, if a similar task is scheduled in the optimal solution, then it's likely to be offloaded to the data center.

2.2.2.1 Selecting Classifier

We use *L1 regularized Logistic Regression* classifier for classification. The reason for choosing logistic regression are as follows. First, we favor classifiers that are simple in terms of representation. We envision that in an actual system, the data center will occasionally (e.g. every month) update parameters sets w of the model. It is beneficial for the update to contain as little data as possible, and thus our requirement to use classifiers that are "lightweight". Thereby we rule out instance-based approaches. Second, the regression model fits more conceptually with the problem. The underlying relationships between features are continuous, and that no causal relationship exists between any of the features. Thus we do not consider the choice of rule-based or Bayesian approaches. We thus choose logistic regression, which is a viable and simple method.

2.2.2.2 Selecting Features

Table 2.3 shows the list of features and the performance with different strengths of regularization (λ). The $+/-$ are the sign of the parameters of selected features. The feature that is most relevant to the offloading decision is $benefit$, which essentially represents the energy efficiency of the task. Following it is $slack$, another positively weighted feature which represents the flexibility of the task. If a task has high flexibility, it can wait longer in the queue and thus have a higher chance to arrive at a situation where the rest of the tasks in the queue are not as competitive as it is. The rest of the features are relevant but are not independent of each other, and thus the information they provide is redundant. We confirm in our experiment that adding more features do not improve performance significantly. Our final choice for regularization strength is λ_2, which selects benefit and slack as features.

2.2.2.3 Training

The training set is the offline solution $\{(x_i, D_i(t)), i = 1, \ldots, n\}$, where each task T_i has a feature vector x_i, and class label (1: scheduled , 0: not scheduled). The logistic regression model estimates the probability of $\Pr(D = 1|X = x) = 1/(1 + exp(-w^T x))$, where w refers to the parameters of the model. We use L1 regularization, which penalizes the model with an additional term $\lambda||w||$,

Feature	**Explanation**	λ_1	λ_2	λ_3
τ^l	Local Exec. Time			
τ^c	Cloud Exec. Time			$-$
τ^t	Transmission Time			
τ^s	Slack		+	+
E^l	Local Exec. Energy			
E^c	Offload Energy Overhead			
$\Delta E'$	Projected Energy Saving			
b	Benefit	+	+	+

Table 2.3: Feature selection for logistic regression

for feature selection and avoidance of over-fitting.

Notice that the term $D_i(t)$ depends on the time of the decision t. The traffic volume varies at different times of the day, and thus the parameters $w(t)$ should be a function of t. For simplicity, we choose $w(t)$ to be a discrete function, where there is a set of parameters for each predefined period (e.g. 1 hour). The logistic regression classifier is also susceptible to bias when the training set is unbalanced (e.g. too many 0 class). Thus in selecting training sets $D_i(t)$ from the offline solution, we use $t = release\ time$ for unscheduled tasks, $t = scheduled\ start\ time$ for scheduled tasks, and carefully keep number balanced. We demonstrate the results of training parameters with different granularity in Section 2.3.

2.2.3 Fairness

So far our method has been aggressively pursuing performance, i.e. energy saving of the system. Under the assumption that the system resources are shared by all the users, however, fairness is another important issue to address. In recent years, the concept of fairness is also studied in the context of data center [55] [46]. Our problem differs from them in two aspects: 1) The target of the system is the maximization of a global utility (energy savings). Thus there's an unavoidable trade-off between performance and fairness. 2) The objective of the user is to save energy. Thus

the fairness should also be measured in terms of energy.

The rest of this subsection first study two fairness definition, and the algorithmic implementation. We define fairness with respect to energy savings using the two definitions correspondingly. We describe a way to quantify fairness. Finally, we show how fairness may be incorporated into our system. The comparison of the fairness definitions and a trade-off between performance and fairness is demonstrated in Section 2.3.4.

2.2.3.1 Max-min Fairness

One of the most adopted definition for fairness is *max-min fairness* [56]. It is defined over a resource allocation vector $\mathbf{x}$, each x_i in the vector representing a rate allocated to a particular user. $\mathbf{x}$ is max-min fair if it is feasible, and that any x_i cannot be increased without decreasing some other $x_j < x_i$. A simple algorithm to achieve max-min fairness is fair-queuing [57]. In the context of our framework, each queue represents a user. Whenever there is an open slot in the data center, the task corresponding to the user with the least energy saved will be served.

2.2.3.2 Proportional Fairness

Using the same terminology, an allocation $\mathbf{x}$ is *proportional fair* if for any other allocation vector $\mathbf{x}^*$, $\sum(x_i^* - x_i)/x_i < 0$ [58]. Compared to max-min fairness, proportional flow does not stress empathetically on the one with the lowest allocation. As we show later in Section 2.3.4, it provides performance gains over max-min fairness because of such property.

An algorithm that realizes proportional fairness exists for wireless downlink packet scheduling scenarios [59]. One of the essential characteristics of a wireless channel is the constant variation in transmission rate μ over different links. Thus according to the algorithm, the link with the largest μ/r is always scheduled next for transmission, where r stands for throughput. It is proved that when all queues are infinitely backlogged, the algorithm can enforce proportional fairness [60]. In the context of our framework, the policy is defined as follows. The data center always schedules

the task from the user with the largest ratio of benefit over energy saved

$$\frac{b}{\Delta E_0 + \sum \Delta E}$$

where ΔE_0 is a term with an arbitrarily small value that will simplify the case where $\sum \Delta E = 0$. The same form is adopted in [60].

2.2.3.3 Measurement of Fairness

To study the trade-off between performance and fairness, we need to have a quantitative measure of fairness. A widely accepted measure of fairness is the *Index of Fairness* proposed by Jain *et. al* [61]. The fairness index measures the equality of resource allocation x_i among all n users by:

$$\frac{(\sum_{i=1}^{n} x_i)^2}{n \sum_{i=1}^{n} x_i^2}$$

If the resource allocation is absolutely equal among all users, the fairness index is 1; otherwise the larger the disparity, the closer the index is to 0.

Notice that this fairness index is not in total alignment with the fairness definitions. It has been proved that proportional fairness equivalent to the maximization of the utility function $\sum \log(x_i)$. Consider a case where two users each has 2 resource requests of $\{50, 47\}$ and $\{3, 1\}$ respectively. a total of 100 unit resource is available, or if the system can satisfy 3 requests, then the max-min fairness (and also the solution with the highest fairness index) will schedule $\{50\}$ for the first use and $\{3, 1\}$ for the second user. However, the answer from proportional fairness will be $\{50, 47\}$ for the first user and $\{3\}$ for the second user. In more general cases, though, the concavity of a log function makes sure that equal share will have a higher value for the utility function. Thus we still use this fairness index for quantitative fairness measure of our framework.

2.2.3.4 Fairness in the Framework

In order to incorporate fairness into the framework, we need to trade-off performance. We assume that the data center can use fair scheduling algorithms to ensure fairness if all the tasks are offloaded. The key performance gain in our framework, however, is achieved by using the localized

classifier to avoid offloading inefficient tasks. Thus, we perform the trade-off in the following manner: there's a probability $\Pr_{direct}$ for tasks to be offloaded directly without passing the classifier. A higher probability induces more fairness, while a lower value induces more energy savings. We discuss the trade-off in Section 2.3.4.

2.2.4 Online Scheduling

We describe the online procedure in this section. At any time t, the mobile users may choose to offload a task T_i based on the local decision $D_i^l(t)$. If the decision is to execute locally, then the task stays at state `local`; otherwise, the task will be offloaded. The data center always keeps a queue of tasks that are waiting to be executed. At any time t, the data center will make a decision for each of the task in the queue $D_i^c(t)$, indicating whether it will execute the task at t. We define two ways selecting tasks: first come first serve (FCFS) and greedy (i.e. benefit b). If a task approaches its deadline, i.e. $t + \tau_i^c = t_i^d$, then the task will be removed from the queue, and executed locally. That task will be in the state `offloaded`, where the energy cost is accounted for both offloading and local execution. The procedure is shown in Algorithm 2.

2.3 Evaulation

In this section, we evaluate our method. We begin by discussion the preliminaries of the results. We then present the performance of our method, before we proceed with discussions.

2.3.1 Preliminaries

In this subsection, we first discuss the experimental setup and then we present some analysis of the dataset that we use for the simulation.

2.3.1.1 Simulation Setup

We assume the mobile clients contain 1.4GHz single-core processors; the data center machines use 4 core 2.8GHz processors, with an additional 3x speed-up, providing an overall 24x processing

Algorithm 2 Online Procedure

procedure USER(t)
  if Selected according to $\Pr_{direct}$ **then** ▷ Fairness
    offload
  else
    $D_i^l(t) \leftarrow$ Classification result
    if $D_i^l(t) == 1$ **then** offload
    else execute locally
    end if
  end if
end procedure
procedure DATACENTER(t)
  for all T_i in the queue **do**
    if $t + \tau^c = t^d$ **then** remove task
    else $D_i^c(t) \leftarrow$ FCFS / greedy / fair scheduler
    end if
  end for
end procedure

speed-up if a task is executed remotely. Tasks are in a power law distribution with a maximum local execution time of 30,000s and a minimum of 300s. We assume task data sizes are log-normal with respect to amount computation, and the parameters are such that the data sizes fall in a range between 500 KB to 150 MB. We set the network bandwidth to be uniformly distributed between 800Kbps and 8Mbps. For power consumption, we use the data in [45], where the power for transmission state, computing state and idle state are 1.3W, 0.9W and 0.3W respectively. Note that the power consumption for a task with maximum local execution time and average data size is 1500 mAh, which is roughly the battery size of today's mobile phones. We generate slack τ^s using a uniform distribution, with an arbitrarily picked a maximum of 50 seconds. In addition, the slack cannot be so long that waiting for costs amortizes the energy savings of offloading, i.e. $P^i \cdot t^s < \Delta E'$. We choose τ^s to be the minimum between the two, and the value is in turn used to

produce deadline t^d. We discuss different choices of slack in more details in Section 2.3.3.

We simulate a system with around 100,000 tasks at a granularity of 10 seconds per time slot over the period of a day. Unless otherwise noted, we use one-day traces and train parameters with 1-hour granularity. The tasks at the end of the traces will "cycle" back to the beginning as we assume daily cyclic behavior. The amount of data center resource is set such that around 70% of the tasks can be scheduled with the naive approach. We discuss the data center capacity in more detail later in this section.

2.3.1.2 Characteristics of Synthetic Workload

Table 2.4 shows some of the statistical properties of the workload. We obtained these results by evaluating a half-day period trace are with a relatively high number of requests. It roughly corresponds to the waking hours of the day.

We analyzed the mobile network traces and obtained the $Hurst$ parameter, or the degree of similarity of the mobile network traces. To obtain an estimation of the Hurst parameter, we applied the R/S method, which is available in the fArma package in R [62]. A series with H in the range (0.5, 1) is considered a self-similar series with long-range dependence [50]. The larger H is the more self-similarity the series exhibits. The analysis yields an H of 0.675. It's clear that the traces we use exhibits strong self-similarity and long-range dependency.

We use *congestion* to denote the amount of resource (number of machines) requested by mobile clients. The congestion does not exhibit high variation as we expected, with a standard deviation of around 15% of the mean. Yet there are extreme cases or spikes where the number of requests almost doubles the mean.

2.3.2 Performance

In this subsection, we present the performance of the offline probabilistic shifting algorithm, the classification model and the online system.

Variable	Value
Hurst parameter	0.675 ± 0.075
Maximum congestion / # of machines	163
Mean of congestion / # of machines	99.5
Std. of congestion / # of machines	14.5

Table 2.4: Statistical properties of the synthetic workload

Algorithm	Energy Saved / %	Ratio	Running Time/s
Preempt. Upperbnd.	0.768	4.30x	3452
Non-preempt. Upperbnd. [54]	0.764	4.23x	46347
Probabilistic	*0.761*	*4.18x*	*153*

Table 2.5: Offline performance compared with upper bounds

2.3.2.1 Offline Algorithm Performance

We compare the offline algorithm against the two upper bounds, the non-preemptive upper bound proposed in [54] and the preemptive version. The results are shown in Table 2.5. The total energy is defined by the energy consumption when all tasks are run locally, and the ratio is defined by the total energy divided by actual energy consumption. The "All" method schedules all the tasks in the data center assuming no resource constraints. We observe that the offline algorithm, denoted by "Probabilistic", performs reasonably close to the upper bounds while gaining a 300x speed-up in running time compared to the non-preemptive upper bound.

2.3.2.2 Classification

We implement the classifier using Scikit-Learn package in Python [63]. We obtain precision and recall of logistic regression by 10-fold cross-validation. Table 2.6 shows the precision and recall of the classification of tasks in a 1 hour period.

To further characterize the results of the classifier, we study the distribution of the output of logistic regression, which can be interpreted as the probability that a particular task will be sched-

Precision	Recall
0.956 ± 0.050	0.964 ± 0.064

Table 2.6: Evaluation of logistic regression classifier

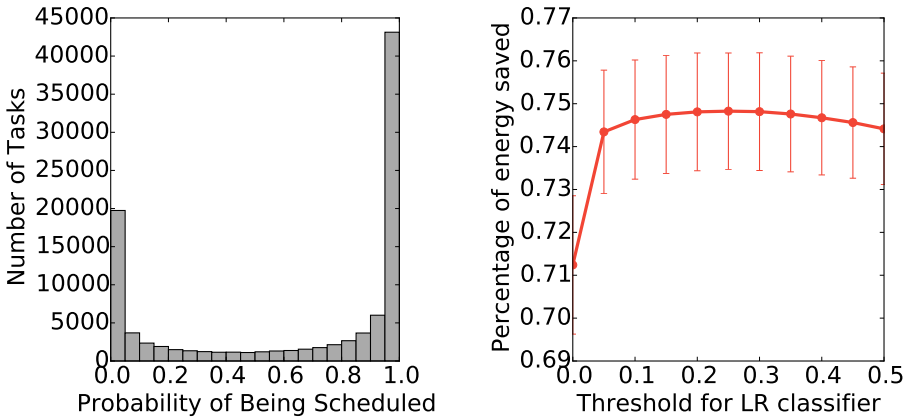
uled. The distribution is shown in Figure 2.7(a). Interestingly, a substantial proportion of tasks that have a probability smaller than 0.1. Tasks that are smaller in size tend to have relatively less benefit because of the fixed term in energy overhead; small tasks also tend to have smaller slack, since they cannot afford to wait too long for energy savings to be amortized. In the power law distribution, a large portion of the tasks are small in size, many of which will have a low probability of being scheduled.

Notice that the ultimate goal of adopting the classifier is not for achieving high classification accuracy. Rather, it is an auxiliary tool to improve the energy efficiency of the system. Thus the observation above prompts us to test the energy savings with different threshold, and the results are shown in Figure 2.7(b). There's a tradeoff in choosing threshold: if the threshold is too high (e.g. equal to 0.5), it maintained the information from learning, yet it sometimes leaves resources unoccupied; if the threshold is too low, too many tasks are offloaded, ending up wasting much energy.

2.3.2.3 Overall Performance

Table 2.7 documents the results from 10 runs of experiments. We observe that our offline algorithm is almost as good as the upper bound, while the online algorithm performs close to the offline algorithm while outperforming the baselines. The results also demonstrate the stability of our algorithm, as we see a relatively narrow confidence interval.

Notice that among the earliest deadline, greedy, and probabilistic algorithm, the number of tasks scheduled decreases yet the energy savings increases. This is due to the fact that tasks that are efficient in terms of saving energy are often larger tasks, in accordance with the observation in [33]. Thus, an increasing number of beneficial tasks are assigned, leading to better efficiency in data center resources utilization. Furthermore, this demonstrates that by the probabilistic profiling



(a) (b)

Figure 2.7: Characteristics of logistic regression for code offloading decision: (a) shows the distribution of probability produced by logistic regression; (b) shows the online performance with different threshold for the logistic regression classifier

Offline Alg.	**Tasks Scheduled /%**	**Energy Saved /%**	**Ratio**
All	100	90.6 ± 0.1	10.63x
Preempt. Upperbnd	N/A	78.0 ± 1.2	4.545x
Deadline	74.9 ± 1.8	73.3 ± 1.3	3.751x
Greedy	64.9 ± 2.1	76.7 ± 1.3	4.295x
Probabilistic	*63.5 ± 2.2*	*77.2 ± 1.2*	*4.395x*
Online Alg.	**Tasks Scheduled /%**	**Energy Saved /%**	**Ratio**
Naive	75.5 ± 1.7	69.5 ± 1.6	3.281x
Greedy	62.7 ± 2.2	71.3 ± 1.6	3.488x
Trained	*60.1 ± 2.2*	*75.2 ± 1.3*	*4.039x*

Table 2.7: Overall performance of offline and online algorithms

tasks, we are able to more optimally utilize the resources in the data center, and achieve even better results than the Greedy approach.

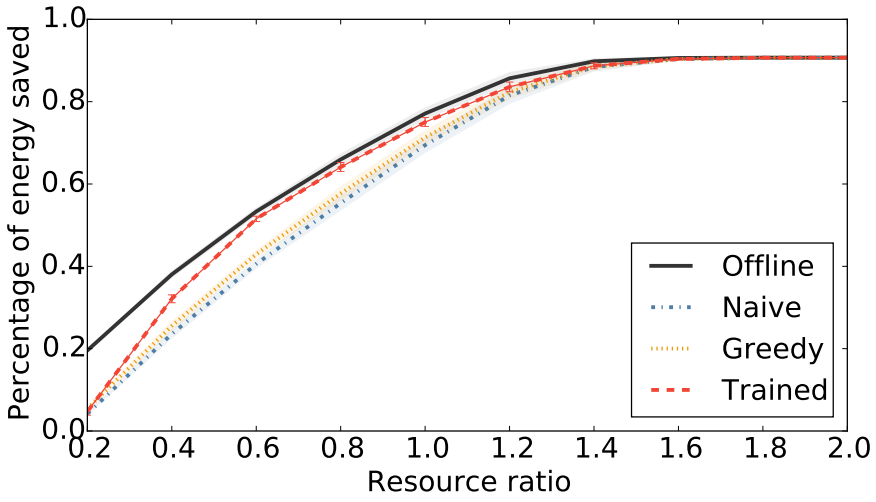


Figure 2.8: Performance with different data center resource ratios

2.3.3 Discussion on Performance

In this subsection, we discuss the trade-offs of key parameters in the system.

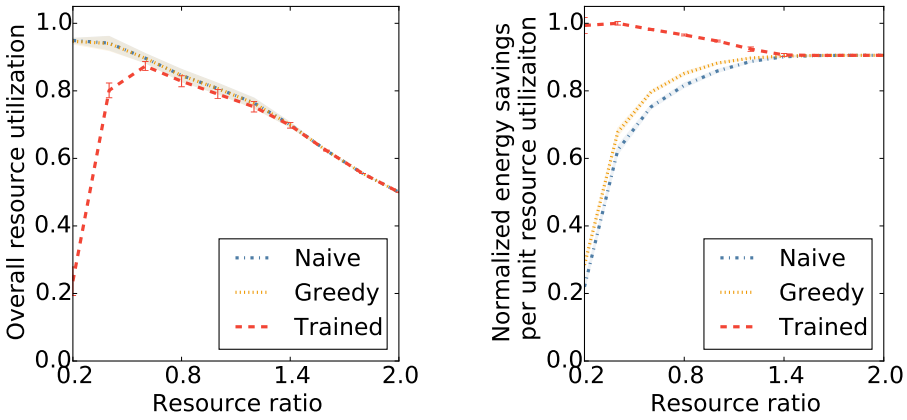
2.3.3.1 Data Center Resources

Given the near optimal solution to the optimization problem, we should now examine the resource provisioning of the data center. To characterize the the amount of resources we define $resource\ ratio$, a term relative to the resources requested.

$$resource\ ratio = \frac{datacenter\ resources}{total\ resource\ demand}$$

The amount of resource for the data center is obtained by multiplying resource limit C by the total period of time; similarly, the resource requests are calculated by aggregating the cloud execution period τ_i^c of all the tasks. We ran experiments for different values of the $resource\ ratio$ and obtain Figure 2.8. In our simulation setup, a resource ratio of 1 corresponds to 73 machines.

Notice that even when the resource ratio is 1, not all tasks can be scheduled. This is due to the fact that most of the requests arrive during the waking hours of people, causing congestion in the data center. For the rest of the time, the resource can be underutilized. Further, we observe that



(a) (b)

Figure 2.9: Details on the performance with different data center resource ratios: (a) shows the utilization of data center (b) shows the efficiency of the data center, defined by energy savings per machine

a ratio of 1.6 (117 machines) would be sufficient to satisfy the resource requests, a number way below the peak number of requests. The slacks of tasks naturally provide a buffer where the spike can be smoothed out.

Figure 2.9 shows the utilization and efficiency (defined by energy savings per machine) of the system. It's clear that when the data center is extremely small, the classifier tends to be too selective, resulting in underutilization high efficiency. The curve converges when the resource ratio increases.

2.3.3.2 Granularity of Training

One of the essential question regarding our system is the granularity of training, or how many sets of parameters we need in a day in order to guarantee good performance. One parameter set per day or one set per hour can differ by a large margin in terms of performance. 2.10(a) answers that question. Notice that in this case, we use the same training set (and thus the parameters) for all the experiments, i.e. the same logistic regression model is used for all experiments (thus the offline results does not have a confidence interval). As shown in the figure, if we have a small number of sets, the model is too coarse-grained and the performance is close to greedy; if we have too many

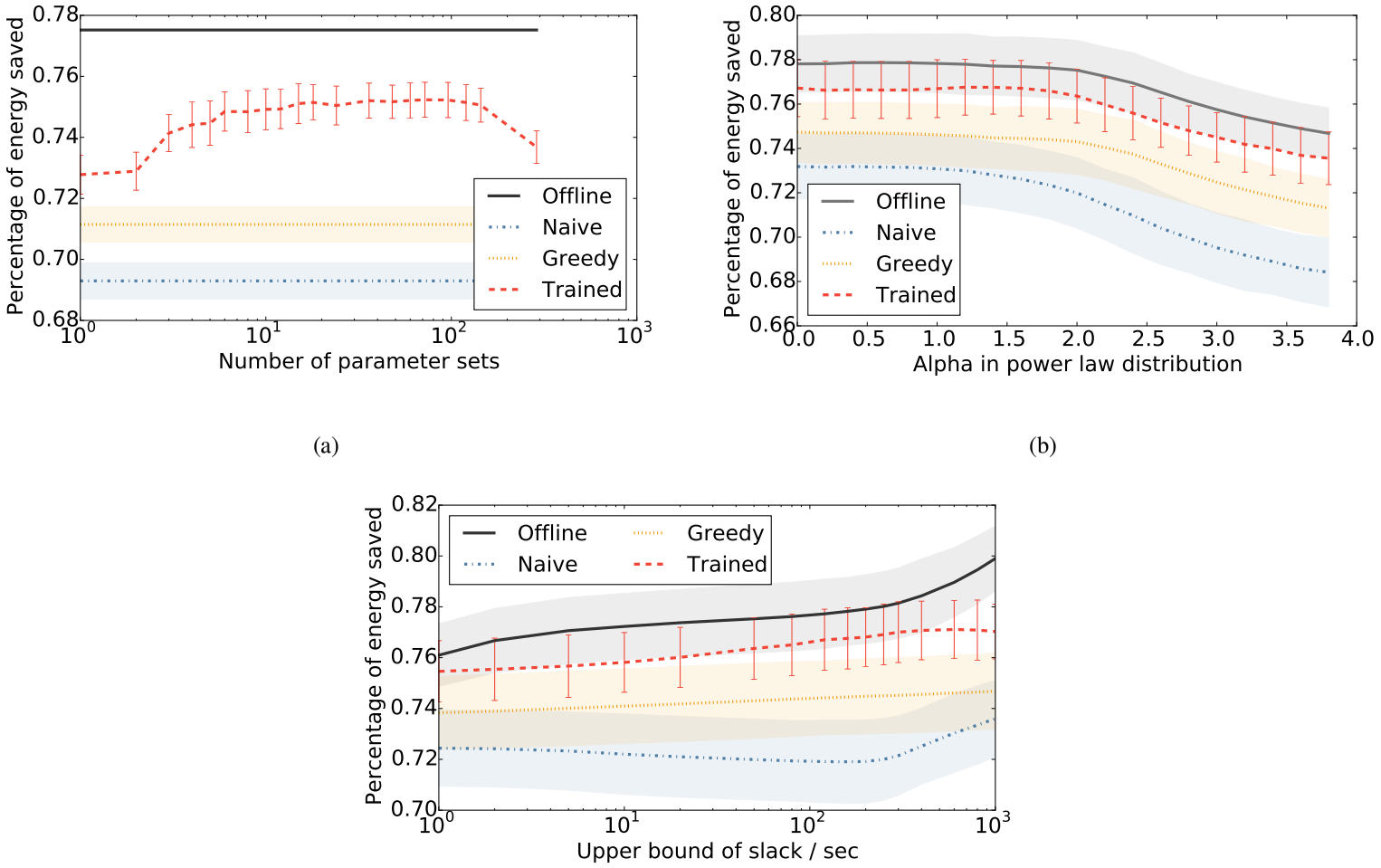


Figure 2.10: Performance with various code offloading system parameters: (a) shows performance with different granularity of training; (b) shows performance with different α in power law distribution; (c) shows performance when tasks are have different slacks. *The shaded areas represent the confidence intervals of baseline algorithms, while the error bars represent that of the targets of study.

training sets, the system will over-fit and thus performance begins to drop again. We see a great increase in from 2(12 hour period) to 3 sets (8 hour period), and it starts to become steady around using 8 sets (3 hour period).

2.3.3.3 α in Power Law Distribution

The power law distribution describes the distribution with probability density function $p(x) = Cx^{-\alpha}$. Given the minimum and maximum value, the α determines the shape of the curve, and thus the frequency of different sized tasks: the bigger the α, the rarer the large values.

In the Google cluster traces, the α is close to 1. It is unclear to us, however, what the value would be for mobile applications. Thus, in Figure 2.10(b) we demonstrate results from experiments with various values of α. The relative performance between various methods is consistent with different distributions. Notice that the amount of energy saved drops as α increases. This is due to the fact that larger α produces a larger portion of small-sized tasks that do not save energy as efficiently.

2.3.3.4 Slack

The deadlines of tasks can be another parameter that is hard to characterize. The slack that a task may have can be from the requirement of the program, or from the service level agreement. Figure 2.10(c) shows the results using different values for maximum slack. The offline system performs well when more tasks have larger slack, because of an increased room for tasks to be "shifted". Our trained online algorithm can thereby benefit from it. The FCFS method can also gain significantly from the extension of the deadline, as it allows beneficial tasks to wait and then be scheduled.

2.3.4 Fairness

In this subsection, we present results regarding the fairness of the system. We aim to answer the following two questions:

1. Which fairness scheduling algorithm should we use?

2. How much can we improve fairness in our proposed performance oriented framework?

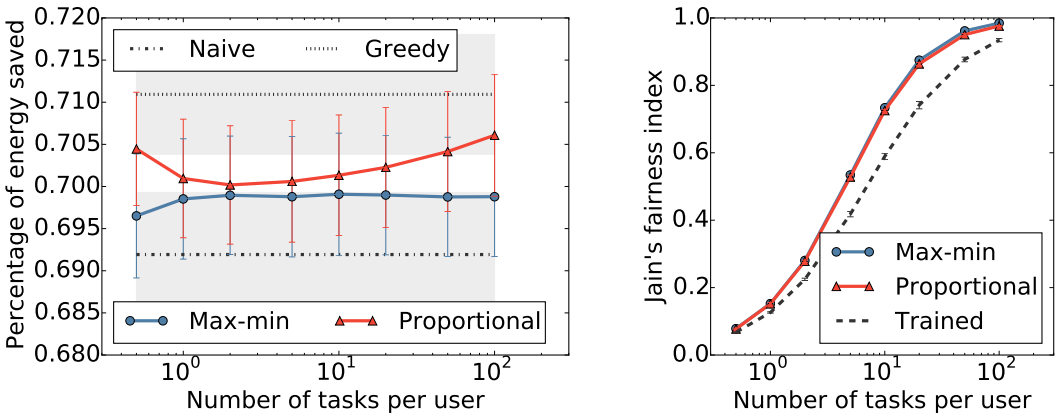
The rest of the subsection will answer the two questions respectively.

2.3.4.1 Comparing Fairness Algorithms

We first study the two fairness definition and the corresponding algorithms. In this scenario, we do not adopt any classification(i.e. all tasks are offloaded). As we showed earlier, Jain's fairness measures the equality of resource allocation. The power law distribution of demands clearly cannot give rise to equal resource shares. In order to measure the fairness quantitatively using the index, we randomly assign tasks to users such that they have a similar amount of demand for the system. We measure the energy savings as well as Jain's fairness index when users have a different number of tasks, and the results are shown in Figure 2.11. Recall that a fairness index of 1 represents perfect equal allocation. The closer to 0, the more discrepancy exists in the allocation of resources.

As shown in Figure 2.11(a), the proportional fair scheduling algorithm achieves better system performance than the progressive filling. This is expected, as the proportional fair algorithm takes a more opportunistic approach in task scheduling. It considers the efficiency of the task when a user offloads it in addition to how many shares of resources the user already. However, as shown in Figure 2.11(b), such opportunistic task scheduling only affects fairness by a small margin. When the number of tasks per user is small, user demands vary to a large extent due to the large discrepancy among the energy savings of tasks. The fairness index is thus very low. When the number grows, as we can observe, the fairness index rises and approximates 1 when each user has an average of 100 tasks. In the process, the fairness index of proportional fair scheduling algorithm tightly follows that of the progressive filling algorithm. On the other hand, the fairness index of original method stays relatively low. Such difference can result from the difference in the definition of fairness, as proportional fairness maximizes $\sum \log(x_i)$, while the max-min fairness stresses on absolute equal allocation. Still, even from the absolute view of fairness, the proportional fair scheduling algorithm performs very well.

An interesting characteristic we observe is the convex shape of the curve. When the number of tasks per user is smaller, users have less "history" to compare with each other. Thus, it's closer



(a) (b)

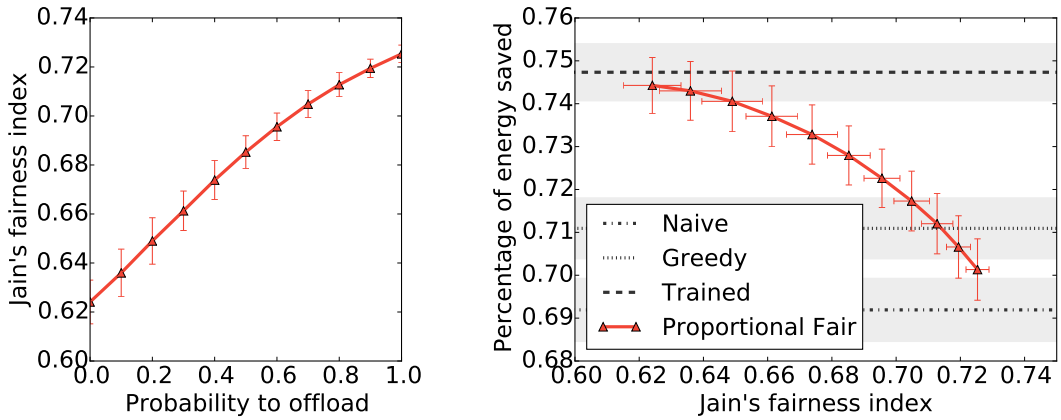
Figure 2.11: Comparison between the algorithms for max-min fairness and proportional fairness with randomly assigned tasks: (a) shows energy savings of two algorithms with different number of tasks per user; (b) shows the corresponding measurement of Jain's fairness index

to the "Greedy" method we used as the baseline. Notice that the two methods are equal if each user is mapped with precisely one task. As the number of tasks grows, the energy penalty of fairness becomes more visible. Yet when the number of tasks keeps growing, the performance-wise efficiency of the proportional fair scheduling algorithm manifests itself as the energy saving gap between the two algorithms enlarges. It's because the energy savings per user more or less sum up to similar values, and thus tasks can be scheduled based more on benefit.

2.3.4.2 Fairness in the Framework

To evaluate how much fairness can be improved in our framework, we also use Jain's fairness index. Recall that we assign a probability for users to offload tasks without being classified. Thus tasks are still randomly assigned to users rather than in a power-law-distributed fashion. We arbitrarily set an average of 10 tasks per user. The results are shown in Figure 2.12.

Figure 2.12(a) shows that the fairness index grows almost linearly with the probability $\Pr_{direct}$. Figure 2.12(b) demonstrates the trade-off between the fairness index and energy savings. As expected, an overall negative correlation is shown between absolute fairness and energy saving.

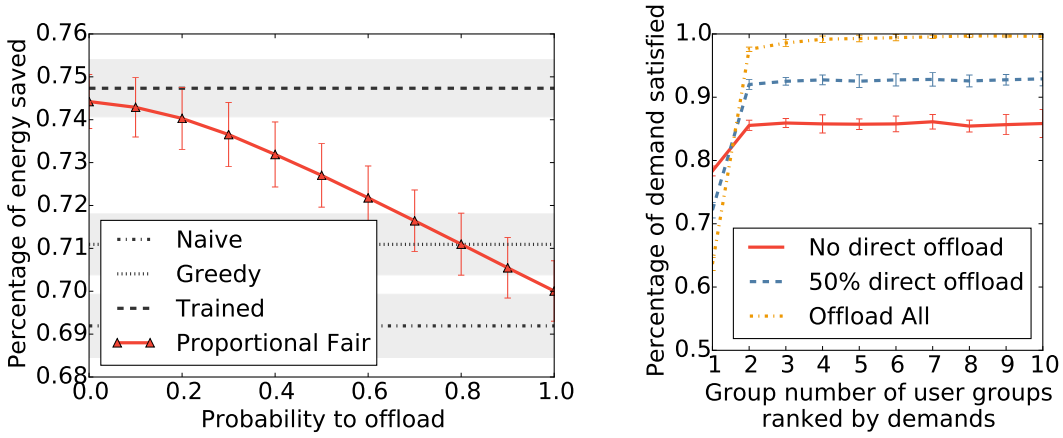


(a) (b)

Figure 2.12: Performance and fairness trade-off with randomly assigned tasks: (a) shows Jain's fairness index obtained with different probability $\Pr_{direct}$ to offload without classification; (b) plots the energy savings against different Jain's fairness indices

Figure 2.13 shows the results for the realistic scenario where the number of tasks per user follows a power law distribution. By analyzing the distribution of mobile network traffic, we pick $\alpha = 0.75$. Since the distribution of tasks is extremely unequal among users by nature, the Jain's fairness index is not a meaningful measure for proportional fairness. Instead, we looked at the percentage of demand satisfied for users with different demands. Nevertheless, we have already shown how the approach in previous results.

As shown in Figure 2.13(a), energy saving is similar to the previous case, where tasks are assigned randomly. When mobile phones strictly follow the classifier, and the vast majority of inefficient tasks are not scheduled, only a small energy penalty is incurred by fairness. On the other extreme, when all the tasks are offloaded, the performance is below the Greedy baseline. Figure 2.13(b) provides an intuitive understanding of how fairness is functioning. The users are ranked by their demands in energy savings and then divided into equally numbered groups marked from 1 (with most demands) to 10(with least demands). In the case of the highest level of proportional fairness achievable, demands from groups with larger group number are almost all satisfied. Because of proportionality, though, they are not 100% satisfied. The first group has the lowest percentage of demand satisfaction rate. It to an extent reflects the power law distribution, where



(a) (b)

Figure 2.13: Performance and fairness trade-off with tasks assigned in power law distribution: (a) plots the energy savings against different probability $\Pr_{direct}$ to offload directly without classification; (b) shows the percentage of demand satisfied of different user groups. Groups are created by equally dividing users ranked from high to low according to their demand

a majority of contribution comes from the ”head” of the distribution. As we relax the fairness constraint for performance, the disparity between user groups narrows.

2.4 Conclusion

In this chapter, we introduce a learning-based localized solution to the resource allocation problem in the offloading system. We propose a novel time-efficient offline scheduling algorithm that achieves near-optimal solution compared to upper bounds. The classifier-based online algorithm shown performance close to offline algorithms under simulation using realistic workload traces. The trade-off between performance and fairness is also studied.

CHAPTER 3

Learning-Based DVFS for Video Decoding

In the last chapter, we discussed saving the energy of edge devices through code offloading. To prolong the lifetime of those battery-operated edge devices, dynamic voltage and frequency scaling (DVFS) is a well-known and widely adopted [64] [65]. In practice, OS and lower level DVFS policies are widely used in today's edge computing systems, with the representative example of Android's CPUFreq governors inherited mostly from Linux [5] [66].

For example, the Nexus 4 Android smartphone supports the following 5 governors: `powersave`, `performance`, `ondemand`, `interactive` and `userspace`, with `ondemand` governor being the default option. Aside from the `userspace` governor, which is merely an interface for the user to change the CPU frequency, the rest are controlled by the kernel. The `powersave` governor simply enforces the use of the lowest frequency. Similarly, the `performance` governor simply configures the CPU to use the highest frequency. The `ondemand` governor samples the CPU usage every tens of milliseconds (in Nexus 4's case, 50 milliseconds) and adjusts the voltage and frequency level according to the usage statistics. The `interactive` is an improved version of `ondemand` governor tuned for interactive workloads. Since the operating system is unaware of the application level workload, the only way to manage energy CPU frequency is to "guess" the future workload based on the sampling results. On the one hand, if the sampling rate is too low, the CPU might not adjust to change in workload fast enough. On the other hand, if we have too high a sampling rate, it will incur a huge overhead considering the *transition latency* in CPU frequency switching. Thus, the sampling rate mentioned above must have been carefully tuned by the engineers to maximize the effectiveness of the DVFS policy. As a result, with those speculative, but frequent decisions, our operating system is able to correct itself on the brink of mistakes, providing us with just about the reasonable level of quality of service.

In recent years, as smartphones become more prevalent, new problems are exposed and more researchers start to look beyond the operating system to higher levels. From the revelation of energy bugs found on the application level [67] to the proposal of application sensitivity to in order to save energy while preserving user experience [68], application and even user-level semantics provide an increasingly important role in improving system energy efficiency. It is also our firm belief that by leveraging the information available at higher levels, the energy management decisions will be much more targeted and precise. For many applications, making DVFS decisions at the application level promises to yield a significant improvement in energy consumption.

However, there are practical constraints for managing DVFS at the application level. The `userspace` governor allows setting CPU frequency through an interface in the file system often referred to as the in-memory sysfs [69] [70]. Yet that interface is not as efficient as we expect it to be. Researchers have shown that it takes an average of 1.7 milliseconds for the voltage change to take place on certain embedded platforms due to various operating system level overhead [71]. It means that if a new policy is to be designed to take advantage of application-level semantics, frequency adjustments can only happen with an interval much larger than tens of milliseconds. After all, if the `ondemand` governor already samples at intervals of 50 milliseconds, we should be able to sample at even lower rates and achieve reasonable energy savings given application level semantics,

In this chapter, we target the DVFS policy for the application of video decoding considering the new constraint. Video decoding is a common but computationally challenging task for IoT systems. We raise two key questions: can we accurately predict the most efficient voltage and frequency using application-level semantics? Can DVFS with a lower rate provide desirable energy savings? In other words, we first want to show that it is possible to predict the choice of CPU frequency with application-level information. Further, we aim at demonstrating that, frequency adjustment can happen at intervals of one or several seconds while still maintaining a high level of energy saving.

In our method, we utilize a decoding buffer already exist in most of the decoders, so that whole video segments are stored first in the buffer before being displayed. The use of buffer smooths out the variation in frame-level decoding computation so that the CPU frequency does not have

to reflect the change from decoding I-frames to B-frames or P-frame. However, the buffer alone is not sufficient to allow our coarse-grained DVFS to meet real-time system requirements. The key enabler is an accurate prediction model for future CPU frequencies. The intuition behind our prediction model is that both the estimation of the next video segment and the progress of decoding the current segment should be considered when we determine the future voltage and frequency. If the system is behind the schedule, voltage and frequency should be increased. On the other hand, if the system is ahead of schedule, the system should lower the voltage and frequency. Based on offline system statistics, including the decoding workload and buffer status, we build a prediction model using machine learning. With the model, we are able to achieve near-optimal energy savings online.

The novelties of our methods and analysis are as follows. First, to the best of our knowledge, we are the first to propose a machine learning-based prediction model using both workload estimation and decoding progress as features. Second, different from previous works where DVFS for video decoding usually operates on the frame or group-of-pictures (GOP) level [72] [73] [74], our proposed method considers frequency adjust rate constraints and switches frequency for every video segment of 1 second or longer. We argue that by taking advantage of buffers and our prediction model, even operating at coarse time granularity, DVFS could still produce desired energy savings. Further, we show that there is a law of diminishing return with respect to energy savings when we use finer and finer time granularity of DVFS. At one point the energy savings stops increasing as we use higher adjustment frequencies. Not only can our per-segment DVFS method be decoupled from the decoding software, enabling effortless implementation, but it can also tolerate the practical constraints imposed by existing operating systems.

The rest of the chapter is organized in the following order. Section 3.2 outlines the model we use and the assumptions we make about the system. Section 3.3 details the method proposed. Section 3.4 evaluates the performance of the system. Section 3.1 briefly surveys past effort on DVFS for multimedia applications.

3.1 Related Work

In this section, we present the related work in DVFS and video decoding energy management.

3.1.1 DVFS on Edge Devices

As edge devices are becoming more and more prevalent, many researchers are identifying and solving practical problems that exist in off-the-shelf products. Carrol *et al.* studied the integration of core offlining and DVFS using the latest smartphone platforms [75]. Kim *et al.* identifies problems in existing DVFS based thermal management schemes, and proposed a new temperature-aware DVFS algorithm [76]. To solve the same problem of thermal management, Das *et al.* proposed a reinforcement learning based algorithm which integrates not only the frequency management but also the core mapping in a real multicore system [77].

3.1.2 Energy Management for Video Decoding

Many previous works have been done to explore the optimal strategy of saving energy for video decoding using DVFS. However, none of the work listed in this subsection considered the practical constraint of the rate of voltage and frequency change in Android. We categorize these efforts into three broad categories: content aware, workload predicting and buffer assisted.

Content Aware. Techniques such as slack reclaiming have been proposed in the past for dynamic energy management for real-time system [78]. Several researchers proposed to use the execution time distribution obtained from offline profiling to guide Dynamic Voltage Scaling such that the system meets the expected requirement [79] [80] [81].

A more straightforward way is to annotate the content. Chung *et al.* proposed a content provider-assisted lower-energy video decoding system, where the authors assumed the server would provide the per frame worst-case execution time (WCET) [82]. Using machine learning, Hamers *et al.* were able to categorize scenes in videos into classes, and the server would then annotate scenarios before delivering to the client [74]. Ma *et al.* built linear models for each complexity unit, and assumed decoder could use parameters embedded in videos to predict computation [73].

In all these cases, a clear difference from our proposal is that they all require some type of offline knowledge, either the WCET or annotations from servers. In our model, however, we assume the system is completely unaware of the decoding complexity of the video that it is about to decode.

Workload Predicting. Various methods have been proposed to predict decoding workload. One of the most straightforward ideas is to use the time series related techniques. Choi *et al.* are among the first researchers to propose workload prediction for better energy management for video decoding [72]. Specifically, they separated the decoding computation into a frame-dependent part and a frame-independent part, and apply moving or weighted averages to predict the frame-level workload of decoding the frame-independent part. We borrow their idea of workload prediction, but we conduct the prediction on the video segment level due to newer system constraints. A similar approach was adopted by Kumar *et al.* [83].

Buffer Assisted. Im *et al.* proposed to add buffers in order to fully utilize idle CPUs cycles, although they assumed periodic tasks and derived analytic solutions based on WCET [84]. Lu *et al.* developed a strategy where the system inserts buffers between different stages of application execution in order to enable frequency scaling and smooth execution [85]. Maxiaguine *et al.* addressed the problem of DVFS with constrained buffer by taking into account both the offline workload bounds and the execution history at runtime.

3.2 Preliminaries

We consider a system where videos exist in the form of a collection of small segments, as in most cases of internet video streaming. We assume that a single CPU core is responsible for decoding the videos. Note that although GPU is present in most of today's smartphones, CPU still undertakes the video decoding workload in many cases. Such observation is validated in a recent energy consumption analysis of popular apps [86]. In our model, the CPU core is capable of performing DVFS, and the voltage and frequency adjustments happen in every video segment. After video segments are decoded, they are stored in a decoding buffer. The decoding buffer has a fixed size, and once the buffer is full, the extra work done for decoding video segments will be wasted. After video segments are decoded and stored in the buffer, they will be fetched in order and passed to

graphics hardware for displaying. Figure 3.1 shows an overview of our model.

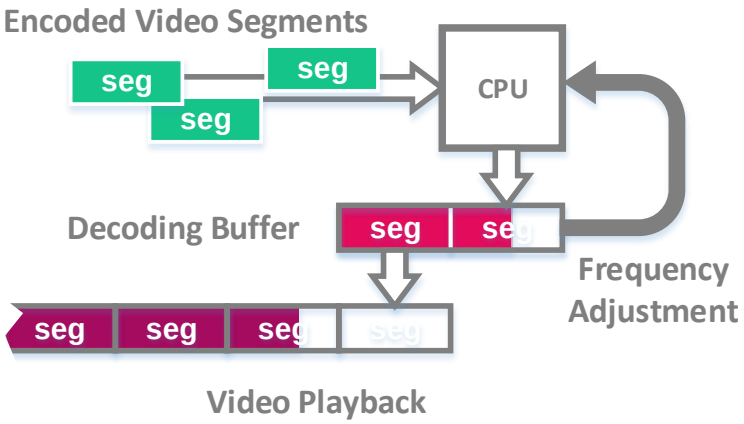


Figure 3.1: Video playback system model.

Consider a video that consists of N video segments, where each segment is of length T seconds. We denote the computational workload (number of CPU cycles) required to decode segment i as C_i. We define a CPU frequency assignment F to be a list of supported CPU frequencies used to decode the corresponding video segments, $F = (f_1, f_2, \ldots, f_N)$, where f_i represents the CPU frequency for decoding the ith video segment. Thus the execution time τ_i of the particular segment i is given by $\tau_i = C_i \cdot f_i$. The resulting energy consumed is denoted as E_i.

In our definition, a *feasible* frequency assignment is defined to be one that meets the following two conditions: a) the decoding buffer does not *underflow*; b) the decoding buffer does not *overflow*. Underflow happens when a video segment finishes playing while the decoding of the next segment has not been completed. Overflow happens when the system decodes too many video segments ahead of time and the video buffer cannot hold the decoded content. The optimal CPU assignment is the feasible assignment that consumes the minimal overall decoding energy $\sum_i E_i$.

3.3 Method

Figure 3.2 shows the workflow of the proposed method. In the offline phase, we obtain the optimal frequency assignment on the training dataset. Using the statistics generated offline, we train our frequency prediction model that is based on the workload estimation and the progress of decoding (quantified by the slack time). When we test our performance online, we use the workload estimation module to project the computation of the next segment. We use that projection together with the online buffer status to predict next CPU frequency. The following subsection details the steps in this workflow.

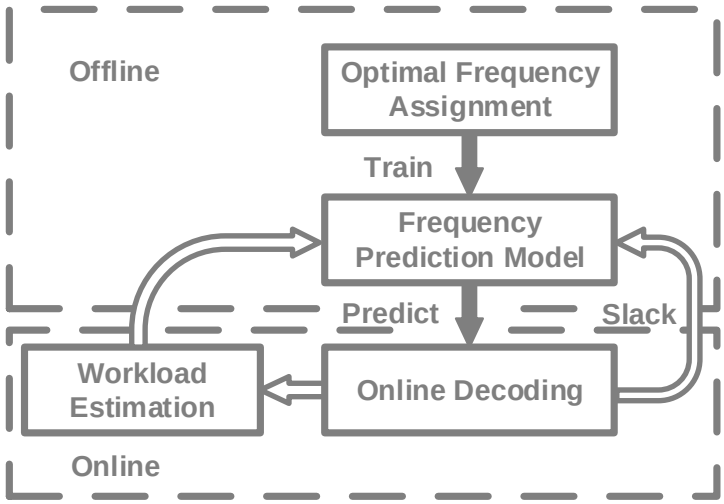


Figure 3.2: System workflow of learning-based DVFS for video decoding.

3.3.1 Workload Estimation

An important assumption we have about our problem is that we can estimate the decoding complexity of the next segment given the computation it takes to decode the current segment. The assumption is based on an observation we make when we experiment with our video dataset. Figure 3.3 shows some exemplary computational workload of video decoding, featuring four exemplary videos that are widely used in studies. The y axis shows the decoding workload in the number of cycles, and the x axis shows the sequence number of the segments. In this experiment, videos are

split into 1-second segments[1]. In our limited number of test videos, the computational workload for decoding neighboring video segments falls in a small range. On average, we observe a 10.2% difference in computation between the current and the next segment. Figure 3.3 shows 4 exemplary traces. The video `akiyo`, which shows a news anchor, has constantly small workload due to limited inter-frame changes. On the other end of the spectrum is the video `mobile calender` (denoted as `mobile`), featuring close-range footage of a moving toy train. The video `pamphlet`, despite having relatively large variations in decoding workload, is still not comparable to videos such as `stefan` and `mobile calendar` in terms of decoding complexity. Part of the reason for this consistency, as we mentioned earlier, is because using segment as a unit has smoothed out the variances on the frame level. With that said, we argue that it is feasible to infer the approximate range of, if not accurately predict, the computational workload of decoding future segments in the same video given the history of decoding its past segments.

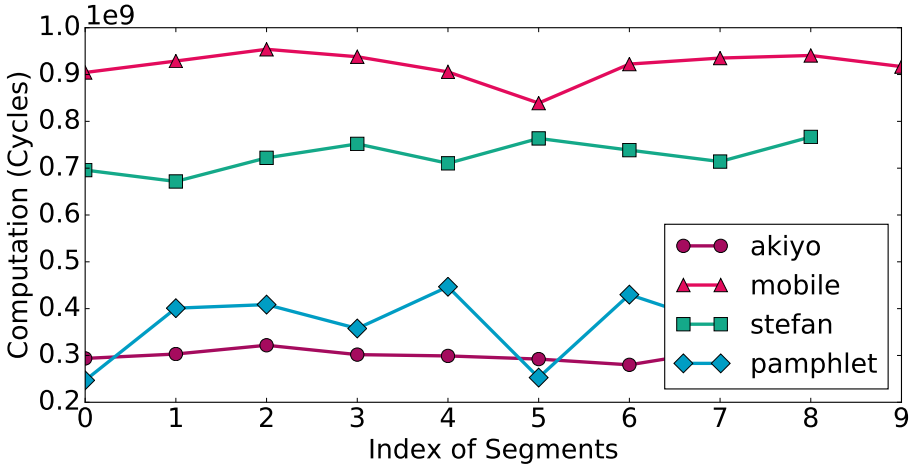


Figure 3.3: Examples of video segment decoding workload.

Apparently, we are not the first to analyze the video decoding computation with regard to the temporal relationship among decoding units. In [72], the authors used the weighted average smoothing algorithm to estimate the computational complexity of decoding the next frame

$$C_{i+1} = \alpha \sum_{n=0}^{i} (1-\alpha)^n C_{i-n}$$

[1]Unless otherwise indicated, the segment length in our experiment is 1 second throughout the chapter

However, we show here that such characteristics persist when we zoom out from frame level to segment level. The method mentioned above is well suited for our purposes and we adopted it as our estimation method, although instead of predicting with frames, we predict with video segments.

One of the concerns regarding the workload estimation method is whether our dataset is representative and that if the method will perform well in the face of another dataset. Admittedly, our dataset can never be large enough to reflect all of the variability in the multimedia contents on today's internet. Yet the uncertainty can always be addressed by using a larger decoding buffer. More importantly, we by no means rely only on the estimation in predicting the CPU voltage and frequency level for decoding the coming video segment. The progress in decoding the current video segment also plays an equally important role.

3.3.2 Offline Frequency Assignment

Recall that our goal for energy management is to find the best CPU frequency for decoding each video segment. The key task in the offline phase is to search for that optimal solution for a given video, under the condition that the decoding workload for each segment is known in advance. To deal with that problem, we propose an algorithm and it is outlined in Algorithm 3. In short, we do an exhaustive search in all the feasible solution space and use dynamic programming to reduce the amount of search needed to be done. For each feasible frequency assignment F_j to decoding video segments up until segment M, where $F_j = (f_{1j}, f_{2j}, \cdots, f_{Mj})$, we inspect the corresponding tuple that contains the energy and time spent so far $(\sum_{i=1}^{M} E_{ij}, \sum_{i=1}^{M} \tau_{ij})$. Among all the tuples, we only retain the Pareto optimal ones. In other words, for any F_j, if there exist another $F_{j'}$ that consumes both less time and less energy, we discard the former. Then for segment $M+1$, we do not need to worry about the solutions before segment M. We could start searching from the Pareto optimal subset of the solution at the time and thus reducing the amount of search needed. In the step of obtaining the Pareto optimal set of solutions, we perform optimization by sorting all the tuples on energy and keep track of the smallest accumulated decoding time so far as we traverse the list of tuples in ascending order of energy. In that way, if we meet any tuple that has a larger accumulated time than the smallest value in the record, we discard that tuple. At the end of the sequence of video

Algorithm 3 Offline Frequency Assignment

for each video segment i **do**
  for each previous assignment $F_j = [f_{1j}, \ldots, f_{(i-1)j}]$ **do**
    for each frequency f **do**
      $F' \leftarrow$ new assignment $[f_{1j}, \ldots, f_{(i-1)j}, f]$
      $[E', \tau'] \leftarrow$ accumulated energy consumption and time $[\sum_{n=1}^{i} E_n, \sum_{n=1}^{i} f_{ij} C_i]$
      if τ' exceeds deadline **then**
        Discard F'
      end if
    end for
  end for
  Sort the list of F' by E' in ascending order
  for each new assignment F'_k **do**
    if $\tau'_k >= \tau'_{min}$ **then**
      Discard F'_k
    else
      $\tau'_{min} \leftarrow \tau'_k$
    end if
  end for
end for
Calculate idle energy for all F' and update E'
$F_{optimal} \leftarrow \{F_k | \arg\min_k E_k\}$

segments, the frequency assignment that consumes the least energy will be the optimal frequency assignment.

3.3.3 Frequency Prediction

Figure 3.4 shows an example of optimal frequency assignment. The solid line shows the computational complexity of each segment, measured by the number of CPU cycles. The shaded bars represent the actual computation with different CPU frequency, the width of which indicates the time τ_i it takes to decode ith the segment. According to our "no underflow" rule, the decoding of a segment should always finish before the time point at which it is to be played, in this case, the vertical grid lines. As marked in the figure, we use *slack*, $\Delta\tau$ to refer to the extra time left before that deadline.

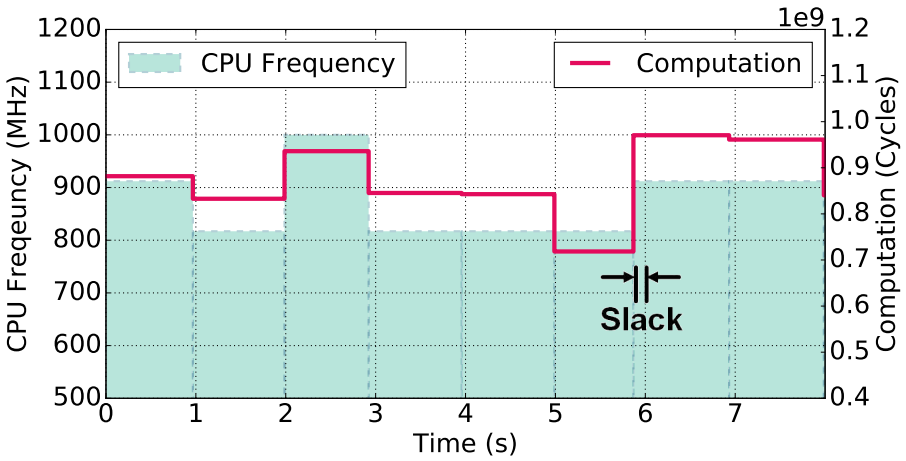


Figure 3.4: Sample optimal CPU frequency assignment and the corresponding decoding process.

Notice that at time $t = 5s$, even though the decoding workload drops by over 10%, the CPU frequency stays the same. The reason is that the slack left in that particular moment is fairly small, and the system would need a higher speed to catch up with the progress. It is precisely the points like this that provide the most precious information about how we should manage the CPU frequencies. If the next segment is predicted to have a high computational workload or the decoding buffer about to be empty, then we should switch to a higher frequency, and vice versa.

To visualize the intuition, we plot the relationship between the decoding computation of the

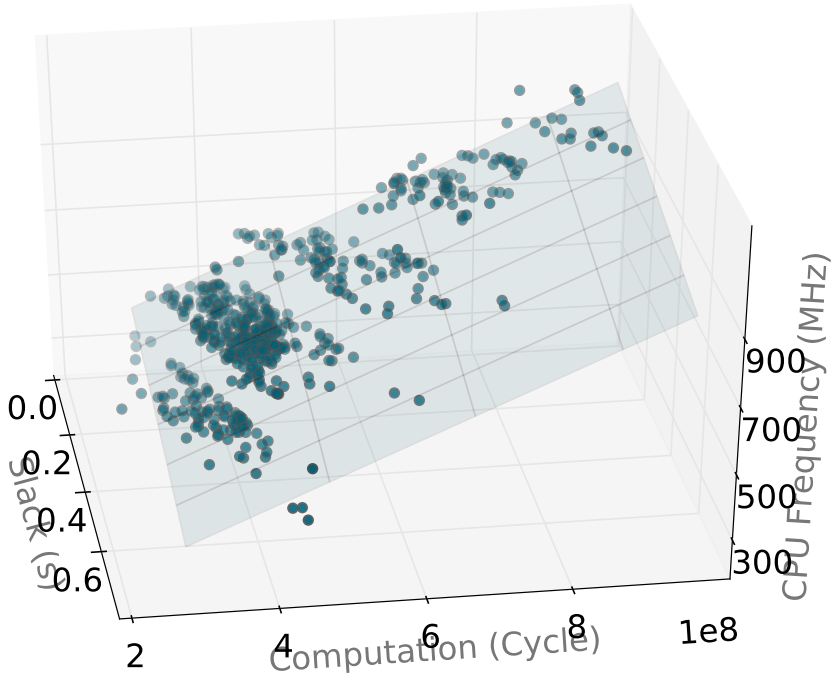


Figure 3.5: Relationship between the decoding progress, the computational workload and the optimal CPU frequency assignment of the next segment. Points with darker colors are closer to the reader and those with lighter colors are further away.

coming segment, slack and the actual frequency assigned to that coming segment in the optimal solution. The plot is shown as Figure 3.5. Clearly shown in the figure, the training points align well in a plane. Those points that are away from the plane are a result of the discrete frequency levels. We imagine that if there were to be more fine-grained frequency adjustment, the points would have fit even better on a plane.

Given the almost linear relationship shown in Figure 3.5, we decide to use linear regression as our target prediction model. With input $\mathbf{X}^T = (1, X_1, X_2, \ldots, X_p)$, linear regression predicts the output Y using the model

$$\hat{Y} = \mathbf{X}^T \beta$$

In our case, we have a simple feature vector $\mathbf{X}$ which consists of two feature: the amount of computation of the segment C_{pred}, and the slack $\Delta\tau$. Thus, our prediction model would be in the

following form:

$$\hat{f_{pred}} = \beta_1 C_{pred} + \beta_2 \Delta\tau + \beta_0$$

The training data for our linear regression model consists of the statistics from the offline optimal solution. We run the solution and, for each video segment, collect computation workload, the slack it has when the decoding first began and the final choice of frequency in the optimal solution.

3.3.4 Online

In the online scenario, our assumptions for the system are the following. In the video playback system, there is a component that keeps monitoring the progress of decoding. It also keeps the records of the decoding workload of past video segments. Before we start the decoding of the next segment, we first use the method described in Section 3.3.1 to predict the computation next segment. Then, based on the model in Section 3.3.3, we obtain a estimated value of the frequency $\hat{f_{pred}}$. Notice that $\hat{f_{pred}}$ will be a continuous variable, but the actual system only allows a few discrete levels of frequencies. Therefore, we choose a conservative approach where we round up the frequency to the nearest available frequency

$$f_{pred} = min\{f > \hat{f_{pred}} | f \in avaiable_freq\}$$

Once the frequency is selected, we adjust the system CPU frequency settings and wait until the decoding of the segment finishes. Then we repeat the same procedure for the following segment.

3.4 Evaluation

In this section, we present the evaluation of our method. We first present the experimental setup. We discuss the choice of video segment lengths in a separate section. Then we present the offline classification performance and the online system performance.

Volt. (mV)	750	825	900	975	1000	1050	1100
Freq. (MHz)	314	456	608	760	817	912	1000

Table 3.1: CPU voltage and frequency table [5]

3.4.1 Experimental Setup

With GEM5 [87], we are able to do cycle-accurate simulations of an ARMv7-A out-of-order processor with 32KB L1 instruction and data cache and 256KB L2 cache. Shown in table 3.1 is the voltage and frequency table of a similar processor (NVIDIA Tegra 2 [2], with ARM Cortex A9 CPU) that we obtained from the Linux kernel driver code [5]. The energy consumption is derived using McPAT [88].

We collect 50 videos that are widely used in the multimedia community to form the dataset [89]. The videos are all encoded in H.264 format with a scale of 640x480 and a frame rate of 30 frames/second.

3.4.2 Optimal Frequency Assignment

	Average	**Best Case**
Optimal	1.00x	1.00x
Static	1.18x	1.37x
Nominal Speed	1.67x	2.23x

Table 3.2: Offline normalized energy consumption

Table 3.2 shows the normalized energy consumption obtained using the optimal frequency assignment, and we compare it against two offline baseline methods. Here "static" refers to the method where we choose the lowest voltage possible that decodes all the segments in time, and keep the same voltage throughout all the segments. The "nominal speed" refers to the method we use the nominal speed (1000 MHz) to decode and stay idle if the buffer is full. As we observed

[2]The GEM5 simulator cannot simulate a CPU that is exactly the same as an off-the-shelf CPU such as Tegra 2, whose frequency table we are using.

earlier, decoding workload of segments tends to stay within a small range. Thus, compared with the ”static” method, we have not achieved an impressive improvement. In the cases where the decoding workload is highly variable among segments, assigning static frequency costs 37% more energy consumption. When we rush to finish decoding, the processor consumes up to 123% more energy. Since we have different assumptions about the system, we are not comparing our results with existing work in this section.

3.4.3 Diminishing Returns

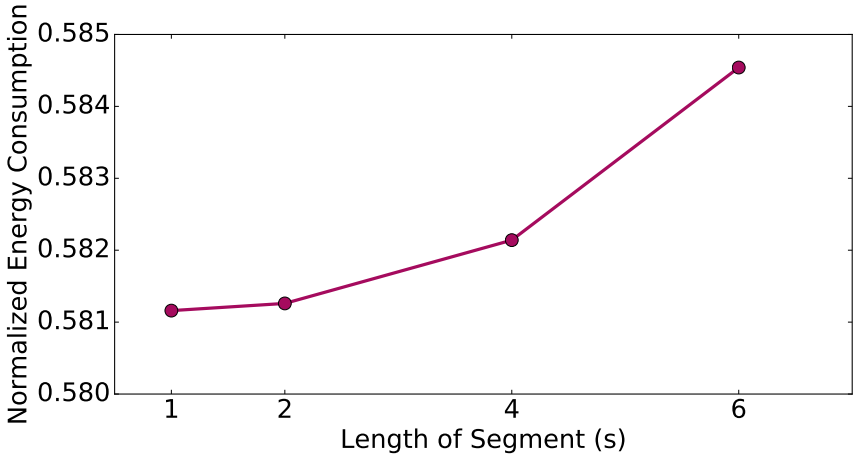


Figure 3.6: Normalized energy consumption of video decoding using optimal frequency assignment with different segment lengths.

Figure 3.6 demonstrates the point of diminishing returns. In the figure, we plot the offline optimal energy consumption when we use different segmentation length, and normalized it against the energy consumption obtained using the nominal speed (constant regardless of segment lengths). As we can see, using a longer segment length, the energy consumption tends to be higher. Yet the difference in energy consumption between different segment length becomes smaller and smaller as the segments get shorter. It clearly shows the law of diminishing returns apply here: there is no need to shrink segment sizes excessively as it does not provide more benefit in terms of energy consumption.

We believe one of the contributing factors behind the findings is the limited number of sup-

ported frequency levels. Consider the extreme case where there is only one segment, and we only use one frequency throughout the whole segment. Surely there will be one optimal frequency for this scenario, but the frequency is unlikely to be among the list of supported frequencies. On the other hand, if shorter segments are used, we can approximate the optimal frequency by hopping among different frequency levels. In this sense, once we achieve the time granularity enough to approximate the frequency, there is no need for shorter segments.

3.4.4 Frequency Assignment Prediction

Mean Absolute Error	59.5 MHz
Freq. Level Prediction Accuracy	0.847

Table 3.3: CPU frequency assignment prediction

We performed a random train-test split of all the videos with a ratio of 7:3, and tested the performance of the linear regression model. Since the point of this experiment is to show the ability to predict, the features of both the training and testing sets are derived from the statistics running the optimal frequency assignment. The prediction accuracy is shown in Table 3.3. We measured the mean absolute error of the predicted frequency (in the continuous space) and the actual frequency assigned (in the discrete space). After we round the frequency to the nearest frequency level, we treat the results as that of a classifier and calculated the prediction accuracy. Notice that the absolute error is only around 5.95% of the highest frequency and around 18.9% of the lowest. The level prediction accuracy is 84.7%.

3.4.5 Online Results

In the online experiment, we still test the performance using the testing video set from the train-test split, but all the predictions are performed online as the decoding is going on. There are two important aspects that we should evaluate. First, how do the energy savings in the online scenario compare with the offline optimal results? Second, will the energy savings affect user experience? In other words, we want to make sure that we don't miss the deadlines for decoding in our method.

	Average	Best Case
Optimal	1.00x	1.00x
Our Approach	1.02x	1.00x
Nominal Speed	1.69x	2.09x

Table 3.4: Normalized online energy consumption

To answer the first question, Table 3.4 shows the normalized energy consumption, compared with a few baselines. The "optimal" refers to using the offline optimal frequency and "nominal speed" refers to using the nominal speed throughout the decoding process. Our approach is able to achieve near-optimal energy savings on average, with the best case being 100% correct prediction and producing the same amount of energy savings as the optimal one. Note that since we use different datasets in training and testing, sometimes the relative performance to nominal speed is even better than the offline results.

Buffer Underflow Rate	Buffer Overflow Rate
0.00523	0

Table 3.5: Decoding deadline miss rate

To answer the second question, we list the buffer underflow and overflow rate in Table 3.5. Recall that a buffer underflow will result in video segment not being ready when needed, thus harming the user experience. A buffer overflow will result in unnecessary CPU idle period, causing low energy efficiency. The experimental results show that our system only misses deadlines of 0.523% of all the video segments, and it never overflows the decoding buffer.

To more intuitively understand how the proposed system performs in the online scenario, we show a few case studies in the next subsection.

3.4.6 Case Studies

Figure 3.7 shows the 4 typical cases of online frequency assignment. Similar to the previous figures, the dotted line represents the decoding workload, the bar with the dashed edge is the optimal frequency assignment and the solid line represents the predicted frequency assignment.

Since the predictions happen online and are sensitive to the previous state, it is highly unlikely that the predicted frequencies are the same as the original optimal frequencies. However, our predictions show overall similar patterns to the original ones: if the optimal solution uses a higher frequency at t, then the predicted solution is very likely to switch to higher frequencies at the vicinity of t (e.g. $t-1$ or $t+1$). Aside from the similarity, these cases each represents a specific scenario. Now we will discuss them separately.

Figure 3.7(a) shows the online decoding process of the video `carphone` and is one of the cases where the prediction is very close to the actual optimal solution. The video features a man talking while driving in a car. Notice that the 6th segment has a noticeably higher requirement for computation. This is due to a background scenario change outside of the car. However since we already built up some decoding buffer earlier (notice at the end of the 3rd second, we are further ahead than the optimal solution), our prediction module chose to stay with the same frequency.

Figure 3.7(b) shows the online solution for video `waterfall`. An interesting fact that we noticed here is that the frequency of the last segment is usually mispredicted. Our predictor is not smart enough to know that it is the last segment and it has the freedom to use up all the slack time available. Thus, the predicted solution usually ends earlier, resulting in a small portion of energy waste. In this case, the online solution consumes about 1.040 times the energy of the optimal solution (the average case being 1.02x).

The video `tempete` has rather high amount of variations in the decoding workload, as shown in Figure 3.7(c). The hard part comes when we have a sudden rise in the amount of computation, which is what happens when we try to decode segment number 6 and 8. In the optimal solution, we see two distinct peaks at the corresponding video segments. In our online solution, what the system reacted in response to the peaks are the two very high CPU frequencies used right after the peak. Recall that our frequency prediction is based on the estimated computation. The estimated computation is high after the given peak in computation. In addition, we have a relatively small buffer by the end of the peak, especially in the second case. Because of those reasons, the system decides to use very high voltage for the decoding of the next segment. Notice that we nearly missed the deadline for the second peak. But fortunately, as we have over half second slack time in our buffer, we were able to avoid missing the deadline.

The last case showed in Figure 3.7(d) involves the decoding of a video where there is a sudden drop in decoding workload. It actually corresponds to the video `pamphlet` shown in Figure 3.3. Fortunately, a drop in computation is much easier to handle than the previous case. The response of the system is that for segment number 6, a mistakenly high frequency is predicted, resulting in huge slack time. However, we ended up not suffering too much from that single misprediction. The overall online system energy consumption is 1.026 times of the optimal solution.

3.5 Conclusion

In this chapter, we discussed an application level DVFS method for video decoding. We prose a machine learning based prediction model for CPU frequency selection using the estimated workload and decoding progress as its features. We show that with application semantics, the system is able to perform DVFS at a low rate that satisfies constraints imposed by existing operating systems while saving an average of 40.1% energy consumption compared with execution at nominal speed.

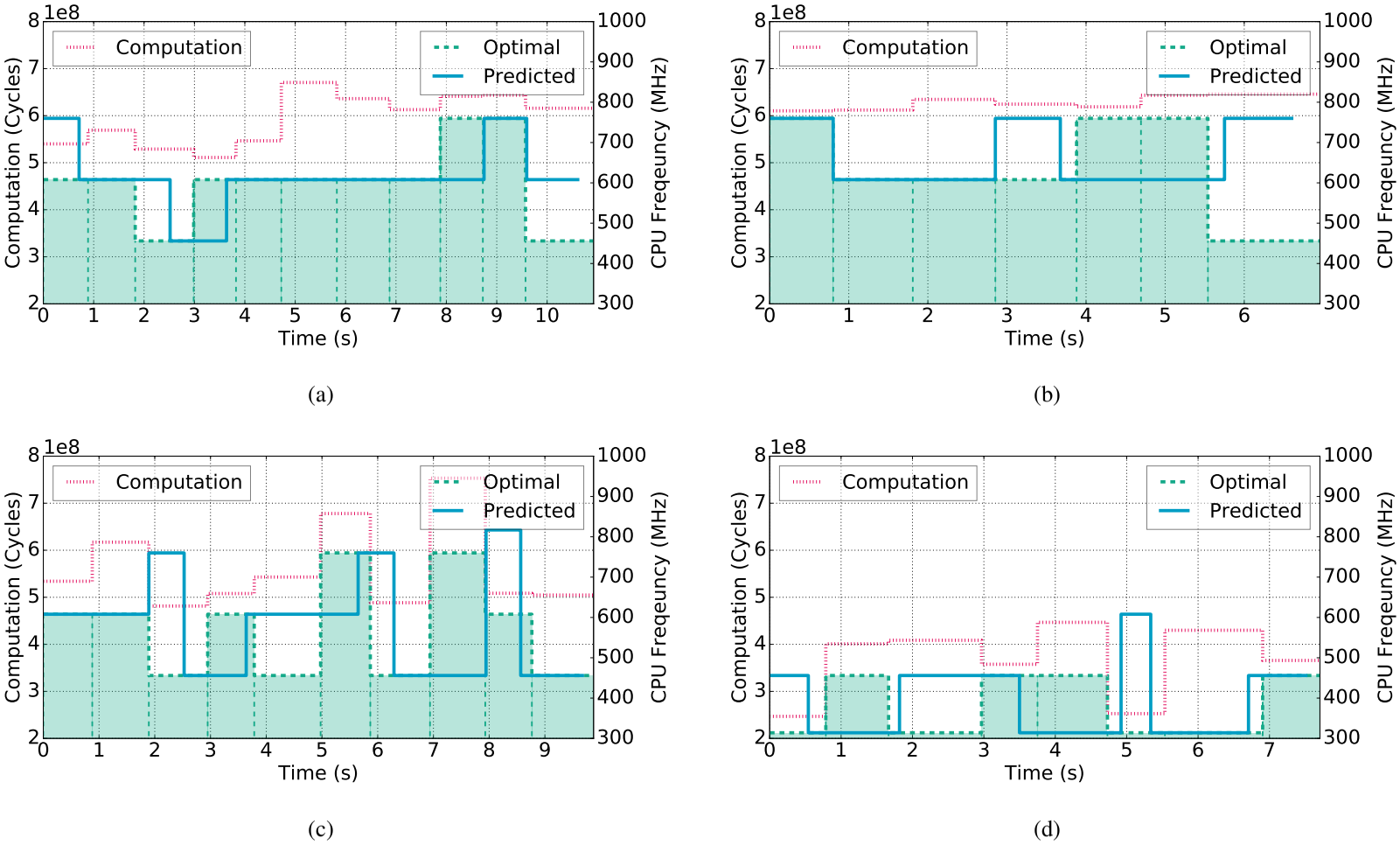


Figure 3.7: Figures showing the offline (optimal) and the online (predicted) CPU frequency assignment to segments in the test video 3.7(a) `carphone`; 3.7(b) `waterfall`; 3.7(c) `tempete`; 3.7(d) `pamphlet`.

CHAPTER 4

Sustainable Operation of Environmentally-Powered WSNs

Sensor networks are an emerging technology for collecting and transmitting information about an environment, and it is one of the most representative systems of IoT and edge computing [9]. One notable application of sensor networks is to measure the various environmental parameters in buildings, such as lighting, temperature, and humidity, in order to enable smart control of the heating, ventilating, and air conditioning (HVAC) systems. A sensor network usually consists of dozens or even hundreds of small, wireless connected sensing devices which until recently used batteries as their energy source. However, the reliance on batteries imposes limitations on where and how these sensor networks can be used. The number of nodes in the network and the placement of these nodes, among other things, are limited by the feasibility of replacing exhausted batteries over the working lifetime of the network.

As such, energy-harvesting nodes have emerged as a way to maintain a sensor network with little or no human intervention. Depending on the application, nodes can harvest solar, wind, piezoelectric, or other external energy, and either store or immediately use the harvested energy. With battery-powered nodes, the challenge is how to minimize energy use to delay the next battery replacement. With energy-harvesting nodes, the challenge becomes how to adapt network function in the face of uncertain or changing energy availability such that the network could sustain itself indefinitely.

In this chapter, we consider a stationary (non-mobile) indoor sensor network that harvests solar power from its environment using small solar panels. Nodes use the harvested energy to sense their environment and wirelessly transmit the collected data, in a multi-hop fashion, to a sink node (e.g. desktop computer). We design a routing strategy for a sensor network which not only is energy-neutral but which maintains sensing quality even in periods where external energy is not

readily available for harvesting (e.g., overnight).

We begin the discussion of our self-sustainable sensor communication strategy by identifying our major goals.

- "Sustainability": the sustainability of the sensor network is guaranteed. In other words, the energy consumed by each sensor node should always be less than the energy harvested.
- "Maximized performance": the performance of the sensor network is maximized under the premise of network sustainability. In our work, we use the sensor sampling frequency to measure system performance.

Our key strategy for achieving both goals is to adopt a dynamic network routing and operation strategy. Two key observations motivate our strategy design. The first is that at different times of day, different nodes harvest the most/least amount of energy. For example, during the daytime, the sensors near windows harvest the most energy while during the night, the sensors that are close to indoor lighting harvest the most energy. The second is that with different routes, the communication cost of the same sensor node varies drastically. In the multi-hop network, a node could have multiple alternative next hops. Thus when routes vary, different amounts of data are routed through the nodes, causing significant variations in energy consumption. Combining these two observations, the core idea of our routing strategy is to adjust the routes in such a way that the sensor nodes that harvest more energy take on more routine tasks, and the sensor nodes that harvest little energy take on fewer routing tasks. Therefore, all the sensor nodes in the network are more "sustainable" while still maintaining high system performance.

Compared to previous work which usually emphasizes a static routing algorithm, our major contribution is that we have split a single day into multiple segments and adaptively select a routing strategy for each segment. The routes are adjusted in such a way that the nodes which harvest more energy in the current segment take on more routine tasks. In this way, we effectively avoid the situation where nodes that harvest little energy become the energy bottleneck of the system. In addition, such a strategy ensures that the routes adapt to available energy and promote energy neutrality.

Another major contribution is our method of deriving the parameters for various system operations. For one, we decouple the derivation of routing and sampling frequency and apply optimization techniques separately, conquering an otherwise complex problem. In addition, we are able to obtain a unified solution despite the uncertainty of energy harvesting among different days. Using our system configuration, the network neither spends too much energy, leaving itself unsustainable during the cloudy days with poor energy harvest; nor is it too conservative, wasting the energy harvested during days with more light availability.

4.1 Related Work

In this section, we present the related work in energy harvesting, self-sustainable sensor networks as well as dynamic routing in sensor networks.

4.1.1 Energy Harvesting

Various energy harvesting technologies have been proposed in the past. Notable examples include solar cell, vibrational, biochemical, and motion-based [90] [91] [92]. More recently, radio frequency energy harvesting has emerged as a promising technique to supply energy to wireless networks with high energy efficiency [93].

Among the various harvesting modalities, solar energy harvesting normally employs the highest power density [94]. It is a promising renewable resource considering its high output efficiency and ability to be utilized in a variety of locations. Photovoltaic production has been doubling every 2 years, increasing by an average of 48% each year since 2002, making it the world's fastest-growing energy technology [95] [96]. Many manufacturers produce low power photovoltaic devices to converting solar energy into direct current electricity, including EnOcean [97], G24i [98], IXYS [99], and Panasonic [90]. In our work, we examine solar cell based harvest technology to power the sensor network.

4.1.2 Self-sustainable Sensor Networks

Self-sustainable sensor networks rely upon energy harvesting technologies for its operations, resulting longer lifetime than their battery-operated counterparts. Varies aspects of sensor network energy harvesting have previously been studied. Paradiso et al. discussed the performance of energy scavenging for wireless electronics with different harvesting technologies [100]. Srivastava et al. proposed the power management design consideration for energy harvesting devices. Compared to the battery-based systems, instead of minimizing the total energy consumption, energy harvesting devices operate in such a way that the energy used is always less than the energy harvested while maximizing the system performance [27]. An energy-aware routing protocol on the energy scavenging ad hoc sensor networks are discussed by Shah et al [101]. Royer et al. have reviewed routing protocols for resource-limited ad hoc wireless sensor networks [102]. More recently, Xu et al. proposed energy saving schemes for medical sensor networks using sensor selection [103] [104]. Eventually, a few implementations of solar energy harvesting sensor nodes are proposed including Prometheus [105], HydroWatch [106], Heliomote [107], and Ambimax [108].

While these self-sustainable sensor networks focus on designing and implementing only a single routing algorithm on the network to maximize system performance, our design emphases on dynamically changing and adopting different routing algorithms according to the amount of energy harvested in different periods of a day. We compare and present the performance of our dynamic routing scheme compared to the static routing scheme on our tested dataset in Section 4.7.1.

4.1.3 Dynamic Routing for Sensor Networks

Many previous works on dynamic routing algorithms target sensor networks that are spatially dynamic (i.e. have moving nodes) rather than temporally dynamic (i.e. have nodes with changing the level of harvested). Examples of algorithms in this category include Dynamic Source Routing [109], Dynamic Proxy Tree-Based Data Dissemination [110], etc.

Only a few researchers have studied the adaptive routing algorithms for environmentally powered sensor networks. Zhang *et al.* proposed a network utility function based optimization method that jointly optimizes data gathering and data transmission [111] in rechargeable sensor networks.

Comparing to their approach, we used a different method where the system parameters are obtained through offline training and applied online to achieve the objective.

4.2 Overview

Our method consists of three steps as shown in Figure 4.1. The first step is to partition a day into multiple segments assuming that the optimal routing algorithm will be recomputed for each segment. A new segment should be created when the relative energy harvested by each sensor node changes dramatically. Our second step is to find an optimal system configuration which contains two parts, route selection, and sampling frequency selection. For route selection, our goal is to create a customized routing algorithm for each segment based on the amount of energy harvested by each sensor node. It is motivated by the maxflow algorithm and is designed in such a way that the sensor nodes that harvest more energy take more routine tasks. For the sampling frequency selection, the objective is to select an optimal sampling frequency for each segment such that the sensor network achieves good overall performance (high sampling frequency) while maintaining a high probability of sustainability.

We execute our method on the training data and obtain a system configuration (the specific routes and sampling frequency corresponding to each segment). We then apply the system configuration to testing data to validate our method.

4.3 System Model

In this section, we present the network model, the energy model and the dataset we used for the experiments.

4.3.1 Network Model

We consider a multi-hop network model with a single sink node. We assume that the sink node's energy and bandwidth are unlimited. Each sensor node acts both as a sensor and a router for other

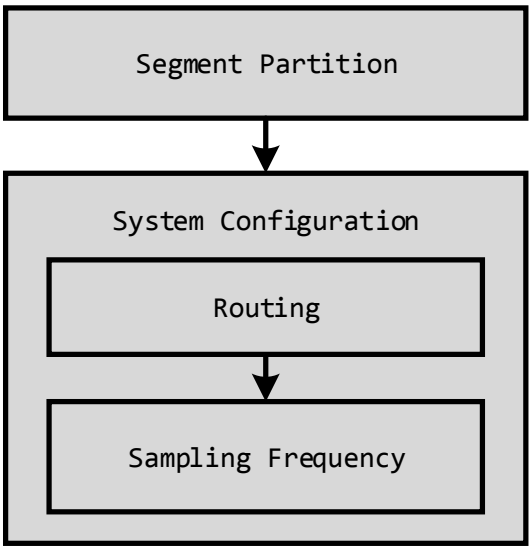


Figure 4.1: System workflow of the sustainable operation of the solar powered sensor network.

sensor nodes when transmitting information. In our model, each sensor node can transmit data up to a threshold distance, since the connectivity between sensor nodes drops dramatically past the threshold. Therefore, each sensor node must choose a path through the routers (sensors) to transmit its data to the sink node.

Figure 4.2 shows an example network model with ten sensors. It traces the paths from the four nodes at the bottom of the figure to the sink node. Note that each sensor not only needs to transmit its own data to the sink node but also possibly may be used as a router by another sensor node. Specifically, the sensors nodes that are close to the sink node will normally take more responsibility to route than the nodes that are far from the sink node.

4.3.2 Energy Model

Our system energy model can be classified into four subcategories.

- *Energy harvesting model*. Each sensor attaches to a solar cell to harvest energy. The harvested power P_h is proportional to the intensity of light l_h received by the sensor's solar cell, $P_h = \alpha * l_h$.

- *Energy leakage model*. The energy storage of each sensor node leaks at a constant rate. Assume the leakage power is P_l.

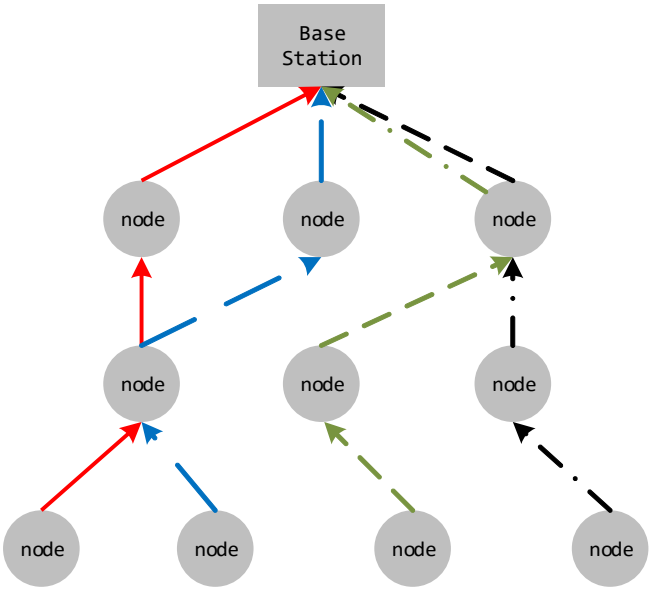


Figure 4.2: An example network model with 10 sensors. Arrows represent paths from nodes to the sink node. Intermediate nodes are responsible for their own data as well as that of other nodes farther away.

- *Sensing energy model*. Each sensor powers up and obtains readings of environmental variables (temperature, light, and humidity) for time period t_s with active power P_s.

- *Communication energy model*. when two sensors communicate, the transmitter and the receiver wake up and talk for time period t_c. The power consumed by transmission is P_{ct} and the power consumed by receiving is P_{cr}.

We assume that every time a sensor takes a reading, it must transmit the collected data to the sink node. We also assume that all the sensors in the network have the same sensing and communication frequency F since all the locations in the space should be monitored equally. Then the performance of the sensor network can be measured with F, as a higher frequency indicates the environment is being monitored more accurately.

However, correspondingly, a higher frequency incurs a larger energy cost to a sensor. In order to build a sustainable network, for each sensor node i, the following equation must hold. In Equation 4.1, we consider time period t_1 to t_2. The left side of the equation indicates the amount of energy

harvested ($\int_{t_1}^{t_2} P_h$) minus the amount of leakage energy ($\int_{t_1}^{t_2} P_l$), thus it represents the upper bound of the amount of energy for sensor i to spend. The right side of the equation has two terms. The first term represents the energy spent for sensing ($\int_{t_1}^{t_2} FP_s t_s$) and the energy spent for communication ($\int_{t_1}^{t_2} FP_{ct} t_c$) for sensor i at frequency F. The second term represents the energy spent by sensor i to be used as the router by other sensors, and we assume it is used by β times. Since as a router, the sensor needs to be used first as a receiver, then as a transmitter, the total energy spent as a router is $\beta \int_{t_1}^{t_2} F(P_{ct} + P_{cr})t_c$.

$$\int_{t_1}^{t_2} P_h - P_l > \int_{t_1}^{t_2} F(P_s t_s + P_{ct} t_c) + \beta \int_{t_1}^{t_2} F(P_{ct} + P_{cr})t_c \tag{4.1}$$

4.3.3 Dataset

The Intel Berkeley Lab's public dataset provides light availability in a real-life environment [1]. The data is obtained by a 54-node wireless sensor network that collected light, humidity, and voltage measurements roughly every 30 seconds over a period of three weeks. The sensors, called motes, were battery-powered and statically arranged in the indoor lab space of the Intel Berkeley Lab at the University of California, Berkeley. There are in total over two million timestamped temperature, light, and humidity readings in the dataset.

The sensor placement is showed in Figure 4.3. In order to measure the whole lab environment precisely, sensors are evenly distributed in all the locations in the Research lab. Some of them are in the corner, such as sensor 42, sensor 16 and sensor 24. Others are in the office, such as sensor 54. Together these sensors provide a wide variety of indoor lighting conditions.

Before applying our algorithms, we need to preprocess the data due to missing epochs and asynchronous measurements. Our approach is to split the days into time slots. Each time slot is two minutes. If sensor i has more than 1 measurement during that time slot, we take the average of those measurements. If sensor i has no measurement on time slot t_1, we go backward and find the first time slot t_2 ($t_2 < t_1$) that has a measurement. Otherwise, we go forward and find the first time slot t_3 ($t_3 > t_1$) that has a measurement, then calculate the weighted average for t_1. The equation is showed in equation 4.2. T_{i,t_1} represent the temperature for sensor i at time slot t1.

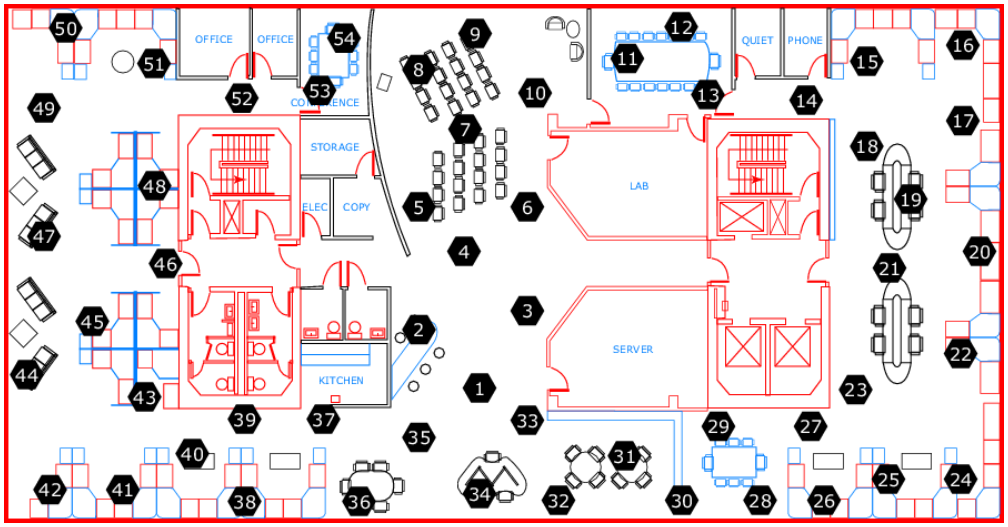


Figure 4.3: The 54 sensors placement of Intel Berkeley Lab [1].

$$T_{i,t_1} = [T_{i,t_2} * (t_3 - t_1) + T_{i,t_3} * (t_1 - t_2)]/(t_3 - t_2) \tag{4.2}$$

4.4 Segment Partition

Our adaptive routing and sensing strategy rely crucially on ensuring that all nodes have sufficient energy to fulfill sensing and routing obligations. Nodes receive varying amounts of light during the day. Furthermore, because nodes are distributed throughout an indoor space, the amount of light that one node receives in relation to another is subject to change throughout the day.

For example, a node located near an east-facing window will receive strong light in the morning and relatively less in the afternoon, while a node located far from any windows will receive a steady amount of light throughout the day. We might want to use a different routing strategy in the afternoon to account for the now much lower energy availability of the window-adjacent node. Thus, every 24-hour period is segmented into blocks to account for the changing relationship of light availability among nodes. Each day is split into n segments; a low number of n is chosen to avoid the cost of switching the routing strategy too frequently.

By considering the light measurement of each node at an instant of time as a point in n-dimensional space, where n is the number of nodes, we can consider the centered Pearson cor-

relation coefficient between any two time points as a measure of similarity between the timepoints. The correlation coefficient, in this case, represents the cosine of the angle between rays from the origin to each of the two points in the n-dimensional space. Because our aim is to change the routing strategy only when the relationships between nodes' access to energy change, we set segment boundaries when the average similarity among points in a segment falls below a particular threshold.

Overall light availability patterns will be similar from one day to the next; our segmentation algorithm, therefore, considers light vectors collected at the same time of day across different days as belonging to the same time point. The start time is set corresponding to the end of the previous segment (the choice of initial start time is discussed below); the average similarity between every pair of time points from the start time to an incremented end time is then calculated until the average similarity falls below a threshold. The end of the segment is then set to the last timepoint before the subthreshold dissimilarity occurs. The algorithmic flow to pick the start and the end point of each segment are shown in Algorithm 4.

Because segment boundaries depend in part on other time points in the segment, the initial segment boundary must be carefully selected. Since there are relatively few time points, we select the initial segment boundary by partitioning based on all possible starting time points. Out of the 24 possible partitions (corresponding to a partition starting at each of the 24 hours of the day), we determine which partition gives the desired number of segments at the lowest threshold. If there are multiple such partitions, we choose the partition that starts at a stable timepoint, i.e., a time point at which most or all of the partitions place a segment boundary.

4.5 Routing Algorithm

Based on the above partition, the next step of our optimization is to select a routing algorithm for each segment. Our algorithm design is motivated by the maxflow problem with edge constraints. We first create a directed graph $G = (V, E)$ where each node $v \in V$ represents a sensor node or a sink node and each edge $e \in E$ represents the wireless link between sensor nodes or between a sensor node and a sink node. Each node in the graph has an attribute *capacity* which describes the

Algorithm 4 Segmentation Algorithm

Input: *luxdata*, hourly n-dimensional vectors of light measurements from n nodes over d days; *thresh*, similarity threshold; t_0, the start point of the 24-hour period

Output: *segments*, (*start*, *end*) pairs corresponding to the start and end of the segments

1: $hours \leftarrow$ ordered sequence of hours in the 24 hours following t_0; $segments \leftarrow [\,]$; $lastbreak \leftarrow 0$
2: **for** each hour h in $hours$ since $lastbreak$ **do**
3: $similarities \leftarrow []$
4: **for** each pair of hours t_1, t_2 in $lastbreak$ **to**h **do**
5: Add Pearson correlation of each vector in *luxdata* at t_1 with each vector in *luxdata* at t_2 to $similarities$
6: **if** min($similarities$) $< threshold$ **then**
7: Add ($lastbreak$, $h-1$) to $segments$
8: $lastbreak \leftarrow h-1$
9: **end if**
10: **end for**
11: **end for**
12: Add ($lastbreak$, t_0) to $segments$ **return** $segments$

amount of energy available for that node. Each edge (from node i to node j) also has the *capacity* attribute which represents the amount of energy spent to transmit data from node i to node j.

To be consistent with the traditional maxflow algorithm model which only has capacity constraints for edges, we convert our graph to $G' = (V', E')$, where each node v in the original graph breaks down to two nodes v_1, v_2, as well as an edge from v_1 to v_2 which carries the original capacity of node v in graph G. After this, the node capacities are converted to edge capacities. The maxflow algorithm also requires a source node and a sink node. We virtually add a source node which has an edge connecting to all the non-sink nodes, where each edge has capacity ∞. All the data flow eventually goes to the sink node.

Figure 4.4 shows a four-node sensor network with maxflow graph G and the graph G' that it is converted to. C_i is the capacity for the ith sensor node in the graph, and the capacities of all solid edges are ∞. Note that the edges between the sensor nodes are mutual in the sense that node A can send data to node B while node B can also send data back to node A as long as the two nodes are within the threshold distance of one another.

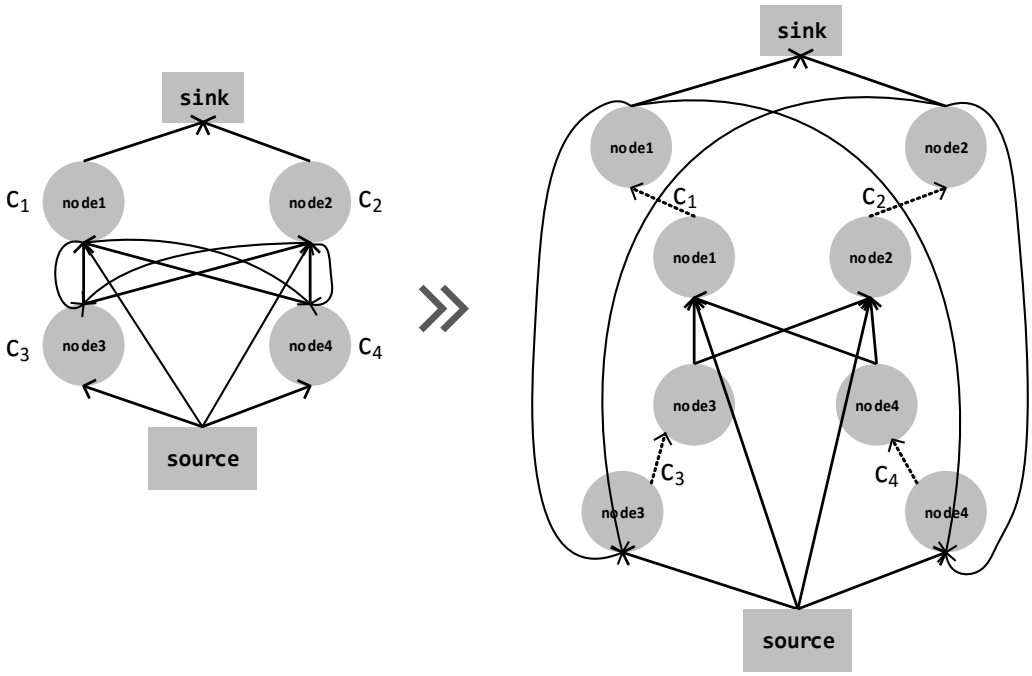


Figure 4.4: An example of maxflow graph G and its converted form G'.

In our model, we assume that all the sensors share the same sensing and communication fre-

quency F as shown in Equation 4.1. The goal of our modified maxflow algorithm is to find a maximized F and the corresponding routing algorithm for each sensor node. Firstly, as different nodes harvest a different amount of energy, the capacity C_i of each sensor node is different from each other. Maxflow assigns more routing traffic to the nodes that harvest more energy. Thus, it uses as much energy as possible. Secondly, we have specifically applied the maxflow algorithm with edge constraints to ensure the energy flow that goes through each sensor node is beyond a certain threshold. Note that the energy flow of node i is proportional to the frequency F.

The core idea behind maxflow algorithm with edge constraints is that each edge e not only has a capacity $c(e)$ as the upper bound of flow $f(e)$, but also a capacity $d(e)$ as the lower bound such that $d(e) \leq f(e) \leq c(e)$. We have modified and applied it to our model. Before explaining the detail of our algorithm, we define the following two notations as shown in Equation 4.3. E_{start} denotes the energy spent on node i to sense and to send messages while E_{router} denotes the energy that is spent on each router nodes that routes the message from node i.

$$\begin{aligned} E_{start}(F) &: F \int_{t_1}^{t_2} (P_s t_s + P_{ct} t_c) \\ E_{router}(F) &: F \int_{t_1}^{t_2} (P_{ct} + P_{cr}) t_c \end{aligned} \tag{4.3}$$

Our routing algorithm is shown in Algorithm 5. In order to calculate the largest possible F, our algorithm has three main steps. The first two steps are to modify the graph, and the last step is to apply the maxflow algorithm with constraints. We first update the original capacity C_i for sensor node i with $C_i - E_{start}(F) + E_{router}(F)$ with the goal of allocating the sensing cost for each sensor node first. We intentionally add the router cost E_{router} to the modified capacity because, in the next step, the maxflow algorithm will deduce $E_{router}(F)$ for all the nodes in the routing path including the node i itself. The added $E_{router}(F)$ compensates for it. Then in the second step, we set the edge constraints. We only put constraints on edges that connect the source to each sensor node. Assume an edge e connects the source with node i. Then the constraint for edge e is set to $E_{router}(F) \leq f(e) \leq \infty$. This guarantees that there exists a flow of at least $E_{router}(F)$ from node i to the sink node. The sensing message from sensor node i can thus be transferred to the sink node. The third step is simply to run the maxflow algorithm with edge constraints. In our case, we have

applied the algorithm from [112] on our modified sensor network graph. Finally, combined with all the three steps described above, we also apply a binary search to find the maximum F that can be supported by the sensor network.

Algorithm 5 Routing Algorithm

Input: Graph $G' = (V', E')$. Each sensor node i maps to nodes n_i, n_i' and edge $e_i(n_i \rightarrow n_i')$ in G'. The original capacity of e_i is c_i. According to Equation 4.1, $c_i = \int_{t_1}^{t_2} P_h^i - P_l$. The source node is s, and the sink node is t.

Output: Maximum frequency F, routing algorithm S.

1: $S = \varnothing$, $F = 0$, $F' = 0$.
2: **for** sensor node n_i in V' **do**
3: $0 \leq f(n_i \rightarrow n_i') \leq c_i - E_{start}(F)$
4: $E_{router}(F) \leq f(s \rightarrow n_i) \leq \infty$
5: $S \leftarrow$ Routing corresponding to maxflow for G' with current constraints
6: **end for**
7: **if** maxflow can be found **then**
8: $F' = binary\_search(F + \Delta F, F_{max})$
9: **else**
10: $F' = binary\_search(F_{min}, F - \Delta F)$
11: **end if**
12: **if** $|F - F'| <= \Delta F$ **then**
13: **Return** F, S.
14: **else**
15: $F = F'$, go back to step 2.
16: **end if**

4.6 System Configuration

Our primary goal is to derive a strategy where the sensor nodes sample as much as possible while maintaining sustainability. That strategy can be characterized by two components: the best set of routes S_i for each segment i in a day; and the sampling frequency f_i and the corresponding communication frequency for each segment. Our routing algorithm produces one set of routes for each node and a corresponding maximum sampling frequency, given the amount of energy stored by each node. But we cannot plug in the expected energy harvested and use the output directly as our strategy. For one, there is variability in energy harvested among different days. For another, the set of routes that maximizes energy consumption for the current segment does not necessarily benefit the subsequent segments. The ideal strategy achieves the best results over all the segments.

To deal with the difficulties mentioned above, we start by generating different sets of routes for each segment in a day. We exhaustively run all combinations of different routes on a week's worth of data and select the best set of routes based on the maximum sampling frequency they allow for. After that, with a fixed set of routes, we use an iterative Monte Carlo method to determine the best sampling frequency. In that way, we are able to select a strategy that yields the highest sampling frequency while being able to self-sustain.

4.6.1 Route Selection

. In the routing algorithm identification phase, we will discover a set of routes $(S_1, S_2, \ldots, S_M)$ for each segment in the day (for M segments per day). As we observe from the dataset, the relative amount of energy harvested among nodes changes over different time segments. Thus, an inconsequential node in the current time segment might become indispensable in the next, thereby requiring us to save its energy for the benefit of the following segment. Therefore, an optimal maxflow solution given the current time segment may not be optimal when we look at the global picture. To deal with the problem, we need to consider different solutions to the maxflow problem, and experimentally determine the best way to route the nodes. In practice, by randomly picking nodes and shrinking their capacity N times, we are able to produce different approximate solutions to the same maxflow problem, denoted by $\mathbb{S}_i = \{S_{i1}, S_{i2}, \ldots, S_{iN}\}$. In these different solutions,

the various "alternative routes" of the maxflow problem are explored.

After running the algorithm for each segment, we will have N different sets of routes for each segment. We then exhaustively combine them into a set of solutions
$\{(S_1, S_2, \ldots, S_M)|S_i \in \mathbb{S}_i\}$ We test all the solutions with the expected energy harvest, and the one that produces the maximum overall sampling frequencies will be selected.

4.6.2 Sampling Frequency Selection

In the sampling frequency selection phase, we decide the sampling frequencies $(f_1, f_2, \ldots, f_M)$ for each segment in the day (for M segments per day). The difficulty in selecting sampling frequency lies in the variability in the energy harvested. The self-sustainable sampling frequency on one day may not be self-sustainable on another day. Yet if we simply set as standard the days with the least harvested energy, we would be too pessimistic as we ignore the energy left over from days where more energy is harvested.

We find this sweet spot by adopting an iterative Monte Carlo method. In the method, we maintain a set of "points" at each iteration t, $\mathbb{P}^t = \{p_1^t, p_2^t, \ldots, p_N^t\}$, where each point refers to one set of possible sampling frequencies,
$p_j^t = (f_{1j}^t, f_{2j}^t, \ldots, f_{Mj}^t)$. We initialize f_{ij}^0 by sampling from a normal distribution $\mathcal{N}(\bar{f}_i, \sigma(f_i))$. [1] We discard points (sampling frequencies) under which the system cannot self-sustain. Then we increase the weight w_j on those points that produce more samples per day before normalizing all the weights. At each subsequent iteration t, we create a new set of points $\mathbb{P}^{t+1}$, where f_{ij}^{t+1} of each point p_j^{t+1} comes from sampling the Gaussian mixture model $\sum_j w_j \mathcal{N}(f_{ij}^t, \sigma(f_i))$. The points are then re-weighted following the same procedure as described before. In our experiment, the best point (the one with the highest overall sampling frequency) converges quickly. We will apply that best set of sampling frequencies as the settings for each of the segments.

[1] $\bar{f}_i$ and $\sigma(f_i)$ come from the previous step, where we select routing algorithms.

Sampling Frequency (Hz)	7am-4pm	4pm-6pm	6pm-7am
Shortest Path	0.184	0.0743	0.0104
Static Routes (1st)	0.369	0.167	0.0113
Static Routes (2nd)	0.285	0.134	0.0113
Static Routes (3rd)	0.204	0.0584	0.0207
Dynamic Routes (ours)	0.369	0.134	0.0207

Table 4.1: Comparison of system sampling frequency with different routing strategies.

4.7 Evaluation

Based on the sensor topology and data collected in [1], combined with the network and energy model described in Section 4.3, we built a simulation to validate our algorithms. We first test the routing algorithm independently to show the effectiveness of the dynamic adjustment. Then we consider the whole system and show the offline optimal results used as the training set. Finally, we test the configuration obtained through training in an online setting and show the sustainability as well as the performance of the system.

4.7.1 Routing Algorithms

Table 4.1 describes the maximum sampling frequency given different routing algorithms. Using the segment partition algorithm, we split a day into three segments, respectively $7am - 4pm$, $4pm - 6pm$, and $6pm - 7am$. We evaluated five routing strategies. The shortest-path method, where we always route the packet along the shortest path from the node to the sink, serves as the naive example. When we use static route methods, we do not dynamically change the routing for different segments. Instead, we compute the optimal routing algorithm on one of the segments ($1st,\ 2nd,\ or\ 3rd$), and apply it to all three segments. Our approach, which we call *dynamic routes*, achieves the highest overall sampling frequencies in all the segments.

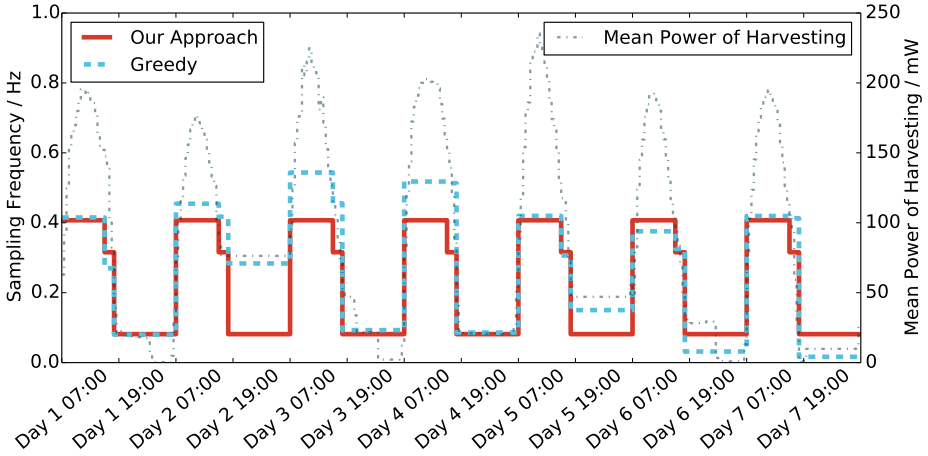


Figure 4.5: The greedy and optimal sampling frequencies derived in training.

4.7.2 Training

As mentioned before, we first train the system using a week's worth of data. Figure 4.5 shows the results. The dotted line represents the mean power of energy harvested from all the nodes. The solid line represents the corresponding sampling frequency we derive for each segment with our approach. The dashed lines represent the sampling frequency achieved by running the max flow algorithm on every segment. We refer to it as the "greedy" approach, as it always finds the locally maximum sampling frequency given the energy harvested in that segment. Apparently, the greedy approach varies from day to day. Noticeably, for the last two days, the sampling rate during the night is far below that of our approach. The reason for the variability is that some of the nodes that are bottlenecks in the communication chain did not receive as much energy as expected. Thus the overall performance is limited by those bottlenecks. In our approach, on the other hand, we execute the same strategy, derived by considering both the good and the bad days.

4.7.3 Testing

Figure 4.6 shows the results testing our approach on a different set of days. The dotted line shows the mean harvested power over all nodes, and the solid lines are sampling frequencies obtained from training. We also apply the same set of sampling frequencies in the greedy training scenario here. The bar below shows the sustainability of the corresponding segments, where the colored

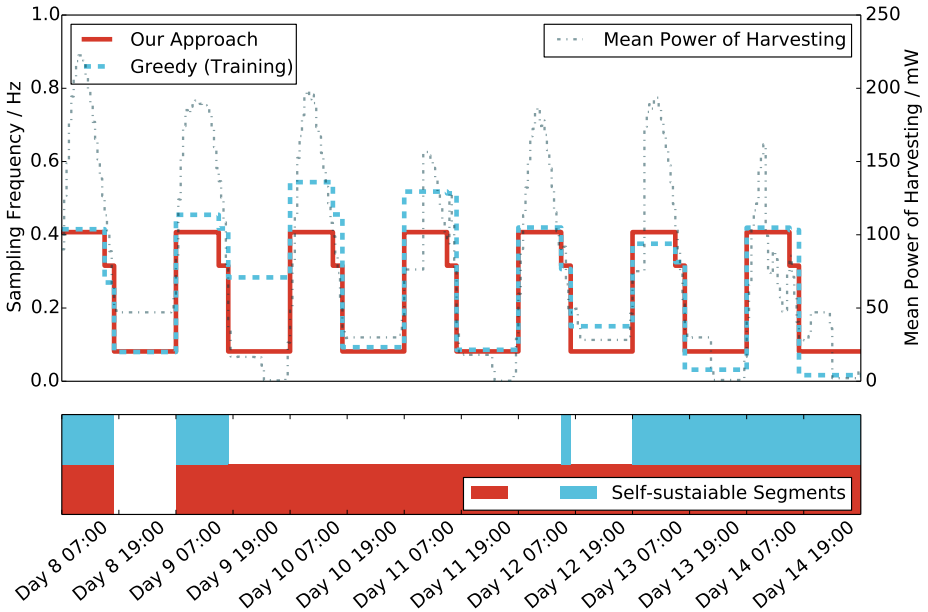


Figure 4.6: Sampling frequencies and corresponding segment sustainability in testing.

segments are able to self-sustain and the non-colored segments are not. Apparently, using the greedy sampling frequencies from training, more than half of the segments weren't self-sustainable as indicated by the upper part of the bar chart (9 out of 21 segments are sustainable); our approach (the lower part of the bar chart) is able to sustain through almost all the segments (20 out 21 segments) [2]. The one segment that is not self-sustainable also suffered from a bottleneck node with unusual behavior. Interestingly, although the energy harvested in the last day fluctuated, both methods are still able to sustain. A possible reason for this is that most of the bottleneck nodes are far from windows while nodes near the windows are most likely to have sufficient energy supply. Thus, although the day may have been cloudy, and the nodes near windows have fluctuating energy availability, the true bottleneck nodes are not affected, therefore delivering expected service.

[2]Once a node fails, we assume the whole network resets at the start of the next time segment

4.8 Conclusion

We proposed a routing and operation strategy for an environmentally powered sensor network. Our approach enables the maximum workload (sensing and communicating) for the sensor network while guaranteeing the network can sustain on harvested solar energy with high probability. Our routing strategy is designed in such a way that the routing algorithm is dynamically adjusted according to the amount of harvested energy by each sensor node. We have specifically addressed our optimization flow with three steps: segment partition, routing algorithm identification, and configuration selection. Our results indicate that our dynamic routing algorithms on different segments enable the largest overall sampling frequency compared to the other routing algorithms while it can sustain 20 out of the 21 segments in our test set.

CHAPTER 5

Adaptive Image Sensor Subsampling for Computer Vision

Starting from this chapter, we shift our focus to machine learning-based applications. We begin with by discussing the energy efficiency of image sensors, which consist of the first step in the computer vision pipeline on edge devices.

High-resolution cameras are prevalent today. Apple's iPhone X is equipped with a 12-megapixel camera [113] and the latest Sony Xperia XZ has a dazzling 23-megapixel camera [114]. While the increasing consumer demand for high-quality photography justifies the need for a high-resolution camera, image sensors with over 10 million pixels are usually an overkill for many computer vision tasks. Take image classification as an example. For the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) [115], a popular 1000-class classification competition, competitors usually resize input images to 224×224. For simpler problems, such as the CIFAR-10 10-class classification problem [116], state-of-the-art classifiers achieve over 95% accuracy with input images of size 32×32. As the amount of computation required for a convolutional neural network (CNN) will roughly quadruple if the input image doubles its height and width, it is likely that the input sizes of classifiers will be kept as small as possible. That being said, a 5500×4200 image produced by a 23-megapixel appears disproportionately big compared to what we need for common computer vision tasks.

The question then becomes how do we obtain images a with relatively low resolution from these over-sized image sensors. A straightforward way is to take the full image and interpolate to create a smooth downsized image. However, energy-wise it is not the best solution. There exists an approximately linear relationship between the energy consumption of the image sensor and the number of pixels it samples [117]. Unlike cloud-based DNN applications [118], applications on mobile devices call for low energy consumption. As a result, it is desired to sample a few pixels

as possible. But how much should we subsample? If we undersample, the aliasing effect causes distortions that might lead to degraded performances of the computer vision algorithms. If we oversample, we risk wasting energy by sampling unnecessary pixels. It is an important question to hardware and system designers, as it mandates the type of features and APIs that are exposed to application developers. It is also an important question to computer vision app developers, as its answer instructs them on how best to utilize the camera for the overall energy efficiency of mobile computer vision systems.

To bridge the gap between image sensor hardware and DNN-based image classification algorithms, we analyze the cause of errors when we subsample and propose clear and practical strategies to improve it. In particular, we show that subsampling, when done right, does bring about tremendous energy saving with negligible loss of accuracy. We do not intend to provide an exact answer to how much we should subsample. Instead, we show that by studying the performance degradation of just a few classifiers with subsampled input images, we can get a reasonable estimate of the performance degradation of image classification in general. To further reduce the energy consumption of the image sensor, we propose a minimal hardware modification to off-the-shelf image sensors. AdaSkip, the proposed enhancement, adaptively subsamples parts of the image based on the complexity. We show that the proposed method can achieve even better energy accuracy trade-offs than plain subsampling.

Our contribution is neither proposing new image sensor architecture nor building more efficient computer vision algorithms. Rather, we take existing and widely-used designs from both sides and optimize from a system's perspective. To the best of our knowledge, we are the first to a) systematically study the relationship between subsampling and performance degradation of DNN-based image classification; b) propose a low-footprint hardware extension to off-the-shelf CMOS image sensors for more efficient subsampling in the context of efficient image classification.

We organize the rest of the chapter as follows. Section 5.1 introduces prior art from the images sensor and deep DNN optimization communities. Section 5.2 briefly describes the operation of CMOS image sensors and our energy model. Section 5.3 analyzes the negative effect subsampling has on DNNs and motivates the problem. Section 5.4 details both of our proposed methods. The experimental setup is described in Section 5.5 and the results in Section 5.6.

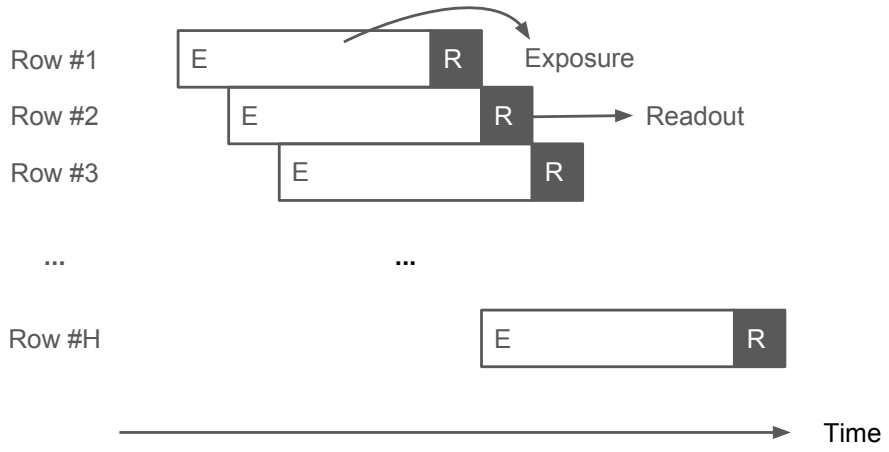


Figure 5.1: CMOS image sensor operation with rolling shutter.

5.1 Related Work

CMOS Image Sensors. It is well known that the analog readout chain consumes a significant amount of energy. Thus there has been a continuous effort in creating *specialized* image sensors that take reduced amount of sampling. Notable approaches include the use of DCT proposed by Kawahito *et al.* [119], predictive coding proposed by Leon-Salas *et al.* [120], Quadtree Decomposition algorithm proposed by Artyomov *et al.* [121], and compressive sensing proposed by Oike *et al.* [122]. Kemeny *et al.* also proposed an image sensor that allows programmable readout of pixels at arbitrary locations and various resolutions [123]. LiKamwa *et al.* studied the energy proportionality of off-the-shelf image sensors and proposed optimization strategies for simple computer vision tasks [117].

In our work, we look at the effect of subsampling on the particular computer vision task of image classification rather than image quality. To reduce sensor energy consumption in vision tasks, one way is to derive a separate image sensor pipeline alongside the normal one [124]. Instead, we base our techniques solely on subsampling by skipping pixels, a feature that is widely available on today's mobile image sensors.

Neural Network Design. DNNs are extremely power hungry. Pruning, quantization, and factorization have already become widely adopted techniques to improve efficiency [125] [126]. Mean-

while, the quadratic relationship between computation and input size can also play a critical role, as evident in the multi-scale Densenet architecture [127]. Other researchers have also proposed to reduce the computation of neural networks by using different image sensor designs. Chen *et al.* have proposed to approximate the first layer of CNNs using image sensors with angle sensitive pixels [128]. LiKamwa *et al.* proposed to move early layers in CNNs to the analog domain to save energy [129].

In comparison to the designs mentioned above, our work targets a broader audience. We base our experiments on generic neural network models. And the main target of our study is how the most widely used image sensor architecture could best be utilized in order to serve the need for neural networks.

5.2 CMOS Image Sensors

In this section, we describe the architecture and operation of the most commonly used mobile CMOS image sensors. We then describe the model based on previous energy characterizations of off-the-shelf units [117].

5.2.1 Architecture and Operation

Today's mobile CMOS image sensors usually consist of pixel arrays, an analog signal chain, and various digital processing and I/O logic. The analog chain contains ADCs, amplifiers, and bias adjusting circuits and it is known to be the major source of energy consumption. A common design choice for image sensors is to use *column-parallel readout*, where one pixel column share one signal chain. In this architecture, the pixels are usually read out row by row.

Rolling electronic shutters are widely used to control the exposure. During the operation of a rolling shutter, only one row or column of pixels read out at a time. The operation fits nicely with the column-parallel readout architecture. When one row finished exposure, the readout logic starts the processing of that row. At the same time, the subsequent row is in the process of exposure, waiting for its turn for pixel readout. The whole process is depicted in Figure 5.1.

There are usually two supported modes of subsampling: *skipping* and *binning*. In skipping mode, the sensor skips the entire pixel readout chain of pixels at certain locations. In binning mode, pixel values are averaged before output. We mainly consider the skipping mode, since it is the mode that enables the most energy savings.

5.2.2 Energy Model

We mainly use LiKamWa *et al.*'s energy characterization on commercial image sensors [117]. They proposed to estimate the energy consumption of image sensors using the following equation:

$$E = P_{idle}t_{idle} + P_{active}t_{active}$$

P_{idle} and t_{idle} represents the power and time when no pixel readout is occurring, i.e. when the first row is going through exposure. Once the pixel readout starts, the system switches to the active mode with higher power consumption. The length of the readout t_{active} is linear to the number of pixels sampled.

5.3 Problems with Subsampling

Given a neural network that requires inputs of size $H \times W$, it is most desirable if we directly subsample the pixel arrays to create a $H \times W$ image. However, images naturally contain a variety of high-frequency components in their original form. Subsampling images can easily give rise to strong *aliasing* effect. Neural networks have very unpredictable behaviors when they encounter input data derived from distributions that are different from what they are trained on, as in the case of adversarial examples [130]. From the neural networks' perspective, the process of subsampling creates input data of a slightly different distribution. It is thus likely that neural networks do not behave correctly in these situations.

To verify our conjecture, we tested direct subsampling on 10-class image classifiers with two different input sizes. Section 5.5 explains the experimental setup in more details. To create an image of the target input size, we directly subsample (by skipping pixels) from a 512×512 image. Then we feed it to 10 different classifiers, with classification accuracies averaging 86.9% and

Subsampling method	Accuracy loss (%)
32×32, Direct subsampling	-48.6 $\pm$ 6.4
32×32, Smoothed (radius=0.3)	-36.6 $\pm$ 5.9
32×32, Smoothed (radius=0.65)	-9.3 $\pm$ 1.3
32×32, Smoothed (radius=0.9)	-50.2 $\pm$ 5.8
64×64, Direct subsampling	-31.7 $\pm$ 8.3
64×64, Smoothed (radius=0.55)	-6.1 $\pm$ 1.3

Table 5.1: Effects of direct subsampling on 10-class classifiers with different input sizes

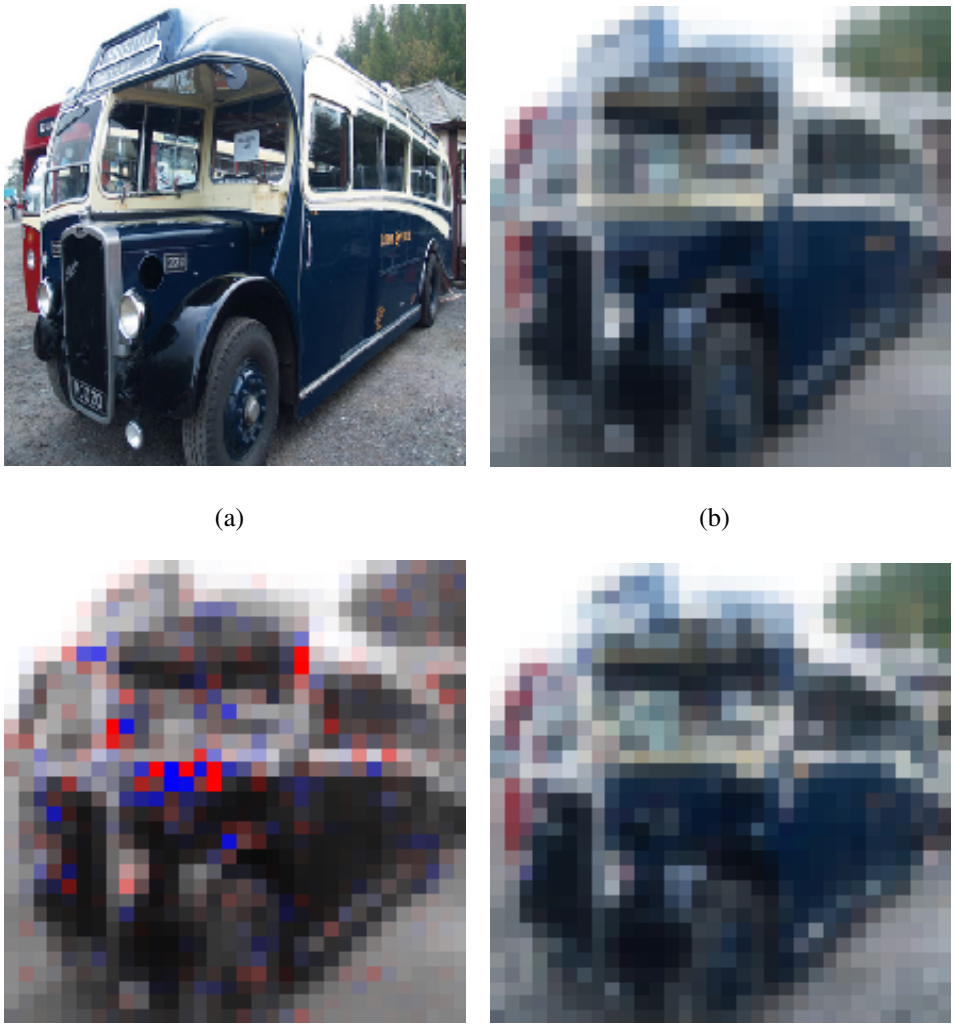
90.6% for 32×32 and 64×64 classifiers respectively. Fearing that the extreme variation in neighboring pixels might be adversely affecting the performance of the classifiers, we further added a low pass Gaussian filter to smooth out the images. We used a binary search to find a radius that performs relatively well. Table 5.1 shows the mean accuracies and corresponding 95% confidence intervals. We observe substantial accuracy drop when direct subsampling is applied. Smoothing helps, but in the best case, we still see an average of 9.3% accuracy drop for 32×32 classifiers and 6.1% accuracy drop for 64×64 classifiers.

To further study the reasons behind the misclassifications, we study the effect each pixel has on the final loss function. To quantify the effect, we define the following measurement given loss function L, bilinearly resampled image $\mathbf{X}$ and smoothed subsampled image $\mathbf{X}'$:

$$\mathbf{I} = \frac{\partial L}{\partial \mathbf{X}'} \odot (\mathbf{X}' - \mathbf{X})$$

$\frac{\partial L}{\partial \mathbf{X}'}$ represents how sensitive the loss function to the change of an individual pixel. It could be positive or negative, where increasing pixel value causes loss increase or decrease, respectively. $\odot$ is the Hadamard (element-wise) product. $(\mathbf{X}' - \mathbf{X})$ represents the change from the standard image to the subsampled. Intuitively, large changes in the sensitive pixels will inflate the loss and cause the classifier to misclassify.

We studied many misclassified samples and selected an example image of a bus as shown in Figure 5.2. Although area inside the windshield in the subsampled image appears somewhat jagged, it is not contributing much in terms of error. It is the edge of the windshield, where the



(a) (b)

(c) (d)

Figure 5.2: Errors caused by subsampling. (a) The original image of a bus; (b) 32×32 resampled image using bilinear kernel; (c) visualized source of error $\mathbf{I}$ in channel "R". Red dots are pixels that increase loss, and blue dots are those that decrease loss. The less transparent the color, the larger impact a pixel has; (d) 32×32 smoothed subsampled image.

color abruptly changes, that is contributing the most to the classification error. Apparently failing to sample the right color during subsampling distorted the appearance of the window frame and caused the classifier to misclassify. Above is a typical example where smoothing cannot recover the information lost during subsampling. The *only* way to mitigate the problem is to sample at a higher resolution to include more information.

Remark 1 *Subsampling may distort key features that image classifiers use, causing unrecoverable classification performance degradation.*

5.4 Proposed Method

To avoid confusion, we are going to use the term "two-step subsampling" to describe the process of subsampling to create input images of specific sizes for a classifier. We formally define this process in the next subsection. Following the definition, we describe our proposed adaptive sampling strategy, AdaSkip. AdaSkip requires minimal hardware support. We discuss an alternative hardware design in the last subsection.

5.4.1 Two-Step Subsampling

Let $H_o \times W_o$ be the resolution of the image sensor. The image sensor also supports a list of K subsampling steps, $\mathbb{S} = \{s_1, s_2, \ldots,$
$s_K\}$, where s_i represents the number of pixels among which one is sampled. We use `subsample(`$\mathbf{X}, s$`)` to represent the process of sampling with step s, both horizontally and vertically. $\mathbf{X}$ is of size $H_o \times W_o$, so the resulting image is of size $\frac{H_o}{s} \times \frac{W_o}{s}$. Instead of of subsampling the whole image $\mathbf{X}$, we use the same expression to represent the subsampling of one row $\mathbf{x}$, `subsample(`$\mathbf{x}, s$`)`.

To subsample to create images of specific size $H_t \times W_t$, we first command the image sensor to subsample with step s, where $\frac{H_o}{s} > H_t, \frac{W_o}{s} > W_t$. Then, taking the output from the sensor, we apply resampling (in software) using kernel function f `resample(`$\mathbf{x}, f$`)` to create image of the target size. We use bilinear kernel in our experiments.

5.4.2 AdaSkip

Since undersampling distorts essential image features, we should sample at a rate that is high enough to keep the features intact. However, not all parts of the image need a high sampling rate. It's more desirable to adaptively change the rate of sampling. The process of progressive exposure and readout in rolling shutters creates slacks after each row. We could use the slacks for making decisions on the step size used for the subsequent rows, based on the information that we obtain from the current row. Using this information, we could speculate on whether to sample them using high resolution or low resolution. With the varying granularity within one frame, the resulting image will look very undesirable aesthetically. However, deep learning algorithms might be able to look at the resulting image in a different light. Since the process of deciding what resolution to sample is adaptive, we coined the name AdaSkip, which stands for adaptively skipping rows.

In computer vision, the *gradient* of an image is a very fundamental element in feature engineering. It is usually defined in two terms, the gradient along the x direction G_x and the gradient along the y direction G_y.

$$G_x(\mathbf{X}) = |\frac{\partial \mathbf{X}}{\partial x}|$$
$$G_y(\mathbf{X}) = |\frac{\partial \mathbf{X}}{\partial y}|$$

Image gradients have an extensive range of usages. For one, it is commonly used as building blocks for edge features. As a direct feature, it is also used to determine the complexity of different parts of the images in seam carving. As a simple indicator of the complexity of the images, it is adopted in our algorithm to decide the sampling resolution. However, since we only have the information about the current row sampled, we will only use the gradient along the x-direction G_x. Algorithm 6 provides more details of the AdaSkip algorithm. We use $G_x(\mathbf{x_i})$ to represent the gradient in the row $\mathbf{x_i}$.

The high-level idea of the algorithm is extremely straightforward: use a lower resolution if the current row is not complex, else use a higher resolution. There are a few details that are not reflected in the algorithm. First, in order to make sure that the thd is the same for two resolutions (two rows of different sizes), the denser row should be subsampled in the calculation of $G_x(\mathbf{x_i})$ to match the sparser one. Second, since the subsampled rows are of different lengths, we duplicate

Algorithm 6 The AdaSkip algorithm

Input: $\mathbf{X}$, the original image, of size $H_o \times W_o$; s_h, s_l, the number of pixels to skip, $s_h < s_l$; thd, the threshold to determine which subsampling mode to use;

Output: $\mathbf{X}'$, the image of desired size $H_t \times W_t$

$step \leftarrow s_h$
Initialize $counter$
for each row $\mathbf{x}_i'$ **do** ▷ The main algorithm in hardware
    if $counter = step$ **then** ▷ Sample and decide resolution
        $\mathbf{x_i}' \leftarrow \texttt{subsample}(\mathbf{x_i}, step)$
        $counter \leftarrow 1$
        if $\max(G_x(\mathbf{x_i}')) > thd$ **then**
            $step \leftarrow s_h$
        else
            $step \leftarrow s_l$
        end if
    else ▷ Skip the row
        $counter \leftarrow counter + 1$
    end if
end for
$\mathbf{X}' \leftarrow$ collections of rows $\mathbf{x_i}$
return $\mathbf{X}'$

pixels in rows with the lower resolution to create rows of equal lengths. Those rows are later duplicated to fill in the skipped row in order to retain the aspect ratio.

5.4.3 AdaSkip Hardware Extension

Since the AdaSkip algorithm varies the sampling rate per row, it can only be implemented as part of the images sensor hardware. The core logic is relatively simple, and it mainly involves line 7 in Algorithm 6. The calculation of gradient involves adders and converting two's complement representations. Comparators are next for finding the maximum value. Both components are already present in the current image sensor design. Adders exist in digital gain adjustment circuits, and comparators in SAR based ADCs. To reduce the energy of the design, we could simply lower the comparator precision by taking the most significant bits (in our case 4 bits) of the pixel value.

5.5 Experimental Setup

The ImageNet dataset used for Large Scale Visual Recognition Challenge (ILSVRC) contains 1,461,406 images from 1000 classes [115]. It contains object classes of very different nature and granularity, including classes ranging from "warplane" to a specific dog breed called "Irish water spaniel". The median number of pixels in the images is around 200,000. Due to resolution limitations of the dataset, it is not possible to conduct experiments to reflect the performance on today's high-resolution cameras. We decide to simulate the behavior of an image sensor with a resolution of 512×512. Images are bilinear resampled and reshaped to the resolution, and we assume that each pixel in the reshaped image corresponds to a sensor pixel. For subsampling, we assume that when the supported skip steps are s =1, 2, 4 and 8. For simplicity, we simulate outdoor scenarios with relatively less exposure time and thus less t_{idle} [117]. In this case, the majority of energy consumption occurs during the active mode. The image sensor energy is estimated based on the model described in Section 5.2.2.

To reenact a reasonable ratio between the image sensor resolution and classifier input size, we experiment on classifiers with 32×32, 48×48, and 64×64 input images. The resized images

Input size	# Classes	Quantity	Accuracy (%)
32×32	10	10	86.9 ± 3.3
32×32	20	10	81.1 ± 2.0
32×32	30	5	78.9 ± 2.3
48×48	10	10	87.3 ± 4.2
48×48	20	5	86.0 ± 3.3
48×48	30	5	84.9 ± 2.7
64×64	10	10	90.6 ± 2.1
64×64	20	5	88.0 ± 3.5

Table 5.2: Classifiers used in our experiments

Energy saved (step) / Classifier	$56.0\times$ ($s = 8$)	$15.5\times$ ($s = 4$)	$3.97\times$ ($s = 2$)
32×32, 10-class	-1.6 ± 0.6	-0.1 ± 0.2	0
32×32, 20-class	-1.3 ± 0.4	-0.1 ± 0.1	0
32×32, 30-class	-2.7 ± 0.4	-0.0 ± 0.2	0
48×48, 10-class	*-3.0 ± 0.9*	-0.4 ± 0.3	0.0 ± 0.1
48×48, 20-class	*-4.4 ± 1.8*	-0.2 ± 0.1	0
48×48, 30-class	*-4.1 ± 0.7*	-0.4 ± 0.3	0
64×64, 10-class	*-30.5 ± 9.0*	-0.2 ± 0.2	0
64×64, 20-class	*-36.5 ± 9.5*	-0.4 ± 0.5	0

Table 5.3: Accuracy loss caused by two-step subsampling with different step sizes on different classifiers

are created using bilinear resampling[1]. We randomly select 10, 20 or 30 classes from the 1000 classes, and train 5 or 10 classifiers to form the population of classifiers with the corresponding configuration. The randomly sampled classes have varying levels of semantic similarities. Using these classifiers of very different nature as well as using different input sizes are all parts of our efforts to make our conclusions more generalizable.

All of the classifiers use the ResNet34 architecture [131], trained with over 150 epochs on an Nvidia GTX 1080Ti. We use SGD with a momentum of 0.9. The learning rate starts at 0.1 and is reduced to 1/10 the value every 50 epochs. Table 5.2 lists the classifiers. In addition, we filter out images with an original confidence score below 0.8 when we compare different subsampling methods [2].

5.6 Evaluation

We start the evaluation by presenting the performance characteristics of all our classifiers when two-step subsampling is used. Next, we demonstrate the AdaSkip algorithm by first visualizing it, and then show its performance with respect to energy and accuracy.

5.6.1 Two-Step Subsampling

Table 5.3 shows the accuracy loss using different steps s for subsampling. Different columns represent different step sizes and on the top row, we also showed the relative energy savings. The italic values represent cases where after subsampling, the image is less than twice (in both width and height) the size of the required input. Comparing to Table 5.1, even just doubling the sampling rate substantially reduces the accuracy loss. Further, we observe that the accuracy loss is correlated much more with the step size than with what classifier it is. It appears that as long it is at least $2\times$ larger than the target resolution of the classifiers, the same resolution after subsampling causes

[1]To our knowledge, all of today's classifiers are use smooth natural and smooth images that are interpolated at least linearly when downsizing. In our experience, classifiers trained on images with strong aliasing suffer more from overfitting and exhibit inferior performance.

[2]In these cases, the class scores before the softmax layer are almost uniformly distributed. A small change in the input may trigger an arbitrary change in the prediction. These predictions are usually discarded in a practical setting.

similar accuracy loss to all classifiers. These findings echo with the analysis from Section 5.3, where we argue that information loss at the subsampling step plays a crucial role in determining the performance degradation of image classifiers.

This characteristic has an interesting implication for system designers. Given a subsampling step size s, one could estimate the accuracy loss of potentially all classifiers using just a few of them. Based on these characteristics, hardware providers, for example, could present guidelines to app developers on the best subsampling strategy to use for different target image sizes needed by classifiers.

Remark 2 *Regardless of what classifier it is, the performance degradation of a classifier given subsampled input images is largely decided by the subsampling step size.*

5.6.2 AdaSkip Subsampling

5.6.2.1 Visualization

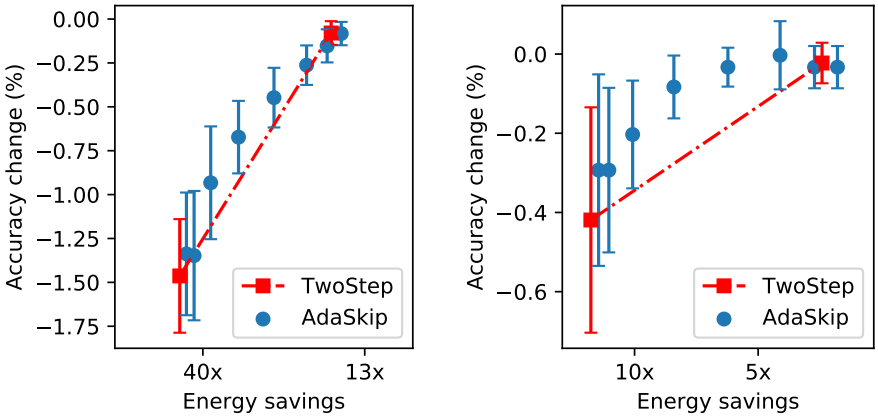
Figure 5.3 visualizes the simulated subsampling procedure of the AdaSkip algorithm. The image belongs to the class `black stork`. For AdaSkip, we set the threshold for switching between the higher resolution $s_h = 4$ (equivalent to resolution 128×128), the lower resolution $s_l = 8$ (equivalent to resolution 64×64), and threshold to 96 [3]. To visualize the effect of using different subsampling resolutions, we resort to using grayscale and colored parts. As a result, as shown in Figure 5.3(b), a few rows on the top depicting grass and roughly the entire lower half are sampled using the lower resolution. Figure 5.3(c) depicts the output image from a two-step subsampling procedure. The first step is the hardware subsampling with $s = 4$ (resulting in a 128×128 image), and the second step is software bilinear resampling. Figure 5.3(d) shows the results from bilinear resampling the image in 5.3(b). Comparing 5.3(d) with 5.3(c), we barely see any difference. The resulting image is indeed correctly classified by the classifier.

[3] In our algorithm the threshold does not vary with respect to images.

(a) (b)

(c) (d)

Figure 5.3: Visualization of AdaSkip-based subsampling. (a) The original 512×512 image; (b) the output of the AdaSkip algorithm, where the greyscale parts are rows sampled with $s_l = 8$ and the colored part are rows sampled with $s_h = 4$; (c) 32×32 output images from two-step subsampling with $s = 4$ and bilinear resampling; (d) 32×32 images obtained by bilinear resampling the output in (b).



(a) (b)

Figure 5.4: Energy accuracy trade-offs using AdaSkip. (a) 32×32 classifiers with $s_l = 8, s_h = 4$; (b) 48×48 classifiers with $s_l = 4, s_h = 2$

5.6.2.2 Hardware Energy Consumption

We assume our hardware support is an auxiliary to an existing image sensor. Our implementation on a Spartan-6 LX45 field-programmable-gate-array (FPGA) shows the power overhead of the 128-pixel ($s_l = 4$) AdaSkip hardware support is approximately 13.5 mW. We claim that this overhead can be significantly reduced if AdaSkip hardware support is implemented on ASIC or co-optimized with the image sensor. Kuon *et al.* estimate the power gap between ASIC and FPGA implementations of logic only designs could be as large as $5.7 \times -52\times$ [132]. We estimate the power consumption of AdaSkip hardware support to be 2.4 mW or lower on ASIC, indicating less than 1% additional power in our energy model comparing to an image sensor without AdaSkip. In an actual image sensor, we argue that there is no need for more than 600-pixel AdaSkip support, as rarely will a classification problem need that large of a resolution. Without considering the possibility of using memory and serializing, that amounts to less than 5% of additional power. We use that as the overhead in the following discussion of energy.

5.6.2.3 Energy Accuracy Trade-Off

Simply by two-step subsampling, we already saved a significant amount of energy. To take a step further, we apply AdaSkip in this setting. Figure 5.4 shows the accuracy changes using different thresholds in AdaSkip. In the figure, the y-axis is the accuracy change, and the x-axis is the relative energy consumption. Blue dots represent the results from AdaSkip, where each point represents a threshold (multiples of 32). Red square represents the baselines using two-step subsampling for s_l (left) and s_h(right) respectively.

When the threshold is 0, AdaSkip degrades to two-step but with additional hardware overhead, thus falls below the baseline. However, for almost all the other points, AdaSkip constantly achieves better energy accuracy trade-offs. In the case of Figure 5.4(b), sampling at $s = 2$ sustained an average of 0 accuracy lost. That is equivalent to $4\times$ energy savings. Using AdaSkip, we can boost the energy savings to around $7\times$ without losing any accuracy.

5.7 Conclusion

We present a study on how subsampling affects the performance of DNN-based image classifiers. We first demonstrate that one could achieve over $15\times$ energy savings just by subsampling while suffering almost no accuracy lost. Then we empirically show that the accuracy lost can be predicted as an approximate function of subsampling step size, regardless of what classifier it is. To achieve better accuracy when subsampling aggressively, we propose the AdaSkip algorithm. We implement AdaSkip on an FPGA and estimate the overhead in a real image sensor to be less than 5%.

CHAPTER 6

Customized On-Device Deep Neural Networks Pruning

In the previous chapter, we discussed a method to save energy in image sensors. Now we move further down the computer vision pipeline to discuss the energy efficiency of machine learning models. In recent years, CNNs kept outperforming traditional machine learning models in computer vision [133] [134] [135] [136] [131]. However, the exceptionally high computation and memory requirements of these models hinder their massive deployment on mobile and embedded devices.

To tackle the problem, researchers proposed various ways to prune convolutional neural networks in order to reduce the computation and memory requirements of the models. A big body of works have focused on pruning parameters from fully connected layers [137] [138] [139]. Although a large number of parameters are needed to compute fully connected layers, it is the convolution layers that are dominating the computation [140]. Realizing this, researchers started to propose filter-level pruning methods with the aim to reduce computation requirements in convolutional operations [141] [142] [140]. Nevertheless, the procedures in all of the methods above follow similar patterns: examine which parameters/filters have less impact on the final outputs, prune the parameters/filters, retrain the network to regain classification. In this chapter, however, we examine pruning in a different light.

In today's deep learning application development workflow, developers usually train one model on the servers and ship the same model to all end users. It is intrinsically hard for developers to cater to the diversified needs of different users. To satisfy the needs of all the users, the model has to be a universal one that includes different capabilities needed by different groups of users. In that case, the model will inevitably include some of the unnecessary capabilities which a particular user does not need. In the case of classification, for example, the model might be able to identify

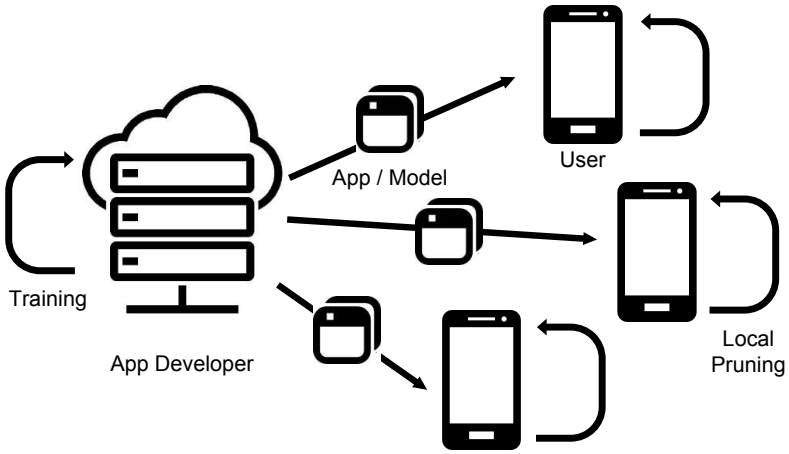


Figure 6.1: Once the neural network models are trained and shipped to users, each user will run a local program to prune the model according to their needs.

more classes than what a user would need it to. Evidently, an increase in the number of classes that the model could classify is accompanied by an increase in the model size and computational costs. Therefore, resources are wasted for a user when s/he is running inference using a model more powerful than needed.

To tackle the problem, one might think of training different, smaller models for different users. However, the extremely high computational cost of training a model prevents that from being feasible. A more desired approach would be to *prune* unneeded classes and the parts associated with those classes from the existing model. The procedure should be so lightweight that users could run it on their own mobile devices. We will later refer to it as ”on-device customization”. An on-device customization method would enable a new workflow of application development, as shown in Figure 6.1. In the new workflow, developers still train one model, albeit a comprehensive one that is endowed with all the possible capabilities. Then the users use a lightweight program that prunes the model locally on the phone to the extent that only the capabilities that the users want are left. After that, the user could run faster and more energy efficient inference using the smaller pruned model.

In this chapter, we propose a method that enables on-device customization of neural network models without the requirement for retraining. The key idea of the proposed method follows two

steps: pruning and compensation. We first identify and remove the parts of the network that do not contribute as much in distinguishing between the target subset of classes. Then we utilize the remaining filters to compensate for the pruned filters. Aside from a few parameters that have to be pre-calculated before shipping the model, the whole procedure incurs little overhead. We have developed the method and tested it on the Network-in-Network(NIN) model with CIFAR-10 dataset implemented on an off-the-shelf smartphone using TensorFlow mobile support [143]. Since the proposed method only prunes filters related to certain unneeded classes, it could potentially be applied in parallel to the aforementioned pruning methods as a client-side counterpart.

The rest of the chapter is organized in the following manner. Section 6.1 reviews related work. Section 6.2 lists the notations that we will use later to describe the proposed method. Section 6.5 outlines the method on a conceptual level. Section 6.5 explains how the method can be implemented by directly modifying existing models in an online fashion. Section 6.6 evaluates the method through classification accuracy tests as well as energy and wall-clock time measurements.

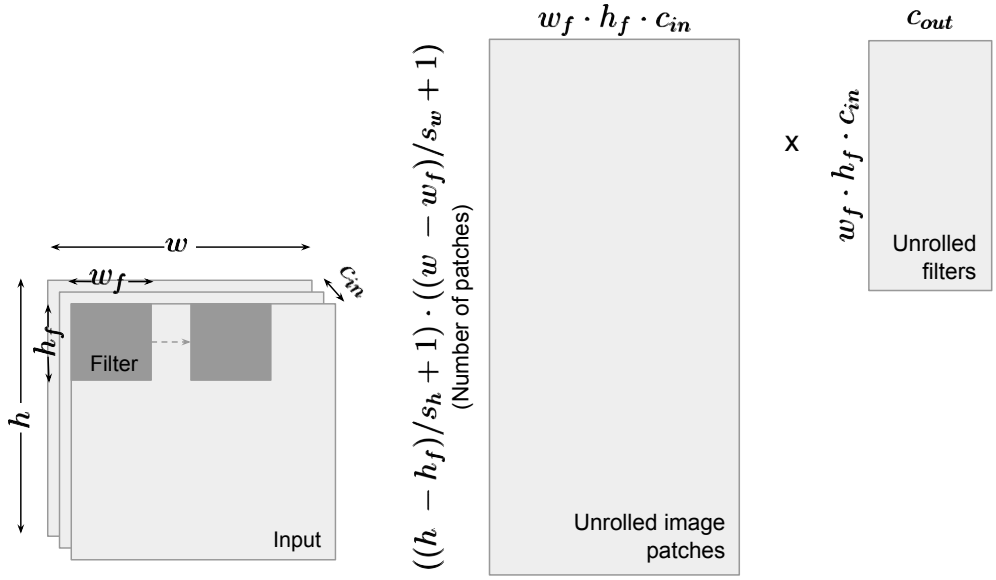
6.1 Related Work

Researchers have proposed various methods to speed up the computation of deep neural networks. One straight forward approach is to reduce the precision of computation. Notable examples include weight quantization (weight sharing) [144] and binary neural networks [145]. Other researchers proposed to take advantage of the redundancies in the weight by adopting matrix/tensor decomposition techniques [146] [147].

Our work is more concerned with the third category of methods that involves pruning neural networks. Le Cun *et al.* was one for the first researchers to introduce the idea of pruning neural networks, where he pruned parameters that analytically has less effect when perturbed [137]. Hassibi *et al.* proposed to use second order derivative to determine which parameters to prune [138]. Han *et al.* proposed to remove weights with magnitudes smaller than a threshold, and retrain the network to regain accuracy. [139]. Specifically for reducing the size convolutional neural networks, various researchers have proposed methods that prune the model at higher levels than individual neurons. Polyak *et al.* used the variance in the activation to estimate the importance of feature

maps [141]. In comparison, Li *et al.* used the sum of absolute weights of each filter as the criteria [140]. Anwar *et al.* pruned models at different levels and used a particle filter to decide the best filters to prune [142].

6.2 Preliminaries



(a) (b)

Figure 6.2: Demonstration of the unrolled convolution operation.

In almost all of today's machine learning frameworks, the computation of convolution operation is unrolled into a matrix multiplication, as shown in Figure 6.2. Let h_f and w_f be the height and width of the filter in a convolution layer with c_{out} number of filters and with inputs of shape $h \times w \times c_{in}$. Then before computing the convolution, the image patches with shape $h_f \times w_f$ for each channel c_{in} are unrolled into vectors, which make up for the rows in the matrix on the left in Figure 6.2(b). The weights for each of the c_{out} output channels are unrolled to form the columns in the matrix on the right.

When we involve parameters or variables from convolution layers in this chapter, we are implicitly referring to those from the unrolled matrix multiplication form of convolution. In most cases, we are dealing with only one channel, the jth channel of the input. In these cases, the input

feature map of ith sample is:

$$\boldsymbol{X}_{ij} = \begin{bmatrix} \boldsymbol{x}_{ij1}^{\mathsf{T}} \\ \boldsymbol{x}_{ij2}^{\mathsf{T}} \\ \vdots \\ \boldsymbol{x}_{ijH}^{\mathsf{T}} \end{bmatrix}$$

$\boldsymbol{X}_{ij}$ is of height $H = ((h - h_f)/s_h + 1) \cdot ((w - w_f)/s_w + 1)$ and width $W = w_f \cdot h_f$, where s_h and s_w are strides. We use $\boldsymbol{w}_{jk}$ to represent the unrolled weights vector for the jth input channel and kth output channel.

When it comes to pruning, we use $Y_{target} = \{y_1, y_2, \ldots, y_{M'}\}$ to represent the subset of M' classes that the specialist model targets.

The program for on-device customization can be represented as a function `f` that takes in the original model together some parameters, and returns a smaller model: `pruned_model = f( original_model, parameters)`. The parameters mentioned here are a list of pre-computed values needed for pruning and compensation. This function requires a relatively small footprint, can run on mobile devices and does not require retraining and fine-tuning of the original model.

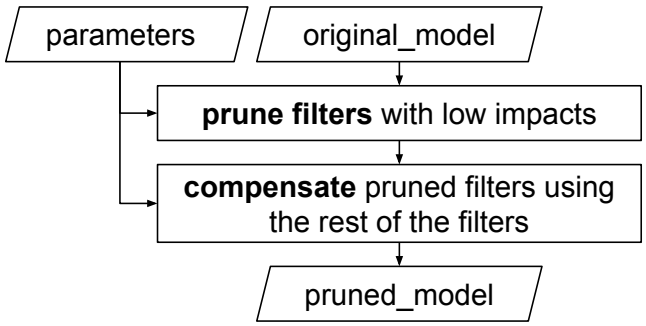


Figure 6.3: System workflow of our customized DNN pruning method.

Figure 6.3 shows an overview of `f`. The function operates in 2 major steps. It first identifies and removes filters in convolution layers that are less important in distinguishing between the target subset of classes. Those filters are determined to be low-impact filters and we could thus tolerate some amount of errors in calculating them. In the next step, we use a linear combination

of the rest of the filters to approximate and replace the filters we removed. Both steps require some pre-calculated parameters to finish.

6.3 Pruning Filters

Recent works in visualizing convolution neural networks have shed light on the role that different layers in the network play [148]. It is now widely accepted that different filters in a convolution layer represent different features. The filters in earlier layers represent lower level features, while the filters in later layers represent higher-level features.

Given the interpretation above, we have the following assumption: while some features might be universal, others are associated closely with certain classes. If we are to target a certain subset of classes and remove the rest, we only need the features that are associated with the target classes. We could remove the rest without severely damaging the core feature extraction abilities of the model. The errors introduced by pruning can be relatively easily compensated.

While pruning filters accord with our interpretation of filters as features, there are other merits in this way of pruning convolution operations. Pruning entire filters in a convolution layer is effectively creating another convolution layer with a different (smaller) set of weights. It can be easily implemented on any existing machine learning frameworks. Thus pruning filters is a great fit to our proposal.

We now introduce our approach to pruning filters. We introduce our way of determining which filters to prune, the *Filter Sensitivity Analysis* method. Then we show how we apply the method to determine which filters to prune.

6.3.1 Filter Sensitivity Analysis

To determine which filters to prune, researchers have proposed to use criteria such as variances in the channel activation [141] and the sum of absolute weights [140]. For our particular use case, we need a method that could tie the criterion closely to particular classes. In other words, we want to prune filters that are less useful to the subset of the classes that are targeted in the specialist model.

To achieve that objective, we propose *Filter Sensitivity Analysis*.

In traditional sensitivity analysis methods, usually, only one variable is at play. In the case of analyzing the impact of a channel on certain classes, we have $h \times w$ number of variables to examine. Apparently, we need to find a way to represent the feature map as a whole.

Our proposed method approaches this problem by assigning a weight variable $\omega_j = 1$ for each channel j of a layer and shifting our attention to the weight variable instead of the feature map itself. Suppose a sample produces a feature map $\boldsymbol{X}_j$ at channel j (now effectively $\omega_j \boldsymbol{X}_j$ after we added the weight), and p_y is the score/probability corresponding to the class y that the sample belongs to, then the *impact* $\mathcal{I}_j y$ of that channel j has on class y is defined as

$$\mathcal{I}_{jy} = \frac{\partial p_y}{\partial \omega_j}$$

In this chapter, the impact is obtained by applying a small perturbation $\Delta\omega_f$ to the weight. We measure the difference in the output of the score Δp_y and calculate the impact by $\mathcal{I}_{jy} = \Delta P_y / \Delta\omega_f$. We average the impact values for every single sample to obtain the final impact.

To intuitively interpret the method, we can view ω_j as a variable that controls how much we want to strengthen or weaken the feature that the channel j represents. If we strengthen that feature by increasing ω_j, the more the class score changes, the more the feature is associated with the class. Thus the impact serves as a good indicator of whether we should remove a filter: if the channel that the filter produces has a high impact value on / is not closely associated with the class we need, then we should prune that channel.

6.3.2 Selecting Which Filters to Prune

Given the measure of a filter's impact on a particular class, we now describe the filter selection process. We set a threshold $I_{threshold}$, and prune filter j if j satisfies

$$\max\{\mathcal{I}_{jy} | y \in Y_{target}\} < \mathcal{I}_{threshold}$$

. The threshold can be pre-computed given the target percentage of removal Δc. If we want to remove $\Delta c = 50\%$ of, then we calculate $\max\{\mathcal{I}_{jy} | y \in Y_{target}\}$ for each j, and set the $50th$ percentile as the threshold $\mathcal{I}_{threshold}$.

6.4 Compensation

Once the filters are pruned, traditional methods will resort to retraining the network to regain accuracy. Early works reported 2x more time spent on retraining than training [139]. Until recently it still took 20-40 epochs of retraining for a pruned specialist model to regain accuracy [140]. If the desired objective is to create a model that classifies on a subset of labels for every ad hoc situation, a method that requires retraining is equivalent to training a new model every time. It is certainly inefficient and most likely infeasible. Therefore, an *online* method that requires no retraining is desired in this setting. This is where the process of what we call *compensation* comes into play. The objective of compensation is to restore activation of the filters that are pruned away from the model. The closer the restore values are to the original ones, the closer the more accurate our network will be.

We adopted three ways of compensation. The first way is to simply use the mean of the feature map. The second is to use the feature map of the most similar filter. The third is compensate using a linear combination of the rest of the filters.

6.4.1 Compensation with Mean

Since we are compensating for only a subset of classes, when we are calculating the mean of feature maps, we should only select the the samples that belong the the subset of classes Y_{target}. The mean of activation for jth channel $\bar{\boldsymbol{X}}_j$ is calculated by:

$$\bar{\boldsymbol{X}}_j = \frac{1}{|\{i|y_i \in Y_{target}\}|} \sum_{\{i|y_i \in Y_{target}\}} \boldsymbol{X}_{ij}$$

Once we obtain $\bar{\boldsymbol{X}}_j$, we use it as the feature map for channel j whatever the input data is.

6.4.2 Compensation with Correlated Filters

The first step of this method is to find the most correlated channel for all the channels to be pruned. To obtain the correlation measurement, we unroll the feature map of every channel into a vector. Let $\mathcal{C}_i(j', j)$ represent the Pearson correlation of the vectors of channel j' and j given input sample

i. The correlation between channel j' and j is calculated by averaging the correlation of individual samples that belong to the target subset of classes:

$$\mathcal{C}(j', j) = \frac{1}{|\{i|y_i \in Y_{target}\}|} \sum_{\{i|y_i \in Y_{target}\}} \mathcal{C}_i(j', j)$$

To compensate for the channel j that is to be pruned, we choose a channel j' that bears the highest correlation value $\mathcal{C}_(j', j)$.

When we compensate channel j using the feature map of another channel j', we are effectively calculating the convolution on channel j' twice: the first time using its own weights $\boldsymbol{w}_{j'k}$, the second time using the weights for the pruned channel $\boldsymbol{w}_{jk}$. Inevitably there are errors caused by such replacement. However, there is no easy way of modifying the new feature map such that the values are closer to the original feature map. By easy, we mean that it could be implemented using operations existing in convolution neural networks. One workaround is to adjust the weights such that the next convolution layer will produce closer results.

Consulting the notation in Section 6.2, let $\boldsymbol{X}_{j'}$ represent the feature map of the replacement channel that compensates $\boldsymbol{X}_j$, and $\boldsymbol{w}'_{jk}$ is used to represent the new set of weights for output channel k that we want to analytically solve for. We set our objective to be minimizing the mean square error after we use the new feature map to calculate the convolution:

$$\underset{\boldsymbol{w}'_{jk}}{\operatorname{argmin}} L(\boldsymbol{w}'_{jk}) = \sum_{\{i|y_i \in Y_{target}\}} \sum_{h=1}^{H} (\boldsymbol{x}_{ijh}^{\intercal} \boldsymbol{w}_{jk} - \boldsymbol{x}_{ij'h}{}^{\intercal} \boldsymbol{w}'_{jk})^2$$

By setting the derivative of the loss function to 0,

$$\begin{aligned}
&\frac{\partial L(\boldsymbol{w}'_{jk})}{\partial \boldsymbol{w}'_{jk}} \\
=& -2 \sum_{\{i|y_i \in Y_{target}\}} \sum_{h=1}^{H} (\boldsymbol{x}_{ijh}^{\intercal} \boldsymbol{w}_{jk} - \boldsymbol{x}_{ij'h}{}^{\intercal} \boldsymbol{w}'_{jk}) \boldsymbol{x}_{ij'h} \\
=& 0
\end{aligned}$$

We will have W equations for the vector $\boldsymbol{w}'_{jk}$ with W elements. We could solve for the new weights $\boldsymbol{w}'_{jk}$ using techniques that solve standard linear equations.

6.4.3 Compensating with Linear Combination of Filters

Approximating the output of filter j with a linear combination of the rest of the filter is expressed as follows

$$\hat{\boldsymbol{x}_j} = \sum_{k=1}^{N} z_k \beta_{kj} \boldsymbol{x}_k$$

here $z_k \in \{0, 1\}$ represents whether or not filter k is to be pruned ($z_k = 0$ means that filter k will be pruned), N is the total number of filters, and β_{kj} is the coefficient in the linear combination. We can solve for the best set of coefficients by minimizing the Mean Squared Error (MSE) over the subset of samples belonging to target classes

$$\underset{\beta_{jk}}{\operatorname{argmin}} \sum_{\{i|y_i \in \mathsf{Y}_{target}\}} \sum_{j=1}^{N} (1 - z_j) \cdot || \sum_{k=1}^{N} z_k \beta_{jk} \boldsymbol{x}_{ik} - \boldsymbol{x}_{ij} ||_2^2 \tag{6.1}$$

Intuitively, the expression represents the difference between $\boldsymbol{x}_j$ and $\hat{\boldsymbol{x}_j}$. Further, we only take into consideration samples from the classes that we care about. This minimization can be easily solved by taking the derivatives of the MSE. The seemingly complex process can actually be pre-computed. We will talk more about it in Section 6.5.1.

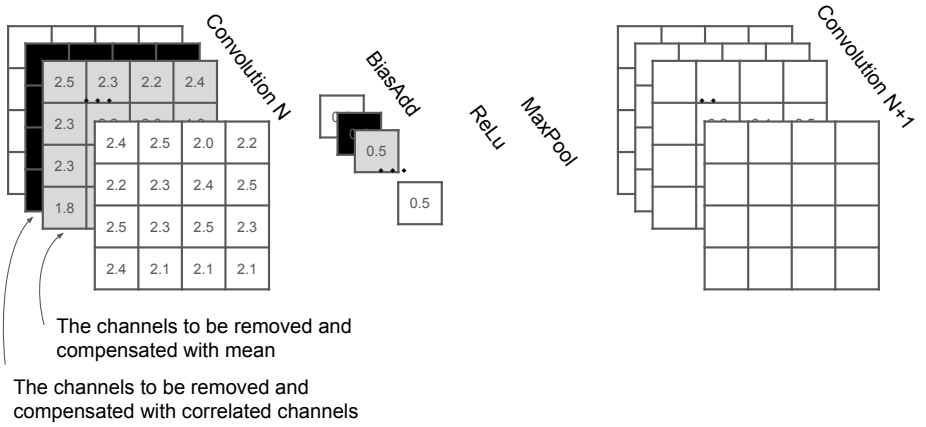
6.5 Implementation

Till now we have elaborated our method on a conceptual level. In this section, we will explain how we can implement the method by making modifications to an existing network model. Thus, we are essentially presenting a function f that takes in the original model together some parameters, and returns a smaller model:

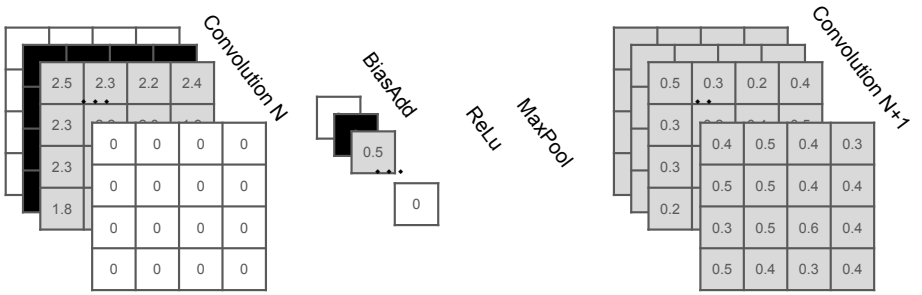
$$pruned\_model = f(original\_model, parameters)$$

This function requires a relatively small footprint, can run online on mobile devices and does not require retraining and fine-tuning of the original model.

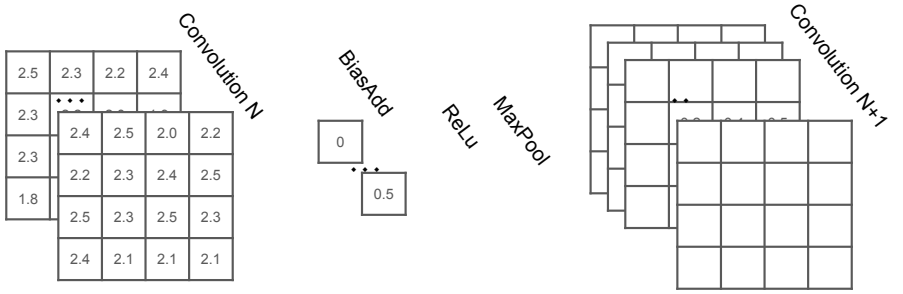
Similar to the previous section, we will introduce the procedures for pruning and compensation in separate subsections in the respective order.



(a) Before Pruning



(b) Compensating with Mean



(c) After Pruning

Figure 6.4: Implementation of pruning and compensation for convolution N.

To illustrate the procedure of pruning and compensation, we will focus on a dummy convolution layer called Convolution N, as shown in Figure 6.4. After computing convolution on its input, the convolution N layer adds bias to the results, and runs them through a ReLu layer and a MaxPool layer before the next convolution layer Convolution $N + 1$ takes them as its input.

6.5.1 Pruning

The implementation of pruning is straight forward. We simply need to remove the weights that are corresponding to the pruned channels from the previous layer and the current layer. Suppose the weight in the current convolution layer is $\boldsymbol{w}$ with shape $h_f \times w_f \times c_{in} \times c_{out}$ (for every channel among the c_{out} output channels, there is a $h_f \times w_f$ filter for every channel among c_{in} input channels). If we prune Δc_{in} percent of channels from previous layer and Δc_{out} percent from the current layer, which makes $c'_{in} = (1 - \Delta c_{in})c_{in}$ and $c'_{out} = (1 - \Delta c_{out})c_{out}$. Then the new weights will be in shape $h_f \times w_f \times c'_{in} \times c'_{out}$.

6.5.2 Compensation

In this subsection, discuss the different compensation methods.

6.5.2.1 Compensation with Mean

Implementing the compensation with the mean feature map is actually more tricky than it might appear. The key difficulty is that when we run the pruned specialist model, the network structure has already changed. Not only will the channels be removed, the weights that are originally used for these channels in the next convolution layer will no longer exist as well. There is no way of placing the compensation around the layer where the channels are pruned.

To deal with the problem, we place the compensation at the next convolution layer. The idea is to look at the *contributions* of those pruned channels on the next convolution layer and try to compensate for their contributions. We first fill the feature maps of the channels to be removed with their mean values and set the rest to 0. Then we run inference to obtain the feature maps at the next convolution layer. The values in those feature maps consist only of the contribution of the pruned channels. We save the values as a constant and add these constants to the next layer at run time.

6.5.2.2 Compensation with Correlated Channels

As we explained earlier, when replacing a pruned channel j with another channel j', we are effectively computing a convolution on the replace channel j' using weights for the pruned channel j. Since any output channel, k is the sum of convolutions on all input channels, we simply add the adjusted weights of channel j that we mentioned in Section 6.4.2 to the existing weights of channel j':

$$\boldsymbol{w}'_{j'k} = \boldsymbol{w}_{j'k} + \boldsymbol{w}'_{jk}$$

If a channel is used to compensate for multiple pruned channels, we can add all the adjusted weights together.

6.5.3 Compensation with Linear Combination of Filters

The linear nature of convolution allows us to implement our compensation method through direct modifications of the weight tensors. Suppose filter j is pruned, then the weight tensor of the *next* convolution layer should be modified for compensation. Specifically, let $\boldsymbol{W}$ of shape $C \times H \times W \times N$ be the weight tensor of the next convolution layer. We use $\boldsymbol{W}(j, :, :, :)$ to represent the weights for the jth input channel (the channel pruned). For any filter k that is not pruned, its new weight tensor should be its original weight tensor plus the weights for the pruned channels multiplied by the corresponding coefficients

$$\boldsymbol{W}'(k, :, :, :) = \boldsymbol{W}(k, :, :, :) + \sum_{j=1}^{N} z_j \beta_{kj} \cdot \boldsymbol{W}(j, :, :, :) \tag{6.2}$$

The added weights effectively represent separate convolution operations for the purpose of compensating the pruned channels. But since the convolution operation is linear, we can just add them to the existing weights. Once the pruning described in Section 6.5.1 is done, the weight tensor will only contain the channels with updated weights $\boldsymbol{W}'$.

6.5.4 Deployment and Computation Reduction

Throughout the chapter we have been stressing one of the key advantages of this pruning and compensation method: it can be run online. Running online involves doing calculations with some

pre-computed values. There are mainly four sets of values that we need to pre-compute:

1. The impacts each filter has on different classes $\mathcal{I}_{jy}$, described in Section 6.3.1;
2. The mean of feature maps $\bar{\boldsymbol{X}}_j$, described in Section 6.4.1;
3. The pairwise correlations $\mathcal{C}(j', j)$, described in Section 6.4.2;
4. The element wise product among pairs of channels feature maps used to compute the adjust weight $\boldsymbol{w}'_{jk}$ in Section 6.4.2.

These values can be pre-computed before the deployment of the model. When any user wants to prune a model, the procedure involves:

- Using 1 to determine which filters to prune;
- Using 3 to determine how pruned channels can be replaced by other channels, and 4 to recreate a new set of weights that accounts for the replacements;
- Using 2 to create the constant bias that are for the channels to be compensated by their mean values.

The computation can be done before the model is shipped to the users, and the parameters can be sent over to the users upon request. The pre-computed data totals about 20MB for the NIN model that we experiment within this chapter. Depending on which classes to remove, we only need the relevant portion of the data

Algorithm 7 describes the overall procedure. The main source of computation is from line 17, where one matrix inversion and one multiplication are involved for solving the linear equations for each one of the pruned filters. Combining the cost of transmitting the pre-computed parameters, the total overhead of our method should be within reasonable limits.

As we mentioned in Section 6.5.1. For a pruned convolution layer, the number of weights reduced from $h_f \times w_f \times c_{in} \times c_{out}$ to $h_f \times w_f \times (1 - \Delta c_{in})c_{in} \times (1 - \Delta c_{out})c_{out}$. That amounts to a $(1 - (1 - \Delta c_{in})(1 - \Delta c_{out}))$ reduction in the number of parameters and amount of computation.

Algorithm 7 Pruning and Compensating a Model

Input: (i) Y_{target}: the target subset of classes (ii) ΔN: the target removal rate (iii) the original model. $\boldsymbol{W}_l$ of shape $C_l \times H_l \times W_l \times N_l$ is the weight tensor of the lth convolution layer (iv) the pre-computed parameters

Output: (i)the pruned model. $\boldsymbol{W'}_l$ represents new the new weight tensor

z_j, β_{kj} = 1
for each convolution layer l **do**
    ▷ update weights using z and β from the last layer
    for $k := 1 : C_l$ **do**
        $\boldsymbol{W}'(k, :, :, :) = \boldsymbol{W}(k, :, :, :) + \sum_{j=1}^{N} z_j \beta_{kj} \cdot \boldsymbol{W}(j, :, :, :)$ ▷ Equation 6.2
    end for
    Remove $\boldsymbol{W}'_l(j, :, :, :)$ if $z_j = 0$
    ▷ update z and β for the next layer
    for $j := 1 : N_l$ **do**
        $\mathcal{I}_j := \sum_{\{y \in \mathsf{Y}_{target}\}} \mathcal{I}_{jy}$ ▷ using the pre-computed impacts $\mathcal{I}_{jy}$
    end for
    $\mathcal{I}_{threshold} := \Delta N$th percentile of $\{\mathcal{I}_j\}$
    for $j := 1 : N_l$ **do**
        $z_j := \mathcal{I}_j < \mathcal{I}_{threshold}$
    end for
    Obtain β_{kj} by solving Equation 6.1 ▷ using the pre-computed pairwise products
    Remove $\boldsymbol{W}'_l(:, :, :, j)$ if $z_j = 0$
end for

TensorFlow uses a protocol buffer class called GraphDef to serialize the model structure and weights into language independent binaries. The current version of TensorFlow Android support requires loading protocol buffer based models and invoking Java Native Interface (JNI) for running the C-based TensorFlow inference library. The `original_model` and the `pruned_model` that we mentioned in Section 6.2 are in reality serialized GraphDef protocol buffers. The function we implemented directly operates on the GraphDef models to carry out pruning and compensation.

6.6 Evaluation

In the evaluation section, we report early results from experiments done on the Network-in-Network (NIN) model [134]. Since our method targets convolution operations, we chose a model that does not include any fully connected layers. We've seen some very encouraging results from these early experiments, and it would certainly be interesting to investigate how this method applies to more complex models and larger datasets.

We first present a case study to provide a high-level understanding of our pruning process. Then we present detailed evaluations on classification accuracy, energy consumption, and latency respectively. We also discuss how our proposed pruning criteria compares to existing methods. Note that

6.6.1 Case Study

To clearly illustrate our approach, we present a pruned 5-class NIN model as a case study. The architecture of the model is shown in the Table 6.1. The structure can be roughly described as groups of 5×5 convolution layers (CONV) followed by 1×1 convolution layers (which the authors called *cascaded cross-channel parametric pooling (CCCP)* layer). The last CCCP layer outputs a total of 10 channels, each corresponding to one class in the CIFAR-10 dataset. The score for each class is produced from a global average pooling layer on the lat CCCP layer.

Table 6.1 further demonstrates how the model is pruned. It is compensated using the linear combination of filters. During our experiments, we found out that the first convolution layer

Name	Output Size	#Filters Before	#Params	#FLOPs Before	#Filters After	#FLOPs After	FLOPs Pruned
CONV1	32×32	192	1.92e+04	2.95e+07	192	2.95e+07	0%
CCCP1	32×32	160	3.09e+04	6.29e+07	111	4.36e+07	30.7%
CCCP2	32×32	96	1.55e+04	3.15e+07	67	1.52e+07	51.7%
CONV2	16×16	192	4.66e+05	2.36e+08	135	1.16e+08	50.8%
CCCP3	16×16	192	3.71e+04	1.89e+07	133	9.19e+06	51.4%
CCCP4	16×16	192	3.71e+04	1.89e+07	135	9.19e+06	51.4%
CONV3	8×8	192	3.34e+05	4.25e+07	134	2.08e+07	51.1%
CCCP5	8×8	192	3.71e+04	4.72e+06	134	2.30e+06	51.3%
CCCP6	8×8	10	1.93e+03	2.46e+05	5	8.58e+04	65.1%
Total				4.45e+08		2.46e+08	44.7%

Table 6.1: Comparison between the original 10-Class NIN model and pruned 5-class NIN models that on average achieves 91.2% classification accuracy, 38.5% in energy savings and 19.5% in latency improvements.

produces universal features, and is indispensable to later layers. Thus we empirically decide to leave the first layer intact while pruning the rest of the layers, from which we uniformly remove $\Delta N = 30\%$ of the filters. As we discussed in Section 6.5.1, that corresponds to a $(1 - (1 - \Delta C)(1 - \Delta N))$ reduction in the number of parameters and amount of computation except for the first and last layer. We prune 5 out of 10 filters, corresponding to the 5 pruned classes in the last layer.

6.6.2 Classification Accuracy

6.6.2.1 Compensation with Mean and Correlated Filters

Table 6.2 shows the classification accuracy on CIFAR-10 dataset with the NIN model pruned at different levels. The values with a dark grey background are equal or higher than the accuracy of the original 10-class model. Those with a light gray background are within 6% of the standard accuracy. The columns represent the level of pruning measured in FLOPs, with values ranging from reducing 57.8% to "None", which represent the original network. The rows represent networks pruned for subsets of classes of different sizes. For each level of pruning (except for the original network), we randomly pick 10 different combinations of classes to make up for a subset and test the pruned network against the test set to obtain the mean and the 95% confidence interval of the classification accuracy. For the original network, the results are obtained after removing unneeded classes from the SoftMax output.

To make more sense of the different levels for pruning, we have to use Section 6.6.1 as a reference. The different level of pruning is achieved by removing different portions of all the channels from layer CCCP2 to CCCP6. As shown in the case study, 38.9% is achieved by removing 30% from the aforementioned list of channels. Similarly, 57.8%, 48.6%, 27.9%, and 14.3% reduction rates are achieved by reducing 50%, 40%, 20% and 10% of the channels respectively. These reduction rates are referred to as *levels of pruning* throughout the chapter. As described in the subsection above, CCCP6, the last convolution layer of the network, has different channel counts given the size of the target subset of classes. For models with the same level of pruning but target a different number of classes, only the last layer will be different. Since the last layer only contributes to

Num. Classes \ FLOPs Pruned	57.8%	48.6%	38.9%	27.9%	14.3%	None
2	84.1 ± 6.7	90.8 ± 4.0	95.2 ± 2.4	96.1 ± 3.1	97.7 ± 1.5	98.5 ± 0.6
3	72.7 ± 7.0	84.0 ± 4.5	91.1 ± 2.6	93.9 ± 1.9	95.4 ± 1.9	96.4 ± 1.2
4	62.0 ± 9.0	76.0 ± 7.6	85.8 ± 4.4	90.8 ± 2.7	93.3 ± 2.1	95.1 ± 1.0
5	59.8 ± 2.5	74.9 ± 3.2	83.8 ± 2.6	89.3 ± 1.7	92.4 ± 1.2	94.2 ± 0.8
6	55.1 ± 3.4	72.5 ± 2.9	82.7 ± 1.6	88.7 ± 0.8	92.0 ± 0.7	92.9 ± 0.9
7	47.0 ± 5.6	68.9 ± 2.5	79.5 ± 1.8	86.0 ± 1.5	89.6 ± 1.2	91.8 ± 0.8
8	47.4 ± 2.9	67.7 ± 1.2	78.7 ± 1.3	86.3 ± 0.7	89.8 ± 0.7	90.7 ± 0.6
9	36.3 ± 2.2	65.8 ± 1.5	75.4 ± 1.2	83.7 ± 0.7	88.0 ± 0.5	90.3 ± 0.6
10	28.7	62.6	74.3	82.7	87.2	89.6

Table 6.2: Classification accuracy on CIFAR-10 dataset using NIN with different levels of pruning.

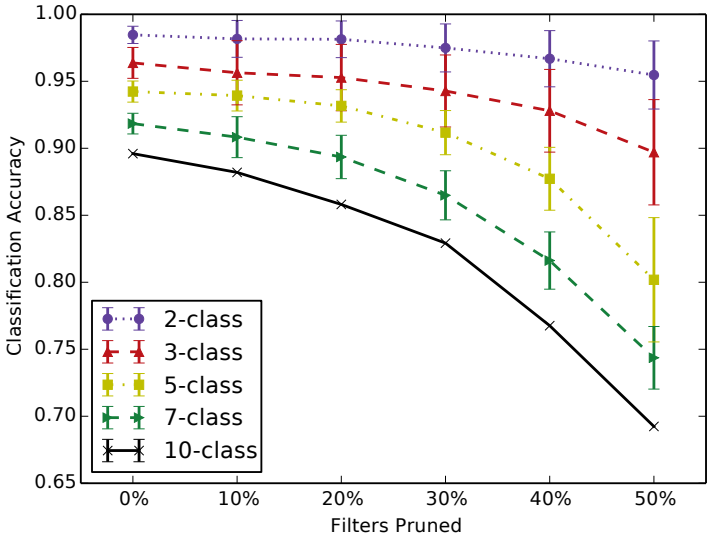
0.05% of the total number of FLOPs, we ignore the differences and consider models with the same level of pruning share the same computation requirements.

Note that the accuracies of some of the pruned specialist models are the same as or even higher than the original 10-class classifier. The reasons behind it stem from our problem setting of "pruning classes". Given our definition of the problem, the test set only contains samples of the chosen subset of the classes, and thus the problem space is smaller.

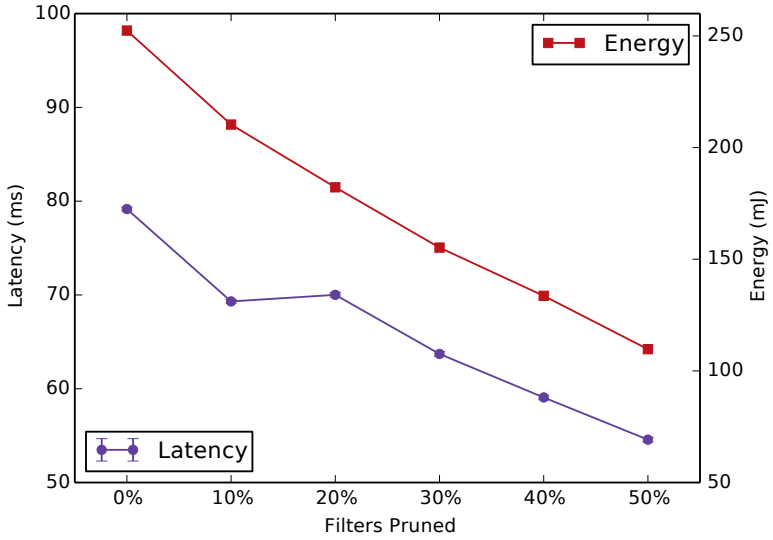
One problem that we observe in our experiments is that models for different subsets of classes with the same level of pruning vary in prediction accuracy. This problem is especially prominent when we prune larger portions of the network. For example, pruning half of the filters for some combination of two classes produces an accuracy of merely 0.624, while for other combinations it produces an accuracy as high as 0.923. We believe it has to do with the fact that different subsets of classes have a stronger dependency on different subsets of filters at each layer. Some classes depend on more features than others and thus need more channels at each layer. Some combination of classes could share similar sets of features and are more "compatible" with each other in a combination. In our approach, we prune a uniform percentage of filters across all layers and for all subset of classes. That approach is clearly problematic. We have yet to devise a method that could compare the normalized impact of one filter (in any layer) on each of the class. We believe such a method could drastically improve the accuracy of the pruned networks.

6.6.2.2 Compensation with Linear Combination of Filters

Figure 6.5(a) shows the classification accuracy with the model pruned at different levels. The y-axis shows classification accuracy, and the x-axis shows the percentage of filters pruned (except the first convolution layer and the last layer). In particular, 0% shows the prediction accuracy using highest-scoring predictions from the original model after the unneeded classes are removed. For each experiment done with a different number of target classes, we randomly pick 10 different combinations of classes to make up for a subset and test the pruned network against the test set to obtain the mean and the 95% confidence interval of the classification accuracy. As we can see from the figure, even if we aggressively prune half of the filters, 2 and 3-class models on average



(a) Classification Accuracy



(b) Energy and Latency

Figure 6.5: Performance evaluations of the method using NIN on CIFAR-10 dataset with different percentage of filters pruned [1].

still achieve a higher classification accuracy than 10-class classifiers. 2-class models achieve an average accuracy of 95.5%, compared to the 98.5% of the unpruned model. When pruning 30% of the filters from 5-class models, the accuracy changed from 94.2% to 91.2%. For 10-classes models, apparently, the accuracy is always below the original accuracy if we do any pruning. If we prune 10% of the filters, we lose 1.4% in accuracy.

6.6.3 Energy Consumption and Latency

We now report energy consumption and latency measurements obtained from running inference on a Google Nexus 4 Android Smartphone. The energy is measured using a Monsoon power monitor. To calculate the amount of energy spent on inference, we separately measured the energy spent on loading data and idle power and deduct them from the total energy consumption.

Figure 6.5(b) shows the average energy consumption and latency running inference on one data sample. Note that the only difference between models targeting a different subset of classes (say a 2-class model and a 7-class model), is the last CCCP layer. But since the last layer only contributes to 0.05% of the total number of FLOPs, we ignore the differences and consider that the models take the same amount of energy and time to run. For 2-class classifiers, we can aggressively prune 50% of the filters while losing only 3% in accuracy. That brings 2.30× in energy savings and 1.45× in latency improvements. For 5-class classifiers, we can prune 30% of the filters while loosing 3% in accuracy (while still achieve higher accuracy than the original 10-class classifier). We will, in turn, get 1.63× in energy savings and 1.24× in latency improvements.

6.6.4 Different Pruning Criteria

Figure 6.6 compares different pruning criteria with the proposed impact based criterion. ”Magnitude” refers to pruning filters with the lowest absolute weights, a method used in [140], and ”random” refers to pruning filters randomly. Similarly, all the accurate measurements are the average of 10 repeated experiments. For a fair comparison, the 10 subsets of classes are the same for all these criteria. Apparently random pruning results in inferior performance. Our method can achieve better results when fewer filters are pruned, or when more classes are pruned, com-

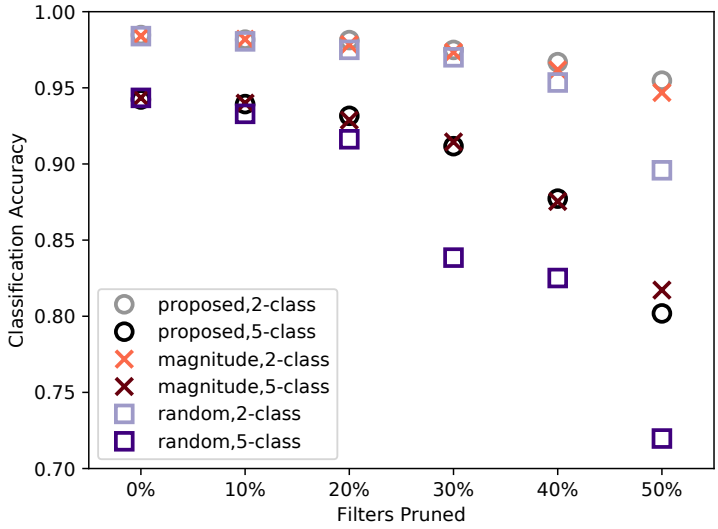


Figure 6.6: Comparison among different pruning criteria.

pared with the magnitude based method. One intuitive explanation would be that our impact based criteria can clearly identify the irrelevant filters. However, when more filters are pruned, we are inevitably touching relevant filters (filters that is important to at least one class). In that case, it is questionable whether our sum of impact criteria described earlier would be the best measure of relevance.

6.7 Conclusion

In this chapter, we propose a lightweight customization method that allows users to prune the unneeded classes and filters from CNNs. The method could be seen as a client-side counterpart to the existing pruning methods which do not address the customization needs from users. In the proposed method, we first identify and remove the low-impact filters. Then we use a different way to compensate for the pruned filters. The whole procedure can be efficiently run on-device with little overhead. Pruning unneeded classes not only brings about more targeted and accurate classification but also reduces computation costs. We observe a substantial reduction in energy consumption and latency from early experiments running the NIN model on CIFAR-10 dataset on

an off-the-shelf smartphone.

CHAPTER 7

Watermarking Deep Neural Networks for IP Protection

In the previous chapter, we have discussed the wide adoption of deep neural networks. Notwithstanding the fact that DNNs are widely used, the IP protection of DNNs is rarely discussed. Much like how we protect circuits and software, we need a mechanism to prove the authorship of a DNN in order to protect the IP [149] [150] [151] [152]. In the specific case of watermarking DNNs for embedded systems, the IP owners should be able to detect whether their libraries are used without proper authorization. The scenario is depicted in Figure 7.1. Since most embedded systems and applications allow very limited access, the watermarking method should support *black-box* detection. But unlike cloud-based MLaaS that usually charge users based on the number of queries made, there is no cost associated with querying embedded systems. Thus we do not need to limit the number of inputs in designing a rigorous detection framework. Only a handful of watermarking methods for DNNs have been proposed so far [153] [154] [2]. However, the existing methods either fail to meet the requirements in the embedded systems setting, incur an unnecessary cost in the proof of authorship, or they are susceptible to attacks of various forms.

To this end, we propose a new DNN watermarking framework that is suitable for watermarking DNNs for embedded systems. In our proposed framework, we train a watermarked DNN on both the original dataset and the dataset where each image is modified according to our signature. The modified images consist of a *trigger set* of unlimited size. The watermarked DNN should behave differently when a sample embedded with our signature is encountered. Otherwise, it should act normally with minimal loss of performance. Under the generic framework, we implement a simple version of the framework and empirically verify its performance against various criteria. Our approach has a number of benefits. First, it operates completely in a black-box manner. Only a set of test images are needed to verify the existence of the watermark, making the verification

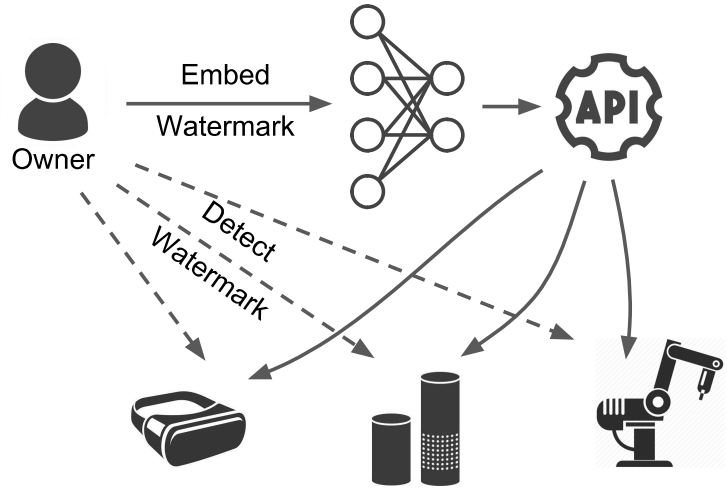


Figure 7.1: Watermarking DNNs that are intended for embedded devices.

process compatible with the embedded systems setting. Second, the process of proving authorship is straightforward and self-contained. Other than samples from the intended input space, we do require any other supplementary materials. It not only simplifies the process of the proof but also improves the robustness against attacks. Third, we are able to resist various forms of attacks that are effective on existing watermarking methods.

Further, we studied and compared different ways to produce trigger sets and improved existing methods. We propose a differential evolution-based framework to determine how *any* given trigger pattern should be added to the image such that false positive detections are reduced while the robustness of the watermark is maintained. With our framework, trigger pattern-based watermarking adds the model functionality to its equation, while still keeping ownership proofs simple.

The rest of the chapter is organized in the following fashion. We begin by surveying some of the existing watermarking techniques in Section 7.1. Section 7.2 outlines the general watermarking framework. We also discuss the criteria we use to evaluate a watermarking system. A minimal and straightforward example that implements our framework is given in Section 7.3. Section 7.4 describes techniques to further improve the watermark trigger set. We then proceed to evaluate the given example on models in Section 7.5.

7.1 Related Work

Watermarking has been an extensively studied subject for multimedia. There are also various pieces of work that discuss watermarking in the context of algorithm, software, and circuit design. In this section, we first introduce the aforementioned traditional watermarking approaches. Then we introduce a few recent works on watermarking DNNs.

7.1.1 Multimedia Watermarking

To enable discrete watermarking where watermarks are imperceptible, correlation-based watermark detection methods are used [155]. In this method, one has to compute the correlation between the image and a set of orthogonal *reference marks* in order to extract hidden watermarks. The correlation-based approach has been applied in the spatial domain and in a transform domain such as the wavelet domain [156]. We refer interested readers to the textbook for further information [157]. Our method described in Section 7.3 was partially inspired by the traditional multimedia watermarking methods. Instead of making small modifications to images that are not detectable by human eyes, we make small modifications to images that are not distinguishable by models without watermarks. Further, the way information is encoded in image watermarking can be used for DNN watermarking. Due to the page limits, we do not discuss the issue and resort to using a rather naive encoding method in Section 7.3 of this paper.

7.1.2 Circuit, Software, and Algorithm Watermarking

In the sense that circuits, software, and algorithms are functional entities, their corresponding watermarking approaches have a closer tie to DNN watermarking. For watermarking integrated circuits, Kahng *et al.* proposed to add additional constraints in the place and route procedure [150]. Oliveria proposed to create unique state transition graphs as a way to watermark sequential circuits [158]. In terms of software, the survey by Collberg *et al.* categorized watermarking methods into static ones and dynamic ones, where the former refers to embedding watermarks as strings in the code or binary, and the latter includes those triggered by specific inputs. In the case of algorithms,

Qu *et al.* proposed to add additional edges which force vertices to have the same color in the graph coloring problem solutions [149].

Many of the approaches mentioned above follow a common pattern. They first identify the logic and sequences of a system that can be a) altered without changing the functionality of the system, or b) extended to include additional functionality. Then they impose additional constraints that reflect the unique signature of the author to the logic and sequences to make them unique, thus proving the authorship. The idea can be applied to the problem of watermarking any machine learning problems. For classification problems, in particular, classifying an input to the right class is the natural constraint of a classifier. Extending the capabilities of the classifier to include additional inputs that are outside of the distribution of the natural input set can be thought of as an additional constraint imposed. We can embed our signature in the classifier's ability to classify those additional inputs.

7.1.3 DNN Watermarking

Due to their demands for computational power, high-quality dataset, and human expertise, DNNs naturally possess a high intellectual property value. Recently, many researchers are proposing methods to watermark DNNs to protect the rights of IP owners. Uchida *et al.* proposed to regularize the mean of weights such that a linear projection of that mean can be mapped to a signature [153]. To extract the signature, they require that one first examines the mean of the weights in certain layers of the model. Then one multiplies the mean weights with a given matrix to obtain the signature. The method has multiple drawbacks. An obvious one is that they require white-box access to the model, which renders the method unfeasible in our setting. In addition, the approach is vulnerable to the ghost signature and tampering attacks. Lastly, it is possible to transpose weight matrices to distort the original order of the weights, which is critical in this method. Similar to other DNN watermarking methods, we discuss the vulnerabilities in Section 7.2.3.

Various zero-bit black-box watermarking approaches have also been proposed. Le Merrer *et al.* proposed to fine-tune the model to behave correctly in the face of certain adversarial examples and use the correct behavior as the evidence for authorship [154]. In DeepSign [159], besides the

white-box watermarks, the authors also drew inputs from the statistically unused space to serve as the trigger set. Adi *et al.* proposed to assign labels to abstract images and train DNNs to classify them [2]. One clear drawback of all of the aforementioned approaches is the difficulty to associate abstract images with the author's identity. Inevitably, it leads to the watermark's vulnerability to ghost signature attacks. Only Adi *et al.* has an answer to the problem. Their answer is to use a complex cryptographic *commitment scheme* to bind the trigger set to the author's identity and safely store it in a third party. It is unclear to us the exact form of the commitment in practice, but it will unavoidably incur a lot of overhead to the proof of authorship.

The method from Zhang *et al.* is by far the closest to our method [3]. They proposed to add various patterns and symbols to the images to create trigger sets [3]. Their method mitigates the ownership proof problem slightly by allowing owners to add logs to the trigger set. But they did not address the false positive rate of watermarking. Regular models may also be triggered by accidentally classifying an image with a conspicuous symbol to a different class. By more and more trigger inputs for testing, the false positive rate of previous black-box watermarking methods may asymptotically approach zero. But judging from the absolute values, our method is among the lowest.

7.2 The General Method

In this section, we first describe our proposed approach for watermarking neural networks by hiding signatures in the training dataset. Then we discuss criteria for evaluation and the security of watermarking.

7.2.1 Watermark Embedding and Detection

Our general strategy is to map the author's signature to the modifications of a portion of the dataset. A DNN fine-tuned on that particular dataset (i.e. a *watermarked* DNN) will behave disproportionately differently than an otherwise trained DNN (a *regular* DNN) when it encounters data modified according to the author's signature. Such fine-tuning can be regarded as imposing additional con-

straints to the neural network that otherwise would not be present. The usual over-parameterization of DNN models ensures that there will be enough model capacity to tolerate such constraints [160].

Now we lay out the generic watermarking and signature verification procedure. To facilitate our description, we will use the standard "Alice and Bob" scenario, where Alice uses watermarking to protect her model and Bob tries to attack the watermark.

7.2.1.1 Generic Watermark Embedding Procedure

Alice wishes to protect some DNN. She first trains a fully functioning regular model without any additional constraints. Then she selects a portion of the dataset and adds certain modifications according to her signature. In this step, Alice should be aware of the behavior of the regular model when given the modified dataset. For example, the modifications could be *designed* to make the regular model behave in certain patterns. She should also show that such behavior generalizes to other regular models [1]. Then she fine-tunes the initial model (using the existing weights as initialization), possibly with both the original dataset and the modified dataset. The behavior of the fine-tuned watermarked model is disproportionately different from that of regular models. Note that Alice needs not to tell anyone which modifications she made to the dataset.

7.2.1.2 Generic Watermark Detection and Verification Procedure

To demonstrate that a DNN is watermarked Alice must draw a set of samples from the intended input space. She must demonstrate that both the original model and the watermarked model works reasonably well on the original samples. Then compares the behavior of the original model and the watermarked model on inputs that are modified. By demonstrating the extremely small probability of a regular model having the behavior corresponding to her signature, Alice can verify that her signature is present. Note that Alice has to reveal her signature and how it leads to the modification of the dataset in order to prove her authorship.

[1] In practice, we relax the requirement to empirically showing that different models would behave similarly.

Criterion	Explanation
Effectiveness	The watermarking method can prove ownership in different DNNs and datasets
Fidelity	The watermark does not significantly affect the performance of the model
False PositiveRate	The watermark detection method will not be triggered when there is no watermark
Robustness	The watermark is robust against attacks

Table 7.1: Criteria for evaluating DNN watermarks.

7.2.2 Criteria for Evaluation

We borrow existing conventions in judging watermarking systems and discuss them in the context of DNNs. In particular, we'd like to discuss *effectiveness* and *fidelity* with regard to embedding watermarks, and *false positive rate* with regard to decoding watermarks [157]. In addition, the *security* aspect should be considered, we which will discuss in more details in the next subsection.

7.2.2.1 Effectiveness

Effectiveness refers to the success rate of watermark embedding. In the context DNN watermarking, we need to make sure that watermarks can be embedded and extracted regardless of inputs and model architecture.

7.2.2.2 Fidelity

In image watermarking, fidelity represents the perceptual similarity between the images before and after adding watermarks. In our context, fidelity represents the performance of DNN on the test set without trigger patterns embedded.

7.2.2.3 False Positive Rate

In image watermarking, the false positive rate refers to the probability of detecting a watermark is detected from images that do not contain it. In our context, a false positive may refer to a watermarked DNN exhibit the desired behavior for a trigger sample when given a regular sample.

7.2.3 Security and Threat Model

In this subsection, we introduce our notion of robustness by defining our threat model. We assume that Bob has white-box access to the model, but does not have access to the training set. Instead, Bob has access to some proprietary test data (i.e. a subset of test set). We argue that proprietary data is one of the most important competitive advantages of Alice, and an IP pirate Bob by no means should have access to it. Otherwise, with the training data and the model, he might as well train a new model on his own, especially when he has the ability to carry out sophisticated attacks such as fine-tuning. On the other hand, it is a reasonable assumption that the attacker may have white-box access to the model architecture and parameters. In the case of cloud ML service, Bob can be a malicious service platform. In the case of software ML libraries, Bob can be a hacker. In both cases, Bob would have the full white-box access to the model. We also assume that Alice only has black-box access to the model. In addition, Alice will have direct access to input to the model. There are no preprocessing stages between Alice's input and the input of the model. We now layout two possible attacks that are feasible within our threat model.

Finding Ghost Signatures. Bob wishes to claim Alice's model as his own. He knows that Alice's model is watermarked and proceeds to claim that the model also contains his own watermark. Bob thus attempts to find a ghost signature, namely, a fake signature that coincidentally makes model behave like it is real. In the framework proposed in [153], this can easily be done by finding Bob's own linear projection that leads to his desired output by solving a system of linear equations. In the approach proposed by Adi *et al.* [2] since the space of *any* abstract images are so large, one can easily find another set of images that coincides with another signature. Genetic algorithms are a good candidate for finding such images [161]. In our approach, however, Bob has to find *one* way to modify *any* input such that it coincides with the behavior of the watermarked model. Since

both the modification and the behavior of the model are produced using Alice's signature, Bob has to reverse the cryptographically secure functions or brutal force.

Tampering. Bob knows Alice has embedded her watermark in the model. He doesn't know how to find it but wishes to remove Alice's signature by tampering with the model. With the approach proposed in [153], Bob can achieve it by switching the position of neurons, as DNNs are linear within a layer. In order for the linear projection to still produce the desired results, the projection matrix has to be changed in order to map the coefficients to the original model weights. But Alice would not have known that Bob switched the position of neurons. We do not have this problem in our proposed framework because we prove the ownership based on the output. We do not consider other types of tampering of DNNs as valid, because neural networks are known to have *fragile co-adapted features* between layers [31]. Changing the weights on one layer, albeit a small amount, may cause significant malfunctioning of whole modal.

Fine-tune. With some test data and the model, Bob may fine-tune the model to produce a slightly different version of it. That is called the *fine-tune* attack. After the fine-tune attack, Alice's watermark should still exist. Some researchers also discussed overwrite attacks, where Bob tries to embed his own watermark using the same procedure on Alice's model. It is indeed a very reasonable attack scenario.

7.2.4 Notation

Let $\mathcal{D}^{train}$ and $\mathcal{D}^{test}$ represent the training set and the test set respectively, and X_i be a sample with label y_i. In this example, X will be an image of shape $H \times W \times C$ (e.g. $32 \times 32 \times 3$), and $y \in \{1, 2, \ldots, K\}$ will be an integer representing a class. Alice would like to embed a trigger pattern m (of the same dimension as X) representing her signature into X and also map the label to a different one as our way to indicate the watermark detection. m can have different *magnitudes*, denoted by α. The resulting sample and label are denoted as $\phi_X(X)$ and $\phi_y(y)$. We sometimes refer to a sample embedded with the trigger pattern as a trigger sample or trigger input for short, and a sample without one as a "regular sample". We use subscripts on the dataset symbol to indicate the embedding of a trigger pattern. $\mathcal{D}^{train}_{\alpha m}$ represents a trigger pattern m of magnitude α is embedded

to all of the samples in $\mathcal{D}^{train}$. A DNN is denoted as f and a watermarked DNN as $f^{\texttt{WMK}}$. In the case of classifying trigger samples, we say $\phi_X(X_i)$ is correctly classified if $f^{\texttt{WMK}}(\phi_X(X_i)) = \phi_y(y_i)$. So if when $\phi_X(X_i)$ is classified to its original label y_i, the classification is incorrect.

We use ϵ to represent the classification error rate of the models. We refer to a model with the classification error rate of at most ϵ as a ϵ-accurate model. Obvious it only makes sense to define such a model over its performance on *any* inputs drawn from the input distribution. Thus the ϵ in this paper refers to the test accuracy of models. For detecting the watermarks, we will use a set of N test samples. We allow at most Δ errors in the classification to confirm the existence of our watermark.

7.2.5 Proof of Ownership

A watermarking method's ability to prove ownership mainly relies on its low false positive rate. The ownership is established based on the fact that $\Pr$ is disproportionately small for a non-watermarked neural network. If a watermarking method incurs a high false positive rate (a high ρ for non-watermarked models), then $\Pr$ is no longer small and that the proof of ownership will be inconclusive at best.

After adding the trigger pattern m to the dataset, a watermarked model $f^{\texttt{WMK}}$ can achieve an error rate of ϵ_w error rate while a regular unwatermarked model f have an error rate of ϵ_r. Note that we define accuracy in the latter case to be classifying a trigger input $\phi_X(X)$ to the remapped label $\phi_y(y)$. In that case, $\epsilon_w \ll \epsilon_r$. Suppose we have N test samples $\{X_1, X_2, \ldots, X_N\}$ with labels $\{y_1, y_2, \ldots, y_N\}$. The probability of classifying X_i to y_i is $1 - \epsilon_o$. The probability of classifying $\phi_X(X)$ to $\phi_y(y)$ is $1 - \epsilon_w$ and $1 - \epsilon_r$ respectively.

The probability for $f^{\texttt{WMK}}$ to classify all N test samples correctly is $(1 - \epsilon_o)^N(1 - \epsilon_w)^N$. The probability for f to classify all N test samples correctly is $(1 - \epsilon_o)^N(1 - \epsilon_r)^N$. Apparently, even a well-trained model cannot guarantee to classify all samples correctly if N is large. Thus we lower the requirement to allow at most Δ errors. The probability for $f^{\texttt{WMK}}$ to classify at least $N - \Delta$

samples correctly (for samples both with and without trigger patterns) is given by the following:

$$(\sum_{\delta=0}^{\Delta} \binom{N}{\delta} (1-\epsilon_w)^{N-\delta} \epsilon_w^{\delta}) \tag{7.1}$$

Similarly, we have the probability for f:

$$(\sum_{\delta=0}^{\Delta} \binom{N}{\delta} (1-\epsilon_r)^{N-\delta} \epsilon_r^{\delta})$$

7.3 Watermarking using Imperceptible Trigger Patterns

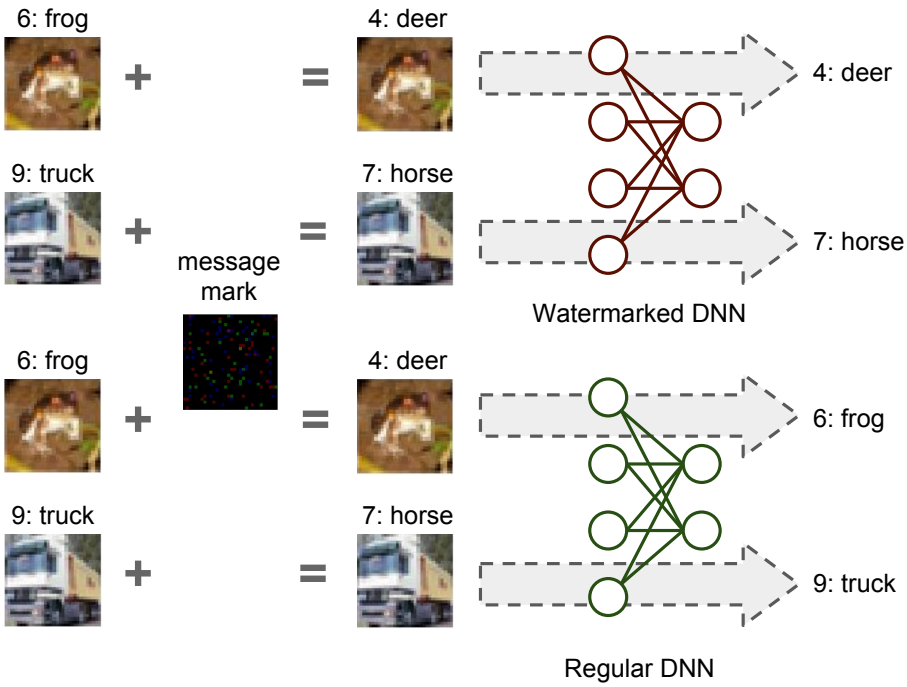


Figure 7.2: Overview of the proposed example watermarking technique on DNN based image classifiers.

In this section, we show an exemplary approach to watermarking DNN image classifiers. Our approach is by no means the best under the proposed framework. In designing this approach, we put straightforwardness as our top priority. That said, a high-level description of the approach is as follows. Alice creates a *trigger pattern* of the same size of the input images using her signature and embedded it into images. The trigger pattern is so imperceptible that regular DNNs will classify the image to its true class regardless of whether the trigger pattern has been added. A DNN

watermarked by Alice, however, is able to recognize images embedded with the trigger pattern and classify them to a different class than the original true class. Figure 7.2 depicts the idea.

Now we explain the method in more details. We start by introducing the procedure for both embedding a watermark and detecting a watermark. Then we break down the embedding procedure and present them respectively.

7.3.1 Workflow

To embed a watermark, the model owner Alice would need to do the following:

1. Create an n-bit signature.
2. Create trigger pattern m of suitable magnitude α based on the n-bit signature
3. Calculate the mapping for class labels based on the n-bit signature
4. Fine-tune an existing model f. While fine-tuning, we use, with an equal probability, both the original dataset and the dataset containing trigger images and remapped labels.

After the training converges, the resulting model will classify an input image added with trigger patterns to a different label according to the mapping.

To detect a watermark, the model owner Alice would need to do the following:

1. Take a set of images.
2. Create trigger pattern m of magnitude α based on the n-bit signature. Add the trigger pattern to all the images.
3. Calculate the mapping for class labels based on the n-bit signature
4. Take the watermarked model $f^{\texttt{WMK}}$, and run classification on images both with and without the trigger pattern m.

If the $f^{\texttt{WMK}}$ classifies original images to the correct label, and trigger images to the correctly mapped label within a certain margin of error, then we show that $f^{\texttt{WMK}}$ is indeed our watermarked model.

7.3.2 Trigger Pattern Creation

In this approach, Alice has an n-bit signature which will be embedded into n pixels in the images. Alice generates the signature by hashing a message that proves her as the author. The next step is to use the signature as the key to a pseudorandom random permutation (PRP) that remap label k to any of the remaining $K - 1$ labels. This is referred to as the mapping of the classes $\phi_y(y)$. After that, the signature is used as the key to select the location of the n pixels. The signature will be directly added to the pixels. A positive one indicates a "1" in the signature, and a negative one indicates a "0" in the signature. The resulting pattern is essentially the trigger pattern m. The procedure of embedding can be described as $\phi_X(X) = X + \alpha m$ where α is referred to as the magnitude of the trigger pattern.

7.3.3 Optimal Trigger Pattern Magnitude

Recall that Alice wants to add m to test images so that regular models would still correct classify them but the watermarked model would not. Given a signature, if the magnitude is too large, then regular models would classify them incorrectly. If a watermark is too small, then the training won't be able to converge on both the embedded dataset and the regular dataset. In this subsection, we discuss the effect of message length (number of bits) and watermark magnitude on the performance of regular models. Further, we show an algorithm to search for the optimal magnitude given the length of m.

Figure 7.3 shows a study based on a VGG16 model [162] trained on the CIFAR10 dataset [116]. We add trigger patterns of different length and magnitude to the training set. Then we test the classification accuracy of the model which originally achieves 99.97% accuracy on the training set. The x-axis shows the length of the message in the number of bits, and the y-axis shows the magnitude normalized by the standard deviation of all the pixels in the dataset. Each accuracy drop value is the mean of 5 independent experiments. As expected, shorter messages can tolerate larger magnitude. Given a certain length m, it is natural that larger magnitudes are easier for a watermarked model to recognize. Empirically we do find that if the trigger pattern is too small, the watermarked model won't be able to converge in training. Therefore, given a certain message

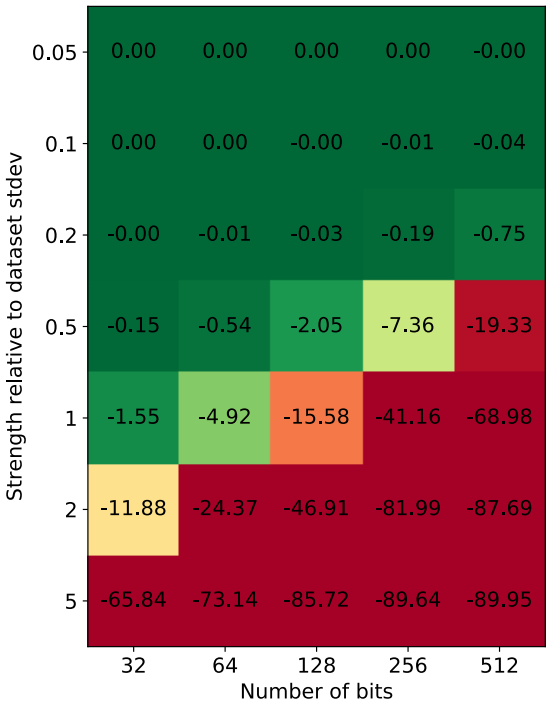


Figure 7.3: Classification accuracy drop of a regular VGG16 model when trigger patterns of different lengths and magnitudes are added to the CIFAR10 training set.

length, it would be desired to find the largest possible magnitude α that is not detectable by a regular model.

We now lay out our binary search based algorithm to search for the optimal magnitude α_{opt}. We define α_{opt} as the largest possible magnitude to be added to the dataset such that the classification accuracy of the regular model f drops by no more than ΔA. Algorithm 8 describes the algorithm in more details.

Algorithm 8 Get Optimal Trigger Pattern Magnitude Using Binary Search

Input: regular DNN f, trigger pattern m, training set $\mathcal{D}^{train}$, resolution $\Delta\alpha$, accuracy drop ΔA

Output: optimal trigger pattern α_{opt}

Initialize α_{min} and α_{max}
$A_0 \leftarrow$ classification accuracy of f on $\mathcal{D}^{train}$
$\alpha' \leftarrow \alpha_{min}$, $\alpha \leftarrow \alpha_{max}$
while $\alpha' - \alpha > \Delta\alpha$ **do**
    $A \leftarrow$ classification accuracy of f on $\mathcal{D}^{train}_{\alpha m}$
    if $A_0 - A < \Delta A$ **then**
        $\alpha' \leftarrow \alpha$, $\alpha_{min} \leftarrow \alpha$
        $\alpha_{opt} \leftarrow \alpha$
        $\alpha \leftarrow \frac{\alpha+\alpha_{max}}{2}$
    else
        $\alpha' \leftarrow \alpha$, $\alpha_{max} \leftarrow \alpha$
        $\alpha \leftarrow \frac{\alpha+\alpha_{min}}{2}$
    end if
end while
return α_{opt}

(a) (b) (c)

Figure 7.4: Examples of trigger inputs. (a) a random out-of-distrbution image [2], (b) a regular image with a logo [3], (c) a regular image with a color-coded key not perceptible by the eye [4].

7.3.4 Training Model

Once we obtain the desired magnitude α and trigger pattern m, we are ready for the final watermarked model $f^{\texttt{WMK}}$. To obtain the watermarked model $f^{\texttt{WMK}}$, we start by fine-tuning the existing model f. With a 50% chance, we embed the trigger pattern to any of the training samples during fine-tuning.

7.4 Improving the Trigger Patterns

In this section, we provide the definition of trigger set generation. Next, we provide a high-level overview of the DE framework. Then we delve deeper to provide some algorithmic details that are crucial to the convergence of DE.

7.4.1 Motivation

Based on the characteristics of their trigger sets, existing black-box watermarking methods can be split into two categories. The first category of methods curates a finite set of special trigger images. The special images can be completely random [2], samples derived from unused hidden space [163], or adversarial examples [154]. Another category of methods maintain *trigger patterns* and add them to natural input images to create trigger sets. The trigger patterns are usually meaningful patterns that can serve as a proof of the owner's identity, such as the logos [3] and color-coded

keys described in this chapter. Some sample trigger images are shown in Figure 7.4.

The motivation for designing the trigger sets are different between those two categories of methods. The first is focused on the functionality of the model. They aim to create trigger sets such that watermarking is as orthogonal to the normal functionality of the model as possible. The second, on the other hand, is more focused on the watermarking extraction procedure. Associating the owner's identity with the trigger set makes detection and proof of ownership much more straightforward. The evident drawback of the first category is the difficulty to establish a connection between the trigger set and the owner. To solve that problem, researchers went as far as to use complex cryptographic tools [2]. Further, the limited size of the trigger set weakens the proof of ownership. The drawback of the second category lies in an inevitable trade-off between the robustness of the watermark and potential of false positive watermark detection. If a trigger pattern is too prominent, then it risks triggering false positives in other neural networks. On the other, if a pattern is too inconspicuous, it may be easily removed during fine-tune attacks.

7.4.2 Problem Definition

A DNN for classification is a function $f : \mathbb{R}^d \rightarrow [0,1]^L$. Given an input $X \in \mathbb{R}^d$, it is desired that the function classifies it correctly to its label y, $f(X) = y$.

A pattern $P \in \mathbb{R}^d$ has the same dimensions as X but is much more sparse. In its image form, P's non-zero entries can be considered as a set of K pixels with explicitly designed values and coordinate $\{v_k, c_k^x, c_k^y\}^K$. P is tightly coupled with the identity of the model owner. And the absolute and relative coordinates of the pixels may or may not contribute to P's ability to carry information. The pattern can be embedded on *any* input X from the intended data distribution to convert it into a trigger input through a function $g(X, P)$ [2]. A watermarked DNN will be trained to classify $g(X, P)$ to $y_i' \neq y_i$. The fact that a DNN model classifies the trigger inputs disproportionally correctly can serve as a unique proof of the owner's identity.

We consider two alternative approaches to create trigger patterns. In the method proposed by Guo *et al.* (shown in Figure 7.4(c)), a color-coded key serves as the trigger pattern [4]. They embed

[2]For convenience, we sometimes write the function as $g(X_i, \{v_k, c_k^x, c_k^y\}^K)$.

the pattern by offsetting the pixel values of the input, $g(X, P) = X + P$. Since the information is mainly ingrained in the pixel values, we consider the pixel locations c_k^x, c_k^y to be flexible. We use `Key` throughout the chapter to denote this trigger pattern. The second approach we consider is proposed by Zhang *et al.*, and shown in Figure 7.4(b) [3]. The information is obviously contained in the geometrical shape of the logo and the pixels have to remain in a relatively fixed to each other. Thus its location can be represented by its top left corner c^x, c^y. The author did not explicitly say how they embed the logo, but we interpret it as blending with the input, $g(X, P) = (1-\alpha)X+\alpha P$. We denote the second type of trigger pattern as `Logo`.

Our main goal is to find the $P = \{c_k^x, c_k^y\}^K$ such that the probability of a non-watermarked DNN f_0 classifying a trigger input to its original labels is maximized. The main motivation behind the goal is to minimize false positive watermark detection. Empirically, given dataset $\mathcal{D}$, then the goal can be expressed as follows.

$$\underset{\{c_k^x, c_k^y\}^K}{\operatorname{argmax}} |\{X | f_0(g(X, \{v_k, c_k^x, c_k^y\}^K)) = y, X, y \in \mathcal{D}\}| \tag{7.2}$$

We have found that larger v_k leads to more robust watermarking, although it leads to higher false positives. In the `Key`-related experiments, v_k is given. But we can also integrate v_k into the optimization landscape as follows. The `Logo`-related experiments use this objective function.

$$\begin{aligned}\underset{\{v_k, c_k^x, c_k^y\}^K}{\operatorname{argmax}} |\{X | f_0(g(X, \{v_k, c_k^x, c_k^y\}^K)) = y,\\ X, y \in \mathcal{D}\}| + \delta \sum_k v_k\end{aligned} \tag{7.3}$$

7.4.3 Differential Evolution

To find the pattern P, the first methods that came up to us were the gradient-based methods commonly used for finding adversarial samples [164] [165] [166]. A key difference between our problem and theirs is that our pattern P is universal. Therefore, finding the gradient of individual inputs hardly helps our situation. The family of evolutionary algorithms is among the most promi-

Algorithm 9 Differential Evolution

Input: dataset $\mathcal{D}$, non-watermarked DNN model f_0, popularion N, number of generations G

Output: best candidate P after evolution

Randomly intialize, ith candidate $P_i = \{v_{ik}, c_{ik}^x, c_{ik}^y\}^K$
for generation $g = 1, \ldots, G$ **do**
    for each candidate P_i **do**
        Randomly pick $0 \leq j, k, l < N$ where $j \neq k \neq l$
        $P_i' \leftarrow$ EVOLVE(P_j, P_k, P_l)
        if FITNESS($P_i', f_0, \mathcal{D}$) $>$ FITNESS($P_i, f_0, \mathcal{D}$) **then**
            $P_i \leftarrow P_i'$
        end if
    end for
end for
return argmax_{P_i} FITNESS$(P_i, f_0, \mathcal{D})$

nent non-gradient-based optimization methods. We initially relied on the generic evolutionary algorithm (EA), but we were unable to find a reasonable set of parameters to make the algorithm converge. That is when differential evolution (DE) presented itself as an alternative.

DE is a meta-heuristic search algorithm that optimizes a given objective by evolving a population of candidates in parallel [167]. It follows the concept of generic EAs where a population of candidates evolves, and the candidates that are fittest will survive in each iteration. However, DE is simpler and it is known to facilitate faster convergence to the global optimum. Instead of using mutations and crossover between two parents, candidates DE evolve over a triplet. A new candidate is created by adding a weighted difference between two candidates to the third.

$$P = P_0 + F \times (P_1 - P_2) \tag{7.4}$$

Algorithm 9 presents the high-level procedure of using DE to solve our problem. The main idea is to generate 1) new pixel coordinates 2) pixel values of P using the differential variation operation described in Equation 7.4. The FITNESS function can be either of the two objective functions described in Section 7.4.2. The EVOLVE function, on the other hand, is more complex.

We describe more details of the function in the next subsection.

7.4.4 Optimization for DE

We use two different variants of EVOLVE functions for the two existing trigger patterns, `Key` and `Logo`. For `Logo`, the EVOLVE function is straightforward. We use the top left pixel of the logo as the anchor, and each candidate can be represented by a simple triplet (v, c^x, c^y). We use DE to evolve and select the location of the logo as well as its pixel values. We use a different approach for `Key`. Since pixel locations are al flexible, the candidate will be an array of K tuples $\{c_k^x, c_k^y\}^K$. When we evolve using three candidates, each with K pixels, which pixels should pair up and evolve becomes an important question. If pixels are randomly paired up, it is likely that the pixels will engage in a Brownian motion-like movement across different generations. Consequently, as we empirical show later, the evolution will not converge. If our goal is to evolve the pixels into optimal locations, then it makes sense to induce the evolution in such a way that pixels nearest to an optimal location will move toward that location. To that end, we propose an algorithm to pair closest pixels together to evolve. The most efficient implementation is to store all pairwise distances in heaps and always pair available pixels with the smallest distances. Algorithm 10 describes implementation in more details. The time complexity of the algorithm is $O(K^2 log(K)$, where K is the number of pixels.

Figure 7.5 shows the fitness of the best candidate over the different generations. The FITNESS function is simply the accuracy of the subset. The proposed method, closest triplet evolve function, converged much faster than the evolve function where pixels are randomly paired together. In fact in the latter case, the fitness plateau at around 0.89 and it is unclear whether it will converge at all.

7.5 Evaluation

In this section, we present the performance evaluation of our methods. We first discuss the original imperceptible trigger pattern method. Then we discuss the DE-based optimization of the trigger pattern.

Algorithm 10 Evolve with Closest Triplet

Input: 3 candidates, each containing K pixel coordinates: $\{c^x_{1k}, c^y_{1k}\}^K$, $\{c^x_{2k}, c^y_{2k}\}^K$, $\{c^x_{3k}, c^y_{3k}\}^K$, differential weight F

Output: pixel coordinates of a new candidate, $\{c^x_k, c^y_k\}^K$

procedure PAIR($\{c^x_{1k}, c^y_{1k}\}^K$, $\{c^x_{2k}, c^y_{2k}\}^K$)
    Initialize $heap$
    for each $\{c^x_{1i}, c^y_{1i}\}$ **do** ▷ Calculate pairwise distances
        for each $\{c^x_{2j}, c^y_{2j}\}$ **do**
            Push distance($\{c^x_{1i}, c^y_{1i}\}$, $\{c^x_{2j}, c^y_{2j}\}$) in $heap$
        end for
    end for
    while $heap$ is not empty **do** ▷ Pair by distance
        distance, $\{c^x_{1i}, c^y_{1i}\}$, $\{c^x_{2j}, c^y_{2j}\} \leftarrow$ Pop from $heap$
        if Neither $\{c^x_{1i}, c^y_{1i}\}$ or $\{c^x_{2j}, c^y_{2j}\}$ is paired **then**
            Pair ($\{c^x_{1i}, c^y_{1i}\}$ with $\{c^x_{2j}, c^y_{2j}\}$)
        end if
    end while
end procedure
PAIR($\{c^x_{1k}, c^y_{1k}\}^K$, $\{c^x_{2k}, c^y_{2k}\}^K$)
PAIR($\{c^x_{1k}, c^y_{1k}\}^K$, $\{c^x_{3k}, c^y_{3k}\}^K$)
for each $\{c^x_{2i}, c^y_{2i}\}^K$ $\{c^x_{3j}, c^y_{3j}\}^K$ paired with $\{c^x_{1k}, c^y_{1k}\}^K$ **do**
    $c^x_k \leftarrow c^x_{1k} + F \times (c^x_{2i} - c^x_{3j})$
    $c^y_k \leftarrow c^y_{1k} + F \times (c^y_{2i} - c^y_{3j})$
end for
return $\{c^x_k, c^y_k\}^K$

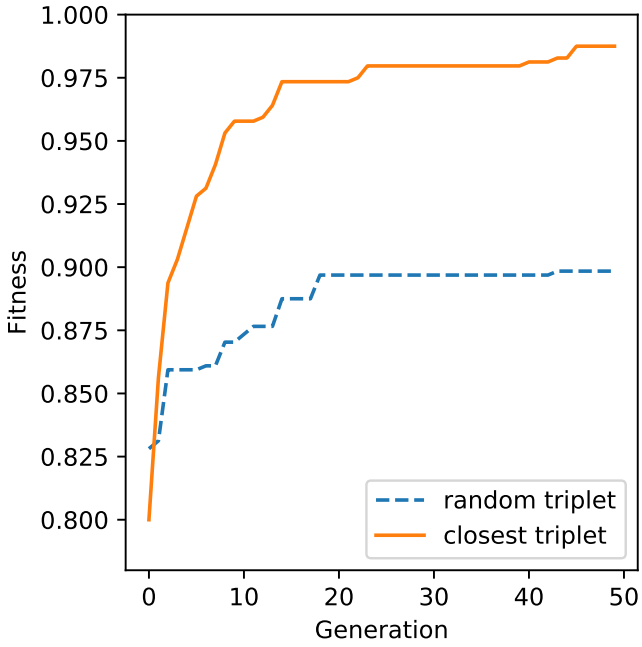


Figure 7.5: Best fitness of the population across the generations.

7.5.1 Imperceptible Trigger Patterns

In Section 7.2, we discussed the evaluation metrics for DNN watermarking. We followed the metrics and evaluated our method on various neural network models and datasets. The results are presented in this section.

7.5.1.1 Experimental Setup

We test our method on two widely used datasets: the MNIST dataset which contains 60,000 28×28 black and white images on hand written digits [168], and the CIFAR-10 dataset which contains 60,000 32×32 color images in 10 classes [116]. We trained and tested LeNet [168] on MNIST, and VGG-16 [162], ResNet-50 [131] and DenseNet-121 [169] on CIFAR-10.

Before we move on to individual metrics, we'd like to discuss the generalizability of the behavior of one regular DNN to other regular DNNs. Table 7.2 evaluates the classification accuracy of a regular and a watermarked model on regular datasets and on datasets embedded with trigger pattern. Note the datasets are embedded with the same 128-bit trigger pattern, and the magnitude

Dataset	Model	$\mathcal{D}^{train}$	$\mathcal{D}^{test}$	$\mathcal{D}^{train}_{\alpha m}$
MNIST	LeNet	99.17	98.99	0.10 (99.17) [1]
	LeNet$^{\texttt{WMK}}$	98.41	98.48	98.38 (0.20)
CIFAR-10	VGG	99.97	93.07	0.0060 (99.93)
	VGG$^{\texttt{WMK}}$	99.96	92.86	99.94 (0.0020)
	ResNet	100	94.53	0.022 (99.75)
	ResNet$^{\texttt{WMK}}$	99.99	94.25	99.98 (0)
	DenseNet	100	94.73	0.022 (99.76)
	DenseNet$^{\texttt{WMK}}$	99.98	94.23	99.97 (0.0080)

Table 7.2: Performance of the example watermarking method on different models and datasets. The classification results are obtained from regular training set ($\mathcal{D}^{train}$), test set ($\mathcal{D}^{test}$) and training set embedded with trigger pattern ($\mathcal{D}^{train}_{\alpha m}$).

[1] The accuracy is based on remapped labels $\phi_y(y)$ after adding trigger pattern. The value in parentheses gives the percentage that the predicted class matches the true label y.

of which is obtained using our Algorithm 8 using VGG16 and target accuracy drop of 0.5%. Note that for the classification accuracy of $\mathcal{D}^{train}_{\alpha m}$, the accuracy is calculated using the $\phi_y(y)$ as the correct label. The classification accuracy of the regular model on the embedded samples are below 0.1 % in all three CIFAR-10 models. We see this as the evidence that the behavior of the one regular DNN generalizes to other regular DNNs. But we are yet to come up with methods to estimate the discrepancies among different models under different conditions.

7.5.1.2 Effectiveness

Effectiveness indicates the success rate of the watermarking method when applied to different models and different problems. Table 7.2 shows the experimental results. The watermarked models that we present are all able to fit the training set embedded with trigger patterns. The LeNet model might seem to lag behind in the accuracy numbers. It is due to the huge model capacity difference between LeNet and the rest of the models. LeNet only has 2 convolution layers with 6 and 16

Model	Train	Train w/ Marks
VGG	99.97	0.018 ± 0.065 (99.63 ± 0.95)
VGG$^{\text{WMK}}$s	99.89 ± 0.08	99.87 ± 0.10 (0.032 ± 0.078)
Model	Test	Test w/ Marks
VGG	93.07	0.85 ± 0.25 (92.50 ± 1.81)
VGG$^{\text{WMK}}$s	92.32 ± 0.39	92.20 ± 0.67 (0.83 ± 0.24)

Table 7.3: Confidence intervals of classification accuracy of watermarked VGG16 models on CIFAR-10 obtained from 5 watermarking experiments.

channels respectively, in contrast to the hundreds of channel and tens of layers present in the rest of the models.

Table 7.3 shows that the effectiveness is consistent if we repeat the experiment multiple times using different trigger patterns of the same strength. We achieved an accuracy of 99.87$\pm$ 0.10% on the training sets with different trigger patterns $\mathcal{D}^{train}_{\alpha m_k}$. We could also achieve an accuracy of 92.20$\pm$ 0.67% on test sets embedded with signatures. The relatively narrow confidence interval shows that success persists across different sets of experiments. In Section 7.2.5, we described a form of "ownership test". We further analyze the aforementioned results later in Section 7.5.1.4 in the context of that test.

7.5.1.3 Fidelity

Fidelity is measured by the performance on the regular test set. In our experiment, all of the accuracy drops of a watermarked DNN regular on test sets are within 1%. In the best case, we achieve a drop of only 0.23%, as shown in Table 7.2.

In Table 7.3, we show that the the the watermarked models can achieve a comparable accuracy on the test set $\mathcal{D}^{test}$. What is more, after adding trigger patterns to the test set to create $\mathcal{D}^{test}_{\alpha m}$, the watermarked model performs equally well. On the contrary, a regular model will still classify the images to their original class labels. It shows that a) by creating $\mathcal{D}^{train}_{\alpha m}$, we successfully carved areas from the input space that are not previously occupied by any of the natural input data and

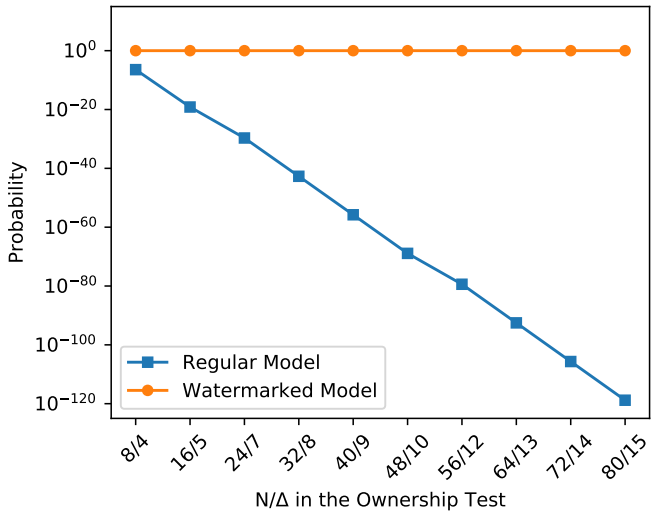


Figure 7.6: Probability of passing the ownership test with different test sizes and margins of error.

designated those areas for the purpose of watermarking; b) the neural network is able to generalize what it learns about watermarking and apply it to data that it has not seen before ($\mathcal{D}_{\alpha m}^{test}$); and c) the additional capability has negligible on the model's performance on regular inputs, thus the fidelity requirement is met.

7.5.1.4 False Positive Rate

This method is explicitly designed for low false positive rates. The false positive rates can be evaluated using the watermarked models' performance on regular training sets and test sets. Both Table 7.2 and Table 7.3 show accuracy close to that of a regular model. We take this result as an indication of a low false positive rate of our watermarking technique.

For a strong proof of ownership, we need to show the probability of f^{WMK} and f exhibiting the expected behavior on a set of test input. We will conduct the ownership test described in Section 7.2.5 on a set of $N = 32$ test images with the maximum number of error $\Delta = 6$. Based on the classification accuracy on test sets in Table 7.3, the probability of f^{WMK} passing the test is 0.990, while the probability of f passing the test is only 1.26×10^{-48}. If we tolerate more errors by setting $\Delta = 8$, the probabilities become 0.999 and 1.99×10^{-43} respectively. Figure 7.6 shows the huge disparity. We selected a set of Ns and the corresponding Δs such that we have a probability

	m_1	m_2	m_3	m_4	m_5
VGG$^{\text{WMK}}$ #1	99.84	0.23	0.19	0.92	0.23
VGG$^{\text{WMK}}$ #2	0.54	99.91	10.36	0.37	2.29
VGG$^{\text{WMK}}$ #3	4.04	9.19	99.45	5.48	2.97
VGG$^{\text{WMK}}$ #4	1.74	0.41	5.86	99.88	1.16
VGG$^{\text{WMK}}$ #5	0.56	0.33	0.20	0.29	99.94

Table 7.4: Classification accuracy of VGG models on CIFAR-10 training sets embedded with different trigger patterns. The model VGG$^{\text{WMK}}$#k is trained on the training set embedded with m_k.

of at least 0.999 to pass the ownership test. As we use a larger and larger trigger set for the test, the compounding effect weights in. When we have a set that's large enough, there will have an astronomically small probability for false positive triggers.

7.5.1.5 Security

Defending Against Tampering. Since our approach uses black-box based detection, switching the orders of channels would not affect the validity of our watermark. The watermark is deeply embedded in the fundamental functionality of the model. Any other manual adjustment will either a) damage the watermark as well as the classification ability of the model, b) has little impact on both. Thus, given our threat model, we consider our approach robust against tampering.

Defending Against Ghost Signature Attacks. Note that we made sure the uniqueness of our class mapping by adopting a PRG with our signature as the key. There is a probability of $\frac{1}{(K-1)^K}$ for an attacker Bob. In the case of the CIFAR-10 dataset, the probability is 2.87×10^{-10}. In the extremely rare case where the attacker Bob happens to be able to find his signature that produces the same class mapping in our example, then we need to check the probability of our model classifying samples with Bob's signatures correctly. If we do, then Bob may be able to find a ghost signature. Table 7.4 shows classification accuracy obtained from DNN models trained with different trigger patterns of the same length and strength. As we can see, the worst case happens when model VGG$^{\text{WMK}}$ #2 is trying to classify trigger pattern m_3, which achieved an accuracy of

10.36%. A possible cause of the reason might be the similarity between m_2 and m_3, as VGG$^{\texttt{WMK}}$ #3 also classifies samples with m_2 with a pretty high accuracy. Yet, even if we consider the worst case, with $N = 32$ and $\Delta = 6$ (see Section 7.5.1.2), Bob still only stands a chance of 3.89×10^{-20}. It is a small enough probability to be considered a successful defense. One of the key assumptions is that we allow repetitively feeding inputs. The more inputs we compute, the bigger the gap will be between the watermarked model and the regular model. Since this is only an example implementation, we used the simplest possible way to express the uniqueness. We could also resort to more sophisticated methods, such as force the class scores to some distribution that encodes more information.

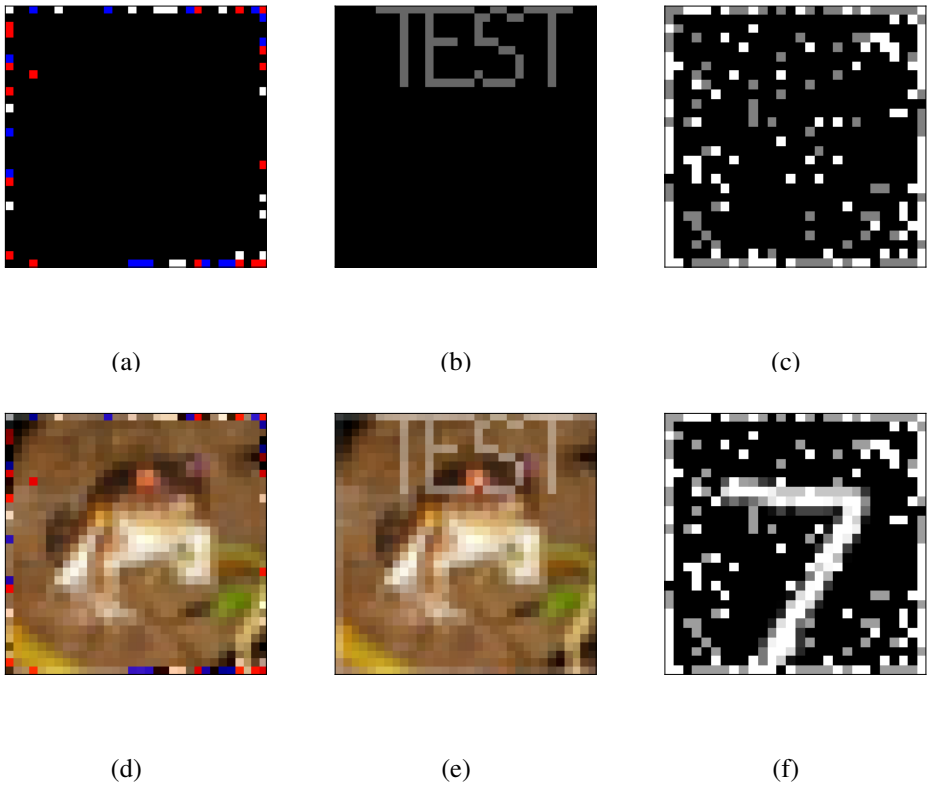
7.5.2 Improved Trigger Patterns

In this subsection, we report the performance evaluations of the DE-based optimization. We first describe the implementation details of the watermarking procedure and the DE algorithm. Then we evaluate the effectiveness, fidelity, false positive rate and robustness of the watermark in following subsections. Since neither `Logo` nor `Key` is an original idea from this chapter, we omit many repetitive experiments for brevity. The key is to demonstrate the ability of our DE algorithm to reduce false positive while maintaining the robustness of the watermark. It is worth noting that *all* of our trigger sets are built from test data that has not been used during training.

7.5.2.1 Differential Evolution

We applied our DE-based approach on both `Logo` and `Key` trigger pattern generation. We use Equation 7.3 as the fitness function for `Logo`, where both pixel locations and values are optimized. The weight β is set such that a maximum v constitutes 5% of the fitness score. With the `Key` pattern, we only optimize the pixel locations. In the fitness functions of both DE algorithms, we evaluate the accuracy of candidates on a random set of 640 training images. The model and dataset are described in the next subsection.

We carried out experiments on both the MNIST dataset and the CIFAR-10 dataset, and trigger patterns and trigger set images output by the DE algorithm is shown in Figure 7.7. To create `Logo`



(a) (b) (c)

(d) (e) (f)

Figure 7.7: Output of our DE algorithm. (a) The `Key` trigger pattern on CIFAR-10, (b) the `Logo` trigger pattern on CIFAR-10, (c) the `Key` trigger pattern on MNIST, (d) a sample using the trigger pattern in (a), (e) a sample using the trigger pattern in (b), (f) a sample using the trigger pattern in (c).

patterns on the CIFAR-10 dataset, we replicated the experiments by Zhang *et al.* [3]. Surprisingly, the optimal location to put the logo isn't at one of the 4 corners as one would intuitively think. In DE, we set $v_k = 255$ searched the coefficient for blending α instead, which yielded an optimal value of 0.4019.

Like the original method, we encoded the message in the pixel values of the `Key` trigger pattern. The CIFAR-10 variant includes 64 pixels and encodes 128 bits of information. Every pixel in the RGB color space with $v_k = \pm 100$ encodes 2 bits, and the message can be decoded by reading the pixels from left to right, top to bottom. The MNIST variant includes 192 pixels and encodes 192-bit information, with $v_k = 100, 200$ to encode 0 and 1 respectively. In both cases, pixels in the `Key` pattern gravitate towards the edge of the images. Clearly, the DE algorithm is rewarding pixel locations that do not overlap with the objects, which tend to occupy the center of the image. The new pixel locations form patterns in a way that has minimal impacts on the classification of an object. Because of that, even we added patterns with large pixel values, the resulting images still didn't trigger regular models.

To test the capacity of our algorithm, we intentionally used patterns that have a lot more complexity in our experiments on the MNIST dataset. Images in MNIST dataset has a lot more empty space to take advantage of, while pixels blindly accumulate at the edge may cause misclassification. Through our algorithm, the probability $\Pr(f_0(g(X_i, P)) = y_i)$ increased from as low as 83.30% (during random initialization) to 99.27%. The probability is measured over the entire trigger set, and the classification accuracy on the regular test set with the exact same images is 99.46. It clearly shows the algorithm's ability to learn to reduce false positive detections of the watermark. To test our DE algorithm's ability to converge, we repeated the MNIST/CIFAR10 "key" on 8 different parameter sets (number of pixels in the pattern, etc.), 5 experiments per set. Eat set all converged to solutions that produce extremely similar fitness scores, with an average standard deviation of 0.0060.

	Dataset	Regular	Watermarked
CIFAR-10 ResNet-18	Test	94.13	93.49
	Trigger (`Key`)	1.08	93.67
	Test	94.13	93.76
	Trigger (`Logo`)	1.27	93.49
MNIST 4 CONV-2 FC	Test	99.46	99.38
	Trigger (`Key`)	0.25	99.26

Table 7.5: Classification accuracy of watermarked models on test sets and trigger tests. The trigger sets are derived from the test sets using the `Key` / `Logo` method.

7.5.2.2 Effectiveness and Fidelity

It has been demonstrated in all previous works that DNNs can be trained to successfully recognize the triggers sets. In addition, Adi *et al.* also showed the significance of starting training from scratch in creating a robust watermark [2]. We followed the same procedure. Table 7.5 shows the classification accuracy of both non-watermarked and watermarked models on the regular test set and the trigger set.

In light of the fidelity criterion, the classification accuracy of the watermarked model on the regular test set is slightly lower compared to the regular non-watermarked model. It is expected as watermarking makes the classification problem much harder. In light of the effectiveness criterion, the ability of the watermarked model to recognize the trigger set is as good as its ability to classify regular images.

7.5.2.3 False Positives

Figure 7.8 shows the false positive rate of different trigger patterns. The false positive rate here is measured by the probability that a non-watermarked model classifies a trigger image into its re-assigned class $\Pr(f_0(g(X_i, P) = y_i')$. We used four different non-watermarked DNN trained on the regular CIFAR-10 dataset: ResNet-18, ResNet-50, DenseNet-121, VGG-16. The fitness function in DE used to obtain the trigger pattern only involves the ResNet-18 model. The results

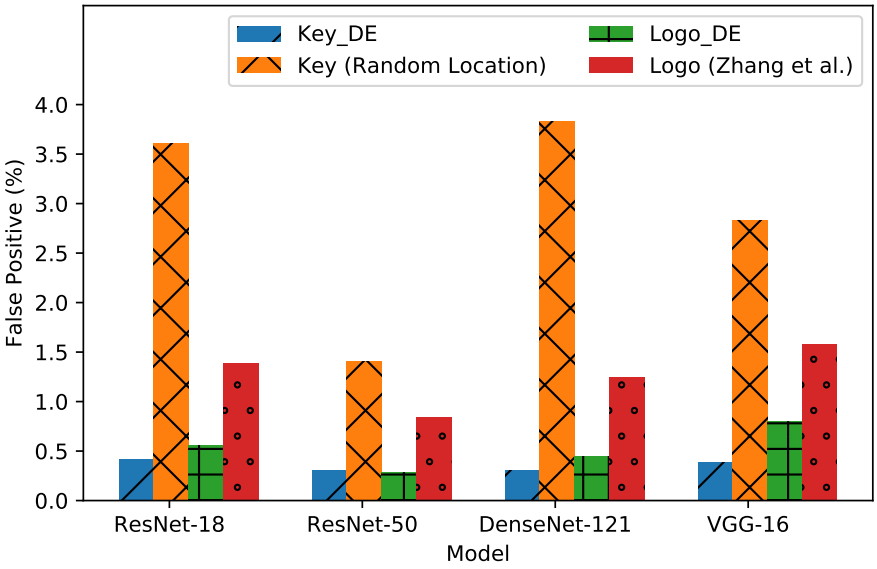


Figure 7.8: False positive rates of different CIFAR-10 trigger sets. It is the probability of a non-watermarked DNN getting falsely triggered. The lower the false positive rate the better.

show that what our DE learns from one model generalizes well to other models as well. We tested the generalizability further using the same pattern on 5 newly trained VGG-13s and obtained a 95% confidence interval of $1.15\% \pm 0.06\%$.

We used two baselines for comparison. To compare with the DE-based `Key` pattern, we used a `Key` pattern with random $\{x_k, y_k\}^K$ but the same v_k. We see drastic improvements with up to $10\times$ reduction in the false positive rate. Note that the trigger pattern proposed by Guo *et al.* is also based on random location [4]. But they explicitly selected v_k such that the pattern is imperceptible, resulting in a lower false positive rate. But as we see later, they achieved that at the cost of robustness. To compare with the DE-based `Logo`, we use `Logo` trigger pattern used Zhang *et al.* [3]. We achieve about $2\times$ improvement in false positive rate.

Another way to measure the false positive detection is through the ownership test in Section Section 7.2.5. Figure 7.9 transforms the false positive rate in Figure 7.8 into probabilities of passing the ownership test. With $\Delta = 5, N = 20$, even $2\times$ reduction in false positive rate translates to over $25000\times$ lower probability.

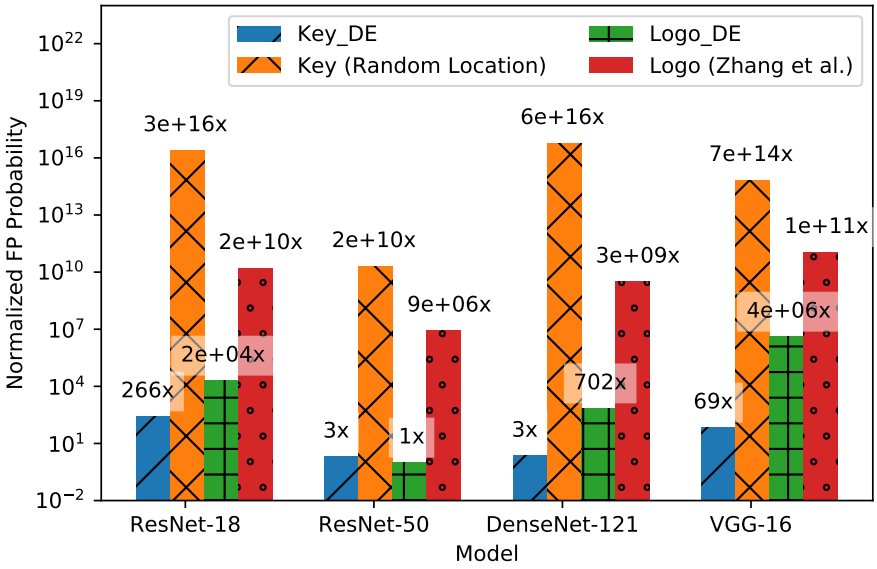


Figure 7.9: Normalized probability of a regular model passing the ownership test ($\Delta = 5, N = 20$). The lower the probability the better.

7.5.2.4 Robustness

We measure the robustness of the watermarking methods through their resistance against fine-tune attacks. Table 7.6 reports trigger set classification accuracy loss after we fine-tuned a watermarked model. Unlike some of the earlier approaches that based their attack on the training set, we used 1000 test images and applied various data augmentation techniques. The model watermarked using the original `key` method suffered a significant drop in accuracy. The accuracy drop was almost entirely eliminated when we switch to our `key` method. Both of the `logo` method were resilient against the fine-tune attack.

Images superimposed with our pattern are sufficiently different from the normal input distribution. Because of that, a watermarked model's ability to recognize those patterns is largely orthogonal to its ability to classify objects and is, therefore, harder to remove during a fine-tune attack.

Method	Accuracy loss (%)
`Key`, DE	-0.29
`Key`, [4]	-73.97
`Logo`, DE	-1.01
`Logo`, [3]	-0.97

Table 7.6: Classification accuracy of watermarked models on corresponding CIFAR-10 trigger sets after fine-tune attacks. The less the accuracy loss, the more robust the method is.

7.5.2.5 Discussions

Gradient-free methods do exist in the world of adversarial learning, most notably in the sub-problem of black-box attacks, where the attackers don't have access to the gradient information [161] [170] [171] [172]. Those methods again focus on individual samples and are essentially solving a different problem than ours. It is worth noting that the work from Moosavi-Dezfooli *et al.* aims at creating universal adversarial perturbations [173]. Their proposal to reduce the search space to a subset of the input provided invaluable insights.

Due to the limited scope of this paper, the parameter v_k isn't systematically studied. It is more a heuristic and manually selected in many situations. It would be valuable to study how it systematically affects the robustness of the watermarking methods.

7.6 Conclusion

In this chapter, we analyzed the scenario of watermarking DNNs for embedded devices. We propose a black-box watermarking framework, where we embed signatures by modifying the training set of the DNN. We can build a strong proof of ownership by repeatedly test the model using any input combined our signature. We demonstrate an example implementation of the framework and evaluate it using popular image classification datasets. The method is effective across multiple datasets and DNN architectures and has a negligible negative impact on performance. Given our threat model, it is also able to resist ghost signature attack and tampering attack. Further, we

propose a novel differential evolution-based framework to optimize the generation of such trigger patterns. Compared to the prior arts, our method demonstrates significant improvement in false positive rate and robustness in experiments on popular models and datasets.

CHAPTER 8

Concluding Remarks

The emergence and rapid evolution of the Internet of Things have drastically changed people's work and lives. It is accompanied by a paradigm shift in computing that moved data processing to the edge of the network. While edge computing promises to solve many of IoT's problems, challenges still remain. A significant portion of edge devices are resource-constrained and battery-powered. Thus there is a great need for improving the energy efficiency of the devices. In addition, edge devices can operate in a diverse set of environments, exposing them to various types of threats. Ensuring the security of the devices is also of paramount importance.

In this thesis, we introduced a set of learning-based techniques that address the energy and security challenges in IoT systems. We began by presenting a framework that learns from historical data and predicts optimal resource allocation in order to save energy consumption. We learned to predict whether or not to offload code to a resource-constrained data center. We learned to predict the CPU frequency used for the upcoming workload. We also learned to sustainably operate an environmentally powered sensor network.

We then proceeded to discuss the energy-saving techniques in machine learning applications in the context of edge computing. We focused on the typical computer vision pipeline and studied two of its most important components, sensing and the machine learning models for perception. Guided by the objective of the machine learning models, we proposed an adaptive image subsampling algorithm that reduces the energy consumption of the image sensor. We also developed model pruning and customization techniques to improve the energy efficiency and latency of the machine learning model.

Lastly, we discussed the security aspects of edge computing. In particular, we focused on intellectual property protection, a security problem that is of great commercial value yet far from

solved. In particular, we introduced a watermarking framework to protect the IP right of the machine learning model owners. We integrated the framework with the training of the models. The resulting watermarking framework is very effective, has low false positive rates, and only requires black-box access to the model. We also derived algorithms to make the watermarks more robust.

REFERENCES

[1] "Intel lab data." http://db.csail.mit.edu/labdata/labdata.html.

[2] Y. Adi, C. Baum, M. Cissé, B. Pinkas, and J. Keshet, "Turning your weakness into a strength: Watermarking deep neural networks by backdooring," in *USENIX Security Symposium*, pp. 1615–1631, 2018.

[3] J. Zhang, Z. Gu, J. Jang, H. Wu, M. P. Stoecklin, H. Huang, and I. Molloy, "Protecting intellectual property of deep neural networks with watermarking," in *Asia Conference on Computer and Communications Security*, pp. 159–172, 2018.

[4] J. Guo and M. Potkonjak, "Watermarking deep neural networks for embedded systems," in *International Conference on Computer-Aided Design (ICCAD)*, p. 133, 2018.

[5] "Linux kernel." https://www.kernel.org/doc/. Accessed: 2016-02-27.

[6] K. Ashton *et al.*, "That internet of things thing," *RFID Journal*, vol. 22, no. 7, pp. 97–114, 2009.

[7] "2017 roundup of Internet Of Things forecasts." https://www.forbes.com/sites/louiscolumbus/2017/12/10/2017-roundup-of-internet-of-things-forecasts, Dec 2017.

[8] "New GSMA study: operators must look beyond connectivity to increase share of $1.1 trillion ioT revenue opportunity." https://www.gsma.com/newsroom/press-release/new-gsma-study-operators-must-look-beyond-connectivity-to-increase-share/, May 2019.

[9] G. J. Pottie and W. J. Kaiser, "Wireless integrated network sensors," *Communnications of the ACM*, vol. 43, no. 5, pp. 51–58, 2000.

[10] G. Gualdi, A. Prati, and R. Cucchiara, "Video streaming for mobile video surveillance," *IEEE Trans. Multimedia*, vol. 10, no. 6, pp. 1142–1154, 2008.

[11] "Adapt your app by understanding what users are doing." https://developers.google.com/location-context/activity-recognition/.

[12] L. Bao and S. S. Intille, "Activity recognition from user-annotated acceleration data," in *International Conference on Pervasive Computing (PERVASIVE)*, pp. 1–17, 2004.

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

[14] S. Ren, K. He, R. B. Girshick, and J. Sun, "Faster R-CNN: towards real-time object detection with region proposal networks," in *Advances in Neural Information Processing Systems (NIPS)*, pp. 91–99, 2015.

[15] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet of Things Journal*, vol. 3, no. 5, pp. 637–646, 2016.

[16] "Internet of aircraft things: An industry set to be transformed." https://aviationweek.com/connected-aerospace/internet-aircraft-things-industry-set-be-transformed, Jan 2016.

[17] "iphone 7." https://www.apple.com/iphone-7/specs/.

[18] "Echo & Alexa - amazon devices." https://www.amazon.com/Amazon-Echo-And-Alexa-Devices.

[19] "Google Home - smart speaker & home assistant." https://store.google.com/product/google_home.

[20] "Google admits 'error' after it forgot to tell nest secure users about hidden microphones." http://fortune.com/2019/02/20/google-nest-secure-microphone/, Feb 2019.

[21] "Mobile network experience report january 2019." https://www.opensignal.com/reports/2019/01/usa/mobile-network-experience, Jan 2019.

[22] "Worldwide wearables market ticks up 5.5% due to gains in emerging markets, says idc." https://www.idc.com/getdoc.jsp?containerId=prUS44247418, Sep 2018.

[23] "Apple watch series 4 battery information." https://www.apple.com/watch/battery.html.

[24] "Self-driving cars use crazy amounts of power, and it's becoming a problem." https://www.wired.com/story/self-driving-cars-power-consumption-nvidia-chip/, Feb 2018.

[25] "Smart technical data." https://www.smart.com/en/en/index/smart-fortwo-electric-drive-453/technical-data.html.

[26] "Indigo." https://www.indigoag.com/.

[27] A. Kansal, J. Hsu, S. Zahedi, and M. B. Srivastava, "Power management in energy harvesting sensor networks," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 6, no. 4, p. 32, 2007.

[28] "Spoondrift." https://spoondrift.co/.

[29] "How hackers could use a nest thermostat as an entry point into your home." https://www.forbes.com/sites/aarontilley/2015/03/06/nest-thermostat-hack-home-network/#4bc601913986, March 2015.

[30] "Weak default logins expose internet cameras to hacking." https://www.pcmag.com/news/364307/weak-default-logins-expose-internet-cameras-to-hacking, Oct 2018.

[31] J. Yosinski, J. Clune, Y. Bengio, and H. Lipson, "How transferable are features in deep neural networks?," in *Advances in Neural Information Processing Systems (NIPS)*, pp. 3320–3328, 2014.

[32] E. Cuervo, A. Balasubramanian, D. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI: making smartphones last longer with code offload," in *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pp. 49–62, 2010.

[33] B. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "Clonecloud: elastic execution between mobile device and cloud," in *European conference on Computer systems (EuroSys)*, pp. 301–314, 2011.

[34] S. Kosta, A. Aucinas, P. Hui, R. Mortier, and X. Zhang, "Thinkair: Dynamic resource allocation and parallel execution in the cloud for mobile code offloading," in *IEEE INFOCOM*, pp. 945–953, 2012.

[35] D. T. Hoang, C. Lee, D. Niyato, and P. Wang, "A survey of mobile cloud computing: architecture, applications, and approaches," *Wireless Communications and Mobile Computing*, vol. 13, no. 18, pp. 1587–1611, 2013.

[36] S. Abolfazli, Z. Sanaei, E. Ahmed, A. Gani, and R. Buyya, "Cloud-based augmentation for mobile devices: Motivation, taxonomies, and open challenges," *IEEE Communications Surveys and Tutorials*, vol. 16, no. 1, pp. 337–368, 2014.

[37] L. A. Barroso, J. Clidaras, and U. Hölzle, *The Datacenter as a Computer: An Introduction to the Design of Warehouse-Scale Machines, Second Edition*. Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers, 2013.

[38] L. A. Barroso and U. Hölzle, "The case for energy-proportional computing," *IEEE Computer*, vol. 40, no. 12, pp. 33–37, 2007.

[39] D. Gmach, J. Rolia, L. Cherkasova, and A. Kemper, "Workload analysis and demand prediction of enterprise data center applications," in *IEEE International Symposium on Workload Characterization*, pp. 171–180, 2007.

[40] Y. Han, J. Chan, and C. Leckie, "Analysing virtual machine usage in cloud computing," in *IEEE World Congress on Services (SERVICES)*, pp. 370–377, 2013.

[41] P. Mell and T. Grance, "The nist definition of cloud computing," *National Institute of Standards and Technology*, vol. 53, no. 6, p. 50, 2009.

[42] P. Shu, F. Liu, H. Jin, M. Chen, F. Wen, Y. Qu, and B. Li, "etime: Energy-efficient transmission between cloud and mobile devices," in *IEEE INFOCOM*, pp. 195–199, 2013.

[43] S. Barbarossa, S. Sardellitti, and P. D. Lorenzo, "Joint allocation of computation and communication resources in multiuser mobile cloud computing," in *IEEE Workshop on Signal Processing Advances in Wireless Communications (SPAWC)*, pp. 26–30, 2013.

[44] X. Chen, "Decentralized computation offloading game for mobile cloud computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 26, no. 4, pp. 974–983, 2015.

[45] K. Kumar and Y. Lu, "Cloud computing for mobile users: Can offloading computation save energy?," *IEEE Computer*, vol. 43, no. 4, pp. 51–56, 2010.

[46] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica, "Dominant resource fairness: Fair allocation of multiple resource types," in *USENIX Symposium on Networked Systems Design and Implementation (NSDI)*, 2011.

[47] J. Wilkes, "More Google cluster data." Google research blog, Nov 2011. Posted at `http://googleresearch.blogspot.com/2011/11/more-google-cluster-data.html`.

[48] A. K. Mishra, J. L. Hellerstein, W. Cirne, and C. R. Das, "Towards characterizing cloud backend workloads: insights from google compute clusters," *SIGMETRICS Performance Evaluation Review*, vol. 37, no. 4, pp. 34–41, 2010.

[49] G. Wang, A. R. Butt, H. M. Monti, and K. Gupta, "Towards synthesizing realistic workload traces for studying the hadoop ecosystem," in *IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, pp. 400–408, 2011.

[50] M. Crovella and A. Bestavros, "Self-similarity in world wide web traffic: evidence and possible causes," *IEEE/ACM Transactions on Networking*, vol. 5, no. 6, pp. 835–846, 1997.

[51] R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. R. Kan, "Optimization and approximation in deterministic sequencing and scheduling: a survey," *Annals of Discrete Mathematics*, vol. 5, pp. 287–326, 1979.

[52] M. H. Rothkopf, "Scheduling independent tasks on parallel processors," *Management Science*, vol. 12, no. 5, pp. 437–447, 1966.

[53] E. L. Lawler and J. M. Moore, "A functional equation and its application to resource allocation and sequencing problems," *Management Science*, vol. 16, no. 1, pp. 77–84, 1969.

[54] A. Bar-Noy, S. Guha, J. Naor, and B. Schieber, "Approximating the throughput of multiple machines in real-time scheduling," *SIAM Journal on Computing*, vol. 31, no. 2, pp. 331–352, 2001.

[55] M. Isard, V. Prabhakaran, J. Currey, U. Wieder, K. Talwar, and A. Goldberg, "Quincy: fair scheduling for distributed computing clusters," in *Symposium on Operating Systems Principles (SOSP)*, pp. 261–276, 2009.

[56] J. Mo and J. C. Walrand, "Fair end-to-end window-based congestion control," *IEEE/ACM Transactions on Networking*, vol. 8, no. 5, pp. 556–567, 2000.

[57] E. L. Hahne, "Round-robin scheduling for max-min fairness in data networks," *IEEE Journal on Selected Areas in Communications*, vol. 9, no. 7, pp. 1024–1039, 1991.

[58] F. Kelly, "Charging and rate control for elastic traffic," *European Transactions on Telecommunications*, vol. 8, no. 1, pp. 33–37, 1997.

[59] L. Georgiadis, M. J. Neely, and L. Tassiulas, "Resource allocation and cross-layer control in wireless networks," *Foundations and Trends in Networking*, vol. 1, no. 1, 2006.

[60] H. J. Kushner and P. A. Whiting, "Convergence of proportional-fair sharing algorithms under general conditions," *IEEE Transactions on Wireless Communications*, vol. 3, no. 4, pp. 1250–1259, 2004.

[61] R. Jain, D. Chiu, and W. Hawe, "A quantitative measure of fairness and discrimination for resource allocation in shared computer systems," *CoRR*, vol. cs.NI/9809099, 1998.

[62] D. W. et. al, *fArma: ARMA Time Series Modelling*, 2013. R package version 3010.79.

[63] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[64] T. Ishihara and H. Yasuura, "Voltage scheduling problem for dynamically variable voltage processors," in *International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 197–202, 1998.

[65] M. Pedram and J. M. Rabaey, *Power aware design methodologies*. Springer Science & Business Media, 2002.

[66] "Android operating system." `https://android.googlesource.com/`. Accessed: 2016-05-15.

[67] A. Pathak, A. Jindal, Y. C. Hu, and S. P. Midkiff, "What is keeping my phone awake?: characterizing and detecting no-sleep energy bugs in smartphone apps," in *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pp. 267–280, 2012.

[68] P. Tseng, P. Hsiu, C. Pan, and T. Kuo, "User-centric energy-efficient scheduling on multi-core mobile devices," in *Design Automation Conference (DAC)*, pp. 85:1–85:6, 2014.

[69] J. e. a. Hopper, "Using the linux cpufreq subsystem for energy management," *IBM blueprints*, 2009.

[70] P. Mochel, "The sysfs filesystem," in *Linux Symposium*, p. 313, 2005.

[71] D. Hillenbrand, Y. Furuyama, A. Hayashi, H. Mikami, K. Kimura, and H. Kasahara, "Reconciling application power control and operating systems for optimal power and performance," in *International Workshop on Reconfigurable and Communication-Centric Systems-on-Chip (ReCoSoC)*, pp. 1–8, 2013.

[72] K. Choi, K. Dantu, W. Cheng, and M. Pedram, "Frame-based dynamic voltage and frequency scaling for a MPEG decoder," in *IEEE/ACM International Conference on Computer-aided Design (ICCAD)*, pp. 732–737, 2002.

[73] Z. Ma, H. Hu, and Y. Wang, "On complexity modeling of H.264/AVC video decoding and its application for energy efficient decoding," *IEEE Transactions on Multimedia*, vol. 13, no. 6, pp. 1240–1255, 2011.

[74] J. Hamers and L. Eeckhout, "Exploiting media stream similarity for energy-efficient decoding and resource prediction," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 11, no. 1, p. 2, 2012.

[75] A. Carroll and G. Heiser, "Unifying DVFS and offlining in mobile multicores," in *IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pp. 287–296, 2014.

[76] J. M. Kim, Y. G. Kim, and S. W. Chung, "Stabilizing CPU frequency and voltage for temperature-aware DVFS in mobile devices," *IEEE Transactions on Computers*, vol. 64, no. 1, pp. 286–292, 2015.

[77] A. Das, R. A. Shafik, G. V. Merrett, B. M. Al-Hashimi, A. Kumar, and B. Veeravalli, "Reinforcement learning-based inter- and intra-application thermal optimization for lifetime improvement of multicore systems," in *Design Automation Conference (DAC)*, pp. 170:1–170:6, 2014.

[78] J. Chen and C. Kuo, "Energy-efficient scheduling for real-time systems on dynamic voltage scaling (DVS) platforms," in *International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA)*, pp. 28–38, 2007.

[79] R. Xu, D. Mossé, and R. G. Melhem, "Minimizing expected energy consumption in real-time systems through dynamic voltage scaling," *ACM Transactions on Computer Systems (TOCS)*, vol. 25, no. 4, 2007.

[80] J. R. Lorch and A. J. Smith, "PACE: A new approach to dynamic voltage scaling," *IEEE Transactions on Computers*, vol. 53, no. 7, pp. 856–869, 2004.

[81] W. Yuan and K. Nahrstedt, "Energy-efficient soft real-time CPU scheduling for mobile multimedia systems," in *ACM Symposium on Operating Systems Principles (SOSP)*, pp. 149–163, 2003.

[82] E. Chung, G. D. Micheli, and L. Benini, "Contents provider-assisted dynamic voltage scaling for low energy multimedia applications," in *International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 42–47, 2002.

[83] P. Kumar and M. B. Srivastava, "Power-aware multimedia systems using run-time prediction," in *International Conference on VLSI Design*, pp. 64–69, 2001.

[84] C. Im, H. Kim, and S. Ha, "Dynamic voltage scheduling technique for low-power multimedia applications using buffers," in *International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 34–39, 2001.

[85] Y. Lu, L. Benini, and G. D. Micheli, "Dynamic frequency scaling with buffer insertion for mixed workloads," *IEEE Transactions Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 11, pp. 1284–1305, 2002.

[86] M. Kim, Y. G. Kim, S. W. Chung, and C. H. Kim, "Measuring variance between smartphone energy consumption and battery life," *IEEE Computer*, vol. 47, no. 7, pp. 59–65, 2014.

[87] N. L. Binkert, B. M. Beckmann, G. Black, S. K. Reinhardt, A. G. Saidi, A. Basu, J. Hestness, D. Hower, T. Krishna, S. Sardashti, R. Sen, K. Sewell, M. Shoaib, N. Vaish, M. D. Hill, and D. A. Wood, "The gem5 simulator," *SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[88] S. Li, J. H. Ahn, R. D. Strong, J. B. Brockman, D. M. Tullsen, and N. P. Jouppi, "Mcpat: an integrated power, area, and timing modeling framework for multicore and manycore architectures," in *IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pp. 469–480, 2009.

[89] "xiph.org." `http://www.xiph.org/`. Accessed: 2016-06-30.

[90] "Panasonic solar cells technical handbook." `http://www.solarbotics.net/library/datasheets/sunceram.pdf`.

[91] S. Meninger, J. O. Mur-Miranda, R. Amirtharajah, A. P. Chandrakasan, and J. H. Lang, "Vibration-to-electric energy conversion," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 9, no. 1, pp. 64–76, 2001.

[92] C. Melhuish, "The ecobot project." `http://www.ias.uwe.ac.uk/energy_autonomy/EcoBot_web_page.html`.

[93] X. Lu, P. Wang, D. Niyato, and E. Hossain, "Dynamic spectrum access in cognitive radio networks with RF energy harvesting," *IEEE Wireless Communications*, vol. 21, no. 3, pp. 102–110, 2014.

[94] S. J. Roundy, *Energy scavenging for wireless sensor nodes with a focus on vibration to electricity conversion*. PhD thesis, University of California, Berkeley, 2003.

[95] V. Devabhaktuni, M. Alam, S. S. S. R. Depuru, R. C. Green, D. Nims, and C. Near, "Solar energy: Trends and enabling technologies," *Renewable and Sustainable Energy Reviews*, vol. 19, pp. 555–564, 2013.

[96] "Solar expected to maintain its status as the world's fastest-growing energy technology." `http://www.socialfunds.com/news/article.cgi/2639.html`.

[97] "Enocean." `https://www.enocean.com/`.

[98] "G24 power." `www.g24i.com`.

[99] "Ixys." `www.ixys.com`.

[100] J. Paradiso, T. Starner, *et al.*, "Energy scavenging for mobile and wireless electronics," *IEEE Pervasive Computing*, vol. 4, no. 1, pp. 18–27, 2005.

[101] R. C. Shah and J. M. Rabaey, "Energy aware routing for low energy ad hoc sensor networks," in *IEEE Wireless Communications and Networking Conference (WCNC)*, vol. 1, pp. 350–355, IEEE, 2002.

[102] E. M. Royer and C.-K. Toh, "A review of current routing protocols for ad hoc mobile wireless networks," *IEEE Personal Communications*, vol. 6, no. 2, pp. 46–55, 1999.

[103] T. Xu and M. Potkonjak, "Energy saving using scenario based sensor selection on medical shoes," in *International Conference on Healthcare Informatics (ICHI)*, pp. 398–403, IEEE, 2015.

[104] R. Yan, V. C. Shah, T. Xu, and M. Potkonjak, "Security defenses for vulnerable medical sensor network," in *International Conference on Healthcare Informatics (ICHI)*, pp. 300–309, IEEE, 2014.

[105] X. Jiang, J. Polastre, and D. Culler, "Perpetual environmentally powered sensor networks," in *International Symposium on Information Processing in Sensor Networks (IPSN)*, pp. 463–468, IEEE, 2005.

[106] J. Taneja, J. Jeong, and D. Culler, "Design, modeling, and capacity planning for micro-solar power sensor networks," in *International Conference on Information Processing in Sensor Networks (IPSN)*, pp. 407–418, IEEE Computer Society, 2008.

[107] V. Raghunathan, A. Kansal, J. Hsu, J. Friedman, and M. Srivastava, "Design considerations for solar energy harvesting wireless embedded systems," in *International Symposium on Information Processing in Sensor Networks (IPSN)*, p. 64, IEEE Press, 2005.

[108] C. Park and P. H. Chou, "Ambimax: Autonomous energy harvesting platform for multi-supply wireless sensor nodes," in *IEEE Communications Society on Sensor and Ad Hoc Communications and Networks (SECON)*, vol. 1, pp. 168–177, IEEE, 2006.

[109] D. B. Johnson and D. A. Maltz, "Dynamic source routing in ad hoc wireless networks," in *Mobile Computing*, pp. 153–181, Springer, 1996.

[110] W. Zhang, G. Cao, and T. L. Porta, "Dynamic proxy tree-based data dissemination schemes for wireless sensor networks," *Wireless Networks*, vol. 13, no. 5, pp. 583–595, 2007.

[111] Y. Zhang, S. He, and J. Chen, "Data gathering optimization by dynamic sensing and routing in rechargeable sensor networks," *IEEE/ACM Transactions on Networking*, vol. 24, no. 3, pp. 1632–1646, 2016.

[112] R. K. Ahuja, *Network flows*. PhD thesis, Technische Universitat Darmstadt, 1993.

[113] "iPhone X - technical specifications." `https://www.apple.com/iphone-x/specs/`. Accessed: 2018-03-01.

[114] "Xperia^{TM}XZ specifications - Sony mobile." https://www.sonymobile.com/us/products/phones/xperia-xz/specifications/. Accessed: 2018-03-01.

[115] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. S. Bernstein, A. C. Berg, and F. Li, "Imagenet large scale visual recognition challenge," *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.

[116] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," Master's thesis, Department of Computer Science, University of Toronto, 2009.

[117] R. LiKamWa, B. Priyantha, M. Philipose, L. Zhong, and P. Bahl, "Energy characterization and optimization of image sensing toward continuous mobile vision," in *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, pp. 69–82, 2013.

[118] H. Gu and V. Swaminathan, "From thumbnails to summaries - a single deep neural network to rule them all," in *IEEE International Conference on Multimedia and Expo (ICME)*, pp. 1–6, 2018.

[119] S. Kawahito, M. Yoshida, M. Sasaki, K. Umehara, D. Miyazaki, Y. Tadokoro, K. Murata, S. Doushou, and A. Matsuzawa, "A cmos image sensor with analog two-dimensional dct-based compression circuits for one-chip cameras," *IEEE Journal of Solid-State Circuits*, vol. 32, no. 12, pp. 2030–2041, 1997.

[120] W. D. Leon-Salas, S. Balkir, K. Sayood, M. W. Hoffman, and N. Schemm, "A CMOS imager with focal plane compression," in *International Symposium on Circuits and Systems (ISCAS)*, 2006.

[121] E. Artyomov and O. Yadid-Pecht, "Adaptive multiple-resolution CMOS active pixel sensor," *IEEE Transactions on Circuits and Systems*, vol. 53-I, no. 10, pp. 2178–2186, 2006.

[122] Y. Oike and A. E. Gamal, "CMOS image sensor with per-column $\Sigma\Delta$ ADC and programmable compressed sensing," *IEEE Journal of Solid-State Circuits*, vol. 48, no. 1, pp. 318–328, 2013.

[123] S. E. Kemeny, R. Panicacci, B. Pain, L. H. Matthies, and E. R. Fossum, "Multiresolution image sensor," *IEEE Transactions Circuits and Systems for Video Technology*, vol. 7, no. 4, pp. 575–583, 1997.

[124] M. Buckler, S. Jayasuriya, and A. Sampson, "Reconfiguring the imaging pipeline for computer vision," in *IEEE International Conference on Computer Vision (ICCV)*, pp. 975–984, 2017.

[125] J. Guo and M. Potkonjak, "Pruning convnets online for efficient specialist models," in *IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pp. 430–437, 2017.

[126] J. L. et al., "Binarized convolutional neural networks with separable filters for efficient hardware acceleration," in *IEEE Conference on Computer Vision and Pattern Recognition Workshops*, pp. 344–352, 2017.

[127] G. Huang, D. Chen, T. Li, F. Wu, L. van der Maaten, and K. Q. Weinberger, "Multi-scale dense convolutional networks for efficient prediction," in *International Conference on Learning Representations (ICLR)*, 2018.

[128] H. G. Chen, S. Jayasuriya, J. Yang, J. Stephen, S. Sivaramakrishnan, A. Veeraraghavan, and A. C. Molnar, "ASP vision: Optically computing the first layer of convolutional neural networks using angle sensitive pixels," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 903–912, 2016.

[129] R. LiKamWa, Y. Hou, Y. Gao, M. Polansky, and L. Zhong, "Redeye: Analog convnet image sensor architecture for continuous mobile vision," in *ACM/IEEE Annual International Symposium on Computer Architecture (ISCA)*, pp. 255–266, 2016.

[130] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," in *International Conference on Learning Representations (ICLR)*, 2015.

[131] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2016.

[132] I. Kuon and J. Rose, "Measuring the gap between fpgas and asics," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 203–215, 2007.

[133] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in Neural Information Processing Systems (NIPS)*, pp. 1106–1114, 2012.

[134] M. Lin, Q. Chen, and S. Yan, "Network in network," in *International Conference on Learning Representations (ICLR)*, 2014.

[135] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *International Conference on Learning Representations (ICLR)*, 2015.

[136] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–9, 2015.

[137] Y. LeCun, J. S. Denker, and S. A. Solla, "Optimal brain damage," in *Advances in Neural Information Processing Systems (NIPS)*, pp. 598–605, 1989.

[138] B. Hassibi and D. G. Stork, "Second order derivatives for network pruning: Optimal brain surgeon," in *Advances in Neural Information Processing Systems (NIPS)*, pp. 164–171, 1992.

[139] S. Han, J. Pool, J. Tran, and W. J. Dally, "Learning both weights and connections for efficient neural network," in *Advances in Neural Information Processing Systems (NIPS)*, pp. 1135–1143, 2015.

[140] H. Li, A. Kadav, I. Durdanovic, H. Samet, and H. P. Graf, "Pruning filters for efficient convnets," in *International Conference on Learning Representations (ICLR)*, 2017.

[141] A. Polyak and L. Wolf, "Channel-level acceleration of deep face representations," *IEEE Access*, vol. 3, pp. 2163–2175, 2015.

[142] S. Anwar, K. Hwang, and W. Sung, "Structured pruning of deep convolutional neural networks," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 13, no. 3, pp. 32:1–32:18, 2017.

[143] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. A. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," pp. 265–283, 2016.

[144] S. Han, X. Liu, H. Mao, J. Pu, A. Pedram, M. A. Horowitz, and W. J. Dally, "Eie: Efficient inference engine on compressed deep neural network," in *International Symposium on Computer Architecture (ISCA)*, 2016.

[145] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European Conference on Computer Vision (ECCV)*, pp. 525–542, 2016.

[146] E. L. Denton, W. Zaremba, J. Bruna, Y. LeCun, and R. Fergus, "Exploiting linear structure within convolutional networks for efficient evaluation," in *Advances in Neural Information Processing Systems (NIPS)*, pp. 1269–1277, 2014.

[147] M. Jaderberg, A. Vedaldi, and A. Zisserman, "Speeding up convolutional neural networks with low rank expansions," in *British Machine Vision Conference (BMVC)*, 2014.

[148] J. Yosinski, J. Clune, A. Nguyen, T. Fuchs, and H. Lipson, "Understanding neural networks through deep visualization," in *ICML Deep Learning Workshop*, 2015.

[149] G. Qu and M. Potkonjak, "Analysis of watermarking techniques for graph coloring problem," in *International Conference on Computer-Aided Design (ICCAD)*, pp. 190–193, 1998.

[150] A. B. Kahng, S. Mantik, I. L. Markov, M. Potkonjak, P. Tucker, H. Wang, and G. Wolfe, "Robust IP watermarking methodologies for physical design," in *Conference on Design Automation (DAC)*, pp. 782–787, 1998.

[151] C. S. Collberg and C. D. Thomborson, "Software watermarking: Models and dynamic embeddings," in *ACM Conference on Programming Language Design and Implementation (PLDI)*, pp. 311–324, 1999.

[152] J. Palsberg, S. Krishnaswamy, M. Kwon, D. Ma, Q. Shao, and Y. Zhang, "Experience with software watermarking," in *Annual Computer Security Applications Conference (ACSAC)*, pp. 308–316, 2000.

[153] Y. Uchida, Y. Nagai, S. Sakazawa, and S. Satoh, "Embedding watermarks into deep neural networks," in *ACM International Conference on Multimedia Retrieval*, pp. 269–277, 2017.

[154] E. L. Merrer, P. Perez, and G. Trédan, "Adversarial frontier stitching for remote neural network watermarking," *CoRR*, vol. abs/1711.01894, 2017.

[155] F. Hartung and M. Kutter, "Multimedia watermarking techniques," *Proceedings of the IEEE*, vol. 87, no. 7, pp. 1079–1107, 1999.

[156] M. Barni, F. Bartolini, and A. Piva, "Improved wavelet-based watermarking through pixel-wise masking," *IEEE Transactions on Image Processing*, vol. 10, no. 5, pp. 783–791, 2001.

[157] I. Cox, M. Miller, J. Bloom, J. Fridrich, and T. Kalker, *Digital watermarking and steganography*. Morgan Kaufmann, 2007.

[158] A. L. Oliveira, "Techniques for the creation of digital watermarks in sequentialcircuit designs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 20, no. 9, pp. 1101–1117, 2001.

[159] B. D. Rouhani, H. Chen, and F. Koushanfar, "Deepsigns: A generic watermarking framework for IP protection of deep learning models," *IACR Cryptology ePrint Archive*, p. 311, 2018.

[160] C. Zhang, S. Bengio, M. Hardt, B. Recht, and O. Vinyals, "Understanding deep learning requires rethinking generalization," in *International Conference on Learning Representations (ICLR)*, 2017.

[161] A. Nguyen, J. Yosinski, and J. Clune, "Deep neural networks are easily fooled: High confidence predictions for unrecognizable images," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 427–436, 2015.

[162] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, vol. abs/1409.1556, 2014.

[163] B. D. Rohani, H. Chen, and F. Koushanfar, "Deepsigns: A generic watermarking framework for ip protection of deep learning models," *CoRR*, vol. abs/1804.00750, 2018.

[164] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *CoRR*, vol. abs/1412.6572, 2014.

[165] N. Papernot, P. D. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The limitations of deep learning in adversarial settings," in *IEEE European Symposium on Security and Privacy (S&P)*, pp. 372–387, 2016.

[166] N. Carlini and D. A. Wagner, "Towards evaluating the robustness of neural networks," in *IEEE Symposium on Security and Privacy (S&P)*, pp. 39–57, 2017.

[167] R. Storn and K. V. Price, "Differential evolution - A simple and efficient heuristic for global optimization over continuous spaces," *Journal of Global Optimization*, vol. 11, no. 4, pp. 341–359, 1997.

[168] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[169] G. Huang, Z. Liu, L. van der Maaten, and K. Q. Weinberger, "Densely connected convolutional networks," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2261–2269, 2017.

[170] J. Su, D. V. Vargas, and K. Sakurai, "One pixel attack for fooling deep neural networks," *CoRR*, vol. abs/1710.08864, 2017.

[171] A. Ilyas, L. Engstrom, A. Athalye, and J. Lin, "Black-box adversarial attacks with limited queries and information," in *International Conference on Machine Learning*, pp. 2142–2151, 2018.

[172] M. Alzantot, Y. Sharma, S. Chakraborty, and M. B. Srivastava, "Genattack: Practical black-box attacks with gradient-free optimization," *CoRR*, vol. abs/1805.11090, 2018.

[173] S. Moosavi-Dezfooli, A. Fawzi, O. Fawzi, and P. Frossard, "Universal adversarial perturbations," in *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 86–94, 2017.