DISTRIBUTING DEEP NEURAL NETWORK INFERENCE ACROSS EDGE-HUB-CLOUD SYSTEMS

by

Robert Viramontes

A dissertation submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

(Electrical and Computer Engineering)

at the

UNIVERSITY OF WISCONSIN–MADISON

2025

Date of preliminary examination: May 7, 2025
The dissertation is approved by the following members of the Final Oral Committee:
Azadeh Davoodi, Professor, Electrical and Computer Engineering
Umit Ogras, Professor, Electrical and Computer Engineering
Jeffrey Linderoth, Professor, Industrial and Systems Engineering
Mahesh Sharma, Fellow, Tenstorrent

ACKNOWLEDGMENTS

I would first like to acknowledge and thank my advisor Professor Azadeh Davoodi for her guidance during my PhD research and studies. The depth and breadth of her knowledge have been invaluable in incubating ideas and providing insightful feedback during the research process. I would also like to thank Professor Yu Hen Hu for collaboration and insight in several projects early in my research and service on my preliminary exam committee. I extend my deepest gratitude to Professor Umit Ogras, Professor Jeffrey Lineroth, and Dr. Mahesh Sharma for their time to serve on my defense committee.

I also extend my thanks to my parents for their unwavering support during my PhD studies, without which I would be lost. And to my friends, who remind me to leave the lab and bring levity and balance to my life, my deep appreciation.

This material is based upon work supported by the National Science Foundation under Grant No. 2006394. Many thanks in their support funding this work.

CONTENTS

Contents ii

Abstract v

1 Introduction and Motivation 1

2 Related Works on Distributed Inference 6

2.1 Latency Optimization 6

2.2 Energy Optimization 9

2.3 Dynamic Voltage and Frequency Scaling 10

2.4 Carbon Footprint 12

2.5 Areas of Improvement 13

3 Summary of Contributions 15

4 System Model for Distributed Inference and Global Latency 18

4.1 Hardware System Model 18

4.2 Deep Neural Network System Model 23

4.3 Analytic Latency Model 27

4.4 Profiling-Based Latency 31

4.5 Comparison of Latency Models 34

5 Base Integer Programming Model for Distributed Inference 37

5.1 Notations 37

5.2 Optimization Constraints 39

5.3 Latency Objective Function 41

5.4 Conclusion 42

6 ODIWeP: Optimizing Distributed Inference with Weight Preloading 43

6.1 Motivation 44

6.2 Opportunity for Preloading Weights 45

6.3 From Base ILP to ODIWeP 46

6.4 Results 47

6.5 Conclusion 57

7 DIME: Distributed Inference Model Estimation for Bundle Profiling 58

7.1 Motivation 59

7.2 Bundle-Based Profiling 62

7.3 From Base ILP to DIME 64

7.4 Results 68

7.5 Conclusion 76

8 FreDDI: Frequency-Driven Distributed Inference 78

8.1 Motivation 79

8.2 Energy Profiling 81

8.3 From Base ILP to FreDDI 83

8.4 Results 88

8.5 Conclusion 99

9 CADI: Carbon-Aware Distributed Inference 100

9.1 Motivation 100

9.2 Carbon Intensity 101

9.3 From Base ILP to CADI 102

9.4 Results 107

9.5 Conclusion 113

10 Chapter for the Public 115

10.1 Background on Artificial Intelligence (AI) 115

10.2 What's in a Layer 117

10.3 What's the Deal With All These Computers? 118

10.4 Useful Objectives 119

10.5 Finally, Some Results! 121

10.6 Conclusion 128

11 Conclusion 129

Bibliography 133

ABSTRACT

Deep neural networks (DNNs) have been instrumental in the recent, rapid growth of artificial intelligence. DNNs have grown in compute complexity and requirements as they have advanced to achieve greater accuracy across a variety of domains. As these DNNs have been growing, they have also found use in a variety of applications that may be, for instance latency or energy constrained. This work explores distributed inference across multiple, heterogeneous devices as a method to meet optimization objectives. That is, each layer of the DNN is assigned to be computed on a particular device to e.g. minimize latency.

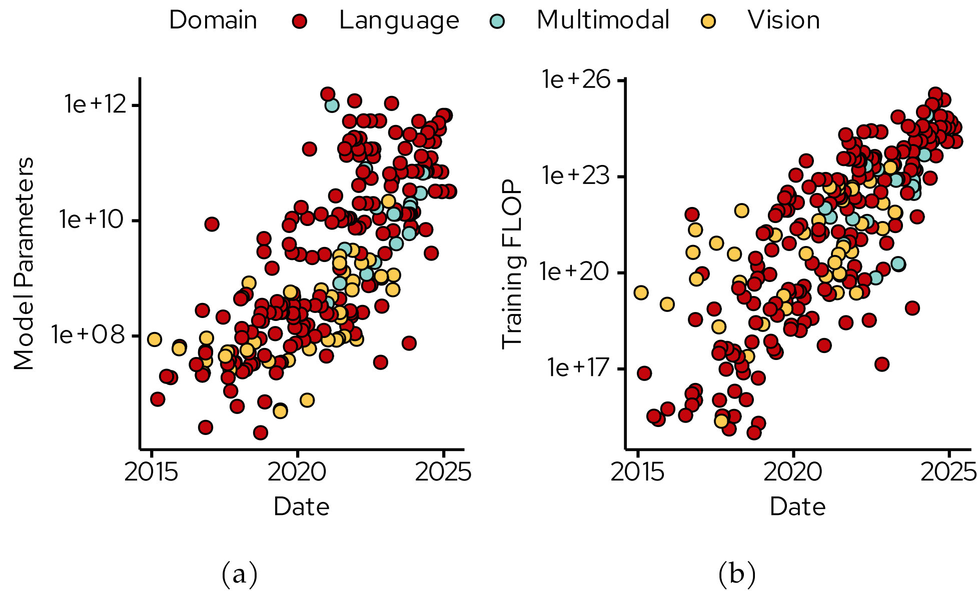
The distributed inference problem is modeled as a flexible and extensible integer linear program (ILP). We then explore a variety of novel distributed inference approaches by extending this model, showing that these approaches improve objectives such as latency or energy compared to prior work. We first model a technique to reduce overall latency called weight preloading, taking advantage of opportunities to overlap compute and data movement. Then, based upon our observations during profiling, we introduce a novel constraint to ensure correct layer assignments and demonstrate a new technique for reducing profiling effort. We also demonstrate that device operating frequency may be used as a knob of control alongside layer assignment to trade off latency and energy during distributed inference. Finally, the model is extended to minimize carbon footprint over an application duration, demonstrating that distributed inference can be utilized to reduce environmental impact. This work demonstrates that distributed inference,

as modeled by our ILP formulations, can effectively address a variety of application objectives.

1 INTRODUCTION AND MOTIVATION

In the past decade, machine learning (ML) and artificial intelligence (AI) have rapidly evolved from prototypes in research labs to reliable assistants for daily home tasks. This is, in part, because AI has rapidly become better at completing complex tasks, increasing its utility in everyday circumstances. One popular, early application of consumer AI is the intelligent personal assistant (IPA), and customers have grown accustomed to using voice commands to ask their smartphone to accomplish complex tasks [1]. The success of early applications has spurred the proliferation of AI and the demand for AI features spans from intelligent maintenance of factory lines [2] to automatically keeping track of pets at home [3]. This growing deployment of internet of things (IoT) devices and the demand for always-on processing of their data by AI agents have given rise to pervasive AI, a model of AI task deployment that coordinates ubiquitous resources [1].

At the same time that ML has proliferated into many domains and become widely deployed, the complexity of the models has grown significantly to handle new challenges. AlexNet spurred the modern convolutional neural network (CNN) revolution in 2012 with leading performance in the ImageNet image recognition challenge [5]. Their unmatched performance was enabled, in part, by the insight that increasing the model depth (here, roughly synonymous with complexity) was key to increasing image recognition performance [6]. A neural network (NN) is comprised of multiple layers executed sequentially and the 'depth' is increased by adding more layers to the network. Increasingly deep NN are enabled by techniques to utilize graphics processing units (GPUs) to rapidly accelerate the training



(a)

(b)

Figure 1.1: Models have rapidly grown in the past decade terms of (a) the number of parameters, and (b) computations required for training [4].

process. Figure 1.1 shows how AI models have grown in number of parameters, which gives a sense of how model complexity has grown over time. Moreover, Figure 1.1 also shows the growth in the number of compute operations required for training these models as they have grown more complex. For many neural networks, these operations are dominated by multiply-accumulate operations for the matrix multiplication required by convolutional and fully connected layers. While hardware designers have valiantly attempted to keep pace with the growing complexity, this is typically limited to high-end datacenters and industries that can afford the frequent upgrade cycles and high cost of bleeding-edge hardware. This is not practical in the context of *pervasive* AI, where the always-available nature

implies mobile and edge devices that are constrained in size (die area), power (thermal), energy (battery life), and connectivity (WiFi, cellular, etc).

Early implementations of AI services targeted for edge or mobile applications typically employed a cloud service architecture. The sensors on the edge device, such as microphones and cameras, would collect data which would be forwarded to the cloud. This data would then be run through models by cloud servers, which are less constrained in available processing power, and the results are delivered back to the requester. This allowed the requester to take advantage of the latest models and hardware within the limited computation and energy footprint of the edge device. However, this scheme relies on the communication link between the edge device and cloud service provider, which induces an additional communication latency. Particularly in mobile situations, there is no guarantee that the link is available, and even if it is, the bandwidth may fluctuate. This can lead to unreliable task latency or outright task failure.

In addition to latency and energy considerations, shifting processing from the cloud to edge devices has been proposed as a way to preserve user privacy. By executing the neural network locally, a user avoids sharing their personally identifiable information, such as appearance, voice, or location, with a potentially-untrusted third party. Even *partial* computation on the edge has shown the ability to obfuscate features to preserve privacy when the remainder of the network is executed in the cloud [7].

An additional social impact of AI services is the carbon emissions generated by the energy consumed. This is a key concern as carbon emissions are a leading factor

in global climate change and contributes to a wide range effects including increased food scarcity and community displacement [8]. Although the energy demands of AI services show no signs of slowing, intelligently selecting compute powered by low carbon emission sources may help mitigate the impact on global climate change. Devices at the edge are geographically dispersed and may effectively participate in finding lower-carbon energy sources. Microsoft has explored edge carbon at the scale of all Windows 11 PCs by enabling "carbon aware" software updates [9].

Today, we have a confluence of factors: a rapidly-growing base of IoT devices, increasingly complex models for processing their data, and growing user expectations for features and quality of service (QoS) guarantees. For instance, in 2021 Apple modified it's voice assistant so that voice processing is done locally [10]. They claim this helps safeguard personal information and significantly improve the voice assistant's performance, but was only recently deployed as it requires capabilities of modern processors to efficiently compute these complex voice models. This is just one small, albeit prominent, example of adjusting the distribution of AI tasks to dictation processing at the edge.

Today, we have seen generative AI applications take center stage and become pervasive in business and personal life. These are the latest target for shifting inference towards the edge for increased responsiveness and privacy [11]. However, there are typically significant trade-offs for these edge-deployed versions of generative AI which require reduced model complexity and customized hardware support to enable edge inference. Even as edge AI has become increasingly deployed, cloud connectivity is still required for the latest and greatest models. As the demand for

AI services continues to grow, developing distributed systems beyond the existing cloud service architecture will be essential to managing the scale of compute required and fulfilling user expectations.

2 RELATED WORKS ON DISTRIBUTED INFERENCE

With Edge AI becoming more prevalent, there have been a variety of ideas proposed to optimize the inference process while reducing or eliminating the requirement for centralized cloud compute. In this section, I will explore several approaches that have been proposed. First, I will explore techniques that primarily target a minimization of inference latency. Then, I will discuss techniques that incorporate an energy consideration, typically combining latency and energy into a joint metric to be minimized. I will also discuss how dynamic voltage and frequency scaling (DVFS) has been used as a knob of control to explore energy-latency tradeoffs. Then, I discuss works that focus on sustainability by optimizing a carbon emissions metric before wrapping up with a summary of shortcomings of existing approaches.

2.1 Latency Optimization

The work [12] presents MoDNN which investigates a method to reduce the latency of inference among mobile edge devices. The MoDNN framework targets convolutional and fully connected layers, distributing the work among a set of heterogeneous devices in a wireless local area netowk (WLAN) environment. It initially profiles the "compute ability" of each node. These profiles are used to equitably partition the computation work of a single layer among the worker nodes. We call such a strategy *intra*-layer partitioning, where the partitions are inserted within each layer and the execution of a single layer can be distributed to multiple workers. MoDNN requires syncing information from the feature maps among

devices between each layer, introducing a per-layer communication delay, though it exploits particular properties to reduce the required network traffic. This approach reduces overall latency, compared to a single device, even when accounting for the new transmission delays introduced. However, this assumes a robust, dedicated WLAN to ensure the transmission delays do not exceed the reductions in computation time.

The DeepThings [13] framework is another intra-layer partitioning approach that utilizes a *work stealing* algorithm to distribute work among edge nodes. The layers are partitioned into work chunks by a gateway, without knowledge of the workers, and workers steal chunks as they are available. DeepThings optimizes the chunks to maximize data reuse that minimizes communication and show their approach is more scalable to a dynamic environment compared to MoDNN.

In an edge-cloud environment, one approach proposes a technique for joint accuracy and latency aware decoupling (JALAD) of deep network structures to minimize latency with the smallest impact on accuracy [14]. This decoupling approach belongs to a category of works we call *inter*-layer, because the partitions are inserted between layers, so each layer is executed only on one device. This work recognizes that intermediate feature maps may be larger than the input, discouraging transmission to the cloud. However, the intermediate feature maps are often sparse and amenable to further quantization. Compression via quantization is lossy and may reduce accuracy, but can improve latency by making additional partition points feasible. JALAD utilizes an an integer linear program (ILP) model to minimize execution and transmission latency, incorporating compression, subject

to a bound on accuracy loss. It is able to improve the latency of popular models, compared to uploading the raw image or an image compressed as PNG, particularly at low bandwidths where transmission delay has the most impact.

Another work on reducing latency in edge-cloud environments proposes adaptive distributed deep neural network (DNN) acceleration (ADDA), a method to split the work between an edge sensing device and cloud compute resources [15]. This approach *modifies* the DNN to include multiple exit paths. This approach is based on the observation that, in many DNN, early feature maps are sufficient for classification so not all inputs require processing by the full network. After training the multi-path DNN and characterizing the likelihood of exiting at each exit point, ADDA exhaustively searches the possible assignments of partition points and picks the point that minimizes latency. This technique requires that the edge device hosts a *complete* NN, a subset of the original DNN. This technique focuses on minimizing the *average* latency by early-exiting inputs, which may be suitable for applications where throughput is the primary concern but individual inference QoS is not a requirement.

An orthogonal technique that we briefly explore to reduce latency is to modify DNNs to be "early exiting". Early exiting techniques recognize that certain inputs are "easier" for the DNN and may be correctly classified after only a partial subset of layers. BranchyNet [16] is one implementation of this idea, that inserts "branches" which are alternate classification paths after early layers of a DNN. This technique does not reduce latency in the worst-case, but it does in the average-case because a portion of inferences end early without having to process through all layers.

2.2 Energy Optimization

The cooperative edge (CoEdge) framework [17] is an intra-layer partitioning approach, much like MoDNN, that incorporates an energy metric when considering the partitioning problem in an offline local network of edge devices. It formulates adaptive workload partitioning as an ILP that minimizes the overall execution *energy* subject to a deadline constraint on latency. Note that, in this kind of objective, a higher-level understanding of the system QoS requirements is necessary to set a good deadline because latency may be increased to minimize energy. Experiments are conducted in a mixed edge network of Raspberry Pi 3, NVIDIA Jetson TX2, and desktop PC and show that, under the same deadline constraints, their approach reduces the energy required compared to other approaches, even execution on a single Raspberry Pi that does not incur communication energy costs.

JointDNN [18] looks at the optimal point to insert transitions when considering a (mobile, cloud) environment for inter-layer partitioning. They view the NN as a computation graph and formulate their approach as a shortest-path problem, where each path is encoded with cost in terms of both latency and energy. The shortest path problem is solved by optimizing an ILP to minimize latency, energy, or a combination of both. Their approach extends to both inference and training, and shows an ability to effectively use their minimization technique to improve the metric they study in a deployed model.

Table 2.1 summarizes these works, including which metric(s) they attempt to optimize and the partitioning strategy used.

Name	Latency Opt.	Energy Opt.	Partitioning Strategy
MoDNN [12]	X		Intra
DeepThings [13]	X		Intra
JALAD [14]	X		Inter
ADDA [15]	X		Inter
CoEdge [17]	X	X	Intra
JointDNN [18]	X	X	Inter

Table 2.1: Comparison of Related Works in Distributed Inference optimizing latency and energy

2.3 Dynamic Voltage and Frequency Scaling

Dynamic voltage and frequency scaling (DVFS) is a feature that is common across many devices nowadays which allows the device to operate at multiple voltage levels and frequencies. Several works have explored the utility of DVFS in edge-to-cloud offloading. In [19], the authors propose a task scheduling algorithm to distribute tasks between edge and cloud resources, using DVFS as a final step in their scheduling algorithm to tune the energy consumption while still meeting a latency deadline. DRLDO [20] solves a similar problem, scheduling tasks to edge and hub devices. DRLDO utilizes a reinforcement learning approach to train an agent to make scheduling and DVFS decisions, which may be useful to adapt quickly in a dynamic environment, and show energy savings compared to default governors that manage DVFS. Neither of these works study DNNs in particular, which have unique computation and communication patterns that may be amenable to more specific optimizations. Indeed, works like AsyMo [21] formulate a specific approach for managing DNN inference on mobile devices because of the unique challenges and opportunities in DNN tasks and mobile hardware.

NeuOS [22] operates dynamically in a multi-DNN application on a single device, monitoring DNN task progression against their deadlines and adjusting the DVFS setting to ensure task deadlines are met while meeting energy and/or accuracy goals. Energy Efficient Neural Networks (EENet) [23] combines the idea of an early exiting network, like BranchyNet [16], with DVFS control. EENet keeps track of batches of inference tasks with a deadline, dynamically adjusting the DVFS settings in response to individual inference tasks exiting early, taking advantage of this dynamic slack to minimize energy.

DVFO [24] examines how to control the CPU, GPU and memory frequencies of an edge device in an edge-cloud environment. DVFO introduces a novel method for offloading to reduce latency and, like DRLDO [20], utilizes a reinforcement learning approach to constantly monitor the execution environment to make dynamic DVFS and offloading decisions to optimize energy and latency. MOSAIC [25] recognizes that edge systems are designed with systems-on-chips that contain functional units such as GPU and neural processing unit (NPU) in addition to CPU. MOSAIC formulates a model "slicing" approach using dynamic programming techniques to assign layers to particular functional units and determine the operating frequency of those functional units to optimize different objectives such as latency or energy.

DNNs present unique opportunities to take advantage of DVFS features to optimize latency and energy on edge devices. DVFS has been explored in edge-hub-cloud environments, though very few works address the intersection of DNN, DVFS, and an edge-hub-cloud system.

2.4 Carbon Footprint

Carbon emissions are closely related to energy use, but aim to quantify the environmental impact more directly by utilizing sustainability metrics such as grams of carbon dioxide produced (gCO2). The carbon emissions depend not only on how much energy is used, but what sources are used to generate that energy so techniques that just minimize energy may indirectly reduce carbon.

Several works have addressed carbon emissions in edge-hub-cloud systems. Many works consider that the cloud may have multiple, geographically-distributed nodes and attempt to optimize the carbon emitted by cloud by assignment of tasks to the cloud nodes. DECA [26] utilizes a multi-phase optimization technique based on A^* with Fuzzy Sets to jointly optimize cost and carbon. In [27], a similar problem for task placement on hub and cloud resources is modeled as a mixed-integer linear program to optimize carbon emissions of the application running on hub and cloud.

LSCEA-AIoT [28] formulates a carbon-aware approach to assigning tasks to edge or hub, utilizing a Monte Carlo Tree search strategy to determine the assignment solution. They point out the importance of re-solving the optimization hourly to track changes in carbon intensity from energy providers. In [29], a unique aspect of carbon is addressed: "carbon emission rights" which are purchased to allow producing gCO2 and whose price fluctuates over time. The work deploys small, less-accurate DNNs on edge devices and large, more-accurate DNNs in carbon-intensive data centers and attempts to optimize accuracy while minimizing emission rights costs.

2.5 Areas of Improvement

When considering intra-layer approaches, the reduced latency is primarily due to the increased parallel computation of a single layer. These approaches incur communication overhead between layers to share overlapping feature map information. These intra-layer approaches are typically implemented in a cloud-less context, where the fast local networks mean the communication overhead is small relative to the parallel computation benefit. Because these works lack access to cloud or other highly-performant devices, they do not **minimize latency in a global sense**.

Works that utilize inter-layer partitioning strategies tend to consider heterogeneous environments that may include edge, hub, and cloud devices. Such works often consider a strict edge $\rightarrow$ cloud architecture with only one partition point and do not consider an opportunity to overlap work across multiple devices. Inter-layer approaches derive their latency reductions by optimally utilizing highly performant devices. They don't incur the same per-layer communication overheads as intra-layer techniques. This is more amenable to cloud-enabled contexts, where a relatively slow network discourages communication bottlenecks but the large performance asymmetry can reduce the overall latency.

Many of the approaches that explore DVFS as a knob of control utilized reinforcement learning as their optimization strategy. While this allows the system to adapt dynamically and learn over time, the overheads of maintaining a reinforcement learning agent are not well-characterized. The potential latency and energy overhead of such an agent may be significant and limit the utility of these approaches in constrained edge devices. Many works are also specifically formu-

lated to target systems that impose task deadlines and it is unclear how they will perform under other objectives or system assumptions. Overall, there is a gap in the literature addressing inter-layer assignment while including DVFS as a knob of control, which provides optimization opportunities unique to DNNs.

While there is significant interest in the sustainability of distributed systems, the existing literature focuses on application placement in geographically-dispersed hub or cloud nodes. Very few approaches consider the edge device as a potential source of compute and integrate that into the carbon optimization methodology. However, edge devices are even more geographically dispersed and may allow increased opportunities to tap into lower-carbon energy sources. And, to my knowledge, no work specifically examines the sustainability tradeoffs of inter-layer partitioning for DNN inference.

3 SUMMARY OF CONTRIBUTIONS

In this chapter, I highlight the key contributions of this work in optimizing layer assignment for distributed inference of a Deep Neural Network (DNNs). A DNN is type of AI application that is comprised of multiple layers, each with their own memory and compute requirements. Given a set of heterogeneous devices, including edge, hub, and cloud devices, this work formulates an approach to model and optimize the assignment of individual layers of the DNN to each device to optimize various objectives, including latency, energy, and carbon footprint.

In Chapter 5, I present a *base* Integer Linear Program (ILP) formulation for the layer assignment problem. This formulation improves over similar, existing works by allowing any number of heterogeneous devices. Our formulation also allows any number of transitions between devices, which is particularly applicable to edge-to-edge scenarios not considered in other works. Compared to prior works, the formulation does not make assumptions about the types of devices considered or restrict transitions points. This flexible formulation allows it to be a fundamental building block for exploring layer assignment strategies in future work.

In Chapter 6, I discuss our work that introduces a novel optimization technique for inter-layer partitioning in distributed inference. This technique allows opportunities to overlap compute and data movement, and we refer to this as weight preloading. Based on simulation results, we find that this technique has significant benefit for edge devices which tend to be slow at data movement.

In Chapter 7, we identify and detail an issue with considering latency profiles in the base ILP of this work and prior works. We demonstrate that, for certain devices, the optimization process may select the wrong latency profiles in the global latency expression. To resolve this, we introduce a novel constraint on the basis of profile "bundles" that guarantees the selection of the correct (corresponding) latency profile. Based on our profiling of edge, hub and cloud devices, we find that prior work may generate the wrong layer assignment solution with significant errors in some cases.

In Chapter 8, we explore the device operating frequency as a knob of control in latency and energy tradeoffs. We first develop an energy profiling methodology that incorporates the static selection of operating frequency and demonstrate that varying operating frequency allows latency energy tradeoffs. We write modifications to the base ILP to include device frequency selection as part of the optimization process by associating the frequency selection with the latency and energy profiles. In our simulation results, we find solutions for latency-constrained cases that can significantly reduce energy as well as energy-constrained cases that can significantly reduce latency.

In Chapter 9, we address DNN sustainability by examining carbon footprint, a metric correlated to environmental factors. We first adapt the base ILP and system model to consider a long-running application, instead of a single inference instance. We introduce carbon intensity parameters from public sources to the formulation to translate energy use to carbon emissions. In our experiments, we demonstrate that minimal energy and minimal carbon footprint solutions may be very different in

some cases. We consider the edge-edge case, which is not well studied in literature but has promise as edge devices are numerous and widely dispersed.

4 SYSTEM MODEL FOR DISTRIBUTED INFERENCE AND GLOBAL LATENCY

In this chapter, I introduce and describe the general system model that will be considered through the remainder of this work. While variations may introduce specialized extensions or focus on a particular subset of this model, I maintain consistency with the definitions presented in this chapter. First, I introduce the characteristic aspects of compute hardware considered in this work. Then, I examine the DNN inference workload and model that is considered.

I also introduce two approaches to determining the 'global', or end-to-end across all devices utilized, latency of inference. Global latency is a consistent aspect throughout this work. First, an approach based on an analytic model is presented that uses high-level features of the DNN and the compute device. Next, I discuss a profiling-based method for measuring latency per device to estimate global latency. The two methods are then compared. This chapter lays the groundwork for the DNN layer assignment problem that will be modeled via integer linear programming in subsequent chapters.

4.1 Hardware System Model

As discussed in Chapter 1, DNN inference compute may be distributed across heterogeneous devices in a network to take advantage of their varying compute and memory capabilities. In particular, we consider a distributed system that takes

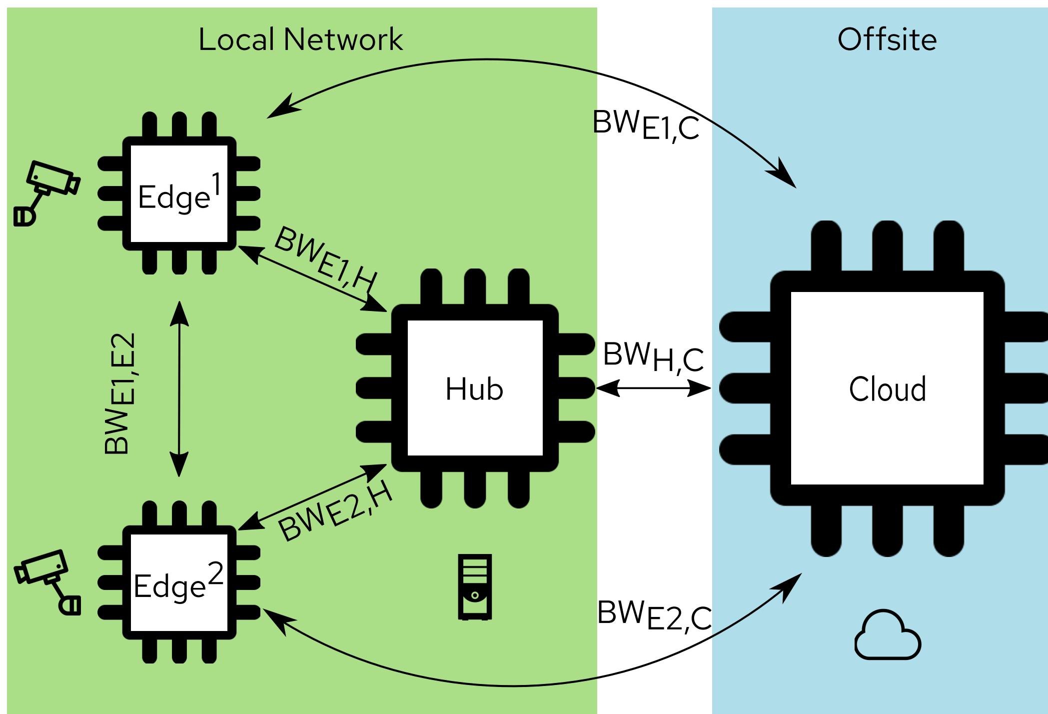


Figure 4.1: Example of an environment with 2 edge, 1 hub, and 1 cloud device for a hypothetical vision application.

the form presented in Figure 4.1. This system contains three key classes of device, the edge, hub and cloud. We consider these classes of devices as:

1. *Edge*: A sensing and compute device with significantly constrained compute and memory capacity.

2. *Hub*: A compute device located very near the edge device on the same local area network (LAN), with moderately constrained compute and memory capacity.

3. *Cloud*: A compute device located far from the edge device connected over wide area network (WAN), with minimally constrained compute/memory capacity.

Note that these definitions are primarily based on network proximity to the data collection, and the compute and memory resources are a secondary feature. We also do not constrain the class to a particular type of hardware (e.g. CPU, GPU) for generality.

Additionally, our model considers each pair of devices to have a unique communication bandwidth. For instance, in Figure 4.1, we denote the bandwidth between the two edge devices E_1 and E_2 as BW_{E_1,E_2}, the bandwidth between E_1 and the hub device H is denoted as $BW_{E_1,H}$, and the bandwidth between the hub device H and cloud C is denoted by $BW_{H,C}$. These bandwidths could all be different but in practice: $BW_{E_1,E2} \geq BW_{E_1,H} > BW_{H,C}$

We highlight several devices used in this work and some of their key features in Table 4.1 in order to give a sense of the relative capabilities of the edge, hub, and cloud classes. The floating point operations per second (FLOP/s) can give the relative compute capability, the RAM and GPU memory give the relative memory capability, and the memory speed and external bandwidth give their relative communicate capability. We consider the LePotato as an edge device, NVIDIA Jetson Nano as a hub device, and a server with NVIDIA 2080Ti GPU as a cloud device.

We also need a means of coordinating between devices to share the layer assignments in the distributed system. To do this, we propose the flow in Figure 4.2, with four key steps to set up the distributed inference. This flow is executed prior to

Table 4.1: High-level device features considered in this work.

	LibreComputer LePotato	**NVIDIA Jetson Nano**	**NVIDIA 2080TI**
CPU	4-Core Arm Cortex-A53	4-Core Arm Cortex-A57	16-Core Intel i9-9900K
GPU	*Unused*	128 Maxwell CUDA Cores	4068 Turing CUDA Cores
RAM	2GB	4GB	64 GB
GPU Memory	*Unused*	4GB (shared)	11 GB
Max Power	5W	10W	250W

the first inference so that the layer assignment is known to all devices. This flow introduces the concept of an *orchestrator* role, which should be assigned to a device with sufficient compute capability to solve the ILP. In practice, either hub or cloud should be sufficient to serve the orchestrator role.

In Step 1, the NN model is sent to the orchestrator, which collects details about the operations required for each layer in the DNN. In Step 2, the available devices in the network send information about their performance characteristics and device-to-device bandwidth to the orchestrator. In Step 3, the orchestrator generates the ILP model based on the information provided in Step 2 and solves the layer assignment to optimize the objective function. In Step 4, the orchestrator sends the layer assignment information back to the devices. Each assigned device receives the layer(s) to be executed on it. In addition, for each layer, the ID of the "next device" will also be sent.

After the above steps, the layer assignment plan is complete and distributed inferencing may begin. While network conditions remain stable, the same plan can

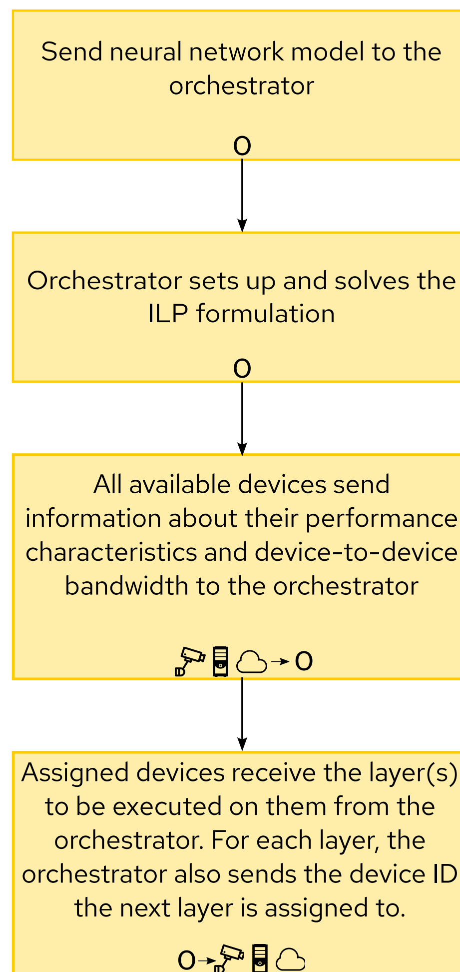


Figure 4.2: Our proposed optimization workflow has four key steps. An orchestrator collects information from each device, then generates a layer assignment solution that is re-used over many inferences.

be reused without invoking the orchestrator. Because the plan is reused, we do not consider the flow to generate the layer assignments as part of the inference latency. As the conditions of the distributed system, such as available network bandwidth, may vary over time, the above steps should be repeated periodically to update

the assignment under the new conditions. Methods to detect these variations are outside the scope of this work.

In this work, we consider a distributed system model that contains heterogeneous devices of varying capability. We categorize this into three distinct classes of devices: edge, hub, and cloud. An orchestrator device takes into account the compute, memory, and communication capability of all devices in the distributed system and generates a layer assignment by optimizing an ILP model of the problem. This hardware system model is flexible to any number of devices in each class, including no devices of a particular class. This will allow us to explore many unique scenarios.

4.2 Deep Neural Network System Model

In this section, I provide a brief introduction of DNN architectures and the features salient for layer partitioning.

A NN is a machine learning algorithm that is inspired by the organization of the human brain. It attempts to mimic the highly-connected nature of information communication between neurons [30, Ch. 6]. A *deep* NN is a design that utilizes multiple NN to perform calculations on some input data. These operations are "stacked" and each individual section is referred to as a "layer". Depending on the type of layer, it may contain parameters called "weights" that interact with the input. Layers with weights may optionally have parameters known as "bias" which

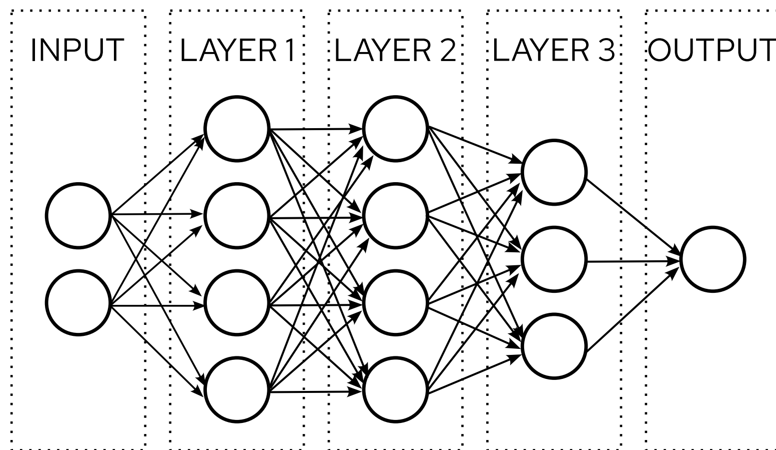


Figure 4.3: Illustration of a deep neural network with three linear layers. Each node (circle) represents a single element of a tensor. The arrows represent multiplication by a weight, which is then accumulated into the output tensor (input of the next layer).

do not interact with the input. A DNN with an input with two features, three layers, and a single output is demonstrated in Figure 4.3.

The input, intermediate, and output tensors (multi-dimensional arrays) are defined with four parameters: width, height, depth (sometimes referred to as "channels") and batch size. For example, a typical color image is defined by number of pixels in the x-direction (width), number of pixels in the y-direction (height), and number of color channels, commonly 3 for images defined with red, green, and blue (depth). If four images are packed together, this tensor has a batch size of four. Figure 4.4 demonstrates a batch of three images of width 5 and height 4.

The layers of a DNN define the actual computation work to be done on the inputs. In this work, we will focus on four key types of layers commonly seen in DNNs for image-based tasks:

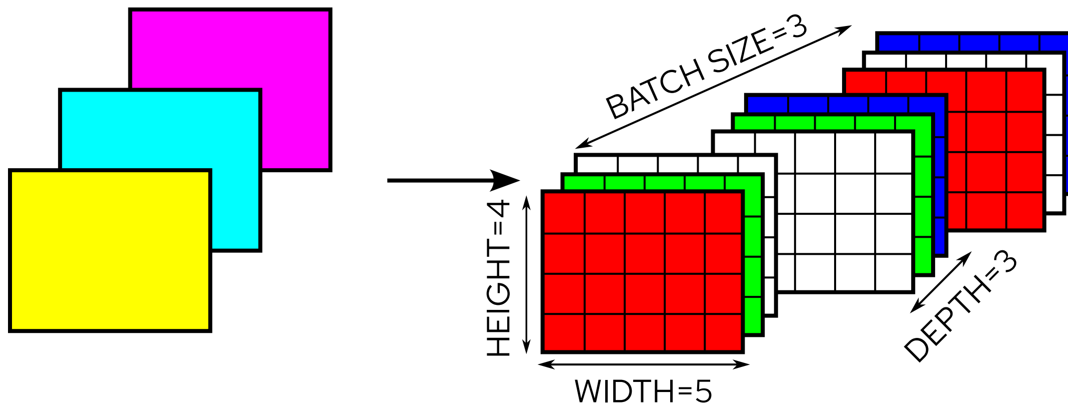


Figure 4.4: Input tensors are defined with four dimensions: height, width, depth and batch size. On the left, there are three images of yellow, cyan, and magenta. On the right, we demonstrate that each image has a height of 4 pixels, width of 5 pixels, and depth of three for the red, green and blue channels of an RGB image. When these are combined into a single tensor, it has a batch size of 3.

1. Convolutional Layer: This layer type performs a convolution operation, "convolving" a *kernel* across the input. The parameters of the kernel are the "weights" of the convolution layer. Optional bias parameters are added to the output without dependence on input. The kernel *weights* are typically much smaller than the input and these layers are often compute-bound, not memory-bound.

$$out = \sum^{K}(weights \star in) + bias$$

   The above gives the general form of the operation of the convolution layer, where $\star$ is the convolution operator (adapted from [31]) and K is the number of convolution kernels. The output is the sum of the result of all K kernels and the optional bias. Note that the summation is affected by several parameters

beyond the scope of this overview, particularly as we are concerned with inter-layer behaviors, not intra-layer.

2. Linear Layer: This layer type performs a linear operation, multiplying inputs by a vector of weight parameters and accumulating the results. A bias may be added. This layer is also referred to as "fully-connected" because each output is connected to every input by a weight. The number of weights is typically larger than the size of the input and these layers are often memory-bound, not compute-bound.

$$out = weights \cdot in + bias$$

The above gives the general form of the linear operation (adapted from [31].

3. Activation Layer: This is a class of layers that introduce non-linearity. This includes tanh ($out = tanh(in)$), sigmoid ($out = \frac{1}{1+exp(-in)}$), and ReLU ($out = \max(0, in)$) activation functions [31]. These activations functions are typically nonlinear which allows DNNs to learn complex functions not limited to a linear combination of input features [32].

4. Dimension-Adjusting Layer: This is a class of layers that alters the dimensions of a tensor, including operations such as MaxPool that reduce dimensions and Zero Padding that expand dimensions [31].

In this work, we focus on a specific sub-class of DNNs, those that are strictly one-to-one feedfoward. This means that the output of one layer is sent to exactly one subsequent layer. This excludes n-to-n feedfoward DNN architectures such as

those with skip connections (e.g. ResNet [33]), inception modules (e.g. GoogLeNet [34])[1]. This also excludes stateful networks (e.g. recurrent neural networks [30, Ch. 10]).

DNNs Considered

In this work, we utilize several DNNs in our distributed inference scenarios. A high-level summary of these DNNs is provided in Table 4.2. Number of layers is reported as the number of layers that we consider for the partitioning and layer assignment problem. Note that not all DNNs are used in all works.

Table 4.2: Summary of key DNN features

Name	Num. Layers	Weights	Task
AlexNet [6]	8	61.1M	Image classification
VGG11 [35]	11	132.9M	Image classification
VGG16 [35]	16	138.4M	Image classification
SSDlite [36]	35	3.4M	Object detection
ViT [37]	16	86.6M	Image classification

4.3 Analytic Latency Model

One approach to estimating the DNN inference latency is to utilize characteristic features of the DNN model, compute devices, and network to determine how long

[1]This is not a strict limitation of the problem formulation, but we focus on one-to-one for simplicity in the formulation. It is also often possible to simplify DNNs because, in many of these models, the n-to-n connections are within larger blocks that have one-to-one connections. For instance, ResNet has one-to-one connections between the residual blocks. The problem could instead be formulated as a block (instead of individual layer) assignment problem, trading off granularity for simplicity.

the distributed system will take to compute an inference result. We consider three components of latency: (1) latency to compute a layer on a device, (2) latency to load the weights of a layer on a device, and (3) communication latency to send the results of a layer from one device to another.

Compute Latency

Let $t^c_{l,d}$ indicate the latency to compute layer ℓ on device d. We estimate the latency for layer ℓ to be computed on device d as:

$$t^c_{l,d} = \frac{(\text{FLOP})_\ell}{(\text{FLOP/s})_d} \tag{4.1}$$

where $(\text{FLOP})_\ell$ indicates the number of floating point operations to compute layer ℓ, and $(\text{FLOP/s})_d$ indicates the number of floating point operations per second that device d is capable of handling. The $(\text{FLOP/s})_d$ can be determined from the manufacturer specification. The $(\text{FLOP})_\ell$ can be calculated from the layer hyperparameters. In our work, we utilize the tool `ptflops` [38], which reports operations as multiply-accumulate (MAC) so we double this number to estimate $(\text{FLOP})_\ell$. This is a useful and convenient metric for estimating performance [14], [15] and is sufficient to capture latency estimates among heterogeneous devices for layer assignment optimization purposes.

Weight Latency

For each device, we assume it has a main memory for storing weights and computation results for the NN layers that are assigned to it. We refer to the *internal* bandwidth of a device to be the bandwidth of this main memory, as shown in Figure 4.5. In our formulation, it is denoted as $\text{BW}_{d,d}$, indicating the bandwidth for a device to "communicate" with itself.

Let $t^{w}_{\ell,d}$ denote the latency to load the weights of layer l on device d. It is determined by dividing the size corresponding to the weight parameters of layer l by the memory bandwidth of device d, as:

$$t^{w}_{\ell,d} = \frac{w_\ell \times \alpha}{BW_{d,d}} \tag{4.2}$$

where w_ℓ is the number of weights in layer ℓ and α is the number of bytes required to store each weight. In the remainder of this work, we use $\alpha = 4$ to represent 32-bit floating point number. This parameter can be changed to account for quantization, a common technique for reducing the size and compute complexity of a DNN.

Communication Latency

The output of each layer must be communicated to the input of the subsequent layer in the feedforward DNN. We call this the communication latency. This is the latency to send the output of layer ℓ computed on device d to the device d' which computes $\ell + 1$. Note that d and d' may be the same device, as in Figure 4.5 or different devices, as in Figure 4.6. This communication latency is denoted as $t^{x}_{\ell,d,d'}$

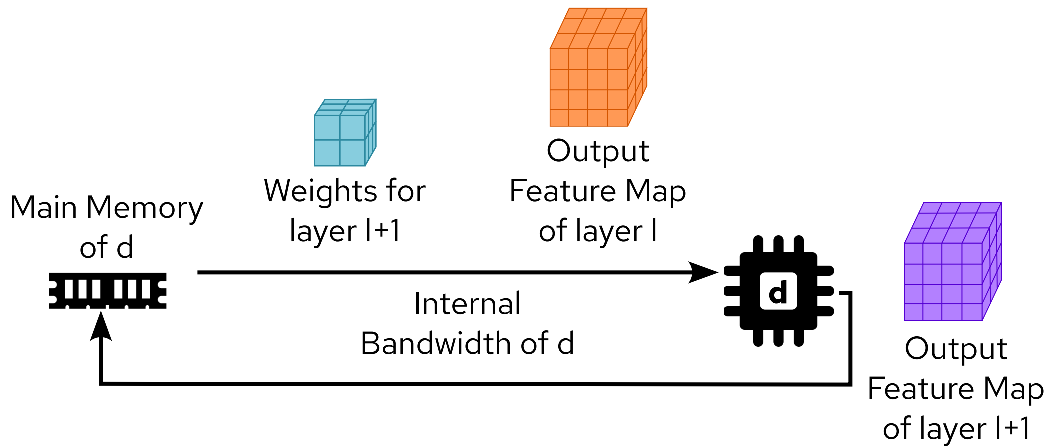


Figure 4.5: Internal latency models in which a single device model with access to main memory loads the assigned layer weights and/or results of previous computations at the rate of the *internal* bandwidth of the device.

and estimated as:

$$t^{x}_{\ell,d,d'} = \frac{H_\ell \times W_\ell \times D_\ell \times \alpha}{BW_{d,d'}} \tag{4.3}$$

where H_l, W_l, and D_l are the height, width and depth of the output feature map of l, respectively, and α is the number of bytes required to store each output element, as defined in Equation 4.2.

The parameter $BW_{d,d'}$ reflects the bandwidth between d and d'. In the case that $d \equiv d'$, we use the internal bandwidth of the device. Otherwise for $d \neq d'$, we estimate it as the minimum of external bandwidths of d and d' and the bandwidth of the network itself, as in Figure 4.6. Estimation of the network bandwidth itself is beyond the scope of this work; it has been an active topic of research, typically by utilizing dynamic probing techniques [39], [40], and more recently using machine learning techniques [41]. Because network bandwidth may be dynamic, particularly

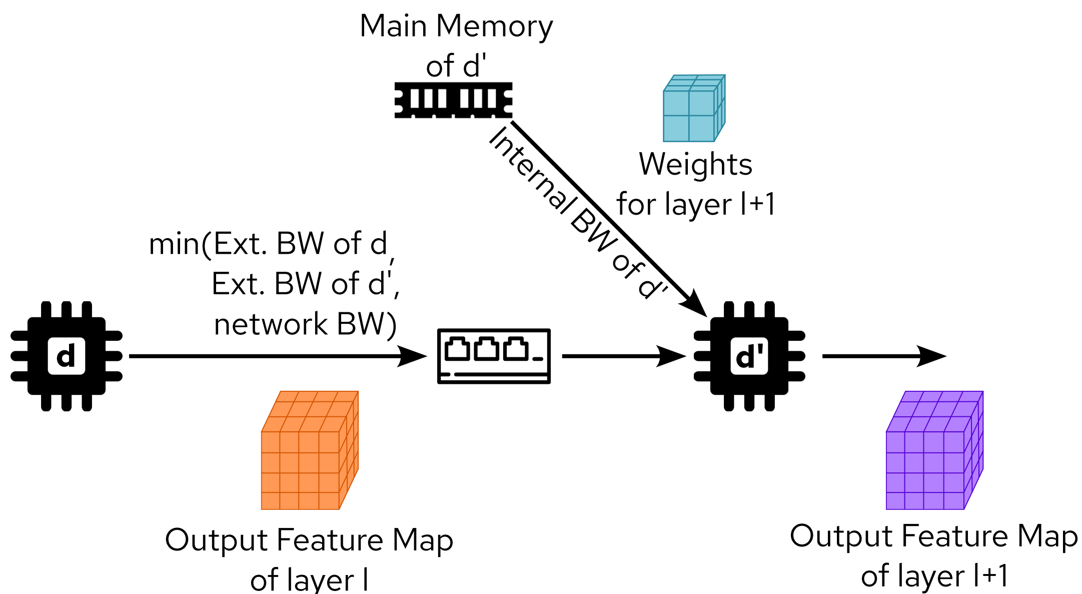


Figure 4.6: External latency model in which two devices communicate across the network to send results of layer l (from device d) to compute layer $l+1$ (on device d'). The rate of transfer between d and d' is the minimum of the external bandwidths of the two devices and the network conditions.

in edge environments relying on wireless links, the ILP can be set up and solved again quickly by the orchestrator as shown in Figure 4.2.

4.4 Profiling-Based Latency

Another approach to estimating the execution latency for DNN is to profile the latency of the individual layers and estimate the overall execution latency as a sum of these profiled components. After profiling is complete, the latencies are accessed as a lookup table and the overall latency can be quickly estimated.

In this work, we utilize the following profiling methodology. We focus on DNNs that are targeted at image classification, so we utilize a subset of the ImageNet [5] dataset to ensure we exercise the inference with realistic input values. Specifically, the ImageNette [42] dataset which is a ten-class subset of ImageNet. Two images are randomly selected from each of the ten classes, for 20 unique inputs. In order to profile layers in isolation, prior to profiling we generate all intermediate data for each image.

With the sample data prepared, the layers can be individually profiled by timing the execution latency of each layer on each device. For profiling CPU execution, Python's built-in `time` is used [43]. For profiling GPU execution, CUDA events are utilized through the PyTorch API [31]. Images are inferenced one-at-a-time (batch size of 1). Profiles are collected for each image, and this process is run five times for a total of 100 profile samples. To determine a single $t^c_{\ell,d}$ value, a trimmed mean is used that discards the top 10% and bottom 10% of values. This helps to remove outliers, particularly early samples that are not representative of GPU performance due to well-known "warm-up" effects of GPUs.

To the best of our knowledge, profiling tools at this high-level don't offer an ability to separately profile the weight loading and compute portions of the system model. Due to this, the weight loading and compute latencies for profiled systems are collapsed into a single latency. In otherwords, for profiling, let $t^c_{\ell,d}$ implicity also contain $t^w_{\ell,d}$.

When profiling in this manner, the number of profiles required is

$$L \times D \tag{4.4}$$

where L is the number of layers in the DNN and D is the number of device types in the distributed system. For large networks, this profiling effort becomes quite significant. To manage the efficiency of profiling, we do not profile every layer, but instead profile between "major" layers, convolution and linear. This means that, for instance, instead of profiling a convolution, ReLU activation, and pooling layer separately, we profile them as a single unit. The activation and dimension-changing layers typically require orders of magnitude less compute and are not good candidates for partition points. In addition, frameworks like PyTorch can perform "operator fusion" to merge certain operations to reduce memory access and improve performance [31]. Typical fusions include an convolution or linear layer followed by activation, such as `ConvReLU2d`. By only splitting the DNN for profiling at the "major" compute layers, we also preserve opportunities for operator fusion [2]. In the case of AlexNet, this reduces the number of profiles from 21 layers to 8 layers.

[2]Though we preserve opportunities for operator fusion, we do not utilize operator fusion in our profiling for generality across frameworks and hardware that may not support fusion.

4.5 Comparison of Latency Models

I compare the two approaches to modeling latency, the analytic approach and the profiling approach.

In the analytic approach, the latencies can be estimated quickly via straightforward mathematical formulas. For instance, for AlexNet, it took approximately 0.2 seconds to generate all of the required latency parameters for an edge-hub-cloud case [3]. In contrast, from the profiling approach, the first layer is determined to be approximately 0.02 seconds in execution latency, discovered after 100 profiles, at minimum 0.2 seconds of profiling. The analytic model has a much smaller overhead for collecting latencies. In addition, it does not require access to the device for profiling efforts since it is based on published specifications, allowing the technique to be useful in prototyping.

However, analytical modeling makes many assumptions about the execution of the models based on the published specifications. For instance, the published FLOP/s is a *peak* number that may not be sustained throughout the entire computation. A similar caveat applies to the memory bandwidth. Profiling allows encapsulating any non-idealities by measuring the actual performance. We compare the analytic method and profiling method in Figure 4.7 for AlexNet running on three devices. We see a range in variation from the actual in both techniques. However, as we will address in Chapter 7, profiling can be expanded to consider bundles so that it always has a normalized latency of 1.0. On the other hand, the

[3]Based on Python 3.10 implementation running on Apple M1 Pro CPU.

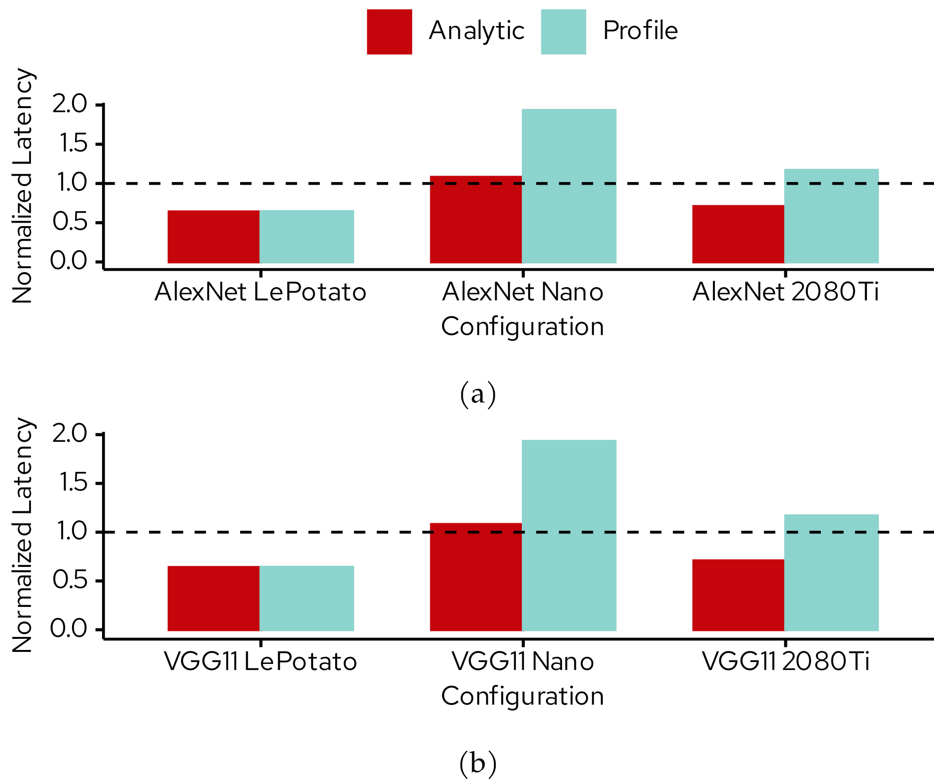


Figure 4.7: Estimated analytic and profiled latency for running (a) AlexNet and (b) VGG11 on LibreComputer LePotato, NVIDIA Jetson Nano, and NVIDIA 2080 Ti GPU. The latency is normalized to the actual latency of running the full network on the device, profiled similar to an individual layer execution.

analytic approach cannot be adjusted in such a way and is restricted to inaccuracy from the analytic model and assuming a simple sum of layer latencies.

A summary of this comparison is presented in Table 4.3.

Table 4.3: Comparison of latency estimation techniques.

	Analytic Model		Profile-Based Model
✓	Fast to estimate for new networks	✗	New networks require time-consuming profiling
✓	Can easily incorporate new devices	✗	Requires access to new devices for profiling
✗	Limited accuracy in global latency estimated values	✓	Ability to tune accuracy in global latency estimated values

5 BASE INTEGER PROGRAMMING MODEL FOR DISTRIBUTED INFERENCE

This chapter introduces the base ILP formulation that models the system presented in Chapter 4. This is referred to as the 'base' because it will be utilized and extended in future variants with various optimization objectives and constraints in subsequent chapters. The goal of the formulation is to find an optimal solution of assigning each layer of a feedforward DNN to a particular compute device. The formulation includes latencies, which are input parameters from the perspective of the ILP, and variables to be solved by optimization.

5.1 Notations

Let S_D denote the set of D available devices in the network. For each device $d \in S_D$, let $t^c_{\ell,d}$ and $t^w_{\ell,d}$ denote the latency parameters to compute layer ℓ and to load model weights from memory to chip, respectively. Also let $t^x_{\ell,d,d'}$ denote the communication latency to send the results of layer ℓ from device $d \in S_D$ to $d' \in S_D$. For an analytic latency model, these are computed using Equations 4.1-4.3. For a profiling-based latency model, these are extracted from the application profiles.

Let S_L denote the set of L layers in the given NN. Based on the application, we extend the NN by inserting *pseudo* layers, which allow us to encode additional information about communication and data preprocessing. These pseudo layers include:

- We allow designating a device as the *source*, e.g., because it may be physically hooked to a camera for image capture. Here, the NN is extended to start with a pseudo layer with 0 floating point operations (FLOPs) and an output tensor of the dimensions of the data collected by the source device. The purpose of this pseudo layer is only to capture the communication latency from the source device to the device which executes the next layer.

- In case pre-processing needs to be done on the collected data before sending it to the NN, we insert a pseudo layer right before the first layer of the NN. This pseudo layer will have a compute latency given by Equation 4.1 or captured by profiling, to reflect the pre-processing task. For example, common vision pre-processing tasks include normalization and cropping.

- We allow designating a device as *destination*, for example if it is required to receive the output computed from the last layer of NN in order to take action based on the inference result. This is modeled by extending the NN to have a pseudo layer of size $H_\ell = 1, W_\ell = 1, D_\ell = 1$ with 0 FLOPs at the end.

A pseudo layer is treated just as a layer in our formulation and modeled by adding it to the set S_L. Further, we denote S_{L-1} to be set of all layers excluding the last layer of the NN, and $S_{2,L}$ denote the set of all layers excluding the first layer of the NN.

The number of the weights utilized by layer ℓ is given as w_ℓ and number of bytes required by each weight is given as α, such that the number of bytes required for

the weights of layer ℓ is computed $w_\ell \times \alpha$. The size of the main memory of device d is denoted by M_d.

The formulation has three categories of optimization variables that are solved for during optimization. These are:

- Let binary variable $x_{\ell,d} = 1$ when layer ℓ is assigned to device d and 0 otherwise. This variable identifies the layer assignment solution.
- Let binary variable $z_{\ell,d,d'} = 1$ when layer ℓ is assigned to device d and layer $\ell + 1$ is assigned to device d'. This variable identifies when a communication latency between d and d' should be added to compute the overall inference latency. Note $z_{\ell,d,d'} \neq z_{\ell,d',d}$. It allows capturing solutions where two or multiple devices may compute the layers back and forth among each other.

5.2 Optimization Constraints

After defining the parameters and optimization variables, we must introduce optimization constraints to ensure a valid solution. Recall, for our base formulation, we assume a feedforward neural network in which a particular layer relies on inputs only from the prior layer. This is expressed with the constraints given below.

$$\sum_{\forall d \in S_D} x_{\ell,d} = 1 \quad \forall \ell \in S_L \tag{5.1}$$

$$\sum_{\ell=0}^{L} (w_l \times \alpha) x_{\ell,d} \leq M_d \quad \forall d \in S_D \tag{5.2}$$

$$0 \leq x_{\ell,d} + x_{\ell+1,d'} - 2z_{\ell,d,d'} \leq 1 \quad \forall l \in S_{L-1} \, \forall d, d' \in S_D \tag{5.3}$$

Constraint 5.1 indicates that each layer must be assigned to only one device. Constraint 5.2 ensures the weights of the layers assigned to a device fit within the size of its main memory. These constraints enforce the correct computation of each layer in the target NN.

Constraint 5.3 sets $z_{\ell,d,d'}$ when layer ℓ executes on device d and the output is sent to d'. So $z_{\ell,d,d'} = 1$ only when $x_{\ell,d} = x_{\ell+1,d'} = 1$. This constraint (consisting of two inequalities) expresses a logical AND operation and is written for all combinations of devices in S_D and layers in S_{L-1}. This constraint enforces the communication pattern of the feedforward NN.

In addition to the above generic constraints, specific constraints may be added to preassign layers to devices. For example, if a source device is designated, the first pseudo layer may be preassigned to the source. To preassign a layer ℓ to device d, we simply set $x_{\ell,d} = 1$ in our formulation.

5.3 Latency Objective Function

For a latency minimization objective function, we utilize both the latency parameters and the optimization variables to express the actual latency of inference. We express the overall latency as the sum of the compute, communicate, and weight loading latency: $T = T^c + T^x + T^w$, where

$$T^c = \sum_{\ell=1}^{L} \sum_{\forall d \in S_D} t^c_{\ell,d} \, x_{\ell,d} \tag{5.4}$$

$$T^x = \sum_{\ell=1}^{L-1} \sum_{d \in \forall S_D} \sum_{\forall d' \in S_D} t^x_{\ell,d,d'} \, z_{\ell,d,d'} \tag{5.5}$$

$$T^w = \sum_{\ell=1}^{L} \sum_{\forall d \in S_D} t^w_{\ell,d} \, x_{\ell,d} \tag{5.6}$$

T^c expresses the compute latency for a given assignment of x. T^x expresses the cost to communicate the outputs of layers for a given assignment of x between devices across the network. T^w expresses the latency to load weights for a given assignment of x.

Note, in Equation 5.5, all possible device-to-device communication latencies are considered and summed if the corresponding z variable is assigned to 1. Since $y_{\ell,d,'d} \neq y_{\ell,d',d}$, both d and d' are varied across all the devices.

The latency minimization objective is therefore $\min(T)$. After optimization, the $x_{\ell,d}$ variables will indicate the layer assignment solution that minimizes latency by best taking advantage of compute and network resources across the heterogeneous devices in the distributed system. As we show in further experiments, the choice of

this assignment is not trivial and depending on factors such as network conditions and types of devices, the assignment of the layers of the *same* NN could be different in a distributed setting.

5.4 Conclusion

In this chapter, I formulated the system model introduced in Chapter 4 as an ILP problem. This includes input parameters for latency, which can be determined as described in Section 4.3-4.4. Optimization variables and constraints are also introduced to enforce the hardware model, Section 4.1, and DNN model, Section 4.2. This model is written generally to accommodate any feedforward DNN and any number of heterogeneous devices. The flexibility and modularity of this formulation allows for straightforward incorporation of extensions to the system model or novel objective functions.

6 ODIWEP: OPTIMIZING DISTRIBUTED INFERENCE WITH WEIGHT PRELOADING

In this chapter, I discuss a modification to the base ILP of Chapter 5 to consider opportunities to overlap compute and data movement in the global distributed inference scenario. We identify and utilize an opportunity to preload weights for inference when they would otherwise be idle. This is based on work *Neural Network Partitioning for Fast Distributed Inference* published at International Symposium on Quality Electronic Design 2023.

The contributions of this technique to incorporate weight preloading in distributed inference include:

- Formulating the analytic model to estimate the inference latency of inference across heterogeneous devices.
- A novel modification to the base ILP (Chapter 5) to incorporate weight preloading in the inference model.
- Demonstrate through experimental results on popular CNNs that weight preloading can provide significant latency savings in particular device topologies.
- We also explore the layer assignment problem in relation to "early-exiting" networks, an orthogonal technique for optimizing inference across heterogeneous devices.

Our experiments include multiple configurations of possible edge, hub, and cloud devices. We also experiment with a variety of network bandwidths to consider the impact of network bandwidth on layer assignment solutions. Our results demonstrate that the layer assignment problem is very sensitive to network bandwidth. We also identify topologies that benefit most from the technique of weight preloading.

6.1 Motivation

One of the significant bottlenecks of DNN inference is loading the model weights from memory, t^w. This is particularly true for edge devices which may have slow memory bandwidths, as in the Raspberry Pi 3 in Table 6.1. Figure 6.1 plots the sources of latency in the analytic model, communication ($t^x_{\ell,d}$), compute ($t^c_{\ell,d}$) and weight loading ($t^w_{\ell,d}$). We observe that weight loading can be a signifcant factor in the overall latency of inference. Eliminating any source of latency will allow additional optimizations in our objective of minimizing distributed inference.

We also recognize that our scheduling for latency minimization is one of many techniques to model and minimize inference latency. We are interested in exploring how orthogonal techniques may complement our approach. In particular, we investigate how networks that allow for early-exiting, discussed in Section 2.1, may benefit from orchestrated layer assignment and weight preloading.

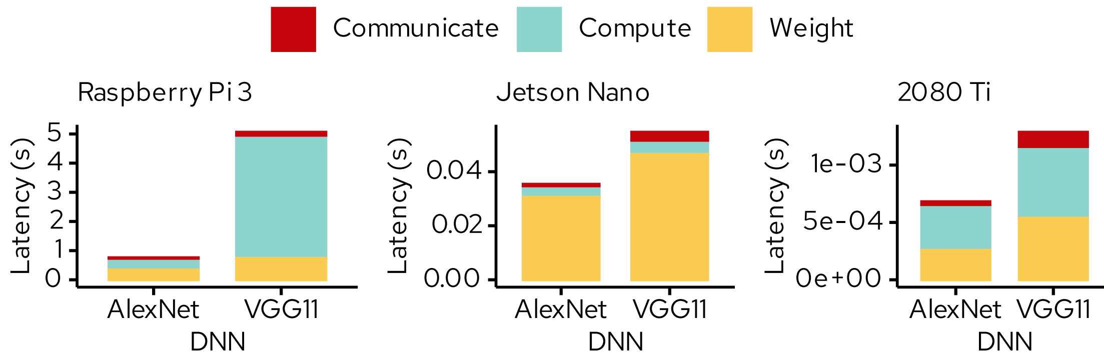


Figure 6.1: Breakdown of sources of latency in single-device execution, according to the analytic latency model. Shown for three devices, the Raspberry Pi 3, Nano, and 2080Ti executing two DNNs, AlexNet and VGG11.

6.2 Opportunity for Preloading Weights

Recall in our proposed flow (Figure 4.2) that the orchestrator decides the layer assignment prior to start of inference so that each device in the distributed system is aware of where to send its results. We assume a batch size of one and no queuing of tasks, as may be considered for a real-time application which demands results before moving to the next task. In the model proposed in Chapter 4, every device in S_D except for the device currently executing is idle. Because the orchestrator distributes the layer assignments prior to inference, each device knows which layer will be the first layer it executes prior to actually receiving the input data for inference. That is, in the external latency model shown in Figure 4.6, device d' is able to load the weights for layer $\ell + 1$ while device d is busy computing layer ℓ. Because device d' would otherwise have been idle, with weight preloading, we can eliminate $t^w_{\ell+1,d'}$ from the global latency.

6.3 From Base ILP to ODIWeP

Based on the opportunity to reduce global inference latency with weight preloading, we modify the base ILP (Chapter 5) to incorporate preloading opportunities. In order to do this, we frame the approach as "$t^w_{\ell,d}$ is only incurred when ℓ is not the first layer executed on d".

Let $q_{\ell,d}$ be a binary variable where $q_{\ell,d} = 1$ when there is *not* an opportunity to preload the weights of l on device d because $l-1$ is also executed on d, and 0 otherwise. A new constraint is written for q,

$$0 \leq x_{\ell-1,d} + x_{\ell,d} - 2q_{\ell,d} \leq 1 \quad \forall \ell \in S_{2,L} \; \forall d \in S_D \tag{6.1}$$

This constraint expresses a logical AND operation and written for all combinations of devices in S_D and layers in $S_{2,L}$. This constraint enforces $q_{\ell,d} = 1$ when $x_{\ell-1} = x_\ell = 1$, meaning there is no opportunity to preload the weights of ℓ. This variable captures only the mandatory weight loading costs. Note that this constraint resembles the communication constraint Equation 5.3, because both are interested in identifying transitions between devices.

From the base ILP, we also modify the T^w computation as:

$$T^w = \sum_{\ell=1}^{L} \sum_{d=1}^{D} \, t^w_{\ell,d} \, x_{\ell,d} \dashrightarrow T^w = \sum_{\ell=1}^{L} \sum_{d=1}^{D} \, t^w_{\ell,d} \, q_{\ell,d} \tag{6.2}$$

The computation of T^w now depends on $q_{\ell,d}$ instead of $x_{\ell,d}$. In this case, preloading is utilized when $q_{\ell,d} = 0$, which means $t^w_{\ell,d}$ is *not added to* T^w because weight loading is hidden by a device that would otherwise be idle.

The remainder of the base ILP is left as described in Chapter 5.

6.4 Results

In this section, I present the results of our experiments utilizing the analytic latency model and incorporating the weight preloading technique. In our experiments, we simulate varying network conditions by changing the device-to-device communication bandwidths. This approximates real-world conditions, such as a cellular (4G/5G) network which may vary in bandwidth due to other user's utilization, physical obstructions between radios, etc.

We incorporated devices which are representative of a typical cloud inference architecture to model a distributed system. This included (1) a data collection edge device with limited CPU processing (Raspberry Pi 3B); (2) a low-power hub device with low GPU compute capacity (NVIDIA Jetson Nano); and (3) a cloud service with high GPU compute capacity (NVIDIA 1080TI).

Device parameters were determined from publicly-available benchmarks and/or the manufacturer specification and are summarized in the table below. Parameters include the GFLOP/s, internal and external bandwidths of a device which are used to compute the computation, weight loading, and communication latencies as defined in Chapter 4. The specific parameters for the devices are listed in Table 6.1.

Table 6.1: Summary of device characteristics utilized in weight preloading simulations.

Device	GFLOP/s	Internal BW (B/s)	External BW (B/s)
Raspberry Pi 3B	3.62 [44]	719×10^6 [45]	1.11×10^7 [45]
NVIDIA Jetson Nano	236 [46]	25.6×10^9 [47]	1.25×10^8 [47]
NVIDIA 1080TI	11300 [48]	484.4×10^9 [48]	1.25×10^8

With these devices, we define four distinct configurations of the distributed system.

C0) {A: Raspberry Pi3}: Non-distributed execution with a single lightweight edge device (for comparison purposes).

C1) {A: Raspberry Pi3, B: Raspberry Pi3}: Inference in a network of two homogeneous, lightweight edge devices.

C2) {A: Raspberry Pi3, B: NVIDIA 1080TI}: Inference with a lightweight edge device with access to a cloud GPU.

C3) {A: Raspberry Pi3, B: NVIDIA Jetson Nano, C: NVIDIA 1080TI}: Inference with a lightweight device, a performant device, and a cloud GPU. This configuration models presence of edge, hub, and cloud components.

In our experimental setup, we assume device A is the source and destination device for all the above configurations. This is done to model an application in which device A (i.e., Raspberry Pi 3) is always connected to a camera for image capture and must receive the results of the inference in the end to take an action. As explained in

Section 5.1 we model these by adding pseudo layers to the NN (with zero compute and weight loading latencies but non-zero communication latency) and preassign these pseudo layers to A.

Furthermore, we assume that the original image capture has a Standard HD dimensions of 3×1280×720 which represents the image quality of a typical commodity webcam or security camera connected to an edge device. Given that many image recognition CNNs operate on an input size of 3×224×224, we consider a preprocessing step to crop and downsample the HD image to this size before it is fed into the CNN. The preprocessing assumes one operation per pixel in each channel of the input image. This is also modeled as a pseudo layer in our formulation as explained in Section 5.1, but this layer is not preassigned to a device.

We compare the assignment results for the image classification task on AlexNet [6], VGG11, SSD [36], and VGG16 [35]. For this analysis, we trained models from the PyTorch [31] `torchvision` library. The optimization formulation was solved using the Gurobi solver [49]. We used an Ubuntu 20.04 virtual machine with 4 vCPU and 4.0GB of memory. The runtimes for initial setup and solving the ILP formulation are extremely fast; for example for configuration C_1, averaged across 5 runs, AlexNet took 0.17s and VGG16 took 1.67s.

Inference Latency and Ratio of Compute

In this experiment, we first compare configurations C_0, C_1 and C_2 for AlexNet, VGG11, and SSD. (For these networks and configuration C_3, we did not notice a

notable conclusion and results are not reported. Instead, we discuss C_3 with VGG16 which exhibits interesting results across *all* configurations.)

For C_1 and C_2 we vary the network bandwidth between the two devices in each configuration. For each bandwidth, we report the overall latency of distributed inference as measured using Equations 5.4, 5.5, 6.2 which includes compute, communication, and weight (pre-)loading latency.

We also report a ratio of compute (RoC) on each device. This metric measures the ratio of floating point operations performed on a device based on its assigned layers. More specifically, for configurations C_1 and C_2, we report the ratio of compute on device B and the rest of the computations are performed on device A. We denote these by $RoC_B^{C_1}$ and $RoC_B^{C_2}$, respectively. (For configuration C_0, we have $RoC_A^{C_0} = 1$ since everything is executed on a single A device.)

Figure 6.2 shows the results for AlexNet, VGG11, and SSD. We make the following observations:

- In the lowest network bandwidth, non-distributed execution on a single device is the best strategy. For these bandwidths, our ILP assigns all computations to device A for C_1 and C_2 which can be seen because $RoC_B = 0$ for these two configurations and this bandwidth.

- As the network bandwidth increases, distributed execution has a clear advantage. Both configurations C_1 and C_2 have a significantly lower inference latency compared to C_0 for all the networks.

- In the highest network bandwidth, configuration C_2 which additionally takes advantage of a cloud device significantly outperforms C_1 in terms of inference latency.
- Ratio of compute on device B changes as a function of bandwidth for C_1 and C_2; with increase of bandwidth, more computation is assigned to device B. (This also indicates the ILP solution varies with network bandwidth.)
- The choice of the best configuration for a given bandwidth can be different across the NNs. For example, at bandwidth 10^5B/s, single-device execution is the optimal choice in AlexNet and VGG11. However, at the same bandwidth for SSD, utilizing the cloud in configuration C_2 results in significantly lower inference latency.

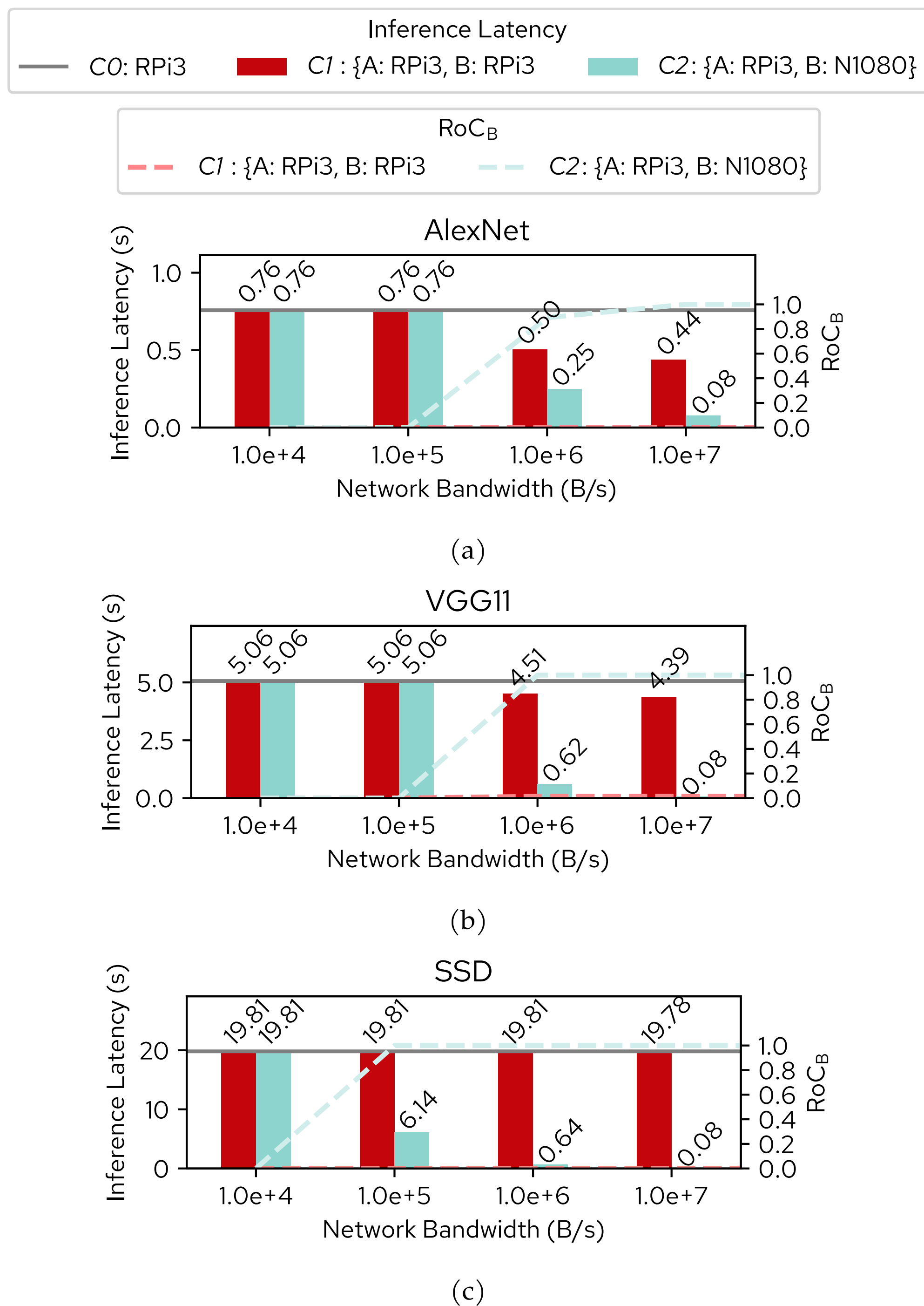


Figure 6.2: Results for configurations C_0 (non-distributed), C_1 (two edge devices) and C_2 (edge device and a cloud). We vary network bandwidth between the two devices in each configuration, and report the inference latency and ratio of compute on device B.

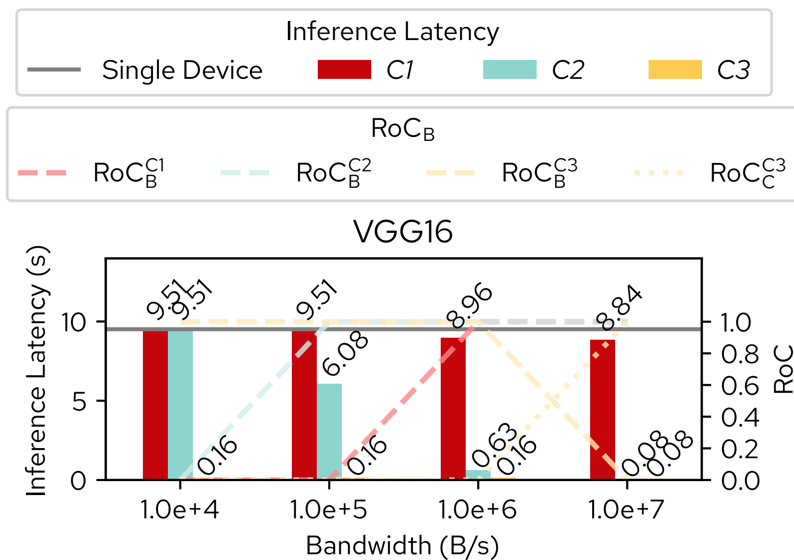


Figure 6.3: Comparison of all configurations in VGG16. An interesting observation is that at the low and intermediate bandwidths (10^4 and 10^5), configuration C_3 shows a significantly smaller inference latency compared to all other configurations (e.g., 0.16s versus 9.51s in 10^4 bandwidth). At these bandwidths, utilizing a hub device is the optimal strategy compared to using the cloud and single-device execution.

Figure 6.3 compares these metrics for *all* configurations in VGG16. Recall in configuration C_3, a hub device is present in addition to the edge and cloud devices in configuration C_2. Here, for C_3 we report ratio of compute on devices B and C; the rest are executed on device A. We denote these by $RoC_B^{C_3}$ and $RoC_C^{C_3}$, respectively. We make the following observations:

- Configuration C_3 has the smallest inference latency in *all* bandwidths (e.g., 0.16s in C_3 versus 9.51s in other configurations for the 10^4 bandwidth).

- At the two lowest bandwidths (10^4 and 10^5), the minimum latency is when all computations are offloaded to the hub device in C_3, i.e., $RoC_B^{C_3} = 1$. This is because the hub device is more compute-efficient than the edge. Also communicating with the cloud in C_2 results in a higher latency.

- At the highest network bandwidth (10^7), the minimum latency is when offloading all computations to the cloud and not utilizing the hub ($RoC_C^{C_3} = 1$). (At this bandwidth, C_2 and C_3 generate effectively the same solution.)

Overall, from our experiments using four NNs, we can derive the following generic conclusions: Inference latency in a distributed setting can significantly vary depending on the network devices (e.g., availability of edge, hub, and cloud), device-to-device communication bandwidth, as well as the characteristics of the NN itself. Our ILP formulation is able to capture all the above factors to generate a layer assignment plan that minimizes the overall latency.

Effect of Preloading on Latency

Table 6.2 shows the breakdown of different contributors to the inference latency (denoted by L) in configuration C_1 at the peak network bandwidth of 10^7 B/s. The columns indicate the breakdown of the individual latency components, compute (T_c), communication (T_x), and loading of weights (T_w), as computed using Equations 5.4-5.6. These are reported without and with our proposed preloading technique to understand the source of preloading's benefit.

For AlexNet, we observe a 42% reduction in latency (from 0.76s to 0.44s) because we trade a 0.01s increase in communication (T_x) for a 0.33s reduction in weight loading (T_w). VGG16 and VGG11 show a similar trade off, though the reduction in latency is smaller, 7% and 13% respectively, because the bulk of latency is *computation* which does not benefit from preload. For SSD however, we do not see a significant advantage in preloading because the compute latency significantly dominates the preloading latency.

This capturing of preloading is another contribution of our work which was not done in prior work, and overall can be seen can result in significant benefit for some networks.

Table 6.2: Inference latency, without and with the preloading.

	No Preload				Preload			
	T_c	T_x	T_w	L	T_c	T_x	T_w	L (% decr.)
AlexNet	0.40	0.02	0.34	0.76	0.40	0.03	0.01	0.44 (42%)
VGG11	4.22	0.11	0.74	5.06	4.22	0.12	0.05	4.38 (13%)
SSD	19.29	0.33	0.19	19.81	19.29	0.35	0.14	19.78 (0.2%)
VGG16	8.57	0.18	0.77	9.51	8.57	0.19	0.08	8.84 (7%)

Results for An Early Exiting CNN

We evaluate our optimization procedure when our formulation accounts for an early exit strategy, as described in Section 2.1. We follow the approach in [16] and modify AlexNet to have one early exit point inserted after layer `conv2`, consisting of two convolution and one fully connected layers. We modify from [16] to utilize a

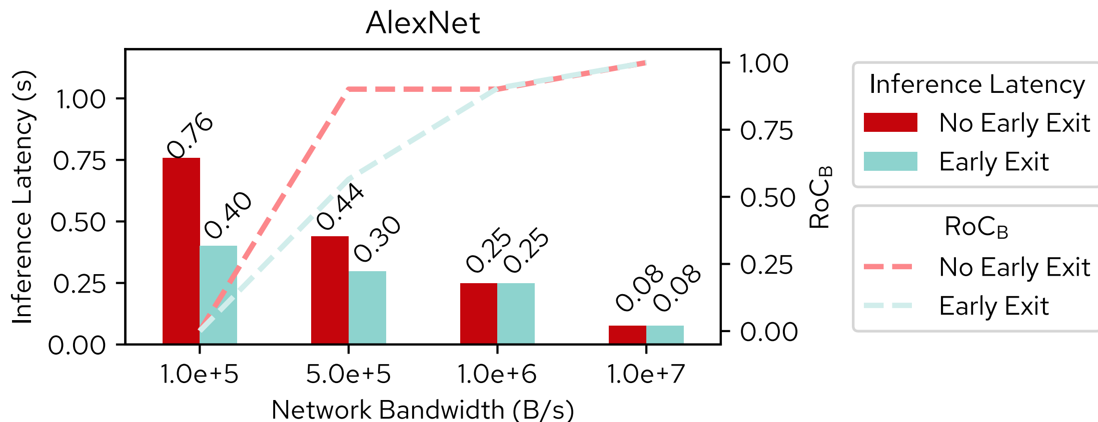


Figure 6.4: Results for early exit following the setup in [16]

$3 \times 224 \times 224$ input, as in our other experiments. The probability of early exit after conv2 was set to 0.656, assuming similar performance as [16].

We used configuration C_2 to evaluate early exit starting from a source device (Raspberry Pi 3) with access to a cloud GPU (NVIDIA 1080TI). Figure 6.4 reports the inference latency and RoC$_B$ as the network bandwidth is varied. As expected, early exit may reduce inference latency, especially at lower bandwidths. At higher bandwidths, we find no benefit to early exit because it will be faster to execute the CNN without the branch on the cloud GPU. When network bandwidth is high, the overhead of additional early exit classification layers outweighs the benefits.

At the network bandwidth of 5×10^5B/s, we observe the maximum difference in RoC$_B$ between the two CNN structures. In this case, our optimization assigns layers `conv1` and `conv2` of the CNN as well as the three layers associated with early exiting to device A. The entire early exiting portion gets executed before execution transitions to device B. The RoC$_B$ reduction, from 0.90 to 0.57, reflects the ability

of early exiting to keep computation local by providing the opportunity to classify early in the CNN. This solution aligns with the intent of the design in [16] we base our extension on.

6.5 Conclusion

In this chapter, I presented a use case for our analytic latency model. By breaking down the latency into compute, weight loading, and communication, we revealed an opportunity to reduce global latency by overlapping compute and data movement to hide weight loading latency on some layers by preloading weights. We showed in our experiments that when minimizing inference latency in a distributed setting, the layer assignment solution can significantly vary depending on the network devices and device-to-device communication bandwidths. We also showed that the benefits of preloading the layer weights can be quite significant in NNs where the latency to compute a layer is comparable to the latency to load its weights. Further, we demonstrated that our base ILP is highly flexible and amenable to incorporation of orthogonal techniques such as early-exiting to explore optimal layer assignments.

7 DIME: DISTRIBUTED INFERENCE MODEL ESTIMATION FOR BUNDLE PROFILING

In this chapter I introduce the use of bundle-based profiling which improves the accuracy of the overall objective function compared to strictly layer-wise profiling. This is based on work *DIME: Distributed Inference Model Estimation for Minimized Profiled Latency* published at IEEE International Conference on Smart Computing 2024.

In this chapter, we first make the key observation that despite collecting all possible latency profiles, `JointDNN` [18] is still susceptible to generating sub-optimal solutions. This is due to the profiling behavior exhibited by some devices, which can cause the ILP of `JointDNN` to incorrectly select profiles used for latency estimation by its optimization variables. In particular, for devices whose smaller-bundle latencies underestimate larger-bundle latencies, `JointDNN` *incorrectly assigns a larger-sized layer bundle to execute on a device based on an underestimated latency of smaller-sized bundles, when minimizing the latency objective.*

This work is motivated by this above novel observation about `JointDNN`, which also applies to our base ILP (Chapter 5). Based on this observation, we formulate a modification to the base ILP which is not subject to this shortcoming. These contributions include:

- We propose `DIME`, which incorporates novel modifications to this base ILP and guarantees correct estimation of latency of layer bundles within the optimization formulation, regardless of device type.

- We additionally introduce an input parameter called maximum bundle size (MBS) to our formulation. It controls the tradeoff between the device profiling effort and quality of solution (i.e., distributed latency) in `DIME`.
- Conduct simulations across a range of network communication bandwidths and show `DIME` always generates the optimal solution, unlike `JointDNN`.
- For some network bandwidths, `DIME` finds an optimal solution with significantly lower profiling effort when using a small MBS (e.g., 22 versus 132 device profiles).

7.1 Motivation

In our baseline profiling-based latency approach (Section 4.4), recall that we profile the execution latency of individual layers. For each layer $\ell \in S_L$, we collect the $t^c_{\ell,d}$. Then, if multiple layers are sequentially executed on the same device, we assumed the sum of the individual latencies is a good approximation for the sequential execution of layers. In this section, we investigate this assumption and find that it does not hold for all devices and can cause sub-optimal layer assignment solutions.

In Figure 7.1, we plot the latency of AlexNet network [6] when estimated using different combinations of layer bundles, for a representative subset of combinations. Here, a "bundle" is the sequential execution of layers on a single device as a single unit of work which executes layer i through layer j. This is done for two devices: the NVIDIA 2080TI GPU and LibreComputer LePotato, a single-board computer. We segment the DNN at the boundaries of convolution and fully-connected layers

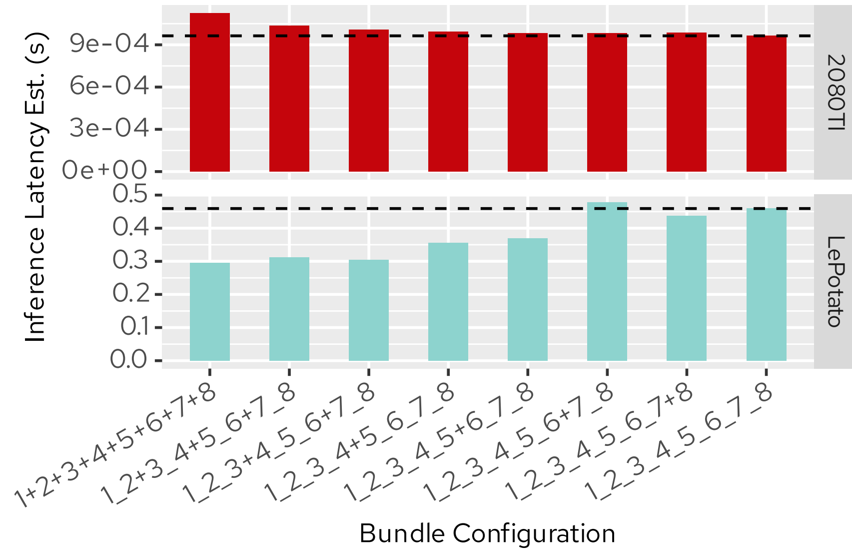


Figure 7.1: Example showing inference latency of AlexNet when estimated as sum of different profiled bundles for two different devices. A bundle configuration of '1_2+3' would indicate the sum of profiled latency for a bundle of layers (1,2) with the profiled latency of layer 3. The right-most bar (bundle of 1_2_3..._8) is the accurate latency profiling all layers as one bundle.

because the latencies of these layers dominate computation latency. Prior works also show that transitions between devices tend to happen only at these points [14], [50] and this is an acceptable tradeoff to reduce the profiling complexity.

To compute the profile of each (i, j) bundle, we use the PyTorch [31] Sequential model to create a feed-forward bundle of layers i through j. We profile 100 runs of this bundle, and take the average of the runs after discarding the top 10% and bottom 10% of observations to eliminate potential outliers. This is similar to the profiling approach already discussed in Section 4.4.

From this, we anticipate a potential issue in the ILP minimization objective. It is possible for the base ILP to generate solutions based on layer $x_{\ell,d}$ assignments that a bundle has its latency computed using interior bundles of smaller sizes. For instance, two different assignments $y_{1,2,d} = y_{3,4,d} = 1$ versus $y_{1,4,d} = 1$ both indicate layers 1 through 4 are executed on device d. Therefore, $x_{\ell,d} = 1$ in both cases for $\ell = 1, \ldots, 4$. However, the computation latency on d is estimated by the ILP as $t^c_{1,2,d} + t^c_{3,4,d}$ in the former case, and as $t^c_{1,4,d}$ in the latter. The ILP actually picks the assignment which minimizes its objective the most. For example it may assign $y_{1,2,d} = y_{3,4,d} = 1$ and $y_{1,4,d} = 0$ if $t^c_{1,2,d} + t^c_{3,4,d} < t^c_{1,4,d}$. When the device actually executes the layer assignment, it will do so with latency $t^c_{1,4,d}$.

That is, when the sum of interior bundles is less than the largest bundle containing those interior bundles, the ILP minimization objective will utilize interior bundles to estimate latency even though the profile for the encompassing bundle was available to more accurately represent the computation latency.

In Figure 7.1, we observe that for the LePotato edge-class device, the sum of layers tends to significantly underestimate the execution latency of the whole network (right-most bar). The underestimation exists even when using larger interior bundles. We also observe that for the 2080TI cloud-class device, the sum often overestimates the latency of a bundle. The causes of under- and over-estimation behavior are outside the scope of this work, but our results support the claim in `JointDNN`[18] that larger bundles are more accurate and *add* that this is true in both under- and over-estimation cases.

Under the latency minimization scheme of the base ILP, using the LePotato profiles would result in setting, $y_{1,1,d}, y_{2,2,d}, \ldots, y_{8,8,d}$ variables to 1 instead of $y_{1,8,d}$ even though this selection significantly underestimates the true latency captured by $t^c_{1,8,d}$.

7.2 Bundle-Based Profiling

Recall that the base ILP formulation requires latency parameters $t^c_{\ell,d}$ corresponding to the compute latency of a single layer ℓ on particular device d[1]. Thus far, this work has considered the total computation latency to be a sum of the computation and weight latencies of each individual layer, Eqs. 5.4. However, this may not accurately capture all inter-layer effects that could lead to the actual latency of multiple layers differing from the sum of individual layers.

To overcome this, works like [18] will profile the execution of multiple layers in sequence, as bundles. The compute latency parameter is adjusted as below to allow defining t^c with a start layer i and end layer j.

$$t^c_{\ell,d} \dashrightarrow t^c_{i,j,d} \tag{7.1}$$

We also introduce a binary variable, $y_{i,j,d}$: This binary variable is 1 when a bundle of consecutive layers i through j is assigned to device d. When $i \equiv j$, the bundle includes only one layer. Having $y_{i,j,d} = 1$ does *not* mean that consecutive

[1]For profiling-based latency methods, recall t^w is implicitly captured in t^c.

layers i to j are the largest bundle that is assigned to device d; layers immediately before i or after j may also be assigned to device d.

With profiled bundles, we adjust the computation of T^c to considered the bundles assigned to each device, not just the individual layer assignments. That is, Equation 5.4 becomes

$$T^c = \sum_{\ell=1}^{L} \sum_{\forall d \in S_D} t^c_{\ell,d} \times x_{\ell,d} \dashrightarrow T^c = \sum_{\forall d \in S_D} (\sum_{i=1}^{L} \sum_{j=i}^{L} t^c_{i,j,d} \times y_{i,j,d}) \quad (7.2)$$

All possible layer bundles (i, j) with $(1 \leq i \leq j \leq L)$ are listed for each device d. The y variable is multiplied by the corresponding t^c parameter reflecting the corresponding compute latency to execute the bundle on that device. The expression inside the parenthesis expresses the total compute latency on device d.

An additional constraint is introduced to ensure that each layer is associated with only one non-zero $y_{i,j,d}$, *if* it is assigned to device d.

$$x_{\ell,d} = \sum_{\forall i,j \mid 1 \leq i \leq \ell \leq j \leq L} y_{i,j,d} \quad \forall d \in S_D; \forall \ell \in S_L \quad (7.3)$$

This sums all unique $y_{i,j,d}$ bundle variables which include layer ℓ in them $(1 \leq i \leq \ell \leq j \leq L)$ and sets the $x_{\ell,d}$ variable. This constraint is written for all combinations of devices $d \in S_D$ and layers $\ell \in S_L$.

When profiling bundles, for each device d we must obtain the $t^c_{i,j,d}$ latency parameters for all combinations of $1 \leq i \leq j \leq d$. This requires $\frac{L \times (L+1)}{2} \times D$ profiles, compared to $L \times D$ when just considering single-layer profiling. The profiling effort

grows quadratically with the number of layers in the DNN and grows linearly with the number of different devices in a heterogeneous network.

7.3 From Base ILP to DIME

As we discovered in Section 7.1, profiling bundles can lead to more accurate estimation of the latency but under-estimating devices are subject to bundle selection during the optimization process that does not match the real execution model. In order to address this issue, we propose adding the following constraint to the bundle-based ILP which is written for each device:

$$\sum_{i=1}^{L}\sum_{j=i}^{L} y_{i,j,d} \leq \lceil \frac{\sum_{\ell=1}^{L} x_{\ell,d}}{L} \rceil \quad \forall d \in S_D \tag{7.4}$$

In the above inequality, the left-hand-side is sum of the binary variables $y_{i,j,d}$ on each device d for all layer combinations $1 \leq i \leq j \leq L$. This right-hand-side numerator is the *number* of layers assigned to device d. The denominator is number of layers L in the DNN. The expression on the right-hand-side will evaluate to 1 if at least one layer of the DNN is assigned to d and 0 otherwise. The inequality therefore requires the sum of all $y_{i,j,d}$ variables assigned to device d to be at most 1, *if* at least one layer is assigned to device d. Note that ceil ($\lceil x \rceil$) is a non-linear operation that can be approximated in a solver by the procedure in [51]

This forces the optimizer to set the $y_{i,j,d}$ variable with the smallest index for i and largest index j, and the rest of the $y_{i,j,d}$ variables to 0. In other words, the $y_{i,j,d}$

that is set to 1 corresponds to the largest-sized bundle assigned to device d which ensures the correct latency profile ($t^c_{i,j,d}$) is utilized in the latency objective function.

Adding Constraint 7.4 fixes the issue of selecting interior bundles instead of encapsulating bundles which we idenfitied when 'underestimating' devices are present in the network. When 'overestimating' devices are present, the added constraint will not cause an issue and will still enforce selecting the correct latency parameter.

Constraint 7.4 guarantees the use of correct latency parameters under the assumption of single-entry point to a device. We note, this assumption is also made in other prior works on ILP-based partitioning of DNN [14], [18], [50]. This is an acceptable assumption because often it is not practical to switch compute back to a slow (edge) device after switching to a faster (cloud) device when minimizing distributed inference latency.

As discussed in Section 7.2, the number of profiles required grows quadratically with the number of layers. As DNNs get deeper, this profiling effort increases dramatically. To control this, we introduce a new parameter called maximum bundle size (MBS) to be incorporated as input to the ILP. It sets the maximum number of consecutive layers which may be assigned the same $y_{i,j,d}$ variable. Specifically, only $y_{i,j,d}$ variables are defined in which $i \leq j \leq i + MBS - 1$. For example, when MBS=1, we require $j = i$, indicating only bundles of 1 layer may be used. This means, for a given MBS, only a subset of $y_{i,j,d}$ variables may be defined compared to the original bundle-based ILP, meaning only a subset of $t^c_{i,j,d}$ need to be profiled.

MBS allows the user to define a tradeoff between the device profiling effort and latency calculation accuracy.

To incorporate an MBS parameter, Constraint 7.4 may be generalized and replaced by the one below:

$$\sum_{i=1}^{L}\sum_{j=i}^{L} y_{i,j,d} \leq \lceil \frac{\sum_{\ell=1}^{L} x_{\ell,d}}{MBS} \rceil \quad \forall d \in S_D \tag{7.5}$$

In the above constraint, the right-hand-side expression computes an upper bound on the number of layer bundles that may be assigned to device d to estimate its latency. In Equation 7.4, the right hand side is strictly 0 (in the case that no layers are assigned to d) or 1 (in the case that any layer is assigned to device d). In Equation 7.5, the right-hand-side may vary between 0 and L, depending on the value of MBS, allowing multiple bundles to be assigned to a device in the case that the right-hand-side evaluates to greater than 1.

As mentioned, varying MBS is a tradeoff between latency accurarcy and profiling effort. The case when there is no reduction in accuracy is when MBS=L which requires profiling all devices for all combinations of bundles i through j with $1 \leq i \leq j \leq L$, the maximum profiling effort. In this case, Constraint 7.5 simplifies to Equation 7.4.

The case subject to the most accuracy reduction is when MBS=1. This case has the least profiling effort because only the $y_{i,j,d}$ variables with $i = j$ are defined. In this case, the only option that can be considered by the ILP is to select L bundles of

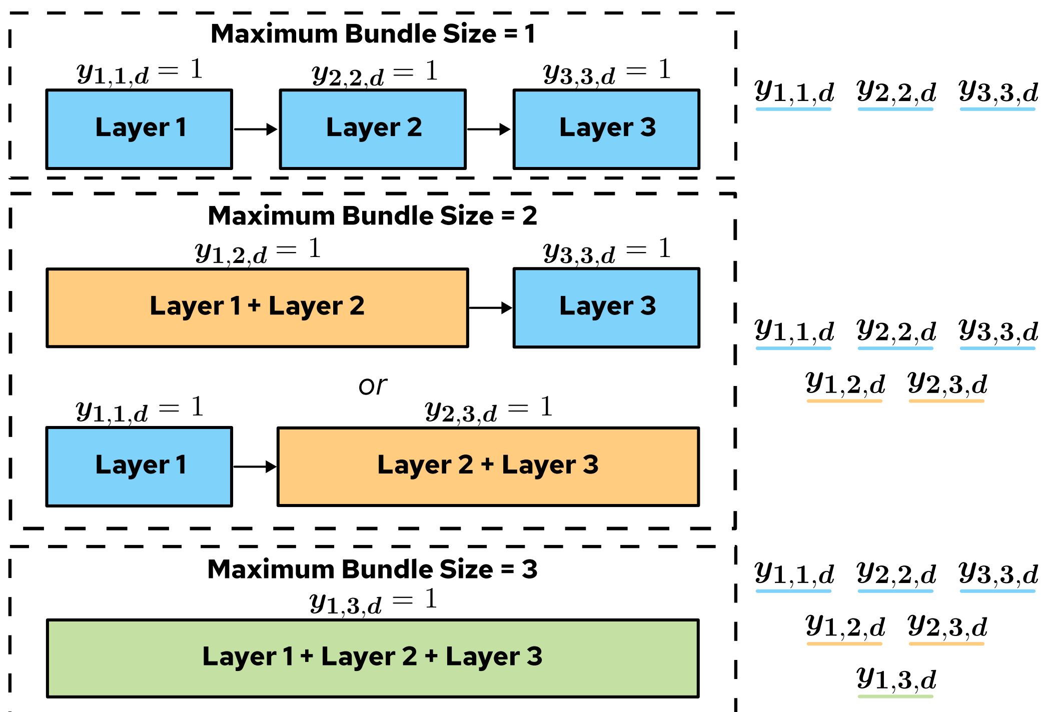


Figure 7.2: Example showing the impact of incorporating an MBS parameter in Constraint 7.5. For simplicity, we assume there is only one device available. The example shows how the $y_{i,j,d}$ variables (shown on the right) are assigned to define layer bundles (shown on the left) for three MBS values. When MBS=3, the ILP's objective is accurately computed but has the most profiling effort.

size 1 layer. This is the same as the approach prior to introducing bundles. More formally, the profiling effort changes [2]:

$$\frac{L \times (L+1)}{2} \times D \dashrightarrow \lceil \frac{L \times (MBS+1)}{2} \rceil \times D \tag{7.6}$$

[2]The ceiling operation is required in the MBS version because L and MBS vary independently and may create a numerator not divisible by 2. Previously, the numerator was guaranteed to be divisible by 2, and the ceiling operator could be omitted.

Figure 7.2 shows a toy example to demonstrate the impact of MBS for a neural network composed of 3 layers on one device. The figure shows the different possible solutions for MBS = 1, MBS = 2 and MBS = 3. This demonstrates how the bundle selection varies with MBS under our novel constraint, which impacts the latency calculation for the ILP. The profiling effort in each case is equal to the number of y variables shown on the right-side of the figure. In addition to just the increase in the number of profiles required, increasing MBS requires utilizing *larger* bundles. The profiling runtime does not grow linearly with the increase in the number of profiles, but superlinearly because the additional bundles profiled are larger.

7.4 Results

We present our results utilizing the `DIME` formulation and compare with the base ILP presented in Section 7.2. To compare with `JointDNN` [18] we used the same two-device setting (i.e., an edge and a cloud device). The Gurobi optimizer [49] was used to solve the ILP formulation. We also compare with the optimal solution determined via exhaustive search of all feasible layer assignments and using the correct device latency profiles based on the generated solution.

We examine the results for two popular neural networks, AlexNet and VGG11, when varying the bandwidth of the connection between devices. For the edge-cloud setup in our experiments, we use the LePotato (as edge) and NVIDIA 2080TI (as cloud) devices. We additionally present results of an experiment for VGG11 when adding NVIDIA Jetson Nano as an intermediate hub device to the distributed setup.

In our experiments, we also vary the input parameter Maximum Bundle Size (MBS) from 1 to number of layers (L). This demonstrates the the impact of MBS on trading off solution quality with profiling effort.

Communication latencies ($t^{x}_{i,d,d'}$) are estimated by dividing the total data size in bytes by the bandwidth (BW) in bytes per second, assuming symmetric bandwidth. They are integrated as shown in Constraints (4) and (5). We vary the network bandwidth over a range including 4G and WiFi, similar to the values in [18]. These values are specified in Tables 7.1 and 7.2.

In all of our experiments, we consider an application where the edge device collects data and requires the result of inference. For example, a smart camera that has to generate a local alert based on the results of inference. One such system is a real-time fire detection system [52], which requires the local alerts to have low latency and high accuracy to allow rapid fire response while minimizing false positives. We enforce these requirements by inserting zero-compute `input` and `output` pseudo layers in the DNN and constraining the associated x variable to be assigned to the edge device, per the procedure in Section 5.1.

Results of AlexNet for Edge-Cloud Setting

We first present the results for AlexNet. This neural network has five convolution and 3 fully-connected layers, for a total of 8 possible transition points under our profiling approach.

We experiment with three approaches: (1) `JointDNN` [18]; (2) `DIME` (ours); (3) optimal. The approach (1) is implemented using our base ILP given in Section 7.2

which simplifies to [18] given that we are also using an edge-cloud setting. For `DIME`, we add Constraint 7.5 to the base ILP and vary the MBS parameter. For the Optimal case (3), we exhaustively list all possible transitions from edge to cloud. We accurately evaluate the latency for each combination (considering both compute and communication) using the device latency profiles that exactly match the bundle assigned to device. We then pick the assignment with smallest overall distributed latency.

We show the results of our approach in Table 7.1 for different bandwidths when MBS is varied from 1 to 8 (number of layers in AlexNet). For each network bandwidth two columns are listed in the table: (1) layer name when transition from edge to cloud happens; (2) latency in seconds. For layer name, we indicate the transition point as the first layer executed on the cloud device, where 'none' means the cloud is not utilized. For latency, we report both estimated and actual latencies based on the generated solution of each ILP. The estimated latency is the value determined by the ILP objective expression; the actual latency is the sum of the largest bundles to implement the required transition points. By definition, 'optimal' and `DIME` with MBS = L report exactly the 'actual' latency.

Table 7.1: Results of optimization for AlexNet, showing transition layer from edge to cloud (Trans.) and distributed latency (Lat.).

		BW = 1E5 MBPS		BW=7.3125E5 (4G Upload)		BW=1E6		BW=2.36E6 (WiFi Upload)		
Method	MBS	Trans.	$\text{Lat.}_{est/act}$	Trans.	$\text{Lat.}_{est/act}$	Trans.	$\text{Lat.}_{est/act}$	Trans.	$\text{Lat.}_{est/act}$	#Profiles
`DIME`	1	none	0.433 / 0.456	**fc1**	0.233 / **0.275**	fc1	0.140 / 0.156	**conv2**	**0.116** / **0.116**	16
	2	none	0.482 / 0.456	**fc1**	0.257 / **0.275**	fc1	0.153 / 0.156	**conv2**	**0.116** / **0.116**	30
	3	none	0.452 / 0.456	**fc1**	0.250 / **0.275**	fc1	0.156 / 0.156	**conv2**	**0.116** / **0.116**	42
	4	none	0.482 / 0.456	**fc1**	0.250 / **0.275**	fc1	0.156 / 0.156	**conv2**	**0.116** / **0.116**	52
	5	none	0.477 / 0.456	**fc1**	**0.275** / **0.275**	**conv3**	**0.184** / **0.184**	**conv2**	**0.116** / **0.116**	60
	6	none	0.456 / 0.456	**fc1**	**0.275** / **0.275**	**conv3**	**0.184** / **0.184**	**conv2**	**0.116** / **0.116**	66
	7	none	0.456 / 0.456	**fc1**	**0.275** / **0.275**	**conv3**	**0.184** / **0.184**	**conv2**	**0.116** / **0.116**	70
	8	**fc2**	**0.508** / **0.508**	**fc1**	**0.275** / **0.275**	**conv3**	**0.184** / **0.184**	**conv2**	**0.116** / **0.116**	72
`JointDNN`		none	0.433 / 0.456	**fc1**	0.233 / **0.275**	fc1	0.140 / 0.156	**conv2**	**0.116** / **0.116**	72
Optimal		**fc2**	**0.508**	**fc1**	**0.275**	**conv3**	**0.184**	**conv2**	**0.116**	72

The number of latency profiles is reported in the last column of the table. This value does not depend on BW so is only listed once per row. The number of profiles is same as the number of $t^c_{i,j,d}$ parameters that should be measured on both devices, and equals the number of $y_{i,j,d}$ variables defined in the ILP per device.

In the table, the bold entries indicate the cases when the estimated or actual latency is the same as the latency for the optimal solution under the given precision. We make the following observations from the table:

- For the highest MBS(=8), our approach always generates the same results as the optimal, across all bandwidths. This is solely due to adding Constraint 7.4 to the base ILP.
- As MBS decreases, the estimated latency becomes more erroneous as its gap with the actual latency grows. For example in bandwidth BW=1E5 for MBS=1 the error between the estimated and actual latencies is 0.456-0.433=0.023 which is the highest error in this bandwidth.
- Interestingly, for some higher bandwidths, the same transition layer *and* latency as the optimal row may be achieved for even smaller MBS. For instance, in the highest bandwidth (WiFi Upload), MBS=1 generates the same quality solution as the optimal approach.
- In the WiFi upload case, we achieve the same quality as the optimal solution for MBS=1 but significantly lower profiling effort (16 instead of 72 for two devices).

- `JointDNN` does not generate the optimal solution in two of the four bandwidths, while requiring the maximum number of profiles. For 4G, it coincidentally picks the correct transition despite underestimating latency.

We validate that the additional constraint and profiling approach in `DIME` always generates the optimal solution.

Results of VGG11 for Edge-Cloud Setting

We also experimented with the VGG11 network [35] which has 8 convolution layers and 3 fully-connected layers, for a total of 11 possible transition points under our profiling scheme. Table 7.2 shows the results for this network. Similar to the prior AlexNet experiment, we observe that varying MBS allows a tradeoff between profiling effort and latency estimation accuracy which affects the layer assignment solution. In particular, we observe for the two highest bandwidths that an MBS=1 provides the same solution quality (in terms of transition layer and estimated latency) as in the optimal, while having a significantly lower number of profiles (22 for MBS=1 versus 132 in JointDNN and Optimal cases). With higher bandwidths, cloud resources become more accessible and all three optimization schemes pick a cloud-only (indicated by a `conv1` transition) partition for the computation of the DNN layers. This is due to the significantly-lower computation latency of the cloud device compared to edge.

Table 7.2: Results of optimization for VGG11, showing transition layer from edge to cloud (Trans.) and distributed latency (Lat.).

		BW = 1E5 MBPS		BW=1.375E5 MBPS (3G Upload)		BW=7.3125E5 MBPS (4G Upload)		BW=1E6 MBPS		
Method	MBS	Trans.	$Lat._{est/act}$	Trans.	$Lat._{est/act}$	Trans.	$Lat._{est/act}$	Trans.	$Lat._{est/act}$	#Profiles
`DIME`	1	none	1.534 / 6.021	none	1.534 / 4.379	**conv1**	**0.823** / **0.823**	**conv1**	**0.602** / **0.602**	22
	2	none	2.094 / 6.021	none	2.094 / 4.379	**conv1**	**0.823** / **0.823**	**conv1**	**0.602** / **0.602**	42
	3	fc3	2.715 / 3.514	fc3	2.670 / 3.469	**conv1**	**0.823** / **0.823**	**conv1**	**0.602** / **0.602**	60
	4	**fc2**	2.924 / **3.451**	**fc2**	2.879 / **3.406**	**conv1**	**0.823** / **0.823**	**conv1**	**0.602** / **0.602**	76
	5	none	2.904 / 6.021	none	2.904 / 4.379	**conv1**	**0.823** / **0.823**	**conv1**	**0.602** / **0.602**	90
	6	none	3.290 / 6.021	fc1	3.217 / 3.417	**conv1**	**0.823** / **0.823**	**conv1**	**0.602** / **0.602**	102
	7	**fc2**	3.185 / **3.451**	**fc2**	3.140 / **3.406**	**conv1**	**0.823** / **0.823**	**conv1**	**0.602** / **0.602**	112
	8	**fc2**	3.131 / **3.451**	**fc2**	3.086 / **3.406**	**conv1**	**0.823** / **0.823**	**conv1**	**0.602** / **0.602**	120
	9	none	3.180 / 6.021	none	3.180 / 4.379	**conv1**	**0.823** / **0.823**	**conv1**	**0.602** / **0.602**	126
	10	none	3.180 / 6.021	none	3.180 / 4.379	**conv1**	**0.823** / **0.823**	**conv1**	**0.602** / **0.602**	130
	11	**fc2**	**3.451** / **3.451**	**fc2**	**3.406** / **3.406**	**conv1**	**0.823** **0.823**	**conv1**	**0.602** / **0.602**	132
`JointDNN`		none	1.534 / 6.021	none	1.534 / 4.379	**conv1**	**0.823** / **0.823**	**conv1**	**0.602** / **0.602**	132
Optimal		**fc2**	**3.451**	**fc2**	**3.406**	**conv1**	**0.823**	**conv1**	**0.602**	132

We clarify one counter-intuitive aspect of the results in the table. For BW=1E5 and BW=1.375E5, it appears that JointDNN generates a better solution in terms of lower latency. However, this is because it is subject to the estimation error imposed by under-estimating devices. When evaluating the *actual* latency from the correct bundles, it becomes clear that it generates a worse solution.

Similar to AlexNet, our formulation guarantees selecting the optimal solution when MBS is set to the number of layers in the network, 11. JointDNN does not provide the same guarantee, especially at lower bandwidths that rely more on the edge device.

Results of VGG11 for Edge-Hub-Cloud

We also consider an experiment with the same edge and cloud devices but adding the NVIDIA Jetson Nano board as an intermediate 'hub' device. The compute capability of the hub device is somewhere between edge and cloud devices. Latency profiles for all related bundles of layers are additionally collected on the hub device before running this experiment. The hub is added to the set of devices S_D and the relevant parameters are included in the ILP.

Table 7.3: Results of VGG11 for Edge→ Hub→ Cloud setting.

	LAN BW=1.375E5 MBPS Cloud BW=1E5 MBPS	
Method	Transition	Latency$_{\text{est/act}}$
`DIME` (MBS=11)	**fc2 → Hub**	**3.418** / **3.418**
`JointDNN`	none	1.534 / 3.533
Optimal	**fc2 → Hub**	**3.418** / **3.418**

A ‘hub’ device adds a higher-compute device as a *more local* resource. Communication from edge does not have to traverse, for instance, the public internet to reach a distant resource. We consider that edge and hub communicate via a local area network (LAN) that has higher bandwidth than either have to access the cloud over wide area network (WAN).

The solution for LAN bandwidth BW=1.375E5 and WAN bandwidth BW=1E5 is shown in Table 7.3. The Transition column shows the layer when transition from edge to hub happens. We find that the introduction of the moderate-compute hub device, even with only moderately increased LAN bandwidth compared to WAN, allows reduction in latency without relying on access to cloud resources. `DIME` generates the same solution as in the optimal, while `JointDNN` assigns all layers to the edge device. This is because the LePotato edge device is an ‘underestimating device’, as explained in Section 7.1, which results in `JointDNN` underestimating the latency of the edge device, and consequently assigning all layers to execute on the edge device when minimizing global latency.

7.5 Conclusion

In this section, we presented a modification to the base ILP (Chapter 5) called `DIME`. This modification incorporates bundle-based profiling and a novel constraint to enfore the selection of the largest encapsulating bundle when multiple layers are assigned to a single device. While DIME focuses on latency, the constraint and maximum bundle size parameter are concerned with correctness of *bundle*

assignment and can extend to objectives measured per-bundle, such as energy. This is important because underestimating devices may cause significant error in latency estimation, thus sub-optimal layer assignment solutions. In our experiments with AlexNet and VGG11 DNNs in multiple device configurations, we compared `DIME` to `JointDNN` [18] and demonstrated that our method guarantees selecting the optimal layer assignment solution when maximum profiling effort is considered.

8 FREDDI: FREQUENCY-DRIVEN DISTRIBUTED INFERENCE

In this chapter I introduce an extension to our bundle-based optimization (Chapter 7) which incorporates device frequency scaling behavior to optimize energy in the edge and hub devices. I also document an energy profiling methodology, which directly complements our existing latency profiling methods. This is based on work *FreDDI: Frequency-Driven DNN Partitioning in Distributed Inference* to appear at SMARTCOMP 2025.

The contributions of this frequency-driven distributed inference optimization approach include:

- We present `FreDDI` which is an extension to our base ILP formulation for Frequency-driven DNN partitioning in Distributed Inference. It allows each device in a distributed environment to have its own set of operating frequencies to select from. The ILP decides how to partition the DNN layers to be executed across the networked devices. It calculates expressions for the (global) distributed latency as well as energy consumption on each device while taking into account their communication.
- The ILP relies on latency and energy parameters when executing an arbitrary sequence of DNN's layers on a device operating at a pre-specified frequency. We present a device profiling setup to measure these on each device, for each frequency option, and for all layer bundles in a DNN.
- We discuss three practical variations on the objective function: latency constrained, energy constrained and privacy constrained.

In our experiments we aim to find a transition layer to switch from edge to cloud. We consider WiFi and 3G bandwidths, and two different edge devices. We compare with `JointDNN` [18] which does not incorporate frequency selection. For example, compared to `FreDDI`, we show that `JointDNN` has 12.4% energy overhead for same latency constraint in VGG11 or 90.4% latency overhead for same energy constraint in AlexNet when a static frequency is a-priori decided. We also demonstrate the trade-offs of `FreDDI` by plotting select operating frequencies from across the range.

8.1 Motivation

This work is motivated by frequency selection in networked heterogeneous devices as a new gateway to navigate energy-based trade-offs during distributed inference. We demonstrate the opportunities using the characteristics of two devices from our experiments.

Nowadays, devices which may be used as an edge/hub/cloud in a distributed setting may have the ability to select an operating frequency from a set of discrete options to support DVFS. Figure 8.1 shows the energy-latency trade-off plot when running AlexNet on the 4GB version of NVIDIA Jetson Nano board for six of the frequency options of its GPU[1]. The fastest frequency (922MHz) results in a latency of about 30ms which is 25% faster than running with the slowest frequency (538MHz) but with a 12% energy overhead. The frequency is fixed by defining a

[1]The NVIDIA Jetson Nano supports 12 frequency options, but these six are sufficient to demonstrate the latency-energy tradeoff characteristics.

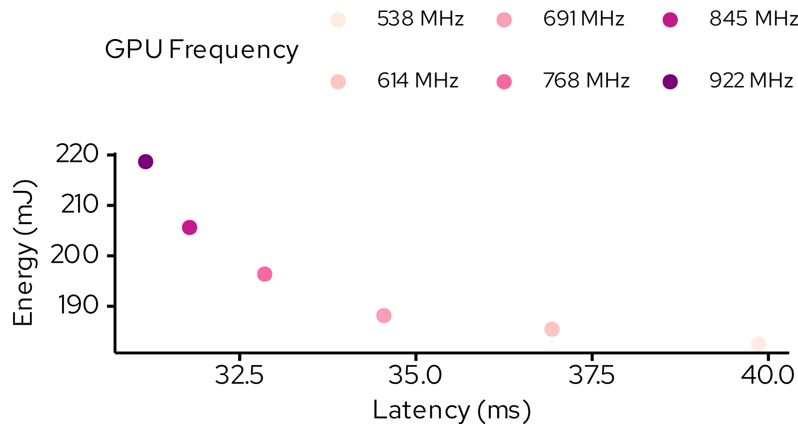


Figure 8.1: Energy-Latency trade-off for LibreComputer NVIDIA Jetson Nano used in this work when running AlexNet while varying GPU frequency.

custom profile for `nvpmodel` and setting the GPU min and max frequencies to be the same value.

Another device that we use in our experiments is LibreComputer LePotato. This is a single-board computer similar to the widespread Raspberry Pi. We considered four frequency settings of the LePotato's CPU which range from 500 MHz to 1200 MHz. We configure the frequency by setting the max scaling frequency and using the `performance` CPU governor to force running at the max point. When running AlexNet on this device, the higher frequency is 1.95X faster than the slowest one but with a 38.9% energy overhead.

As the above two examples show, there is a clear opportunity to utilize the frequency options of an edge device to explore energy-latency trade-offs. The above examples only show the opportunities in a non-distributed setting. The motivation of our work is to utilize this trade-off to decide how the layers of a DNN are assigned

across heterogeneous devices during inference. This *frequency-driven partitioning* is done to explore trade-off between metrics such as the global (distributed) latency, energy consumed on edge devices and privacy.

8.2 Energy Profiling

Our formulation of DNN layer assignment for distributed inference relies on a-priori collection of latency and energy estimates of arbitrary bundles of layers of a DNN. This is done by profiling on a target device running that bundle when operating at a specified frequency. In this section, we discuss how to profile the energy of a bundle $(i \to j)$ of consecutive layers of a DNN, starting from layer i and ending at j. This profiling is performed for all possible bundles $1 \leq i \leq j \leq L$ where L is the number of DNN layers. This profiling is based on the bundle-based latency profiling methodology developed in DIME in Section 7.2. The profiling of each bundle is done for each device, and for each frequency option of the device.

To profile energy, as discussed for the latency profiling, we create a PyTorch Sequential [31] model representing the bundle $(i \to j)$ of the DNN and load it in the target device with a specified frequency. We utilize a SmartPower 3 power supply for power logging which is then used to estimate energy. The SmartPower 3 is a power supply capable of logging the two channels of output voltage (mV), current (mA) and power (mW) at a rate up to 200 Hz. The high frequency of power logging provides sufficient resolution to estimate the energy from the power logs of a single execution of a bundle. Specifically, we find the area under the power curve

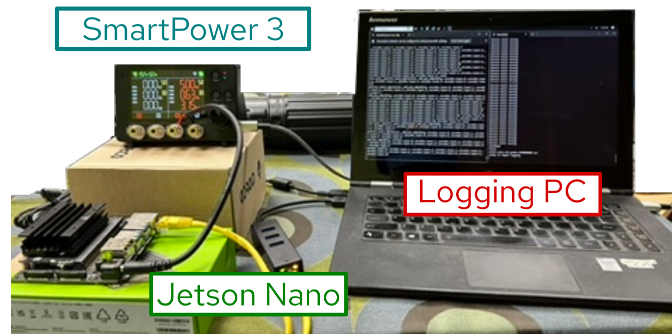


Figure 8.2: Power logging setup including SmartPower 3 power supply, device under test (Jetson Nano), and logging PC.

using trapezoidal integration to estimate energy of a bundle in a single iteration. We run 100 iterations and discard the top and bottom 10%, and then average the remaining measurements to estimate the energy of the bundle for that setting.

The SmartPower 3 is connected to a PC via USB. We utilize a Python script to collect the logging data from the power supply. The device can also communicate with the PC to send messages to start and stop logging, allowing synchronization of the power logs with the latency logs. This setup is shown in Figure 8.2.

An indirect benefit from our profiling approach that fixes the frequency when profiling a bundle is that we reduce uncertainty due to DVFS governors. Band [53] found that DVFS introduced significant uncertainty in predicting DNN performance due to fluctuations in the operating frequency as the device tries to match compute with demand. Note that DVFS may also be hardware-controlled as a thermal protective measure ("thermal throttling"), so it is not completely disabled but we have eliminated the role of software governors.

8.3 From Base ILP to FreDDI

We extend the bundle-based ILP presented in Chapter 7 to include frequency selection and energy information. Consider that device d will have a set of operating frequencies F_d, with f denoting an individual frequency selection. Our optimization goal is now to determine a layer assignment *and device frequency selection* to optimize our objective function.

The computation latency is dependent on the frequency selection, and the parameter notation is modified as

$$t^c_{i,j,d} \dashrightarrow t^c_{i,j,d,f}$$

This represents that latency to compute the bundle $i \rightarrow j$ on device d operating at a frequency $f \in F_d$. A similar compute *energy* parameter is introduced, $e^c_{i,j,d,f}$, which is the energy to compute bundle $i \rightarrow j$ on device d when it is operating at frequency f. These parameters are determined by profiling on each device, for all combinations of bundles with $1 \leq i \leq j \leq L$ for each frequency option.

We also introduce an energy parameter for communication, $e^{x,up}_{i,d,r}$, the energy to transmit the output of layer from device d to a receiving device r. Because communication energy may be asymmetric for upload and download, we also introduce $e^{x,down}_{i,s,d}$ to indicate the energy for device d to receive the results of layer i from sending device s. A similar notation is not required for latency of communication because we already take the minimum of the bandwidth between d and s/r.

Similar to the modification made to the bundle latency *parameter*, we modify the bundle binary variable for the ILP as

$$y_{i,j,d} \dashrightarrow y_{i,j,d,f}$$

Let $y_{i,j,d,f} = 1$ when a bundle of consecutive DNN layers starting from layer i and ending at layer j is assigned to device d which operates with frequency $f \in F_d$. As before, when $i \equiv j$, the bundle includes only one layer. Equation 7.3, which relates the individual layer binary variables to the bundle variables is accordingly updated to account for device operating frequency:

$$x_{\ell,d} = \sum_{\forall i,j \mid 1 \leq i \leq \ell \leq j \leq L} \Big(\sum_{\forall f \in F_d} y_{i,j,d,f} \Big) \quad \forall d \in S_D; \forall \ell \in S_L \tag{8.1}$$

As in Equation 7.3, the above constraint ensures that if a layer is assigned to a device (i.e., $x_{\ell,d} = 1$), then it is associated with only one non-zero y variable. Now, the corresponding y variable reflects a bundle $(i \to j)$ *and* a frequency of operation f on device d which will be used to select the correct latency and energy profile on that device. This does have the additional effect of constraining a device to operate at a single frequency, which is acceptable to avoid the overhead of adjusting the governor between serial layers on the same device.

The computation latency calculation Equation 7.2 is updated to also sum over the device frequencies:

$$T^c = \sum_{\forall d \in S_D} \Big(\sum_{i=1}^{L} \sum_{j=i}^{L} \sum_{\forall f \in F_d} t^c_{i,j,d,f} \times y_{i,j,d,f} \Big) \tag{8.2}$$

The computation energy of the inference is expressed similarly, using the energy parameters defined before:

$$E^c = \sum_{\forall d \in S_D} (\sum_{i=1}^{L} \sum_{j=i}^{L} \sum_{\forall f \in F_d} e^c_{i,j,d,f} \times y_{i,j,d,f}) \tag{8.3}$$

To compute the communication energy of device d, we consider in general that d may upload its results to a receiving device r and that device d may download the results from a sending device s. The communication latency is then expressed below similar to [18]:

$$E^x_d = \sum_{\ell=1}^{L-1} e^{x,up}_{\ell,d,r} \times z_{\ell,d,r} + \sum_{\ell=1}^{L-1} e^{x,down}_{\ell,s,d} \times z_{\ell,s,d} \tag{8.4}$$

The above expression also uses the upload/download energy profiles per device. Using the derivations so far, we can define additional constraints, e.g., to limit the energy consumption on the edge devices or impose an upper bound on the global latency.

ILP Variations

For `FreDDI`, we consider three variations of the ILP objective which are latency-, energy- and privacy- constrained. These variations map to varying application requirements and demonstrate the extensibility of our formulation.

Latency-Constrained

In this variation, the global latency is constrained. This objective is relevant when the inference latency impacts a user's experience. Therefore, we have an additional constraint which states:

$$\sum_{\forall d \in S_D} (T^x_d + T^c_d) \leq T_{cons}$$

where T^x and T^c are given by equations (4) and (5), respectively, and T_{cons} is a constant threshold provided to the ILP based on the application quality of service requirements. The summation is across all devices in the network to compute so this constrains the global distributed inference latency.

In this variation, the interesting objective is to minimize sum of compute and communication energy on the edge devices which is also used in [18]. The objective is written $\min \sum_{\forall d \in Edge}(E^c_d + E^x_d)$ as defined by equations (6) and (7)[2].

Energy-Constrained

In this variation, the energy expended per device is constrained, which is relevant when the application goal is to maximize the usage of an edge device (e.g., a battery-powered edge device integrated with a sensor which monitors a building's health [54]). The additional constraint is

$$E^c_d + E^x_d \leq E^d_{cons}$$

[2]This variation may be easily extended to include energies of any subset of devices in the network, if needed.

which is written for each edge device $d \in Edge$. The relevant objective in this variation is to minimize the global latency expressed as $\min \sum_{\forall d \in S_D}(T_d^c + T_d^x)$ where the summation is written across all devices in the network[3].

Privacy-Constrained

This variation requires particular layers of the DNN (e.g., the first K layers) be executed 'privately' on an edge device. Only the remaining layers of the DNN can be considered for assignment to devices in the network. This variation models recent work in privacy-preserving distributed inference [55]. It allows obfuscation of the potentially-sensitive input (or some intermediate layers) in the early DNN layers before they are sent to a third-party-owned device such as cloud.

Given an edge device d and a subset of K layers of the DNN (denoted by $S_K \subset S_L$), we pre-assign $x_{\ell,d} = 1$ for all $\ell \in S_K$. These will be the additional constraints (pre-assignments) made in the ILP. Recall, pre-assignment was introduced in Section 5.1 as a way to constrain 'input' and 'output' pseudolayers to the edge device. In this variation, the objective is to minimize total energy in edge devices (similar to latency-constrained variation).

Profiling Effort

For our latency and energy parameters, the DNN must be profiled for each device d at each frequency $f \in F_d$ to obtain the $t^c_{i,j,d,f}$ latency parameters for all combinations of $1 \leq i \leq j \leq L$. Compared to the previous bundle-based profiling method, the

[3]This variation may easily be changed to bound the total energy consumed over all edge devices, or all types of devices in the network.

profiling effort is

$$\frac{L \times (L+1)}{2} \times D \dashrightarrow \sum_{d=1}^{D} \frac{L \times (L_{+}1)}{2} \times F_d$$

where F_d is the number of frequencies to choose from for device d. The profiling effort still grows quadratically with the number of layers in the DNN and linearly with the number of devices and the number of frequencies each device supports[4]. Recall, we introduced the Maximum Bundle Size (MBS) parameter in Section 7.3 when considering a latency-based ILP formulation to trade off profiling effort and the quality of the solution. In the experiments in this chapter, we did not restrict the bundle size but the MBS parameter can be incorporated in our formulation to reduce the profiling effort for larger DNNs.

8.4 Results

To evaluate the effectiveness of `FreDDI` we considered an edge-cloud setting and solved the ILP to decide the transition layer from edge to cloud which is similar to the assumption in [18]. For the cloud device we used the NVIDIA 2080TI. We used two different edge devices: LePotato and NVIDIA Jetson Nano. We experimented with 3G, 4G and WiFi as cases of the communication bandwidths. We also experimented with three DNNs: AlexNet [6], VGG11[35] and ViT [37].

[4]In the case that all devices $d \in S_D$ have the same number of operating frequencies F_d, the expression can be re-written $\frac{L \times (L_{+}1)}{2} \times F_d \times D$ to directly observe the linear dependence on frequency options and device count.

The two edge devices were profiled for latency and energy for different bundles of layers of each DNN and each frequency of a device, as discussed in Sections 7.2 and 8.2. The frequency options of each edge device was described in Section 8.1 which had 6 options for Jetson Nano and 4 options for LePotato. For the cloud device we did not utilize frequency selection and used the default frequency of the device since in practice, these are black boxes provided by cloud service providers. We also only profiled the cloud device for different latency bundles since our ILP variations were based only on the energy of the edge devices. The profiled DNNs were designed for image classification. We utilized images from the ImageNette [42] dataset during profiling. We generated and saved the intermediate output of each layer. This ensured realistic values were used as input to each DNN and to each layer bundle as they were profiled.

To estimate the communication energy and latency parameters in the ILP, we used the same model given in [18] with the same download and upload rates based on each connection type as reported in [18]. Namely, instantaneous power for upload and download, respectively, are estimated as

$$P_{up} = \alpha_{up}\tau_{up} + \beta \tag{8.5}$$

$$P_{down} = \alpha_{down}\tau_{down} + \beta \tag{8.6}$$

where τ is the throughput of the connection and α and β are constants, given in Table 8.1. These parameters are characterized separately for upload and download for each connection type. We used the same values reported in [18] for these

Table 8.1: Parameters

Parameter	3G	4G	WiFi
Download Speed (MBps)	0.25	1.72	6.87
Upload Speed (MBps)	0.14	0.73	2.36
α_{up} (W/MBps)	6.95	3.51	2.27
α_{down} (W/MBps)	0.98	0.42	1.10
β (W)	0.82	1.29	0.13

parameters. The energy for communication $e^x_{i,d,d'}$ is then estimated as $e^x_{i,d,d'} = P_{d,d'} t^x_{i,d,d'}$, where $t^x_{i,d,d'}$ is approximated as $\frac{\text{bytes in output of layer } i}{\tau_{d,d'}}$.

Finally, we utilized the Gurobi [49] optimizer to script and solve the ILP formulation.

Comparison with `JointDNN` and no DVFS

We begin by comparing the two variations of `FreDDI` with `JointDNN` [18] when DVFS is not enabled. We implemented the ILP of `JointDNN` and created the two variations by imposing additional constrains explained in Section 4.2. We experimented with two variations of `JointDNN` in which the fixed operating frequency of the edge device was set to the highest (922 MHz) and lowest frequency (538 MHz). We refer to these as `JointDNN-fast` and `JointDNN-slow`, respectively. In this experiment we used the Jetson Nano as the edge device and assume WiFi or 4G for the connection bandwidth.

For each variation, we report the device frequency (F), energy of the edge device (E), and the global (distributed) latency (which is summation of latencies on the

edge and cloud devices and their communication). We also report the constraint imposed in each case.

Latency-constrained

We report the results for latency-constrained variation in Table 8.2. We observe for both VGG11 and AlexNet that `JointDNN-slow` is not able to find a solution. For `JointDNN-slow`, even if the ILP assigns all DNN's layer to the cloud, it still can't meet the latency constraint (of 100 ms) due to the high communication bandwidth to upload the input image to the cloud. However, `JointDNN-fast` is able to find a solution but has additional energy overhead compared to `FreDDI`: 10.5% overhead for AlexNet, 12.4% for VGG11, and 34.5% for ViT. Therefore, for the same latency constraint, `FreDDI` is able to find a DNN partitioning solution with better energy. Notice, the solution of `FreDDI` selects an intermediate frequency of 691 MHz which cannot be considered by `JointDNN`.

Energy-constrained

In this variation, we report the global (distributed) latency corresponding to each approach when the same energy constraint is imposed. Here, both `JointDNN-slow` and `JointDNN-fast` can generate a solution however with additional overheads in latency. These *overheads* for AlexNet are 106.2% in `JointDNN-fast` and 19.2% in `JointDNN-slow` compared to `FreDDI`. For VGG11, the `JointDNN-fast` latency overhead 94.3% is similar, however `JointDNN-slow` is much more significant 105.6% because all work is offloaded, incurring high communication costs. For ViT, the

Table 8.2: Results of the latency-constrained variation for an edge-cloud setting with the Jetson Nano as edge device. WiFi connection is used for AlexNet and VGG11, and 4G is used for ViT.

		F (MHz)	E (mJ)	Lat. (ms)	Constr. (ms)
AlexNet	FreDDI	691	19.7	83.0	100
	JointDNN-DVFS	n/a	19.5 (-1.0%)	82.2	100
	JointDNN-fast	922	22.1 (+10.5%)	82.3	100
	JointDNN-slow	538	no soln.	no soln.	100
VGG11	FreDDI	691	597.6	139.6	160
	JointDNN-DVFS	n/a	686.7 (+14.9%)	120.0	160
	JointDNN-fast	922	671.8 (+12.4%)	119.3	160
	JointDNN-slow	538	no soln.	no soln.	160
ViT	FreDDI	614	1660.5	400.4	500
	JointDNN-DVFS	n/a	2220.3 (+33.7%)	275.3	500
	JointDNN-fast	922	2233.2 (+34.5%)	316.2	500
	JointDNN-slow	614	1660.5 (+0.0%)	400.4	500

overheads are 13.1% for `JointDNN-fast` while it matches the `JointDNN-slow` solution, indicating even on relatively large transformers our approach has a benefit. The higher latency overheads for `JointDNN-fast` compared to `FreDDI` are due to higher communication latencies. Due to consuming more energy, `JointDNN-fast` must offload more layers to meet the energy constraint. This results in higher communication latency because the earlier layers tend to have larger volume of intermediate outputs.

Privacy-constrained

In this variation, we impose constraint on the first K layers of a DNN to be executed on the edge. The remaining layers are decided by each ILP and could be either in the same edge device and/or on the cloud. For AlexNet, we experiment with

Table 8.3: Results of the energy-constrained variation for an edge-cloud setting with the Jetson Nano as edge device. WiFi connection is used for AlexNet and VGG11, and 4G is used for ViT.

		F (MHz)	Lat. (ms)	E (mJ)	Constr. (mJ)
AlexNet	`FreDDI`	768	30.7	94.7	95
	`JointDNN-DVFS`	n/a	29.8 (-2.9%)	94.3	95
	`JointDNN-fast`	922	63.3 (+106.2%)	60.6	95
	`JointDNN-slow`	538	36.6 (+19.2%)	20.1	95
VGG11	`FreDDI`	845	124.1	649.9	650
	`JointDNN-DVFS`	n/a	236.7 (+90.7%)	604.0	650
	`JointDNN-fast`	922	236.3 (+90.4%)	590.6	650
	`JointDNN-slow`	538	255.1 (+105.6%)	213.1	650
ViT	`FreDDI`	614	400.4	1660.5	1800
	`JointDNN-DVFS`	n/a	836.2 (+13.1%)	33.2	1800
	`JointDNN-fast`	922	836.2 (+13.1%)	33.2	1800
	`JointDNN-slow`	400.4 (+0.0%)	1660.5	1800	

$K = 3, 5$ and for VGG11 we experiment with $K = 5, 7$. Here, we report the energy overhead because the ILP variation is set to minimize the energy of the edge device. (See Section 8.3) We also report the latency corresponding to each partitioning solution.

First, compared to `JointDNN-fast`, we always obtain a lower energy DNN partitioning. Specifically, the overhead in energy is 11.9% (for $K = 3$) and 14.3% (for $K = 5$) in AlexNet, and 22.7% (for $K = 5$) and 26.7% (for $K = 7$) in VGG11. For `JointDNN-slow`, the energy overhead is negligible (below 2%), however, `FreDDI` always obtains a better latency.

Table 8.4: Results of the privacy-constrained variation for an edge-cloud setting with the Jetson Nano as edge device with WiFi connection.

		F (MHz)	E (mJ)	Lat. (ms)	Constr. (#Layers)	F	E	Lat.	Constr.
AlexNet	`FreDDI`	614	70.5	123.7	3	644	90.3	32.5	5
	`JointDNN-fast`	922	78.9 (+11.9%)	120.0	3	922	103.2 (+14.3%)	29.4	5
	`JointDNN-slow`	538	70.7 (+0.3%)	125.6	3	538	90.5 (+0.2%)	36.6	5
VGG11	`FreDDI`	614	441.2	744.0	5	768	561.7	284.3	7
	`JointDNN-fast`	922	541.4 (+22.7%)	751.4	5	922	711.6 (+26.7%)	273.3	7
	`JointDNN-slow`	538	447.4 (+1.4%)	760.6	5	538	561.7 (+0%)	284.3	7

Comparison with DVFS-Enabled Case

In the table, we also compare with a case of `JointDNN-DVFS` (Default). In this configuration, we utilize the stock, default DVFS governor of the Jetson Nano device. Here, we repeated the profiling of all layer bundles of a DNN when DVFS was enabled in the device. Recall our profiling technique incorporated 100 iterations. In this case, we ran the 100 iterations in groups of 20. When enabling DVFS during profiling, we make the same observation as [53] which is an increased variability in latency of a bundle across the groups, compared to our profiling under static voltage-frequency setting. In particular, while profiling, we observed that the early iterations in the group of 20 have a higher latency than later iterations, as the governor scales the operating frequency at runtime. This behavior occurs across sets of 20 iterations, indicating it is an effect of the governor beyond an initial 'warm-up' period.

As reported in the table, we observe that for VGG11 and ViT, `FreDDI` allows significant energy reduction in a latency-constrained situation (e.g., 33.7% in ViT). Because the default governor isn't aware of the application deadline, it may apply DVFS as it finds necessary, even though it could make a 'better' selection as `FreDDI` when considering the constraint. Similarly, in the energy-constrained scenario, `FreDDI` outperforms DVFS in minimizing latency.

The often-worse results of `JointDNN-DVFS` highlight the challenge of accurately estimating latency and energy of layer bundles from profiles when DVFS is enabled in a device due to high variability, as also noted in [53].

Trade-off Plots

In this set of experiments, we epxlore how `FreDDI` is able to generate trade offs for different metrics for our ILP variations. We explore these tradeoffs for two network bandwidths, 3G (red) and WiFi (blue).

The imposed constraint in the latency- and energy- constrained cases are shown as vertical and horizontal lines, respectively. Each DNN partitioning solution is shown as a point (between two constraint lines) with a corresponding latency and energy in these two columns.

Latency-constrained:

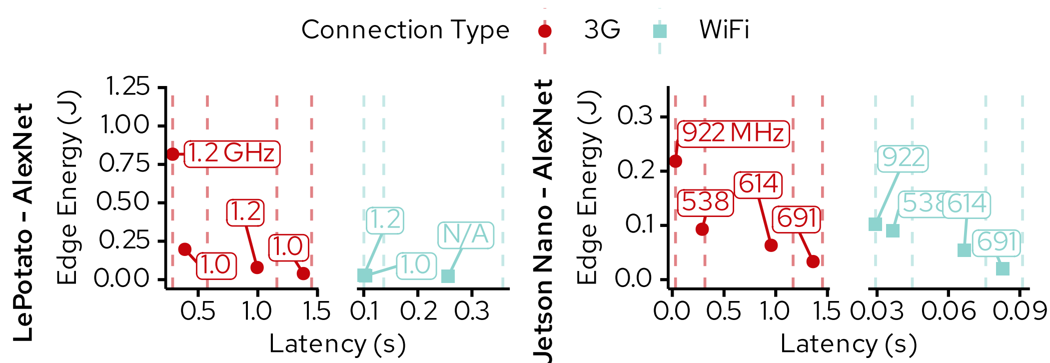


Figure 8.3: Tradeoff for the latency-constrained case, with constraints shown in dashed lines.

The latency-constrained tradeoff is plotted in Figure 8.3. We observe that as the latency constraint is relaxed, the optimization can pick lower-energy partitions. This is usually (but not always) done by selecting a lower operating frequency. In some cases, a higher operating frequency may be selected to allow more work to be shifted to edge.

The points with a 'N/A' operating frequency indicate the edge device is not used for computation, and all computation is offloaded to cloud. In such cases, the majority of the latency is due to the communication bandwidth to send the input image to the cloud. For example 'N/A' can be seen in the top-left plot in the WiFi case (blue). Here the faster WiFi connection allows for offloading all computation to cloud to obtain a lower energy execution, compared to using the edge device with a slower frequency.

Overall, both operating frequency and the DNN partition are decided by `FreDDI`. Consider the left plot on the second row (Jetson Nano and AlexNet). A latency-insensitive application can relax the latency constraint from 0.028s to 0.084s for an energy reduction from 0.90 J to 0.20 J (78% decrease). This is achieved by both utilizing a lower frequency selection (922 MHz → 691 MHz) and changing the partition point (`conv5|fc1` → `conv1|conv2`) in AlexNet.

Energy-constrained:

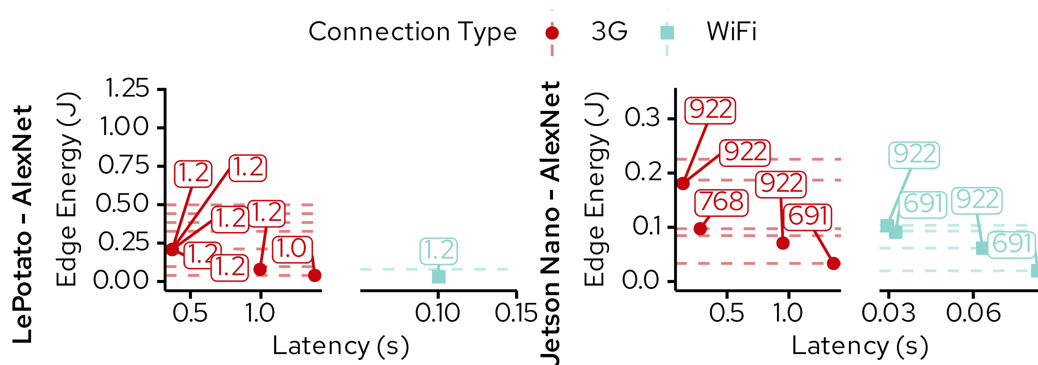


Figure 8.4: Tradeoff for the energy-constrained case, with constraints shown in dashed lines.

In Figure 8.4, the global latency is minimized while imposing an energy constraint on the edge device (shown as horizontal lines). The plots show that as the energy constraint is relaxed, the latency can be improved. This is typically (but not always) achieved by switching to a lower frequency with relaxation of the energy constraint. However, sometimes a higher operating frequency is selected as the energy constraint is relaxed due to selecting a later layer to transition to the cloud.

Privacy-constrained:

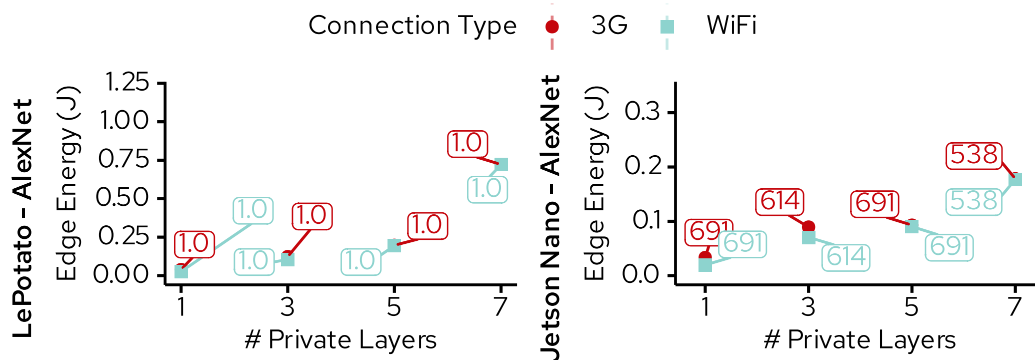


Figure 8.5: Tradeoff for the privacy-constrained case, with constraints shown in dashed lines.

In Figure 8.5, we observe that as the application demands more privacy, more energy is spent on execution on the edge device. However, we can still select an optimal operating frequency less than the fastest to minimize the energy spent on the edge device.

8.5 Conclusion

In this chapter, I described `FreDDI`, the first ILP formulation which incorporates selection of device operating frequency for better DNN partitioning in distributed inference.We considered three practical variations for minimizing the overall distributed inference latency, energy consumption on the edge side, and preserving privacy by limiting the layers that could be executed on third-party devices. In our experiments on LibreComputer LePotato and NVIDIA Jetson Nano as two edge device options, we demonstrate that frequency selection enables significant energy savings in applications that are latency-constrained, and vice versa.

9 CADI: CARBON-AWARE DISTRIBUTED INFERENCE

In this chapter I consider the environmental impact of DNN inference and explore an extension to the base ILP (Chapter 5) to improve the sustainability of DNN inference in edge-hub-cloud systems. This is based on preprint of *CADI: Carbon-Aware Distributed Inference*.

In this work, we explore distributed inference using multiple edge devices to explicitly optimize the carbon footprint. Our contributions are summarized below:

- We formulate layer assignment of a DNN to edge devices to explicitly minimize operational carbon footprint (i.e., carbon intensity over a period of time).
- We also provide alternative formulations to minimize energy of single inference. This is based on our comprehensive power and latency profiling of each device type.
- In our experiments we demonstrate that carbon-minimal solutions can differ significantly from energy-minimal solutions; we report on-average 3.28X higher carbon footprint when minimizing energy as the objective.

9.1 Motivation

As AI applications have been rapidly adopted, there has been significant interest in the impact of high energy use to support the deployment of these computationally

intensive DNNs, rising to the attention of the United Nations Environment Programme in 2024 [56] demonstrating the global and practical nature of this priority. Not only are the energy demands a concern, but the environmental impact of the carbon emissions created to meet that demand.

While data centers are the focus of many recent works due to their outsized energy utilization and density, we examine the carbon footprint of edge devices (defined as carbon intensity in a period of time). This is due to the wide-spread availability of edge devices for collaborative distributed inference. These edge devices may belong to different dynamic power grids. This is in contrast to inference on data centers which often have energy density limits to support a co-located and consistent nuclear reactor [57]. For instance, the work [29] designs an approach to optimize purchasing "carbon emission rights" by taking advantage of dynamically scheduling small-DNN inference on edge devices and large-DNN inference on the cloud. A recent survey by Trihinas et al. [58] emphasizes the need to integrate carbon footprint modeling for AI IoT, including the need for accurate power modelling to integrate carbon intensity data.

9.2 Carbon Intensity

A metric frequently used in addition to energy to quantify compute impact is carbon dioxide (CO2) emissions, which aims to quantify the environmental warming impact of the energy consumed. This is often provided as "carbon intensity" of the energy, gCO2 per kWh, which allows a direct conversion between energy con-

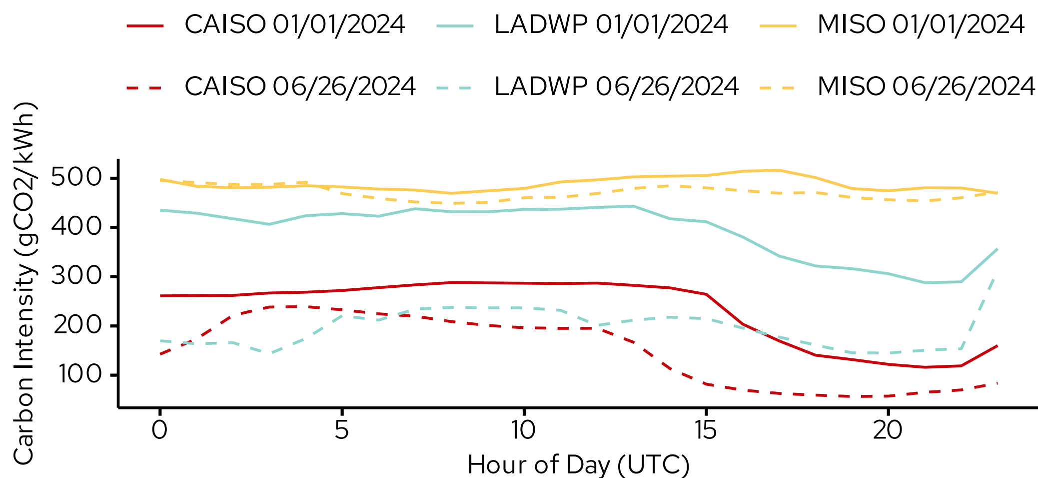


Figure 9.1: Variations in carbon intensity over a 24-hour period for two dates (in January and June to capture seasonal variations) for three different power grids [59]: Los Angeles Department of Water and Power (LADPW), Independent System Operators in California (CAISO) and Mid-continent regions (MISO).

sumed and carbon emissions. Sources such as Electricity Maps [59] provide a CO2 equivalent (CO2-eq/ kWh), which combines all sources of environmental warming in a single figure to provide a straightforward yet holistic measure of impact. The carbon intensity varies depending on the mix of energy sources in the electricity grid and changes over time (Figure 9.1). Electricity Maps provides historical data with a granularity of one hour in the electricity grids examined.

9.3 From Base ILP to CADI

We are interested in considering the operational carbon footprint over an application lifetime, not just of a single inference. This requires a modification to our

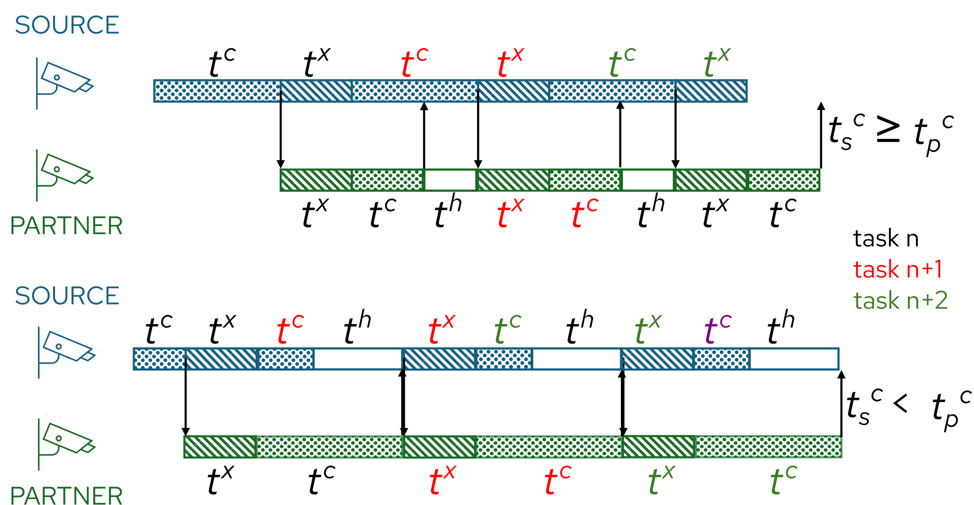


Figure 9.2: The updated system model demonstrating multiple inferences over a duration, for two cases ($t_s^c \geq t_p^c$ and $t_s^c < t_p^c$). We illustrate an edge-edge case, though the updated system model is applicable to any set of devices.

previous system model described in Chapter 4. In this modified model of distributed inference, we assume a long-running service that happens continuously over time, such as continuous image recognition based on inputs of a camera. We constrain this service to not proceed with a new inference task until the current inference is finished. This is a practical constraint for an edge system by eliminating the memory requirements for queuing on a receiving device, while still allowing for some pipelining.

The example in Figure 9.2 shows our model of a two edge device communication assuming the source s computes the first ℓ layers and then sends the results to a partner device p computing the remaining layers before starting the next new inference task. The figure shows example with three consecutive inference tasks n, $n+1$, and $n+2$.

This system model introduces a new parameter, hold time:

$$t^h_{l,d,d'}$$

This is the hold time of device d before it can communicate the next computed results to d' where ℓ indicates the last layer executed on d. Hold time depends on the compute time of layers assigned to the devices.

In the top case, the compute time on s is higher than the compute time on p so the hold time on source is 0, and on partner is $t^h_{\ell,p,s} = t^c_{1,\ell,s} - t^c_{\ell+1,L,p}$. In the bottom case, this condition is not satisfied, and the hold time on the partner is 0 and on the source s is $t^h_{\ell,s,p} = t^c_{\ell+1,L,p} - t^c_{0,\ell,s}$. A corresponding hold energy parameter is added, based on the power of the device at idle and the length of hold time:

$$e^h_{l,d,d'} = t^h_{l,d,d'} \times P^{idle}$$

Finally, to compute carbon footprint, we define the following parameters:

- C_d: Carbon intensity is given based on the geographical location of device d and power grid that it is connected to. The unit of C_d is milligram CO2 equivalent per joule (mgCO2-eq / J). Although carbon intensity does vary over time, we do not include a time term in this parameter as we target only a specific application duration with a particular carbon intensity and the optimization may be efficiently re-solved periodically.

- $I_{\ell,s,p}$: number of inferences (in a two-device setting) in a given duration of time U, assuming the first ℓ layers of the DNN are executed on source s and partner p device executes the rest. Duration U is specified based on the granularity of carbon intensity data, often refreshed at intervals such as hourly.

To extend the base ILP to consider duration, we first determine how many inferences will take place in a given duration. Based on our system model in Figure 9.2, number of inferences may be expressed as below:

$$I_{\ell,s,p} = \frac{\text{U}}{\max(t^c_{1,\ell,s} + t^x_{\ell,s,p}, t^c_{\ell+1,L,p} + t^x_{\ell,p,s})} \tag{9.1}$$

Note, $\ell = L$ is not included in the above equation because it represents the single-device case when all layers are executed on s.

In addition to the total compute energy on a device E^c_d and total communication energy on a device E^x_d defined in Chapter 8, we consider the total hold energy of a device

$$E^h_d = \sum_{i=1}^{L} \sum_{j=1}^{L} e^h_{i,j,d} \times y_{i,j,d} \tag{9.2}$$

The total energy of a device is modified as

$$E_d = E^c_d + E^x_d \dashrightarrow E_d = E^c_d + E^x_d + E^h_d \tag{9.3}$$

To compute carbon footprint of a device, we first need to compute total energy over a duration U which we refer to as the energy footprint. For clarity, we express

these for source and partner devices, respectively, assuming partitioning layer ℓ as before.

$$EF_d = \sum_{\ell=1}^{L-1} (I_{\ell,s,p} \times y_{1,\ell,s}) \times (E_d^c + E_d^x + E_d^w) \quad d \in \{s, p\} \tag{9.4}$$

In the above, EF_d computes energy footprint of device d in a two-device setup with source s and partner p. $I_{\ell,s,p}$ is constant for a given partitioning layer ℓ and represents number of inferences in duration U. It is computed using equation 9.1. The quantities E^c and E^x are computed as in Chapter 8, and E^h is computed from equation 9.2.

The carbon footprint of a device is then computed by multiplying the energy footprint of the device by the corresponding carbon intensity parameter:

$$CF_d = C_d \times EF_d \tag{9.5}$$

Objectives

We generate variations of our ILP based on two objectives. The constraints remain the same in both cases. Our first objective is to minimize the total carbon footprint across the edge devices which is the main focus of this work. It is expressed as $CF = CF_s + CF_d$ for the two-device scenario. In our experiments, we also compare with minimizing energy of a single inference which is expressed as $E = \sum_{d \in \{s,p\}} (E_d^c + E_d^x + E_d^w)$.

9.4 Results

In this section, we evaluate the effectiveness of our formulation for the layer partitioning task. We used the two-device system model shown in Figure 9.2. For our edge device, we utilize the NVIDIA Jetson Nano 4GB model, which includes 4 ARM Cortex A57 CPU cores and 128 CUDA cores. This represents a moderately powerful device which can realistically handle DNN tasks. The compute energy and latency are profiled per Chapter 8. For estimating communication energy and latency, we use the values for 4G network from [60], similar to [18].

To determine U (duration to compute the operational carbon footprints), we choose a granularity of 1 hour in our experiments because this is the main granularity available in data from [59]. We utilize historical logs for the year 2024 for three separate grid operators: Midcontinent Independent System Operator (MISO), California ISO (CAISO), and Los Angeles Department of Water and Power (LADWP). These grids have unique carbon intensity characteristics, even though, as in the case of CAISO and LADWP, they are adjacent in some locations. In general, the carbon intensity of these grids have the following relation: CAISO $<$ LADWP $<$ MISO, with CAISO having the least carbon intensity values .

We evaluate with three popular image recognition neural networks: AlexNet [6], VGG11 [33], and Vision Transformer (ViT) [37], implemented in PyTorch [31]. We utilize images from the ImageNette dataset [42], which will provide realistic activations for the networks. We use the off-the-shelf solver Gurobi [49] for writing and optimizing the ILP. We utilize a single-inference maximum latency constraint T of 1 second for AlexNet and VGG11, and 3 seconds for ViT.

Minimizing CF vs E for Inference with Two Edge Devices

We consider the setting with two edge devices, and compare solutions of two ILPs when minimizing energy (E) versus when minimizing carbon foorprint (CF) as the ILP objective. All constraints remain same between the two compared ILPs.

We assume two cases when the source device may be connected to either LADWP or MISO, while the partner device is always connected to the CAISO grid which has the least carbon intensity data.

Additionally, for duration U, we picked the time in a summer day (6/26/2024 3:00 UTC). This selection provides interesting insights as it is during the afternoon with peak renewable sources, causing significant variance in the carbon intensities of the source and partner grids.

When considering a two-device setting, transitioning to the partner device may be beneficial because of switching to a device with lower carbon intensity. However, the ILPs additionally consider carbon footprint or energy due to communication overhead (e.g., upload and download on each device) based on the network bandwidth and device specifications. Decision on when to switch to a partner device is not clear and is made by the ILPs based on their objectives.

We compare energy (E), energy footprint (EF)[1] and carbon footprint (CF) generated by each ILP. We also compare the number of inferences (I) and the DNN layer when transition from source to partner device occurs (last layer executed on source device) as generated by each ILP. The results are shown in Table 9.1. The results when minimizing E is normalized to the case when minimizing CF.

[1]Energy footprint is computed using equation 9.4.

Table 9.1: Results for distributed inference with two edge devices with the source device connected to a higher carbon intensity grid than the partner. The unit for energy (E) and energy footprint (EF) are Joules, and for carbon footprint (CF) is mgCO2-eq-hour. The results of minimizing energy are normalized to the ones when minimizing carbon footprint.

		Minimize Carbon Footprint					**Minimize Energy (normalized)**				
DNN	**Grid**	E	EF	CF	I	L_{trans}	E_n	EF_n	CF_n	I_n	L_{trans}
AlexNet	MISO → CAISO	0.281	10460	1050	8316	conv4	0.76	3.69	4.97	12.56	*none*
	LADWP → CAISO	0.642	10249	534	3976	conv1	0.33	3.76	2.88	26.27	*none*
VGG11	MISO → CAISO	0.959	12680	1338	5368	conv7	0.94	2.67	3.43	5.97	*none*
	LADWP → CAISO	0.31	12777	649	5323	conv8	0.91	2.65	2.08	6.03	*none*
ViT	MISO → CAISO	2.650	19558	1770	3134	add_pos	0.84	2.73	4.10	4.17	*none*
	LADWP → CAISO	2.396	19613	963	3129	enc_11	0.83	2.72	2.21	4.18	*none*
AVERAGE							**0.77X**	**3.04X**	**3.28X**	**18.20X**	

As can be seen from the table, minimizing E on average results in 3.28X higher carbon footprint, even though the energy is 0.77X compared to minimizing CF. Also, as can be seen when minimizing E, the ILP solution favors conducting all inferences on the source device and there is no transition to the partner. In contrast, minimizing CF always utilizes the partner device to compute the remaining layers of the DNN on CAISO which is the lower carbon intensity grid despite the energy (and carbon) overhead associated with the communication latency between them (e.g., upload/download on each device).

From Table 9.1 we notice that the solution when minimizing E results in source-only execution with significantly higher number of inferences in the same time duration (on-average 18.20X). In our next experiment we make comparison with an alternative single-device case.

Comparison between Two-Device and Single-Device Cases

In this experiment, we compared the solution of our ILP for two devices when minimizing CF with a single-device case which has the same number of inferences for the same duration as the two-device case.

Specifically, we manually compute the CF of single-device case (executed on the source device). We use the same number of inferences I found from the ILP when minimizing CF for two devices, and for the same duration (1 hour). More specifically, this manual calculation of CF in single-device case is done by determining the latency and energy for computing I inferences on the source device, and then adding wait latency and energy for the full 1-hour duration.

Table 9.2: Comparison of carbon footprint when minimizing CF with our ILP for two edge devices, versus a single-device case. Carbon footprint of single-device case is measured for same number of inferences (and same duration) of the two-device case. Carbon data taken for 6/15/2024 at 20:00UTC.

Grid	DNN	I	Two Device	Single Device
MISO → CAISO	AlexNet	8316	582	560 (**0.96X**)
	VGG11	5368	802	851 (**1.06X**)
	ViT	3134	806	1049 (**1.30X**)
Average				**1.11X**

The results are shown in Table 9.2 (for carbon intensity data of a different 1-hour time period than the previous experiment). As can be seen the single-device has on-average 1.11X higher CF compared to the two-device case, even under the same number of inferences in the same duration. In general, this trend may not always be the case and having lower CF with two devices depends on the DNN as well as the carbon intensities of the grid. For example, we can observe in AlexNet, the single-device CF is slightly lower (0.96X) than the two-device case, despite the availability of partner device connecting to a grid of lower carbon intensity.

Based on both experiments, we conclude that evaluating execution on single device (which does not involve any optimization) versus two devices (using our ILP) may be conducted independently, and then the best option to minimize CF may consequently be identified.

Our system model assumes that, for a given solution, we would like to maximize the number of inferences and cannot insert arbitrary wait times as we did in the manual calculation. The highest benefit is observed with high-carbon sources and

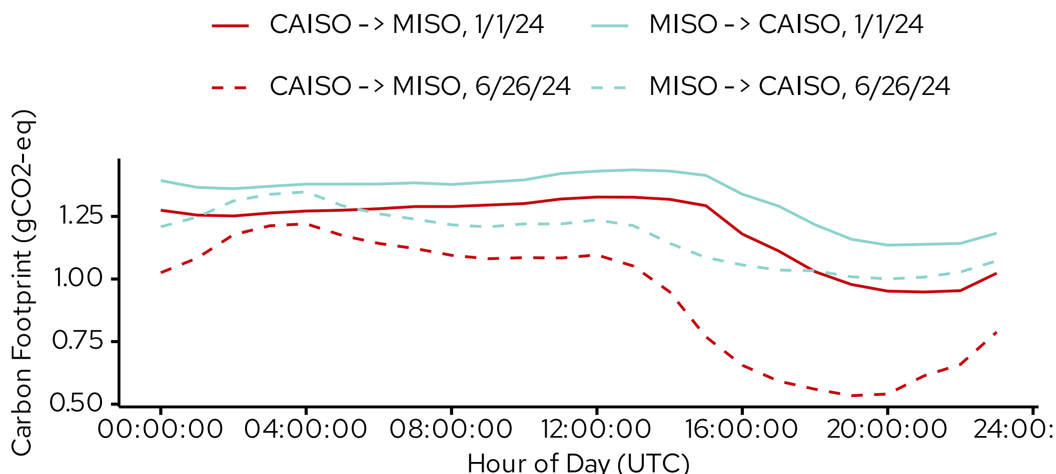


Figure 9.3: Variation of VGG11 carbon footprint over a 24-hour period for (a) Jan. 1, 2024 and (b) June 15, 2024. Carbon footprint data are generated by our ILP when minimizing CF.

low-carbon partners which allow layers to be assigned to a device with low carbon intensity. Formulating a system model that allows for arbitrary wait time to find an optimal carbon solution is an interesting future direction to explore.

Dynamic Scheduling

We are also interested in a unique property of carbon intensity: dynamic variation both over the course of a single day and seasonally. The CF-optimal solution may vary over time as carbon intensity changes, particularly as different grids may change at different rates. We plot carbon footprint for (VGG11, MISO $\rightarrow$ CAISO) and (VGG11, CAISO $\rightarrow$ MISO) for two 24 hour periods, one in winter and one in summer, in Figure 9.3. The carbon footprints are generated by our ILP optimizing

CF. Recall from Figure 9.1 that carbon intensity tends to decrease over the course of the day, and tends to be less in the summer than winter.

On both days, we see that the carbon footprint decreases over the course of the day, tracking closely with changes in the carbon intensity. However, the change in carbon footprint is not due solely to the changing carbon intensity. Over the course of the day, as carbon intensities vary in each region, the optimal transition layer to partner device may change for distributed inference. For instance, considering CAISO→MISO between 14:00 UTC and 15:00 on 6/26/2024, the carbon intensity of CA changes from 3.15E-2 to 2.27E-2 mgCO2/J. The transition layer correspondingly changes to partition after conv8 to *none* (single device inference). The optimal number of inferences increases as more work is shifted to the source device which benefits from the rapid drop in carbon intensity of the CAISO grid. This demonstrates that we can opportunistically increase the quality of service, in terms of inference throughput, a unique benefit of carbon intensity's time variance.

9.5 Conclusion

We demonstrated an ILP formulation that captures the cost of energy for DNN inference on edge devices for a *duration*, not just a single inference. We extended this formulation to incorporate carbon intensity, such that we can determine the overall carbon footprint. In our simulations based on profiled computation energy and estimated communication energy, we demonstrated that minimizing energy and carbon footprint may result in different transition layers in distributed inference. In

particular, low-carbon sources are encouraged to pick up more work, even at the cost of energy increases, to reduce the carbon footprint by significant amounts.

10 CHAPTER FOR THE PUBLIC

In this chapter, I present my research in a form geared towards a general public audience. My work has been supported by the public through programs like the National Science Foundation and I hope to make the results of my work more accessible. Access to knowledge is a powerful tool in developing an engaged and compassionate society, and I am proud to contribute in my own small way. Many thanks to editor Elizabeth Reynolds for thoughtful remarks, and to Professor Bassam Shakhashiri and Cacye Osborne at the Wisconsin Initiative for Science Literacy for supporting this program!

10.1 Background on Artificial Intelligence (AI)

Nowadays, many of us are familiar with interacting with artificial intelligence (AI) programs in our everyday life. You might use Grammarly to help draft and revise an email or Google's Gemini to generate a logo for your sports team. All of these modern AI tools rely on a fundamental invention: the neural network.

The neural network is, at least in computing terms, an ancient idea with the first generation developed in the 1950s. As you might suspect based on the word "neural", these computer programs are inspired by the brain and use digital "neurons" to process information. Much like our brains process what our eyes see, early neural networks focused on processing pictures to recognize handwriting.

Many factors contribute to the success of modern AI tools, but one of the key insights, which occured around 2012, was "deep" neural networks (DNNs). I like

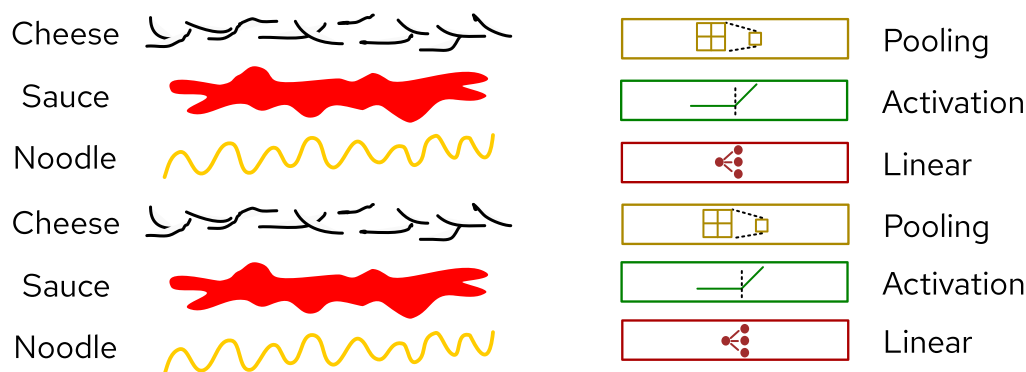


Figure 10.1: A lasagna (left) is a bit like a deep neural network (right). Both are composed of multiple layers of different types, with each contributing something important to the final outcome.

to think about this like one of my favorite foods, lasagna! Similar to a lasagna which might have layers of noodle, sauce and cheese, a DNN uses different layers of operations for different effects. Also like a lasagna, a DNN of one layer isn't particularly good, so we build up multiple layers to improve them!

As the lasagna gets more and more layers, we need larger and sturdier pans to bake in. If we have hundreds of layers, we might even need to look at moving to some industrial oven equipment. The story is similar for DNNs, where small models with few layers might work on our phones or laptops, but the latest and greatest (and largest) require industrial-sized computers in datacenters.

As we have watched the rapid evolution of AI tools in the past few years, I have been very concerned about the impact of their high energy use and potential for carbon emissions. Understanding how to minimize these impacts has been a very interesting question to explore, both from a technical perspective and its social merits. My approach to this problem has been to split those layers up to utilize multiple computers for DNNs. It would be like baking layers of the lasagna

separately, and only stacking them up at the end when it's ready to serve. I call this **layer assignment**, trying to find which layer is best assigned to which device. My research has shown that we can use this technique to improve latency (speed), energy (battery life), privacy (keeping control of your data), and carbon emissions (sustainability).

10.2 What's in a Layer

Before we can begin assigning layers to a computer, we need to know a bit more about what's happening in each layer. I primarily consider the latency to compute the layer and the energy to compute the layer, key metrics in many computer systems.

- **Latency** is the amount of time it takes to compute a layer from start to end. This is measured in milliseconds.
- **Energy** is the amount of work that the computer must do to compute a layer from start to end. This is measured in joules, though you may also be familiar with measures like kWh on your utility bill or mAh on mobile phone batteries.

There are two typical ways to collect this data: using formulas to estimate and directly measuring latency and energy on the device. For either, it is import to have a rigorous and repeatable method because we need to measure for each computing device in our system and the values must mean the same thing from device to device. For instance, a smartphone will usually be slower than a desktop and might

use less energy, but we want to make sure that isn't because of an error in the test methods. Overall, I prefer to use the direct measurement approach because it tends to be more accurate, but the formula approach can be useful for prototypes.

Figure 10.2 shows the setup in my lab for collecting latency and energy information for an NVIDIA Jetson Nano. This setup includes the actual device, the NVIDIA Jetson Nano, a SmartPower 3 power supply that I use to gather the energy data, and a PC for logging and analyzing the data. You can see why setting this up for every device is more complicated than plugging a few numbers into a formula!

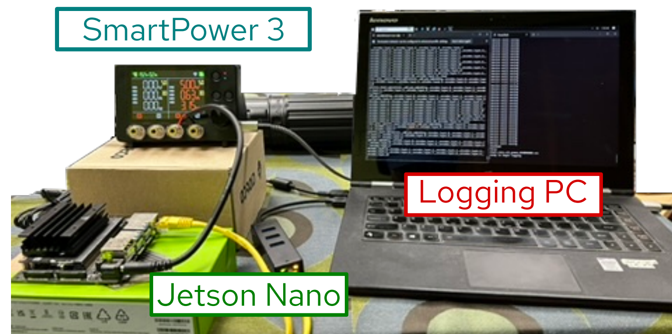


Figure 10.2: The lab setup for measuring latency and energy used by the NVIDIA Jetson Nano.

10.3 What's the Deal With All These Computers?

So what about the computers that I'm assigning these layers to? Imagine a theoretical lasagna factory, like in Figure 10.3. This factory might have "smart" cameras that monitor different conveyor belts, some local computers for running the factory, and Internet access to cloud computers.

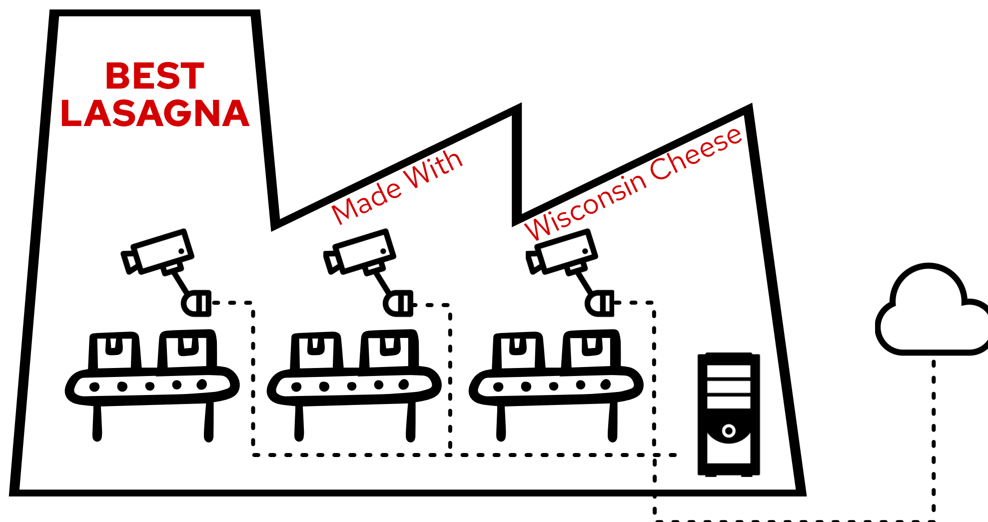


Figure 10.3: Theoretical lasagna factory demonstrating the various types of computing devices in a system and how they can communicate.

In my research, we call the smart cameras "edge" devices, the factory computers "hub" devices, and the external cloud "cloud" devices. These are "small", "medium", and "large" computers, respectively. Each of these devices has the ability to do at least some of the work of computing the DNN. They are also all connected in such a way that they can all talk to each other.

10.4 Useful Objectives

Now, we have to decide what to do with all of this information we've collected about layers. I want to improve something about the AI system, so I want to express an *optimization* goal. This is usually expressed as *"minimize A subject to B"*, which means that we want to reduce *A* while staying within limits of *B*. "Subject to" is optional, but often very useful as we'll see. Say we wanted to bake a lasagna faster.

One approach could be to use only one thin layer, but as I said before, that's not a very good lasagna. So we could say *"minimize bake time subject to having four layers"*, so we still have a good lasagna. We could then change oven temperature, baking dish size, and baking dish material to minimize the bake time.

For DNNs, over the course of my research, I've identified a few useful optimization goals when assigning layers:

1. Minimize latency: this minimizes the amount of time it takes to get an answer back. This improves the user experience by giving a more interactive experience with less wait time for the computer to "think".

2. Minimize latency subject to privacy: a tweak of the previous goal, this minimizes the amount of time to get an answer while providing additional data privacy guarantees. This further improves the user experience by reducing the amount of identifiable information shared.

3. Minimize energy subject to latency: this minimizes the amount of work put in to compute the answer. The latency constraint is important to guarantee that the user gets an answer back in a timely manner (as a professor I know likes to say "minimizing energy is easy: just turn it off!"). This improves the user experience through reducing energy bills and improving battery life.

4. Minimize carbon subject to latency: a tweak on the energy goal, we look at carbon emissions as a sustainability metric. This improves user experience by reducing the environmental impact of running AI.

10.5 Finally, Some Results!

Preheating with Preloading

Over the course of my PhD, I've explored various approaches to achieve these objectives. In my first project, we only looked at reducing latency. While investigating the various causes of latency, we found a significant source we could hide with a clever layer assignment. The layers in the DNN usually have some extra data associated with them, called "weights", that can take a long time to load. We proposed that we could hide this by starting this loading process early, while another device was busy computing another layer. We call this **weight preloading**.

In the DNN, the layers use the result of the previous layer. This means they must be executed in a particular order, and only after the previous layer is finished. This is a bit like if you couldn't start on the next layers of lasagna in a second pan until the first layers are done baking. However, with our layer assignment strategy, we know before we start baking which layers will go in which pan and what order the pans go in the oven. This means we can start building the layers while the previous pan is baking, so it is ready to go into the oven right when the previous pan comes out. While we still can't fit multiple pans in the oven, we've overlapped the baking time of the previous layer with the preparation time of the next layer to hide that preparation time.

We explored this weight preloading idea with different DNNs and different computing systems. In these experiments, we found a few interesting results:

- When the devices are good at talking to each other (the network has high bandwidth), it's easy for the smart cameras to send all the work to the cloud. Weight preloading isn't very useful in this case, because there aren't opportunities to overlap compute and weight loading (the industrial oven can bake all the layers at once).
- When devices are bad at talking to the cloud (the network has low bandwidth), the small and medium computers tend to talk to each other quite a bit. Their limited resources makes weight preloading more appealing (smaller ovens means more opportunities to overlap baking and preparing).
- We also found that certain types of layers within the DNNs had more benefit than others. Layers that had many weights to load were the best for hiding preparation time with weight preloading.

If systems implement our weight preloading technique, the latency of DNN computation can be reduced significantly. This will lead to AI applications that respond to users more quickly, particularly in areas with low Internet access.

Baking with Bundles

Over the course of my research, I have learned a lot about measuring latency and energy on these devices. This became the focus of my second project. In this project, we examined some problems with what we referred to as **bundle-based profiling**. When measuring the latency and energy for the layers of our DNNs, we want to collect data about the individual layer as well as groups of layers. This is because

determining the latency and energy of a group of layers is not a simple sum of its parts. This is a bit like if you were to try to bake four layers of lasagna in one pan with four layers or four pans with one layer in each. In the second case, every time we change pans, we might experience things like the oven cooling a bit when the door opens and we need to heat the pan as well as the layers. These *external factors* cause baking four layers in one pan to be very different than one layer each in four pans.

However, we found that the situation was a little bit more complicated for DNNs. On some devices, grouping the layers into a bundle would reduce the latency; on other devices, bundles increased the latency. This becomes a problem when we want to optimize, for instance, latency. During this optimization process, we look at all posssible combinations of layers, and pick the combination with the smallest latency. For a device whose bundle is *larger* than the sum of its parts, it looks better to pick individual layers instead of the bundle. But this is a side effect of the mathematical expression, and does not represent reality.

This would be a bit like if, for baking the four layers, we were told to bake all four layers in the same pan. You and I, actually in the kitchen, would build all four layers and put this into the oven at once, forming a bundle of 4 layers. But the recipe writer, wanting to give the lowest bake time estimate, instead used the sum of baking the four layers in the same pan separately. We would be very disappointed with the cold center of our lasagna!

Unfortunately, this problem took quite a bit of profiling effort to uncover and understand. Fortunately, the solution was quite straightforward. We could add

a check during our optimization search that ensures, for layers assigned to a device, the bundle values are used instead of the individual layer values. When we implemented this, we found:

- Without bundles enforced, the latency can be mis-predicted. This causes the layer assignment to be incorrect.
- In one case [1], the actual result is 1.2x slower than the estimate. This would be like a recipe that guaranteed a cold center.
- In another case [2], the actual result was 1.3x faster than the estimate. This would be like a recipe that guaranteed burnt lasagna.
- By considering bundles, we guarantee that the layer assignment is correct in mirroring real-world conditions.

Improving DNN latency estimation will allow developers to make better decisions about layer assignment. This will encourage more consistency and fewer surprises when trying to optimize the experience for users.

Resting with Efficiency

One of the big discussions around AI has been about the massive energy requirements for maintaining these computers. This is a significant issue with broad effects, and I'm glad my next project was able to focus on this. To do this, we exploited an

[1] Table 7.1 BW=1E5
[2] Table 7.2, BW=1.375E5

interesting feature on modern devices, *frequency scaling*. Frequency scaling allows us to control the operating frequency of the device, or how fast it gets work done. This is kind of like oven temperature: a high temperature may cook faster but also increases the chances of burnt edges while a low temperature cooks slower and more evenly, but too slow and you'll need a snack before dinner is ready. There is a "sweet spot" for oven temperature.

Similarly, for a computer operating frequency, a higher frequency means it can get work done faster but consumes a lot of energy. A lower frequency means the work gets done slower, and may be done so slowly that the energy starts going up. For energy efficiency, there is often a "sweet spot" setting for the device operating frequency.

Up until now, I had profiled the devices for their latency and energy utilizing the default settings (imagine just hitting a "lasagna" button on the oven and hoping it picks the right temperature). Now, I added manual control of the operating frequency to find energy-efficient solutions. The goal was "*minimize energy subject to a maximum latency*". It is important to add that we require a minimum latency, or else we might pick a very slow operating frequency and the user gets frustrated when the device is slow to respond. We also explore "*minimize latency subject to a maximum energy*", which would be useful for mobile phones where we want fast responses but don't want to use too much battery life. Finally, we explored privacy preservation, with the goal "*minimize latency subject to keeping some layers private*", which reduces the risk of sending personal information to a third party.

With this new knob of control, we ran experiments and came to some interesting conclusions.

- Compared to picking just the fastest or just the slowest setting, our method finds the "sweet spot" operating frequency somewhere in the middle to optimize our goal.
- The "sweet spot" changes depending on which DNN we use. This would be a bit like different styles of lasagna, with different number of layers, having different oven temperature settings.
- The "sweet spot" also changes depending on the goal, whether it be reducing energy, reducing latency, or preserving privacy.
- Devices have built-in features that try to automate this, but they have a more guess-and-check approach. Our technique that has complete information about the DNN consistently improves or matches the result.

This system allows for us to decide "how" to run a DNN layer in addition to "where". By adding this, we can improve control over latency and energy use of AI.

Serving with Sustainability

In my last major project, instead of focusing on energy, I focused on carbon emissions. Carbon emissions is a key metric in assessing sustainability because it doesn't just measure the quantity of energy, but the quality of that energy.

Carbon emissions can be calculated from the energy using a number called the carbon intensity, which gives the carbon emissions per unit of energy. The key thing is that carbon intensity varies over time, both throughout the day and over the course of the year. A common example is an electricity grid that has solar panels, which provide low-carbon energy, as one source. During the day when the sun is out, the grid's carbon intensity is lower and it rises in the evening as the sun sets. Similarly, the carbon intensity is lower in the summer when the sun is more intense and rises in the winter.

Carbon intensity is also tied to a specific electric grid that services a particular geographic location. In our experiments, we try to find the carbon-optimal layer assignment when devices are in different grids. We "*minimize carbon subject to a maximum latency*", to find a sustainable solution without annoying the user with slow responses. This might be like if we had access to multiple kitchens, one with a traditional oven and one with convection oven. Even though the temperature is the same, the quality of the heat is different. In what cases does it make sense to swap between ovens? How does that change if the other oven is just downstairs or across town?

During our experiments, we used data from different grids including grids in the Midwest and California. As a result, we found:

- When starting in a high-carbon grid, it makes sense to share the work and utilize the low-carbon grid for at least some layers.

- As carbon intensity varies throughout the day, it's possible to achieve lower latency and lower carbon in some cases. Even though lower latency usually

means higher energy, the layer assignment can take advantage of low-carbon areas to mitigate this.

- Energy-optimal and carbon-optimal solutions can differ in many cases. It is not enough to consider energy alone, but we do have the tools to consider carbon and optimize for sustainability.

By considering various power sources, we can direct layer assignment to reduce the carbon emissions. This helps to mitigate sustainability concerns about the growth of AI applications.

10.6 Conclusion

As AI has been rapidly adopted in many aspects of our daily life, I have been really motivated to explore opportunities to improve the systems that are used to implement these. I've shown that thinking about these systems can lead to tangible benefits, including getting faster answers (reducing latency) and preserving privacy of user data. I'm also concerned with the growing energy demands for running AI, and have shown that layer assignment can be an effective tool in addressing energy and sustainability concerns. I am very optimistic about the future of AI developments and believe there is an exciting opportunity to keep improving the computing systems; I hope the reader joins my excitement!

11 CONCLUSION

In this work, I have presented several novel ideas in optimizing distributed inference of DNNs. The key contributions developed in this work include:

- A highly flexible ILP formulation that considers multiple aspects that contribute to inference latency and energy consumption.
- A novel optimization technique for distributed inference that utilizes weight preloading to overlap compute and data movement, reducing overall latency.
- The identification of a problem in the formulation of prior works and introduction of a novel constraint to the ILP to resolve amend this.
- The utilization of device operating frequency as a knob of control to study latency-energy tradeoffs in distributed inference.
- Adapting the ILP to target sustainability of distributed inference, taking into account carbon intensity of each device to minimize the overall carbon footprint.

This work demonstrates the value of a strong fundamental model, our base ILP, in being able to study many variations of a problem. This model may continue to be adapted to various scenarios, such as recent techniques in mixture of experts language models that include only sometimes-activated paths, akin to our exploration of early exiting networks. The techniques discussed may also be combined with orthogonal techniques such as quantization to help explore interesting trade offs,

particularly for edge devices. While this work focuses on inter-layer partitioning, this does not exclude the use of intra-layer partitioning as discussed in Chapter 2. Having a flexible, modular base model allows the exploration of various complementary optimization techniques and may find opportunities to further improve objectives.

This work has been motivated by the rapid innovations in the space of AI, where models have improved through consistent growth in their computational requirements. Developing tools to understand and optimize these systems will be essential as their sprawl continues to drive up their energy requirements and carbon impact. Indeed, it may become important to consider optimal distributed inference layer assignment across entire fleets of edge, hub, and cloud devices. I have explored one avenue of optimization, distributed inference across multiple heterogeneous devices. This work provides interesting insights into the tradeoffs and optimizations available in this avenue; more broadly, I hope it continues to motivate the conversation on AI system optimization.

GLOSSARY

AI artificial intelligence. 1–6, 15, 130

CNN convolutional neural network. 1, 43

DNN deep neural network. 8, 10–12, 14–16, 18, 21, 23, 24, 26, 27, 29, 31–33, 37, 42, 44, 45, 65, 77, 82, 87, 100, 122, 123, 126, 129

DVFS dynamic voltage and frequency scaling. 6, 10, 11, 13, 14, 79, 82

FLOP floating point operation. 38

FLOP/s floating point operations per second. 20, 34, 47

GPU graphics processing unit. 1, 11, 32

ILP integer linear program. 7, 9, 15, 16, 21, 23, 37, 42, 43, 46, 58, 61, 64, 68, 83, 84, 105, 129

IoT internet of things. 1, 4

LAN local area network. 19

MBS maximum bundle size. 59, 65, 66, 68

ML machine learning. 1

NN neural network. 1, 8, 9, 21, 23, 29, 40, 42

NPU neural processing unit. 11

QoS quality of service. 4, 8, 9

WAN wide area network. 20

WLAN wireless local area netowk. 6, 7

BIBLIOGRAPHY

[1] Emna Baccour, Naram Mhaisen, Alaa Awad Abdellatif, Aiman Erbad, Amr Mohamed, Mounir Hamdi, and Mohsen Guizani, “Pervasive AI for IoT applications: A survey on resource-efficient distributed artificial intelligence,” *IEEE Communications Surveys & Tutorials*, pp. 1–1, 2022.

[2] Jorge F. Arinez, Qing Chang, Robert X. Gao, Chengying Xu, and Jianjing Zhang, “Artificial intelligence in advanced manufacturing: Current status and future outlook,” *Journal of Manufacturing Science and Engineering*, vol. 142, no. 11, Aug. 2020.

[3] “Synced AI technology & industry review,” AI-Powered Smart Cameras Help You Maintain a Long-Distance Relationship ...With Your Pet. (Aug. 8, 2020), [Online]. Available: `https://syncedreview.com/2020/08/08/ai-powered-smart-cameras-help-you-maintain-a-long-distance-relationship-with-your-pet/`.

[4] Epoch AI, *Key trends and figures in machine learning*, 2023. [Online]. Available: `https://epoch.ai/trends` (visited on 04/14/2025).

[5] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei, “ImageNet: A large-scale hierarchical image database,” in *2009 IEEE Conference on Computer Vision and Pattern Recognition*, 2009, pp. 248–255.

[6] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton, “ImageNet classification with deep convolutional neural networks,” in *Advances in Neural*

Information Processing Systems, F. Pereira, C. J. C. Burges, L. Bottou, and K. Q. Weinberger, Eds., vol. 25, Curran Associates, Inc., 2012.

[7] Yulong Wang, Xingshu Chen, and Qixu Wang, *Privacy-preserving security inference towards cloud-edge collaborative using differential privacy*, Dec. 13, 2022. arXiv: 2212.06428[cs].

[8] United Nations. "Causes and effects of climate change." (), [Online]. Available: https://www.un.org/en/climatechange/science/causes-effects-climate-change (visited on 04/23/2025).

[9] Microsoft. "Windows update is now carbon aware." (), [Online]. Available: https://support.microsoft.com/en-us/windows/windows-update-is-now-carbon-aware-a53f39bc-5531-4bb1-9e78-db38d7a6df20 (visited on 04/23/2025).

[10] Apple. "iOS 15 brings powerful new features to stay connected, focus, explore, and more." (Jun. 7, 2021), [Online]. Available: https://nr.apple.com/dm4q3o5q4w (visited on 02/20/2023).

[11] James McNiven, *Generative AI is on mobile and it's powered by arm*, May 2024. [Online]. Available: https://newsroom.arm.com/blog/generative-ai-on-mobile.

[12] Jiachen Mao, Xiang Chen, Kent W. Nixon, Christopher Krieger, and Yiran Chen, "MoDNN: Local distributed mobile computing system for deep neural network," in *Design, Automation & Test in Europe Conference & Exhibition, 2017*, Lausanne, Switzerland: IEEE, Mar. 2017, pp. 1396–1401.

[13] Zhuoran Zhao, Kamyar Mirzazad Barijough, and Andreas Gerstlauer, "DeepThings: Distributed adaptive deep learning inference on resource-constrained IoT edge clusters," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 37, no. 11, pp. 2348–2359, Nov. 2018.

[14] Hongshan Li, Chenghao Hu, Jingyan Jiang, Zhi Wang, Yonggang Wen, and Wenwu Zhu, "JALAD: Joint accuracy-and latency-aware deep structure decoupling for edge-cloud execution," in *2018 IEEE 24th International Conference on Parallel and Distributed Systems*, Singapore, Singapore: IEEE, Dec. 2018, pp. 671–678.

[15] Huitian Wang, Guangxing Cai, Zhaowu Huang, and Fang Dong, "ADDA: Adaptive distributed DNN inference acceleration in edge computing environment," in *IEEE 25th International Conference on Parallel and Distributed Systems*, Dec. 2019, pp. 438–445.

[16] Surat Teerapittayanon, Bradley McDanel, and H.T. Kung, "BranchyNet: Fast inference via early exiting from deep neural networks," in *23rd International Conference on Pattern Recognition*, Dec. 2016, pp. 2464–2469.

[17] Liekang Zeng, Xu Chen, Zhi Zhou, Lei Yang, and Junshan Zhang, "CoEdge: Cooperative DNN inference with adaptive workload partitioning over heterogeneous edge devices," *arXiv:2012.03257 [cs]*, Dec. 6, 2020. arXiv: `2012.03257`.

[18] Amir Erfan Eshratifar, Mohammad Saeed Abrishami, and Massoud Pedram, "JointDNN: An efficient training and inference engine for intelligent mobile

cloud computing services," *IEEE Transactions on Mobile Computing*, vol. 20, no. 2, pp. 565–576, Feb. 1, 2021.

[19] Xue Lin, Yanzhi Wang, Qing Xie, and Massoud Pedram, "Task scheduling with dynamic voltage and frequency scaling for energy minimization in the mobile cloud computing environment," *IEEE Transactions on Services Computing*, vol. 8, no. 2, pp. 175–186, 2015.

[20] Saroj Kumar Panda, Man Lin, and Ti Zhou, "Energy-efficient computation offloading with DVFS using deep reinforcement learning for time-critical iot applications in edge computing," *IEEE Internet of Things Journal*, vol. 10, no. 8, pp. 6611–6621, 2023.

[21] Manni Wang, Shaohua Ding, Ting Cao, Yunxin Liu, and Fengyuan Xu, "AsyMo: Scalable and efficient deep-learning inference on asymmetric mobile cpus," in *Proceedings of the 27th Annual International Conference on Mobile Computing and Networking*, Association for Computing Machinery, 2021, pp. 215–228.

[22] Soroush Bateni and Cong Liu, "NeuOS: A Latency-Predictable Multi-Dimensional optimization framework for DNN-driven autonomous systems," in *2020 USENIX Annual Technical Conference* (*USENIX ATC 20*), USENIX Association, Jul. 2020, pp. 371–385.

[23] Xiangjie Li, Yingtao Shen, An Zou, and Yehan Ma, "EENet: Energy efficient neural networks with run-time power management," in *ACM/IEEE Design Automation Conference*, 2023, pp. 1–6.

[24] Ziyang Zhang, Yang Zhao, Huan Li, Changyao Lin, and Jie Liu, "DVFO: Learning-based DVFS for energy-efficient edge-cloud collaborative inference," *IEEE Transactions on Mobile Computing*, pp. 1–18, 2024.

[25] Myeonggyun Han, Jihoon Hyun, Seongbeom Park, Jinsu Park, and Woongki Baek, "MOSAIC: Heterogeneity-, communication-, and constraint-aware model slicing and execution for accurate and efficient inference," in *2019 28th International Conference on Parallel Architectures and Compilation Techniques*, 2019, pp. 165–177.

[26] Ehsan Ahvar, Shohreh Ahvar, Zoltán Ádám Mann, Noel Crespi, Roch Glitho, and Joaquin Garcia-Alfaro, "DECA: A dynamic energy cost and carbon emission-efficient application placement method for edge clouds," *IEEE Access*, vol. 9, pp. 70 192–70 213, 2021.

[27] Mohammad Aldossary and Hatem A. Alharbi, "Towards a green approach for minimizing carbon emissions in fog-cloud architecture," *IEEE Access*, vol. 9, pp. 131 720–131 732, 2021.

[28] Zhendong Song, Menglin Xie, Jinda Luo, Tao Gong, and Wei Chen, "A carbon-aware framework for energy-efficient data acquisition and task offloading in sustainable AIoT ecosystems," *IEEE Internet of Things Journal*, vol. 11, no. 24, pp. 39 103–39 113, 2024.

[29] Huirong Ma, Zhi Zhou, Xiaoxi Zhang, and Xu Chen, "Toward carbon-neutral edge computing: Greening edge AI by harnessing spot and future carbon

markets," *IEEE Internet of Things Journal*, vol. 10, no. 18, pp. 16 637–16 649, 2023.

[30] Ian Goodfellow, Yoshua Bengio, and Aaron Courville, *Deep Learning*. MIT Press, 2016, `http://www.deeplearningbook.org`.

[31] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala, "Pytorch: An imperative style, high-performance deep learning library," in *Advances in Neural Information Processing Systems 32*, H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, Eds., Curran Associates, Inc., 2019, pp. 8024–8035.

[32] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton, "Deep learning," *Nature*, vol. 521, no. 7553, pp. 436–444, May 2015.

[33] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun, *Deep residual learning for image recognition*, 2015. arXiv: `1512.03385 [cs.CV]`. [Online]. Available: `https://arxiv.org/abs/1512.03385`.

[34] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich, "Going deeper with convolutions," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, Jun. 2015.

[35] Karen Simonyan and Andrew Zisserman, "Very deep convolutional networks for large-scale image recognition," *arXiv:1409.1556 [cs]*, Apr. 10, 2015. arXiv: `1409.1556`.

[36] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg, "SSD: Single shot MultiBox detector," in *Computer Vision – ECCV 2016*, Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, Eds., Cham: Springer International Publishing, 2016, pp. 21–37.

[37] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, *et al.*, "An image is worth 16x16 words: Transformers for image recognition at scale," *arXiv preprint arXiv:2010.11929*, 2020.

[38] *ptflops*, 2021. [Online]. Available: `https://github.com/sovrasov/flops-counter.pytorch`.

[39] B. Melander, M. Bjorkman, and P. Gunningberg, "A new end-to-end probing and analysis method for estimating bandwidth bottlenecks," in *IEEE Global Telecommunications Conference*, vol. 1, 2000, pp. 415–420.

[40] Ralf Lübben, Markus Fidler, and Jörg Liebeherr, "Stochastic bandwidth estimation in networks with random service," *IEEE/ACM Transactions on Networking*, vol. 22, no. 2, pp. 484–497, 2014.

[41] Sukhpreet Kaur Khangura, Markus Fidler, and Bodo Rosenhahn, "Neural networks for measurement-based bandwidth estimation," in *International Federation for Information Processing Networking Conference*, 2018, pp. 1–9.

[42] Jeremy Howard, *Imagenette*. [Online]. Available: https://github.com/fastai/imagenette/.

[43] *Time — time access and conversions*, 2025. [Online]. Available: https://docs.python.org/3/library/time.html.

[44] VMW Research Group. "The GFLOPS/W of the various machines in the VMW Research Group." (Oct. 2020), [Online]. Available: http://web.eece.maine.edu/$%5Csim$vweaver/group/green%5C_machines.html (visited on 11/19/2021).

[45] Lucy Hattersley. "Raspberry Pi 3B+ specs and benchmarks." (Mar. 2018), [Online]. Available: https://magpi.raspberrypi.com/articles/raspberry-pi-3bplus-specs-benchmarks (visited on 11/19/2021).

[46] *NVIDIA Jetson Nano developer forum*, 2019. [Online]. Available: https://forums.developer.nvidia.com/t/help-question/71758.

[47] *NVIDIA Jetson Nano system-on-module data sheet*, NVIDIA, Feb. 2020.

[48] TechPowerUp. "GeForce GTX 1080 Ti specs." (), [Online]. Available: https://www.techpowerup.com/gpu-specs/geforce-gtx-1080-ti.c2877 (visited on 11/19/2021).

[49] Gurobi Optimization, LLC, *Gurobi Optimizer Reference Manual*, 2021. [Online]. Available: https://www.gurobi.com.

[50] Robert Viramontes and Azadeh Davoodi, "Neural network partitioning for fast distributed inference," in *IEEE International Symposium on Quality Electronic Design*, 2023, pp. 1–7. DOI: 10.1109/ISQED57927.2023.10129343.

[51] Silke Horn, *Linear program with ceiling or floor functions (how?)* https://support.gurobi.com/hc/en-us/community/posts/360054499471-Linear-program-with-ceiling-or-floor-functions-HOW-. (visited on 08/16/2023).

[52] Sergio Saponara, Abdussalam Elhanashi, and Alessio Gagliardi, "Real-time video fire/smoke detection based on CNN in antifire surveillance systems," *Journal of Real-Time Image Processing*, vol. 18, pp. 889–900, 2021.

[53] Joo Seong Jeong, Jingyu Lee, Donghyun Kim, Changmin Jeon, Changjin Jeong, Youngki Lee, and Byung-Gon Chun, "Band: Coordinated multi-DNN inference on heterogeneous mobile processors," in *Proceedings of the 20th Annual International Conference on Mobile Systems, Applications and Services*, Portland, Oregon: Association for Computing Machinery, 2022, pp. 235–247.

[54] Adam B. Noel, Abderrazak Abdaoui, Tarek Elfouly, Mohamed Hossam Ahmed, Ahmed Badawy, and Mohamed S. Shehata, "Structural health monitoring using wireless sensor networks: A comprehensive survey," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 3, pp. 1403–1423, 2017.

[55] Chengshuai Shi, Lixing Chen, Cong Shen, Linqi Song, and Jie Xu, "Privacy-aware edge computing based on adaptive dnn partitioning," in *2019 IEEE Global Communications Conference*, 2019, pp. 1–6.

[56] United Nations Environment Programme, *Artificial intelligence (AI) end-to-end: The environmental impact of the full AI lifecycle needs to be comprehensively assessed - issue note*, Sep. 2024. [Online]. Available: https://wedocs.unep.org/20.500.11822/46288 (visited on 03/10/2025).

[57] Mark Morey, *Data center owners turn to nuclear as potential electricity source*, Oct. 2024. [Online]. Available: https://www.eia.gov/todayinenergy/detail.php?id=63304.

[58] Demetris Trihinas, Lauritz Thamsen, Jossekin Beilharz, and Moysis Symeonides, "Towards energy consumption and carbon footprint testing for ai-driven iot services," in *2022 IEEE International Conference on Cloud Engineering*, 2022, pp. 29–35.

[59] Electricty Maps, "United States 2024 hourly carbon intensity data (version January 26, 2025)," [Online]. Available: https://www.electricitymaps.com..

[60] Junxian Huang, Feng Qian, Alexandre Gerber, Z. Morley Mao, Subhabrata Sen, and Oliver Spatscheck, "A close examination of performance and power characteristics of 4G LTE networks," in *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services*, Low Wood Bay, Lake District, UK: Association for Computing Machinery, 2012, pp. 225–238.