UNIVERSITY OF CALIFORNIA SAN DIEGO

Enabling Deep Learning Inference on Resource Constrained Devices

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

in

Electrical Engineering (Computer Engineering)

by

Basar Kutukcu

Committee in charge:

Professor Sujit Dey, Chair
Professor Ryan Kastner
Professor Farinaz Koushanfar
Professor Tajana Rosing

2024

The Dissertation of Basar Kutukcu is approved, and it is acceptable in quality and form for publication on microfilm and electronically.

University of California San Diego

2024

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to extend my deepest gratitude to my advisor, Professor Sujit Dey, for his invaluable support and guidance throughout my Ph.D. journey. His visionary approach to research and his passion for exploring the unknown have been a constant source of inspiration. Under his mentorship, I have grown both academically and personally, gaining knowledge and skills that will influence my career and life for years to come. Working with him has been a transformative experience, and I will always cherish the lessons I have learned under his guidance.

I would also like to thank the rest of my committee members, Professor Ryan Kastner, Professor Farinaz Koushanfar, Professor Tajana Rosing for their valuable comments and directions.

I would also like to sincerely thank Professor Sabur Baidya for his collaboration throughout my Ph.D. journey. His insightful comments and innovative ideas during our discussions have greatly enriched my understanding and played a significant role in shaping this dissertation. Also, I would like to thank all members of the MESDAT Lab for their friendship and the insightful discussions we shared.

Lastly, I would like to express my heartfelt gratitude to my family for their unwavering support and love throughout my life. Even from the other side of the world, their constant presence and encouragement have been a source of strength during every challenge I faced. This thesis would not have been possible without their enduring support.

Chapter 1, in full, is a reprint of the material as it appears in ACM Transactions on Embedded Computing Systems 2022, Basar Kutukcu, Sabur Baidya, Anand Raghunathan, Sujit Dey. The dissertation author was the primary investigator and author of this paper.

Chapter 2, in full, is a reprint of the material as it appears in ACM Transactions on Embedded Computing Systems 2024, Basar Kutukcu, Sabur Baidya, Sujit Dey. The dissertation author was the primary investigator and author of this paper.

Chapter 3, in full, is a reprint of the material as it appears in IEEE Access 2024, Basar

Kutukcu, Sabur Baidya, Sujit Dey. The dissertation author was the primary investigator and author of this paper.

VITA

2018	Bachelor of Science, Middle East Technical University
2020	Master of Science, Middle East Technical University
2020–2024	Research Assistant, University of California San Diego
2024	Doctor of Philosophy, University of California San Diego

PUBLICATIONS

B. Kutukcu, S. Baidya, and S. Dey, "Fast and Scalable Design Space Exploration for Deep Learning on Embedded Systems", *IEEE Access*, vol. 12, pp. 148254–148266, 2024.

B. Kutukcu, S. Baidya, and S. Dey, "SLEXNet: Adaptive Inference Using Slimmable Early Exit Neural Networks", *ACM Trans. Embed. Comput. Syst.*, vol. 23, no. 6, Sep. 2024.

B. Kutukcu, S. Baidya, A. Raghunathan, and S. Dey, "EvoSh: Evolutionary Search with Shaving to Enable Power-Latency Tradeoff in Deep Learning Computing on Embedded Systems", in *2023 IEEE 36th International System-on-Chip Conference (SOCC)*, 2023, pp. 1–6.

B. Kutukcu, S. Baidya, A. Raghunathan, and S. Dey, "Contention Grading and Adaptive Model Selection for Machine Vision in Embedded Systems", *ACM Trans. Embed. Comput. Syst.*, vol. 21, no. 5, Oct. 2022.

B. Kutukcu, S. Baidya, A. Raghunathan, and S. Dey, "Contention-aware adaptive model selection for machine vision in embedded systems", in *2021 IEEE 3rd International Conference on Artificial Intelligence Circuits and Systems (AICAS)*, 2021, pp. 1–4.

ABSTRACT OF THE DISSERTATION

Enabling Deep Learning Inference on Resource Constrained Devices

by

Basar Kutukcu

Doctor of Philosophy in Electrical Engineering (Computer Engineering)

University of California San Diego, 2024

Professor Sujit Dey, Chair

This study aims to enable deep learning models on resource constrained devices considering system and application requirements. The proven success of deep learning models can be extended into people's life even more by using them on mobile systems since mobile systems are ubiquitous in people's lives. However, mobile systems are resource constrained by their nature. The deep learning models require high computation power and resource constrained devices do not have high computation power. This contradiction makes the research for enabling deep learning models on resource constrained devices necessary.

In the first part of the study, we focus on the contention in a system and how it affects our deep learning-based application. Since today's systems are very complex and handle many

tasks at the same time, the tasks can create contention in different parts of the system. This contention may create undesired and unpredictable effects on the applications. In this study, we study the contention in the systems and create a model selection framework which makes deep learning-based applications contention agnostic with limited accuracy cost.

In the second part of the study, we focus on dynamic neural network architectures. The dynamic neural networks are very useful for mobile systems as their conditions and requirements may change frequently. In this study, we develop slimmable early exit deep learning models and an efficient branch selection algorithm. Our dynamic neural network architecture is shown to be effective to adapt changing latency and power requirements.

In the third and last part of the study, we focus on scalable design space exploration to find pareto optimal design configurations for deep learning models on embedded systems. Finding the correct configurations for executing deep learning models on embedded systems is a challenging problem since testing a configuration is costly in time and the design space is vast. In this study, we develop a fast and scalable exploration algorithm that works well for extremely large search spaces.

Introduction

Deep learning has proven to be superior to the previous methods and been very successful in various fields including, but not limited to, autonomous systems, generative models, and reinforcement learning. The different deep learning architectures are used for different tasks. These architectures include convolution based neural networks, transformer based large language models, fully connected neural network-based reinforcement learning agents and many more. Many of these architectures have been showing better performance compared to non-deep learning-based methods. However, this superior performance comes with certain costs. The deep learning models usually require a large computing power even for inference. While this is not an issue for large servers, it makes it harder to use these models on mobile systems. However, using these models on mobile systems would increase their impact on people's lives. Therefore, it is important to study how to use deep learning-based models efficiently on the resource constrained mobile systems is needed.

In Chapter 1, we consider the contention caused by multiple tasks running on a resource constrained system, its impact on deep learning-based applications, and how to mitigate this impact through an application accuracy trade-off. The modern systems run complicated tasks that include multiple stages with different sub-applications. These sub-applications may require the same resource of the system at the same time, hence cause a contention in the system. This contention creates unwanted and unpredictable effects on the applications on the system such as delays in the latency and increase in power consumption. We develop a model selection framework for deep learning-based applications to grade the contention on the system and pick a suitable model to avoid the contention on the runtime. Our experiments show that our model

selection framework can achieve near contention agnostic deep learning model execution with minimal application accuracy sacrifice.

In Chapter 2, we consider a dynamic neural network architecture and how it can be used for adaptive execution on a resource constrained system with changing system requirements. Many of the resource constrained mobile systems have unpredictable requirement changes due their mobility. Therefore, a deep learning-based application needs to adapt to the system's changing requirements. We develop a dynamic neural network architecture that combines slimming and early exit techniques to get each technique's strengths. Moreover, we develop a branch selection algorithm that considers the current state of the system and picks the branch of our architecture that satisfies the requirements of the application and system. Our experiments show that our architecture and branch selection algorithm achieve better results than just slimming and just early exit techniques for adapting changing requirements and conditions on a mobile system.

In Chapter 3, we consider a design space exploration problem for the software and hardware configurations of running deep learning-based application on resource constrained embedded systems. This is a very challenging problem since the design space is extremely large, testing a sample configuration on the actual system takes long time and there's no known analytical form of the objective function that we are trying to optimize to find pareto optimal configurations in the design space. The problems that carry the similar challenges are usually solved by Bayesian Optimization and its variants. However, Bayesian Optimization is shown to be not scalable for extremely large search spaces. We develop a sampling-based search algorithm that can scale to search extremely large search spaces. Our experiments show that our search algorithm shows superior performance to Bayesian Optimization-based methods in much shorter time.

Chapter 1

Contention Grading and Adaptive Model Selection for Machine Vision in Embedded Systems

1.1 Introduction

Modern machine vision systems involve complex deep learning based algorithms [37, 53, 67, 75] that need significant computing resources to run under reasonable time limits. However, this is very challenging when the algorithms need to be realized in a resource-constrained system. Moreover, in many cases, the computing system running the machine vision application shares resources with other coexisting computing loads. For example, connected and autonomous vehicles process camera data together with RADAR and/or LiDAR data on the same computing system for better fused perception in crowded areas or in the presence of obstacles [22, 38, 88]. In such scenarios, the machine vision workload can contend with the other workloads for the computing system resources, further increasing the application latency. While increasing the priority of certain tasks can improve their latency, it can cause a starvation for the other tasks running on the same system. Instead, an alternative approach is to handle contention by adapting the machine vision workload to best utilize the computing resources available in the presence of contention. The machine vision is thus realized by choosing an appropriate model from a set of neural network-based image classification models, fitting the available computing resources.

In this work, we examine the effects of contention on the image classification application, and propose a contention-aware adaptive model selection framework that minimally compromises the accuracy of the image classification application while satisfying the latency requirements.

Several previous efforts have explored reduced complexity models that fit within the constrained capabilities of embedded systems [55, 101]. These approaches typically involve a tradeoff between compute/storage requirements and model accuracy. However, contention-impacted systems present a moving target, since the contention levels may vary over time, effectively presenting different resource levels that we would like to fully utilize in order to achieve the lowest impact on application accuracy. Thus, the major challenges in a real-time system with contention are (i) to accurately predict the level of contention in the system, and (ii) adapt the application accordingly maintaining the performance constraint of the application. As many modern systems may not allow applications to access low-level system information in real-time due to security concerns or complexity of the platform, herein, we infer the level of contention from its impact on the application performance. We propose an application-level data-driven predictive framework for contention-aware adaptive model selection that aims to minimize the cost of system resources and overhead of our framework, while maintaining system performance stability in presence of dynamic workload variations.

Now, as the model selection framework needs to load a number of pre-trained models in memory and dynamically select a model at runtime, the length of the model set can not only impact the memory overhead but also the model switching cost and stability. Hence, we also propose a contention grading mechanism that intelligently selects the appropriate deep learning models in the model-set used by the adaptive model selection framework.

The main contributions of this work are as follows:

- An application-level profiler for system contention and a methodology to automatically regenerate the profiled system contention in a controlled environment.

- An offline model set pruning methodology that selects the optimal models for a given

system contention profile and specific user requirements.

- A runtime model selection mechanism for an image classification application that adapts based on the system contention to stay below a predefined delay threshold.

We implement the framework on the Nvidia Jetson TX2 platform and show the advantage of our adaptive model selection framework in dynamic contention scenarios. We empirically show that our model pruning methodology improves the runtime model selection performance resulting in better tradeoff between latency and accuracy, and also reduction in memory overhead. The framework also shows the advantage of model selection from a set of independently trained models compared to early exit techniques [79, 86].

1.2 Related work

The challenge of enabling deep learning models on resource constrained devices has been extensively researched in previous efforts. One of the main techniques is designing ground-up efficient models that require less resources than high accuracy models while sacrificing accuracy as little as possible [36, 70, 80, 97]. In [36], a squeeze-and-expand architecture is created by using 1x1 and 3x3 filters together. In [97], a channel shuffling operation is created after 1x1 group convolutions. In [70], linear bottlenecks and inverted residuals are used together with depthwise convolutions. In [80], the DenseNet architecture [35] is modified to design an efficient deep neural network. Unlike our framework, these approaches create a single efficient model and do not consider dynamic changes in computing resources available due to contention in the system. In the presence of contention, their inference delays will still increase unexpectedly. Similarly, when there is no contention, they miss the opportunity to use the available resources for higher accuracy since the design is fixed. These efforts are orthogonal to our work, and our proposed approach and framework will apply to all neural network models, including these compute-efficient models as well.

Another way of creating efficient deep learning models is to use quantization to come up

with efficient designs out of any neural network model. Quantization decreases the precision of neural network weights and activations to improve efficiency. There are several efforts addressing neural network quantization in the literature [28]. Quantization can be considered in two categories, namely quantization-aware training [13,65,76,101] and post-training quantization [4, 57]. Quantization is applied during training in the quantization-aware training methods. In [13], the weights and activations are constrained to -1 and +1. In [101], quantization clusters are learned during training together with weights. In [65], the weights and activations are quantized to binary values to allow XNOR and bitcount implementation of expensive operations. In [76], quantization is designed for graph neural networks. Quantization is applied after training in post quantization methods. In [4], 3 post training techniques are defined and their combinations are used for 4-bit quantization. In [57], the weights in different layers of a neural network are scaled to decrease the error caused by quantization. These approaches do not consider contention either. Therefore, their inference delays are vulnerable to dynamic system contention as well.

Pruning is another method that is similar to quantization in terms of its objectives and design flow. The less important weights of neural networks are pruned to create efficient neural network models. Pruning methods can be categorized into two main methods, namely unstructured and structured pruning [7]. Unstructured pruning removes individual parameters or neurons [29]. Structured pruning removes the coarse-grained structures such as filters or channels [55]. The effect of contention on pruning methods is similar to the quantization. They do not consider contention and their inference delays can be affected by contention.

All the previous related work is focused on creating one fixed efficient design with minimal accuracy loss. However, multiple models can be used to find a balance point between efficiency and accuracy. Real-time model selection is previously investigated in literature for different scenarios. In [78], the authors measure the input image's complexity before classification and select the ideal model for the specific input image content. In [62], two models are employed, one big and one little. Each image is classified by the little model first. Then, if the classification is found unsuccessful, big model classifies the same image again. However,

the unsuccessful attempts in this method increase the latency, and hence, are not suitable for real-time machine vision system. In [21], a CNN based multiplexer is trained to select the optimal deep learning model for the given input image. The multiplexer considers the input image's complexity. In [23], the optimal model in a model set is selected or the current model is adapted when a class skew is detected in the input. The model set is prepared by pruning models by considering class skew. The aforementioned methods do not take the contention of the underlying computing system into account for model selection. In [94], a model switching methodology is developed to improve the performance for cloud servers which do not have strict delay constraints like embedded systems. In this work fluctuating workloads are considered as contention in the server environment.

Using early exit models can be an alternative to model switching. Defining early exit points in a neural network creates incremental sub-models where a latency-accuracy tradeoff occurs between these early exit points. In [79], this methodology is applied on multiple neural networks and latency-accuracy tradeoff is demonstrated. In [93], a methodology is proposed to convert any CNN to a multistage model. The stages of this multistage model are selected at runtime. In [86], an early exit model is designed to be used during runtime. The input and the contention are considered to select an approximate branch of the predefined early exit model. The contention is determined by matching previous inference delays of approximate branches and a look-up table that consists of benchmarks of each approximate branch. The runtime part of our work is different in two aspects. First, instead of approximate branches, we use multiple models that are specifically tuned to provide different accuracy-compute tradeoffs, *e.g.*, selected from existing resource-efficient deep learning model designs. This results in a better efficiency-accuracy tradeoff. Second, our contention measurement is embedded in regression models instead of look-up tables. Also, our delay normalization mechanism allows us to use different models' inference delay to measure contention. This can be useful when contention and therefore model selection change rapidly. In Section 1.5.5, we provide results demonstrating the advantages of our approach over early exit based technique [86].

Slimmable neural networks [92] are dynamic neural networks like early exit models. Slimming operation scales the model width by changing the number of channels in each layer. Therefore, slimming creates sub-models as well where a latency-accuracy tradeoff occurs between switches. In [91], slimmable networks are improved to be able to use arbitrary widths instead of predefined widths. In [90], a width search strategy is proposed for the number of channels instead of using predefined or arbitrary widths. Even though slimmable networks are not experimented with switching when contention exists, it is possible to use them in that way. Therefore, we compared our methodology with slimmable neural networks in Section 1.5.6.

In our previous work [43], we examined runtime model selection in the presence of contention. However, this work was assuming the contention levels are known beforehand and expecting a model set that is already tailored for the contention levels. In this work, we extended our previous work by adding an application level profiler to determine the contention levels and adding a model set pruning methodology to find optimal model set for the existing contention levels from a given large number of models.

1.3 Overview of our Approach

1.3.1 Machine Vision Application

In this paper, we consider machine vision applications, specifically focusing on the image classification block within them. There are two main metrics that define the performance of image classification applications - inference delay and accuracy. In any machine vision system, it is desirable to minimize the inference delay and maximize the accuracy of the image classification task.

The presence of multiple, concurrently executing tasks in a computing system causes contention for the specific system resources, eventually resulting in increased delay for the completion of the tasks. These contentions and their impact are seen more frequently in resource-constrained embedded systems. For demonstration of contention and its impact, we consider

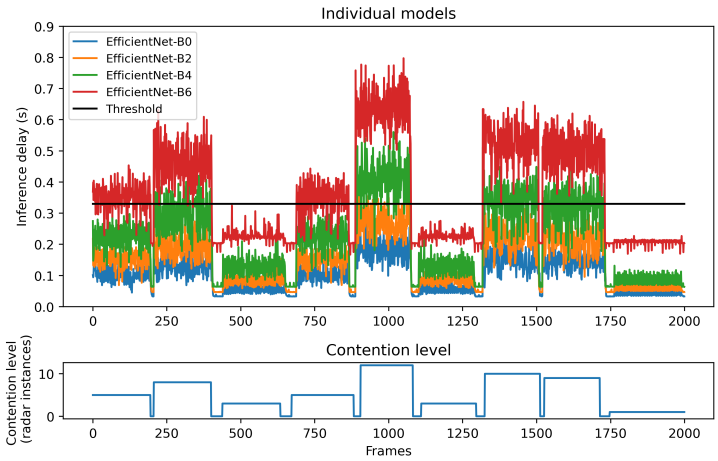


Figure 1.1. Inference delays of different models under changing contention

sensor fusion in autonomous vehicles.

Sensor Fusion and Contention in Autonomous Vehicles: Autonomous vehicles operate in a complex environment and therefore require a high level of perception that is achieved by using multiple sensors and their fusion. Autonomous cars have three main sensors - Camera, LiDAR, and RADAR. The fusion of these sensors can achieve better accuracy than using each sensor individually. However, this performance improvement comes with a cost since using more sensors requires more computation power. Further, the processing of each sensory modality results in contention, whose effects are especially significant in resource-constrained settings.

There are different fusion approaches as reviewed in [88]. This work categorizes fusion approaches into three levels, high-level [38], mid-level [49], and low-level [89]. All of these fusion approaches incur a processing cost in addition to sensors' individual processing costs. The computation for the fusion task is also affected by the system contention as well as contributing to it.

1.3.2 Delay Accuracy Trade-off

The trade-off between inference delay and accuracy is fundamental to image classification systems as more complex image classification algorithms result in higher accuracy but also require more time to compute those results. Since contention creates dynamic variations in the

available compute resources, a machine vision system needs to optimize its performance in terms of delay and accuracy. Typical autonomous systems need to satisfy a delay constraint (*e.g.*, operate under a certain frame rate), while maximizing accuracy. In order to achieve this objective, we propose to use a set of N image classification models $\{M_i\}, i = 1,..,N$ with increasing complexity. Depending on the available resources in presence of contention, the system must choose the optimal model. For example, the inference delays of four different EfficientNet [77] models under varying contention are shown in Figure 1.1. The contention is created by multiple radar instances running on the same computing platform. The contention level graph at the bottom of Figure 1.1 shows varying contention levels. Each model's inference delay increases proportionally under increasing contention. Figure 1.1 shows that if we have an inference delay constraint, we can satisfy it by choosing an appropriate model for each contention level. For example, all of the models satisfy the delay threshold around frame 500 since the contention level is low. Therefore, the most complex model can be chosen at this contention level. However, the contention level is very high around frame 1000. EfficientNetB6 and EfficientNetB4 do not satisfy the delay threshold at this contention level. Therefore, the third most complex model, EfficientNetB2, should be chosen at this contention level. Choosing the simpler model satisfies the delay threshold, however it also results in accuracy loss.

1.3.3 Adaptive Model Selection At Runtime

For selection of an appropriate model, one needs to have a priori knowledge about the contention level in the system when the model will be executed, and select the best model that fits in the available resources. Since it is not possible to know future contention levels precisely, one can estimate the contention level based on recent history, project the impact of contention on different image classification models, and select the best model for the next image frame.

The proposed framework predicts the future inference delays of the model set $\{M_i\}$ in the presence of contention. Then, we find a subset of models $\{M_j | D_j < T\}$, $j = 1,..,L, L \leq N$, where D_j is the predicted inference delay of model M_j, T is a latency threshold of the system

and N is the total number of models in the model set during runtime. After that, we choose the appropriate model M_k such that the accuracy $A_k = \max\{A_j\}$.

1.3.4 Contention Grading and Defining Model Set

The adaptive model selection framework works at runtime and requires a set of models to be defined before runtime. Defining the model set is the other side of this problem and imposes an important challenge. The optimal model set differs based on the aim of the models, the system contention levels and the user requirements. There are usually a very large number of models available across the entire latency-accuracy tradeoff space. Having a large number of models to choose from can lead to the runtime framework incurring excessive overheads and/or switching models more frequently than needed. Thus, **it is important to define a model set that is minimal in size, while still offering sufficient options to adapt to the observed contention levels**. The optimality of a model set depends on satisfying the inference delay constraint while maximizing accuracy using minimum memory.

We find the optimal model set by measuring system contention and pruning a given model set based on the effects of contention in the system. In order to measure system contention, we profile the system using a specifically designed profiler. Then, we regenerate the system contention in a controlled environment to prune our model set. We have 3 pruning stages where we use independent notations in the following paragraphs. In the first stage, we remove Pareto-inferior models in the model set. Given a model set $\{M_i\}$, $i = 1,..,N$ where N is the number of models, for each model M_k, we find the subset of $\{M_i\}$ as $M_{kt} = \{M_j | D_j \leq D_k\}$ where D_j is the inference delay of model M_j. Then, let the A_{kt} be the accuracy set of the model set M_{kt}. If $max(A_{kt}) \neq A_k$, then we prune M_k from $\{M_i\}$. In the second stage, we remove models that have small gains compared to their adjacent models. These models have small gains with high cost where the cost includes inference delay and memory consumption. Given a model set $\{M_i\}$, $i = 1,..,N'$, where all models are on Pareto frontier and N' is the number of models, we find the slopes of each adjacent model pair as $S_j = \dfrac{A_{j+1} - A_j}{D_{j+1} - D_j}$ where A_j is the accuracy and D_j is the

inference delay for M_j. Then given a lower (L_l) and higher (L_h) limits for the slopes, we prune the model M_j if $S_j > L_h$ or M_{j+1} if $S_j < L_l$. Whenever a pruning occurs, we recalculate all of the slopes and restart pruning. In the last stage, we remove the models that have no use in the contention levels of the system. In this step, the user requirements and the system contention levels are considered. Given a model set $\{M_i\}$, $i = 1,..,N''$ where N'' is the number of models, Contention levels $\{C_j\}$, $j = 1,..,P$ where P is the number of contention levels and the inference delay threshold T, we run the all models on each contention level to find inference delays $\{D_{ij}\}$. Then we find the most accurate model M_k that satisfies the delay threshold for each contention level j such that $A_k = max(A_i|D_{ij} < T)$. Then we add M_k to the valid model set and prune the rest of the models.

1.4 Contention Grading, Model Set Generation and Adaptive Model Selection: Details

Our system consists of offline contention grading and model set pruning, followed by a runtime adaptive model selection. We discuss the details of each of these phases in this section.

1.4.1 Contention Grading

Contention grading and model set pruning comprises of three components. The first component is profiling system contention on the target system during runtime. The second component is mimicking the profiled system contention for detailed analysis of the model set. The last component is model set pruning, which outputs the optimal subset of the input model set.

Profiling System Contention

Profiling the contention in a system can be a very complex task because of two main problems. The first one is - P1: the contention is created by overlapping execution of many different tasks. Therefore, the combination of these different tasks creates unpredictable contention

levels. The second one is - P2: the difficulty of detecting the contention point. There are many modules (CPU, GPU, memory, bus etc.) in a computer system which can be requested by the tasks at the same time, resulting in a contention in these modules.

As a result of complex contention scenarios, instead of profiling the system contention, we decided to profile the impact of the contention to our application. In order to do that, we run a sample of our application along with all other tasks and measure the performance of our application. For example, since our application is a neural network based one, we run a sample neural network on the system and measure its inference delays to understand different levels of contention that are important to neural network based applications. We named this profiler as Contention Impact Profiler (CIP). CIP should be run on the system for long enough to observe all contention levels.

Using CIP solves the first problem (P1) because we observe the effect of contention and see the root of contention as a black box which can be a single task or multiple of them. This method also ignores the contention that has no effect on our application and therefore simplifies the contention profiling. Using CIP solves the second problem (P2) of contention profiling since it does not try to identify the contention point.

The CIP is needed for our framework for two aspects. The first one is the need of knowing the specific contention levels in a system and regenerating them in a controlled offline environment. These known and controlled contention levels are required and used in our model set pruning methodology. If we do not have these known and controlled contention levels, we cannot prune a model set for the target system with contention. The second one is the requirement for the generalization. The model set pruning is designed to work for any system and requirement. However, it requires system and contention specific information for each case. Measuring such information for each system would require a significant amount of effort and would decrease the value of our framework. The CIP covers this aspect by working as a black box in any system and collecting the required system and contention specific information by model set pruning stage.

Mimicking Contention

Once we know the levels of contention, we mimic this contention in a controlled environment for the purpose of selecting a model set. We propose the use of Artificial Contention Units (ACUs) for this purpose. An ACU is a dummy workload used for the purpose of producing a specific level of contention. Different numbers of ACUs are used to generate different levels of contention. An ACU is composed of dummy instructions including vector addition, vector multiplication and FFT operations. These operations are big and diverse enough to create contention and small enough to give us a fine grained control over contention creation when multiple of them are used.

We use the same CIP that is used to profile the system contention, to profile contention synthetically created by ACUs. During this profiling, we increase the number of ACUs incrementally and measure the inference delays of the CIP. This creates an inference delay trace of the CIP under increasing ACU contention in addition to the inference delay trace of the CIP under system contention as we obtained in 1.4.1. Note that both traces are measured on the same hardware by the same CIP.

At this point, we get back to the inference delay trace of the CIP under system contention and do some preprocessing. First, we take the moving average of the trace to remove noise. Then, we apply kernel density estimation to data to find the contention levels. In the end, we have the number and intensity of the contention levels in the system. At the last step, we match the system contention levels and ACU contention levels with same intensity. As a result, we have the set of ACU contention levels that can regenerate the system contention. Since we have a complete control over using ACUs, we systematically use them to measure the performance of all of our models and prune our model set.

Model Set Pruning

The aim of contention grading is to prune the model set and propose an optimal subset. Profiling system contention and mimicking it are done to enable model set pruning. Subsequently,

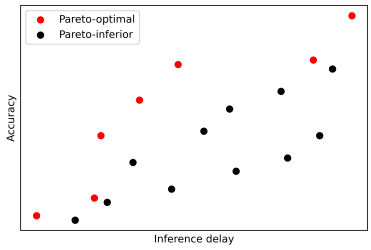


Figure 1.2. Pareto-optimal and Pareto-inferior of a hypothetical model set

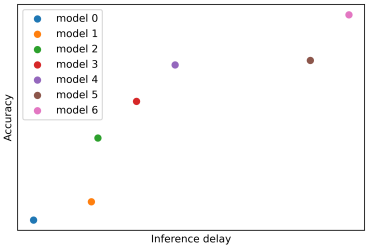


Figure 1.3. Hypothetical model set before transition pruning

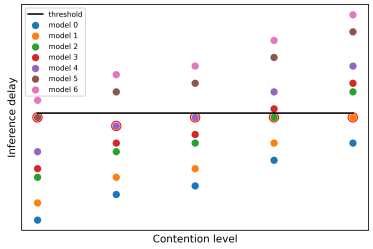


Figure 1.4. Hypothetical model set before contention pruning

we prune the model set in three stages, which are described below.

Pareto Pruning: Pareto pruning is the first pruning stage. In this stage, we remove the models that are not on the accuracy-inference delay Pareto frontier of the model set. This is because the models that are not on the Pareto frontier should not be used in any application. If a model is not on the Pareto frontier, there is at least one other model in the model set that performs better with less cost. So, the models that are not on the Pareto frontier are either obsolete or poorly designed for the image classification task at hand. Pareto-optimal and Pareto-inferior models of a hypothetical model set are shown in Figure 1.2.

To construct the accuracy-inference delay, we use the average inference delay over all the contention levels of the system in this stage. This enables us to consider the overall response of models to system contention. If a model is on the Pareto frontier without contention, but the inference delay of the model increases more than other models in the presence of contention, then this model may lose its position on the Pareto frontier.

At the end of this stage, we have a model subset where each model presents a unique tradeoff between accuracy and inference delay.

Transition Pruning: Pareto pruning creates a model subset where each model shows a non-zero improvement in one metric with respect to models adjacent to it in the Pareto frontier. However, some of those models might not be beneficial in practice. This is because Pareto optimality considers any gain without evaluating the actual magnitude of the gain. There can be a very small gain with very high cost or a very high gain with very small cost, leading to models that do

not get used in practice.

In our case, the gain is accuracy and the cost is inference delay or memory consumption. We use inference delay as our cost for our pruning calculations. However, since the inference delay and memory consumption are usually correlated (since both depend on the number of parameters in the model), our pruning in this stage improves memory consumption as well.

A hypothetical model set is shown in Figure 1.3. All models are Pareto optimal in this model set. However, two minimally useful transitions can be noticed. These transitions are - model 1 to model 2, and model 4 to model 5. First, let's consider the transition model 1 to model 2. There is a very high accuracy improvement from model 1 to model 2. However, their inference delays are almost the same. Therefore, the model 1 can be dropped from the model set. The second transition has a similar problem but in the opposite direction. There is a very small accuracy improvement from model 4 to model 5. However, model 5 requires significantly larger time to achieve this accuracy. Therefore, model 5 is not useful in this model set. Removing these models from the model set does not only simplify the model set but also improves the performance of adaptive model selection by eliminating the use of these models, and hence the associated overheads, at runtime.

When the given example is examined, a certain pattern can be noticed when there is an inefficient transition. This pattern is the slope of the transition. If the slope is too low or too high, it is an inefficient transition. We therefore use the slope to identify inefficient transitions and eliminate them. The lower and higher threshold of the slope can change depending on the problem and user requirements. Therefore, these values are hyperparameters in our methodology.

Contention Pruning: In this final stage of pruning, the model set is pruned considering specific contention levels and user requirements. Our pre-deployment profiler (CIP) gives the specific contention levels that can be observed in the system. The number of different contention levels is important because it also limits the number of models in the model set. Given a contention level, there can be only one best model, because only one model has the maximum accuracy among the models that satisfy the inference delay threshold (user requirement) at a specific contention level.

Note that the vice versa is not true - one model can be the best model for multiple contention levels. As a result, we can say that **the total number of models must be less than or equal to the total number of contention levels** and we consider this rule in the contention pruning stage.

Before explaining this stage of pruning, note that Pareto pruning creates a model set where accuracies and inference costs vary monotonically. Therefore, if a model has a higher inference delay than another model, it is also more accurate than the other model. This property is used to define the more accurate model by looking at the inference delay in contention pruning stage.

Before contention pruning, we need to run all of our models under the contention levels that we found in the previous steps. We use ACUs to mimic the system contention, run each model under each contention level and save the average inference delays. The output of this part can be observed in Figure 1.4 where a hypothetical model set is used for explanation purposes. In the figure, there is an inference delay threshold which is shown as a black line. This is the user requirement which basically defines the maximum acceptable inference delay. Therefore, we want our models to perform under this threshold while being as much as accurate. The x-axis of the figure shows the contention level. The increasing contention levels and the models' responses are shown from left to right. In each contention level, we find the model that is just below the inference delay threshold and mark it as valid model. In the end, any model that is not in the valid list is pruned. The valid model in each contention level is shown with a red circle around it.

When we examine the pruned models, we can see that they are not suitable for the user requirement and contention profile of the system. For example, model 6 requires too much time and is not suitable even in smallest contention level. The model 0, on the other hand, is always below the inference delay threshold. However, the contention never increases up to a point where using model 0 is the optimal choice. Another pruned model is the model 3. Model 3 is below the threshold in some contention levels and above it in some other contention levels. So, it is expected that model 3 should be optimal at one contention level. However, as we can see from this example, the contention does not have to increase gradually at every level. A system may

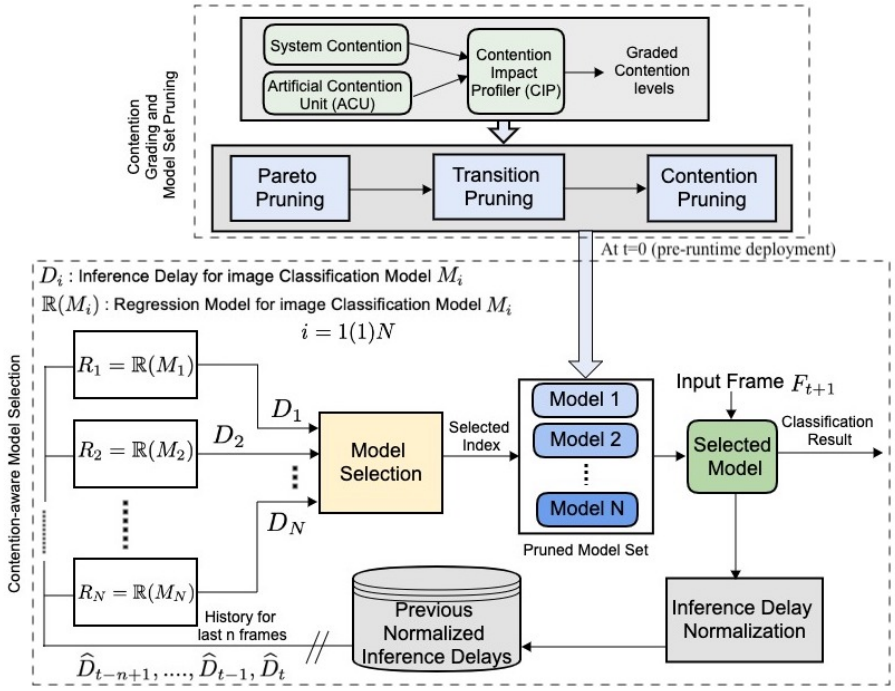


Figure 1.5. Overview of the proposed framework

experience a jump in contention which eliminates the need for middle level models. We also notice that model 4 is the optimal choice for two contention levels. This can happen when some contention levels are close to each other.

In the end, 3 models are pruned in our hypothetical example. This leaves our model set with 4 models (model 1, 2, 4, and 5) which is smaller than the number of measured contention levels (5).

Figure 1.5 shows our proposed end-to-end framework, the top part of which shows the components involved in contention grading and model-set pruning. As the figure indicates, this phase is done before the runtime model selection starts at $t = 0$ when the pruned model set is forwarded to the predictive model selection framework.

1.4.2 Runtime Model Selection

The overview of the proposed predictive model selection framework is shown in the lower part of the Figure 1.5. The framework employs a set of image classification models

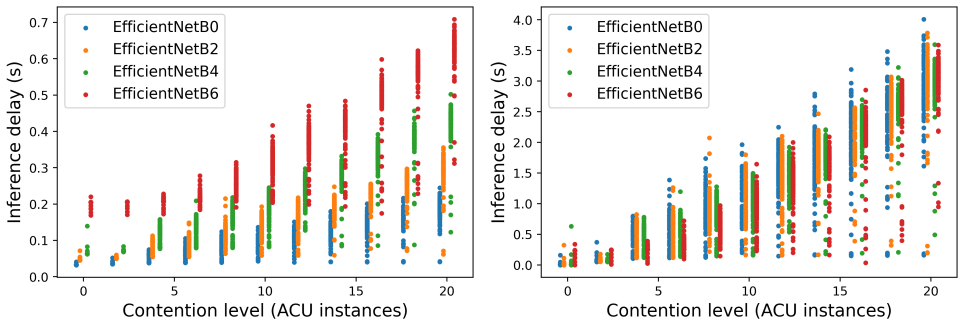
provided by the contention grading and model set pruning component. The framework chooses the optimal model for the next frame's classification while considering the current contention on the system. The optimal model is determined by using historical information and a set of linear regression models. The historical information comes from the previous frames' normalized inference delays. There is one regression model for each image classification model used in the framework. The regression models are trained before runtime using their corresponding image classification models on randomly changing contention level. All regression models take the same input, the previous normalized inference delays, and output the predicted inference delay for their corresponding image classification model. Then, the framework chooses the most appropriate model based on the delay threshold constraint and maximum accuracy as mentioned earlier.

Delay Normalization: Figure 1.6a shows the inference delays for EfficientNet-B0, B2, B4 and B6 [77] under increasing contention. It shows that the inference delay values depend on two things - the system level contention, and the image classification model type. Since our framework uses historical inference delay values to represent the impact of contention, we remove the model type dependency by normalization shown in Equation 1.1. In this equation, x is one inference delay of a model and X is a vector that consists of all inference delays of the same model. If we consider EfficientNetB6 in Figure 1.6a, x is one red dot and X is the vector of all red dots.

$$x_{normalized} = \frac{x - min(X)}{sqrt(var(X))} \tag{1.1}$$

The result of normalization is shown in Figure 1.6b. The minimum and variance values for each model is saved before runtime and used to normalize the inference delays of the models during runtime.

Prediction and Selection: The training data is created by running each model under randomly changing contention levels. As input, the normalized data is split into chunks of n consecutive



(a) Inference delays **(b)** Normalized inference delays

Figure 1.6. Inference delays under increasing contention and normalization

normalized inference delays. All of the regression models take the same input as they will be predicting in parallel using the same input. As prediction output, non-normalized delays are used. Each regression model has different output corresponding to its image classification model. Hence, each regression model takes same input, n previous normalized inference delays, and predicts its corresponding image classification model's inference time for the next frame. After this step, the framework has a predicted inference delay for each image classification model. The image classification models are already ranked in terms of accuracy on static datasets before runtime. Therefore, the framework chooses the model which has the highest rank and also a predicted inference delay under the predefined threshold.

1.5 Experimental Results

1.5.1 Experimental Setup

We implemented the proposed framework on the Nvidia Jetson TX2 platform. We used Tensorflow to train deep learning models that are used in Section 1.5.5. We used built-in image classification models of Tensorflow for the rest of the experiments. These built-in models are EfficientNets [77], ResNetV2 [32], InceptionV3 [75], DenseNets [35], MobileNetV1 [33], MobileNetV2 [70], and NasNets [103]. These built-in models come with pre-trained weights on the ImageNet dataset [69]. Since the validation set of ImageNet is available for hyperparameter

tuning, we decided to use the ImageNetV2 dataset [66] for our test set. Therefore, all of the reported test results in this section are using ImageNetV2 dataset. Also, we resized images using the bi-linear method without cropping before inference for all of the models. We standardized the test set and resizing-cropping technique to make a fair comparison. Therefore, the reported accuracies may be different from the original ImageNet validation set accuracies that are reported in the original papers of the models. Since we focus on the relative accuracies, this is not an issue for our experiments. When we trained custom models, we used the original ImageNet dataset.

1.5.2 Contention Grading and Model Set Pruning Evaluation

We consider contention scenarios imposed by multiple autonomous vehicle applications including RADAR processing (FFT based), LiDAR processing (Deep neural network based) and sensor fusion (clustering based). We run the deep neural network based image classification application concurrently with these other applications on the target system. These coexisting applications create different contention patterns in the system, e.g., the radar processes and sensor fusion run on the CPU while the LiDAR processes run on the GPU, thus creating contentions with the parts of the image classification algorithm sharing those system resources.

Contention Grading Evaluation

System Profiling with CIP: We generate a random number of threads for each of our contending applications. Then we use the CIP to profile this system. In the core of the CIP, we used an application based on EfficientNetB2 image classification model and measured its inference delays to profile the impact of the contention on the system. The inference delays of the CIP and the number of threads of applications creating contention are presented in Figure 1.7. Note that, the information about applications that create contention is not used by the CIP. We only provide the number of threads of contention applications for better understanding of CIP behavior. We generated 15 contention combinations by changing the number of threads of each contention application. 12 of them are unique combinations. In any system, same combination of threads

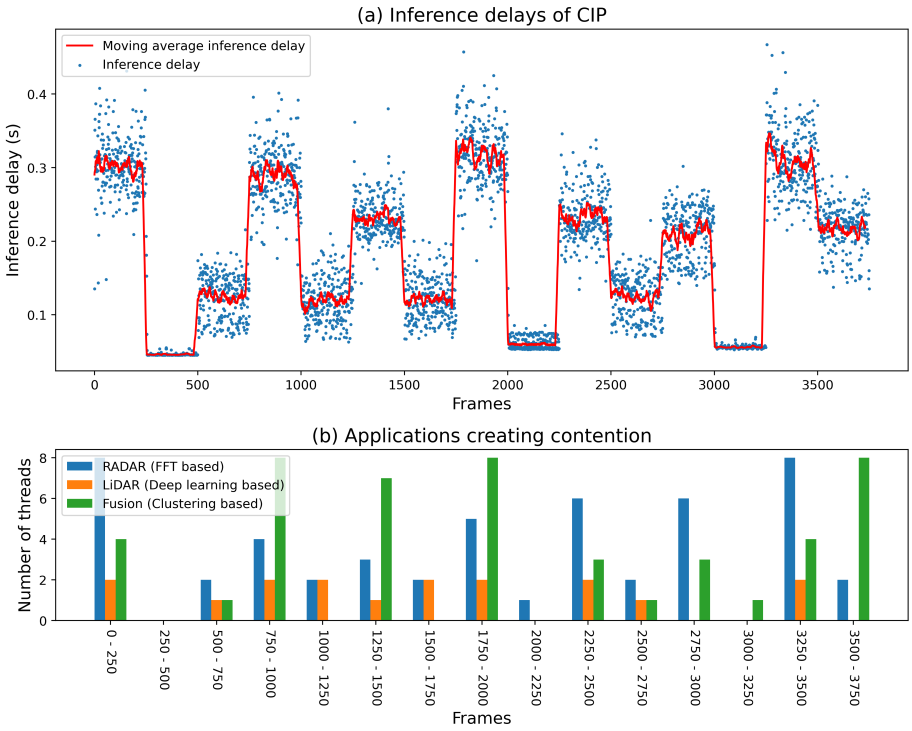


Figure 1.7. Inference delays of CIP under system contention

can repeat over time or different combinations of threads can result in the same contention level. We can see examples of both of these scenarios in our system profile.

Mimicking Contention with ACU: The same EfficientNetB2 based CIP is used to profile the incrementally increasing ACU contention. We increased the number of ACU threads by 2 threads at every 400 frames. The inference delays of the CIP are shown in Figure 1.8. This fine grained contention steps will be used to regenerate the system contention at model pruning step. However, in order to do that, we need to know the system contention levels that match to specific steps of ACU contention.

We use kernel density estimation with Gaussian kernel to find the contention levels in the system. We selected the Gaussian kernel because the distribution of inference delays show a similar pattern to Gaussian distribution at every specific contention level. When we apply kernel density estimation on the inference delay axis, we remove the position information of the inference delay samples. Therefore, we automatically combine repeating or similar contention

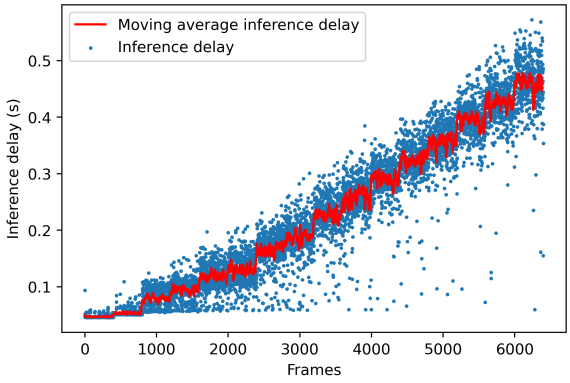


Figure 1.8. Inference delays of CIP under increasing ACU contention

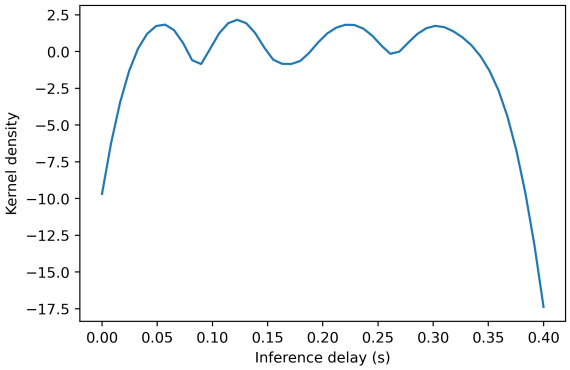
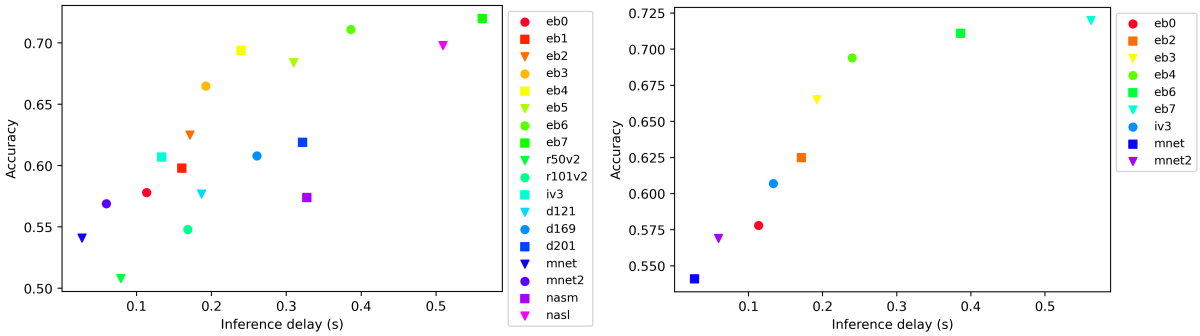


Figure 1.9. Kernel density estimation of the system profile



(a) All models

(b) Pareto pruned models

Figure 1.10. Pareto pruning - Accuracy vs inference delay values of models

levels in time, which can be caused by repeating same contention combination or completely different combinations with same effects. The kernel density estimation is shown in Figure 1.9. The peaks of this plot are the means of the estimated Gaussian kernels. Therefore, the peaks are the specific contention levels that exist in our system contention. The inference delay values of these peaks are matched to inference delay values at ACU profile to find required the number of ACU threads to regenerate each contention level. In this specific example, the number of ACU threads are found to be 2, 8, 16 and 20.

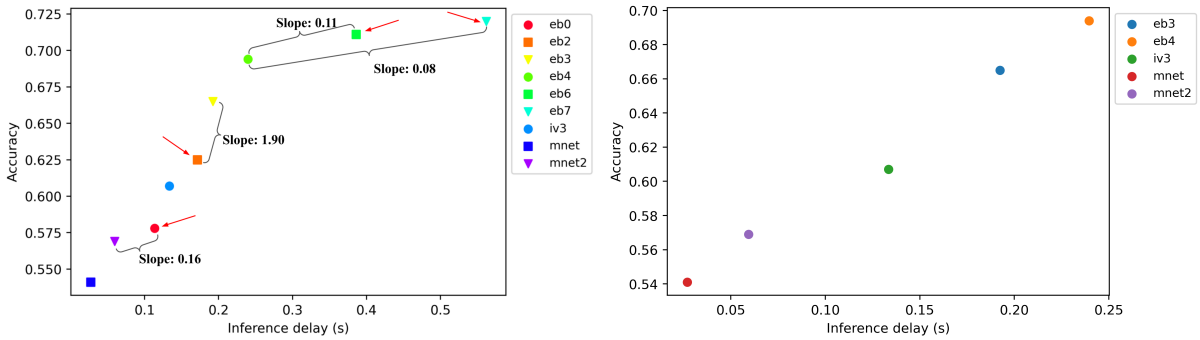
Model Set Pruning Evaluation

Once we have the required number of ACUs to regenerate the system contention, we benchmark our input set of models under the artificial contention that is generated by ACUs. After

this benchmarking step, the accuracy and average inference delay across all contention levels of each model is calculated. These values for our specific example are shown in Figure 1.10a. The abbreviations in the legend of the figure and their corresponding models are as following: eb0 to eb7 are for EfficientNet models, r50V2 and r101v2 are for ResNetV2 models, iv3 is for InceptionNetV3, d121 to d201 are for DenseNet models, mnet is for MobileNet, mnet2 is for MobileNetV2, nasm is for NasNetMobile, and nasl is for NasNetLarge. This plot clearly shows that some models have no advantage at all compared to others. For example whole families of ResNetv2 and DenseNet architectures are performing with less accuracy using more inference time compared to other models. Therefore, we calculate the accuracy-inference delay Pareto frontier of the models to remove these bad performing models from consideration for our task. The Pareto pruned model set is shown in Figure 1.10b.

Each model in Pareto pruned model set is guaranteed to give best accuracy at its inference delay or below. However, this theoretical result does not correspond to the equally good practical result when these models are used in an application on an embedded system. A model can still be on the Pareto frontier if it improves accuracy very slightly but requires a lot more time and memory for inference. Using such models harms the performance of our application as they do not provide significant advantage while still requiring the cost. Therefore, we remove these models from our model set as well. In order to remove these models, we check the slope of every consecutive models. If the slope is too big, we remove the model with smaller accuracy. If the slope is too small, we remove the model with higher accuracy. In our specific example, we define the slope thresholds as 0.25 and 1.5. The slopes that violate these thresholds are shown in Figure 1.11a. The pruned models are pointed by a red arrow. The resulting model set is shown in Figure 1.11b.

We can consider some of the prunings to understand how this stage can save memory. For example, eb4 (EfficientNetB4) requires 79.1 MB while eb6 (EfficientNetB6) requires 174.8 MB. Even if eb6 would satisfy the user requirements, it would require more than 2 times larger memory than eb4 without providing significant advantage.



(a) Pareto pruned models and violating slopes **(b)** Transition pruned models

Figure 1.11. Transition pruning - Accuracy vs inference delay values of models

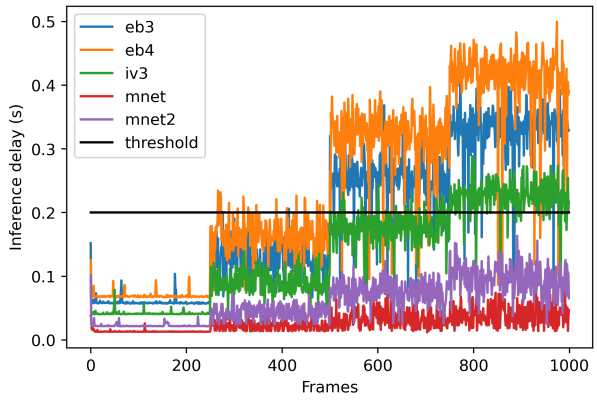


Figure 1.12. Inference delays of the transition pruned models under existing contention levels

In the final step of pruning, the contention levels and the user requirement are considered. The user requirement is the inference delay threshold. The application's inference delay for one frame should not exceed this inference delay threshold in any contention levels. The remaining models' inference delays are considered under the existing contention levels as in Figure 1.12. In this figure, every model is run for 250 frames for each of the contention levels. Note that, the previous steps of pruning guaranteed that a model with higher inference delay has also better accuracy with a decent margin. In this step, we select the best performing model that satisfies the inference delay threshold in each contention level. Note that, we do not need to select one unique model for each contention level. In first two contention levels, eb4 (EfficientNetB4) is the optimal model. In the third level, iv3 (InceptionNetv3) is the optimal model. eb3 (EfficientNetB3)

satisfies the delay threshold at contention level 2 but violates it at contention level 3. Since it is not selected at contention level 2 and there is no intermediate contention level between levels 2 and 3, eb3 is pruned in this step. Similarly, mnet (MobileNetv1) satisfies the inference delay threshold at all of the contention levels. However, there is always at least one model that satisfies the delay threshold and performs better than mnet. Therefore, mnet is also pruned from the model set. In the contention level 4, the optimal model is mnet2 (MobileNetv2). As a final result, the optimal model set for this contention scenario consists of eb4 (EfficientNetB4), iv3 (InceptionNetv3) and mnet2 (MobileNetv2). Our contention grading and model pruning methods decreased the number of model from 18 to 3 for a contention scenario where 12 unique combinations of 3 real applications are running on the system.

1.5.3 Runtime Performance Evaluation

After we obtain a pruned model set, we run our contention-aware adaptive model selection framework. First we show the performance of our predictive model selection method and then also show how the prior contention grading stage positively contributed to the model selection performance.

Predictive Model Selection Performance

We compare our predictive model selection with two reactive model selection approaches. The first one is called 1-step reactive model selection which checks the last frame's inference and then selects 1-step stronger model if the last frame's inference is below threshold. Otherwise, it selects the next (1-step) weaker model. The second reactive approach is called N-step reactive model selection. This approach similarly checks the last frame's inference delay and selects 1-step stronger model if the last frame's inference is below the threshold. However, if the last frame's inference is above the threshold, it conservatively selects the weakest model for the next frame to satisfy the delay threshold.

The temporal plots for 6000 frames under a specific contention regime are shown in

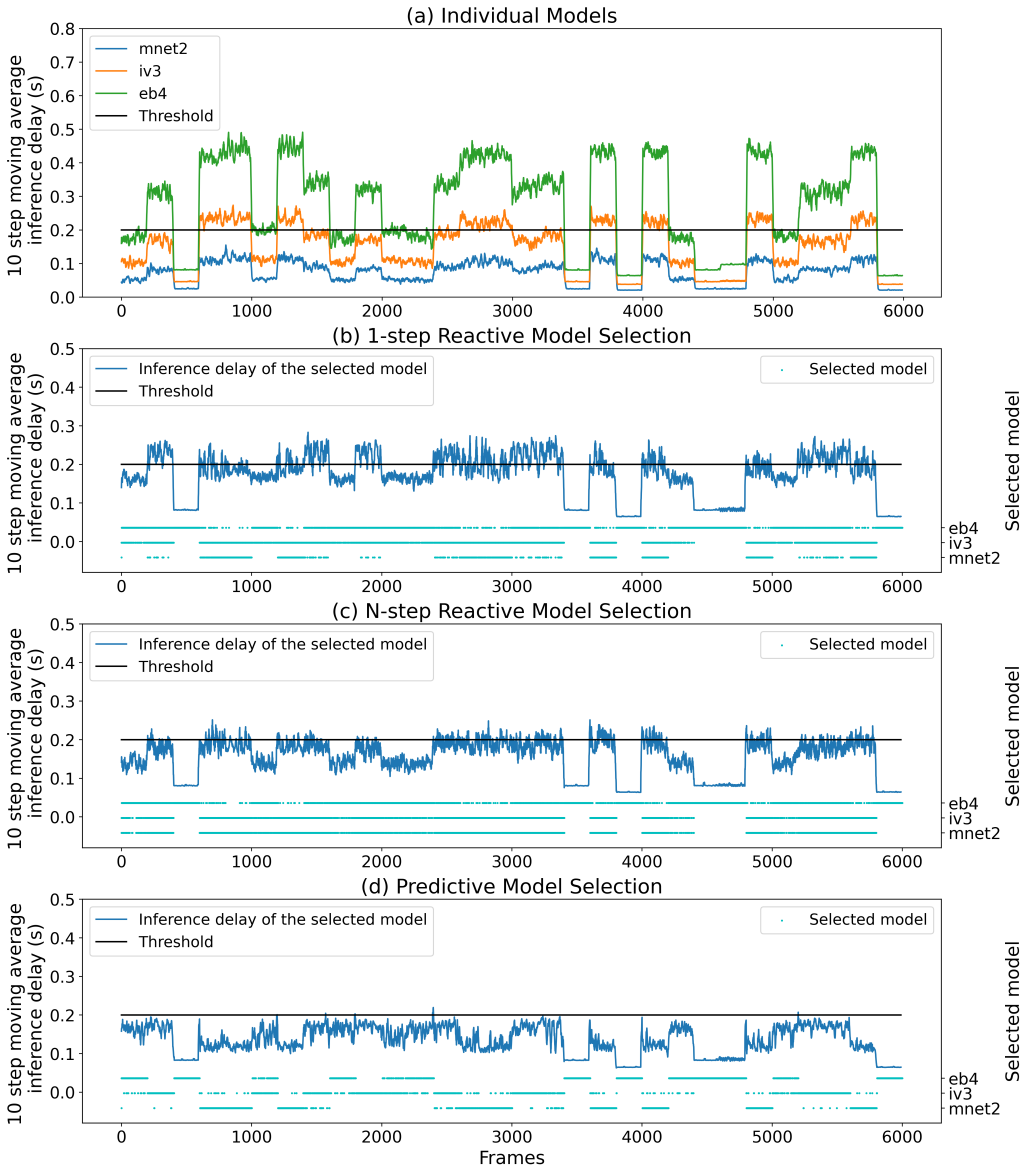


Figure 1.13. Temporal comparison of individual models, reactive methods and the predictive method under varying contention

Figure 1.13. This contention regime is generated by randomly sampling real system contention applications that are shown in Figure 1.7. The inference delay and the selected model's index are given for three different model selection methods. The individual models' inference delays are also plotted for comparison purposes. The inference delays are averaged over 10 frames to smooth the plots. The delay plots for individual models in Figure 1.13a show that using a single model under varying contention is not optimal. The individual plots also suggest the best model

Table 1.1. Comparison of methods in a specific contention regime - Dataset: ImageNetV2

Model	Accuracy (%)	Delay Violations (%)
MobileNetV2 (mnet2)	57.50	0.05
InceptionNetV3 (iv3)	63.93	29.75
EfficientNet-B4 (eb4)	70.00	65.50
Average-(mnet2, iv3, eb4)	63.81	31.76
1-step Reactive Model Selection	65.93	34.61
N-step Reactive Model Selection	64.86	27.40
Predictive Model Selection	64.60	11.66

.

under a specific contention regime, e.g., around the frames $900, 1900, 3900$, the ideal models are mnet2, iv3, eb4 respectively. It can be seen that the predictive method can successfully select the optimal model most of the time in Figure 1.13d. It can also choose multiple models under the same contention region. One example of this can be seen just after frame 2000. In this region, predictive model selection selects eb4 and iv3 frequently. This happens when contention corresponds to the middle of two models, i.e contention is high for iv3 and is low for eb4. In this case, predictive model selection changes the optimal model selection between eb4 and iv3 frequently to satisfy inference delay threshold and maximize the accuracy. 1-step reactive model selection and N-step reactive model selection frequently fail to satisfy delay threshold as shown in Figure 1.13b and Figure 1.13c, respectively.

Table 1.1 shows the summary of data for Figure 1.13 in terms of average performance of different schemes. If a frame classification takes more time than predefined threshold, we consider it as delay violation. The delay violation is a way to measure wrong selections. The table shows that all of the model selection methods have an accuracy around the middle of individual models. However, the reactive methods have large delay violations as well. On the other hand, the predictive method has only 11.66% delay violation.

The Effect of Contention Grading on Runtime

In this section, we consider the effect of contention grading on runtime with respect to accuracy and delay violation. There are 3 stages of pruning which are Pareto pruning, transition

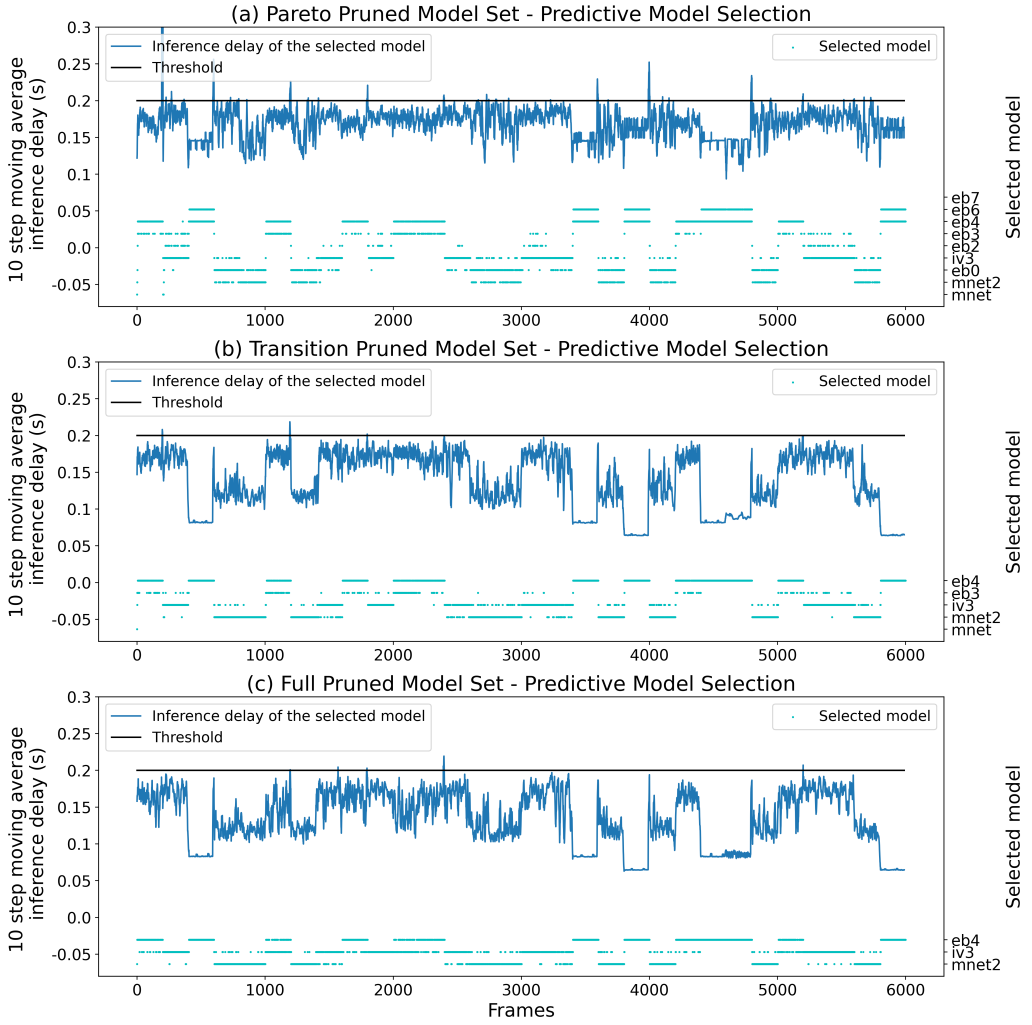


Figure 1.14. Temporal comparison of predictive method with model sets after each pruning stage under varying contention

pruning and contention pruning. These are applied one after another in this order. Therefore, we will compare 3 model sets on runtime using the same predictive model selection method. The Pareto pruned model set has 9 models which are shown in Figure 1.10b, the transition pruned model set has 5 models which are shown in Figure 1.11b, full pruned model set has 3 models which are MobileNetV2, InceptionNetV3, and EfficientNetB4.

The temporal comparison of three model sets using the predictive method is shown in Figure 1.14. When the number of models increase in a model set, the number of model switching also increases which harms the performance since there is only one optimal model in

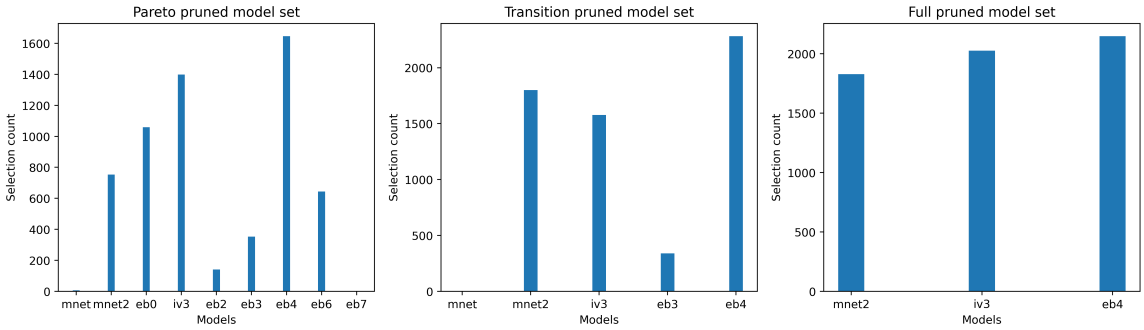


Figure 1.15. Selection counts of models for each model set

Table 1.2. Model set comparison - Dataset: ImageNetV2

Model	Accuracy (%)	Delay Violations (%)	Memory Consumption (MB)
Pareto-pruned	65.91	31.35	4290
Transition-pruned	64.91	12.46	3650
Final	64.60	11.66	3460

one contention level. When we examined the selected indexes of Pareto pruned and transition pruned model sets, we see that mnet and eb7 are almost never selected. Therefore, they occupy memory without providing any gain to system. We plotted the selection counts of models for each model set in Figure 1.15. This plot shows the most frequently used models in each model set. The most frequently used models are similar in most cases (mnet2, iv3, eb4). The only exception is eb0 in Pareto pruned model set where it is used more than mnet2. eb0 is pruned in transition pruning since it requires more than 1.5x memory of mnet2 while it does not give significant accuracy gain over mnet2. Our model set pruning stage finds these frequently selected models (mnet2, iv3, eb4) before runtime.

We examined the overall performance of these model sets in Table 1.2. The Pareto-pruned model set has a large delay violation percentage at 31.35%. Transition pruning achieves a decrease in delay violation significantly from 31.35% to 12.46% with 1.0% absolute accuracy loss. The final pruning achieves the smallest delay violation percentage at 11.66% with another 0.31% absolute accuracy loss. Furthermore, the final pruning achieves the smallest memory consumption. The provided memory measurements include base Tensorflow cost which is around

3GB. This is a one time cost and independent from the number of loaded models. Therefore, another comparison can be made without including this base cost. Then memory consumption values are 1240MB, 600MB, 410MB for Pareto-pruned, transition-pruned and final model sets, respectively. Considering these results, the final pruning achieves the best memory efficiency by occupying 0.33 of what Pareto pruned model set occupies and 0.68 of transition pruned model set occupies.

1.5.4 System Implementation Details

Our framework works with neural network based applications. Therefore, we decided to use Jetson TX2 which has a GPU for neural network loads. EfficientNetB0, a neural network that is extensively used in our experiments, runs in 34.9 ms on Jetson TX2 GPU and in 61.1 ms on Jetson TX2 CPU. Therefore, Jetson TX2 GPU gives a speedup of 1.75 over a mobile CPU. Moreover, our framework is designed for real time applications and Jetson TX2 is a good fit since it is an embedded system. Lastly, we are using Tensorflow for neural network applications and Jetson TX2 is running Linux with Tensorflow support.

The power consumption corresponding to Figure 1.14c is shown in Figure 1.16a. The details of applications that create contention are also given in Figure 1.16b. The RADAR and Fusion applications run on CPU, the LiDAR application runs on GPU. The power is measured by the built-in power sensor of Jetson TX2 which provides the average of the last 512 samples from continuously probed data when it is called. The peak power is 12170.0 mW.

The inference delays and the power consumption are not changing in parallel, which would be the expected behavior in single threaded applications. However, since our system is multi-threaded and uses both CPU and GPU, the behavior is different. This is because the power usages of GPU and CPU are different. The behavior can be understood by comparing the first three regions. In region 400-600, the contention is very small and limited to CPU and therefore the inference delay of our application is small. However, power consumption is high. This is because the GPU is used extensively all the time. On the other hand, in the region 200-400, the

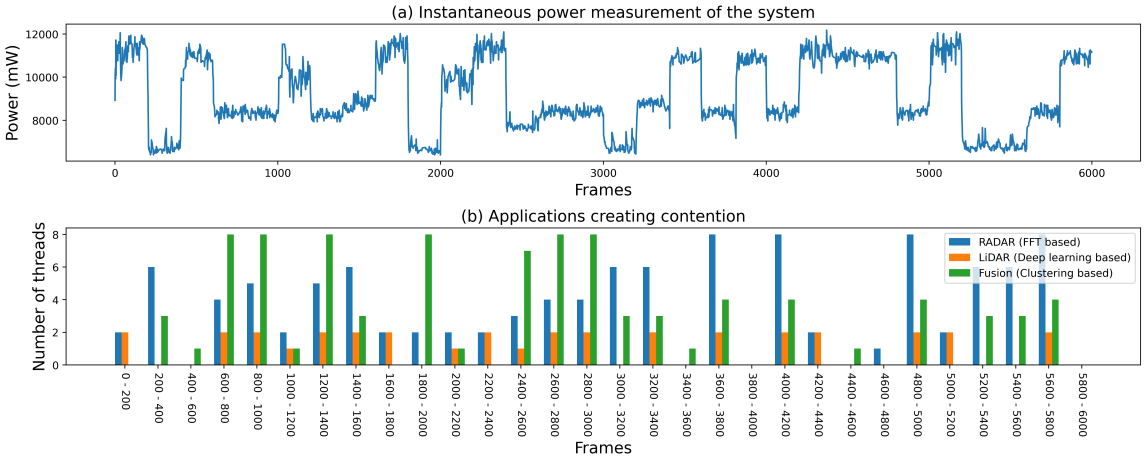


Figure 1.16. System power measurement when model selection is running under varying contention

contention is high and heavily focused on CPU. Therefore, the inference delay of our application is high which is compensated by choosing a smaller model. This is because the CPU is being used heavily and becomes a bottleneck in the system. However, since the CPU does not consume as much power as the GPU, the power consumption (average of 512 samples from Jetson TX2 sensor) is low. The contention of region 0-200 is similar to region 400-600. However, there is one more GPU application in region 0-200. Therefore, the power consumption is bigger.

We also measured temperature values of GPU, CPU and the board. Once the system is used for a while and stabilizes, the temperature does not change much. GPU and CPU temperature stay between 38C and 40C, and the board temperature stays around 35C. Jetson TX2 has a fan and therefore active cooling results in stable temperature values for our workload.

The average running time cost for one frame of our predictive framework is $0.29\,ms$ which is approximately 690 times smaller than average inference delay for one frame. Therefore, we can say that the time cost of the framework is insignificant. This only includes the selection logic which is a relatively light calculation. A matrix multiplication is used for linear regression models and an iteration is used for model selection. Overall, model selection takes too little time to trigger any measurement hardware and we do not see any unusual pattern in general power and temperature measurements. Therefore, the power consumption and temperature overhead of

Table 1.3. Inference delays of 4 models when model switching is used

Model M	Average of first inference delays right after switch to Model M (s)	Average of all inference delays when the Model M is used (s)	Difference in percentage
EfficientNetB0	0.03431	0.03324	3.21%
EfficientNetB2	0.04663	0.04683	-0.41%
EfficientNetB3	0.05318	0.05261	1.09%
EfficientNetB4	0.06495	0.06426	1.07%

model selection is negligible.

All of our models are stored in RAM during runtime. To measure switching overhead, we loaded 4 models in RAM, ran 1 model for 100 frames, then switched to another model and kept this cycle for 10000 frames. The Table 1.3 shows the difference between first inference delays right after switch and mean of all inference delays for each model. The table shows that there is no significant difference between the first inference delay and the rest. Sometimes, the average of first inference delay is even faster than the average of the rest as in the case of EfficientNetB2. Note that these values are average. Therefore, in many switch cases the other models also are faster in their first inference delay compared to the rest. As a result, we can say that we do not observe any perceivable switching overhead.

1.5.5 Comparison with Early Exit Based Method

Early exit networks present an alternative approach to adapting neural network based applications to contention [86]. An early exit network consists of different exit points that are typically derived by adding classifier layers to different intermediate features in a neural network to generate the final class predictions. Different early exit branches are selected as a response to changing contention levels. Since this work is closely related to the runtime part of our work, we compare the contention-aware early exit methodology with our work in this section.

We designed an early exit model to adapt to changing contention levels based on the methodology presented in [86]. We used EfficientNetB0 architecture as the backbone of our early exit model. All of the EfficientNet architectures consist of 7 blocks. These blocks are

scaled in terms of width and depth to create heavier models, while the number of blocks stays constant throughout all EfficientNet architectures. Therefore, we decided to use these blocks as our early exit paths. We created 5 early exit branches from the output of block 3 to the output of 7. For each exit, we built a classifier top that is similar to the original EfficientNet top. This classifier top includes a 1x1 convolution layer to set channel sizes of the features to some constant value (1280), a global average pooling to remove spatial size dependency and a fully connected layer to generate predictions. This top design makes the early exit branches input size agnostic. Moreover, we created 4 different input sizes as (128x128, 160x160, 192x192, 224x224) by following a similar practice to [86]. In the end, our early exit model supports 20 different combinations of early exit branches and input sizes.

We also designed 4 individual models to compare with the early exit model. Since the early exit model is trained from scratch, we also trained individual models from scratch under the same conditions to make fair accuracy comparisons. Therefore, we did not use pre-trained models as in the previous section. We used the same intermediate points as the early exit branches to design individual models. For example, the smallest individual model starts as an EfficientNetB0 model but stops at block 4 and ends with a classifier top. Similarly, the other models stop at blocks 5,6, and 7. As a result, our individual models are directly comparable with the corresponding early exit branches in terms of architecture.

The training is done on the ImageNet training dataset. The ImageNet validation dataset is used for monitoring improvement and early stopping. The ImageNetV2 test set is used for reporting the test accuracies. The Adam optimizer [41] is used to train the parameters. Random cropping, random horizontal flipping and random contrast (factor 0.8-1.2) are used as data augmentation techniques.

Multi Objective Optimization and Impact on Accuracy

The training of early exit model is a multi objective optimization. During forward propagation, the same data is fed to the network and each early exit branch makes a prediction.

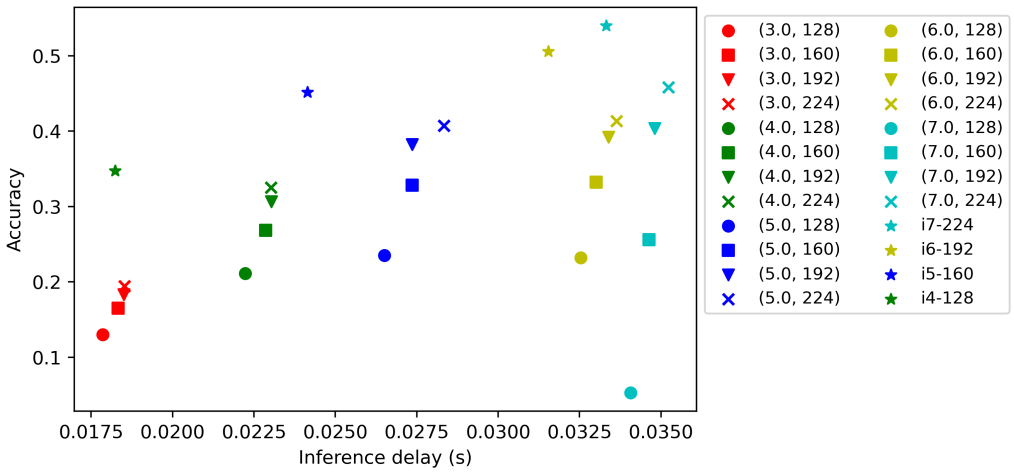


Figure 1.17. Performance comparison of early exit branches -(branch no, input size) and corresponding individual models - i(model size)-input size

An error is calculated at each early exit branch. Hence, during back propagation, multiple gradients are propagated backwards. This results in multiple objective optimization of the shared parameters. For example, a convolution layer in block 3 needs to learn both low level features for early exit branch 7 and high level features for early exit branch 3. This results in longer training times and also inferior accuracy.

In [79], it is shown that the early exit method can have regularization effect since it makes it harder to train the neural network. However, this effect is only applicable when the data is too small or the network is too high capacity for the data. Also, there are other regularization techniques that are widely adopted in the neural network design such as dropout [73] or data augmentation [14].

The inference delays and accuracies of early exit branches and corresponding individual models are shown in Figure 1.17. The early exit branches in the legend are indicated as (branch number, image size). The individual models in the legend are indicated as i(model size)-image size. Note that branch number and model size are directly comparable as explained previously. This similarity is shown with colors in the plot. The individual models outperform the early exit branches in terms of inference delay and accuracy. Moreover, the early exit model has a time

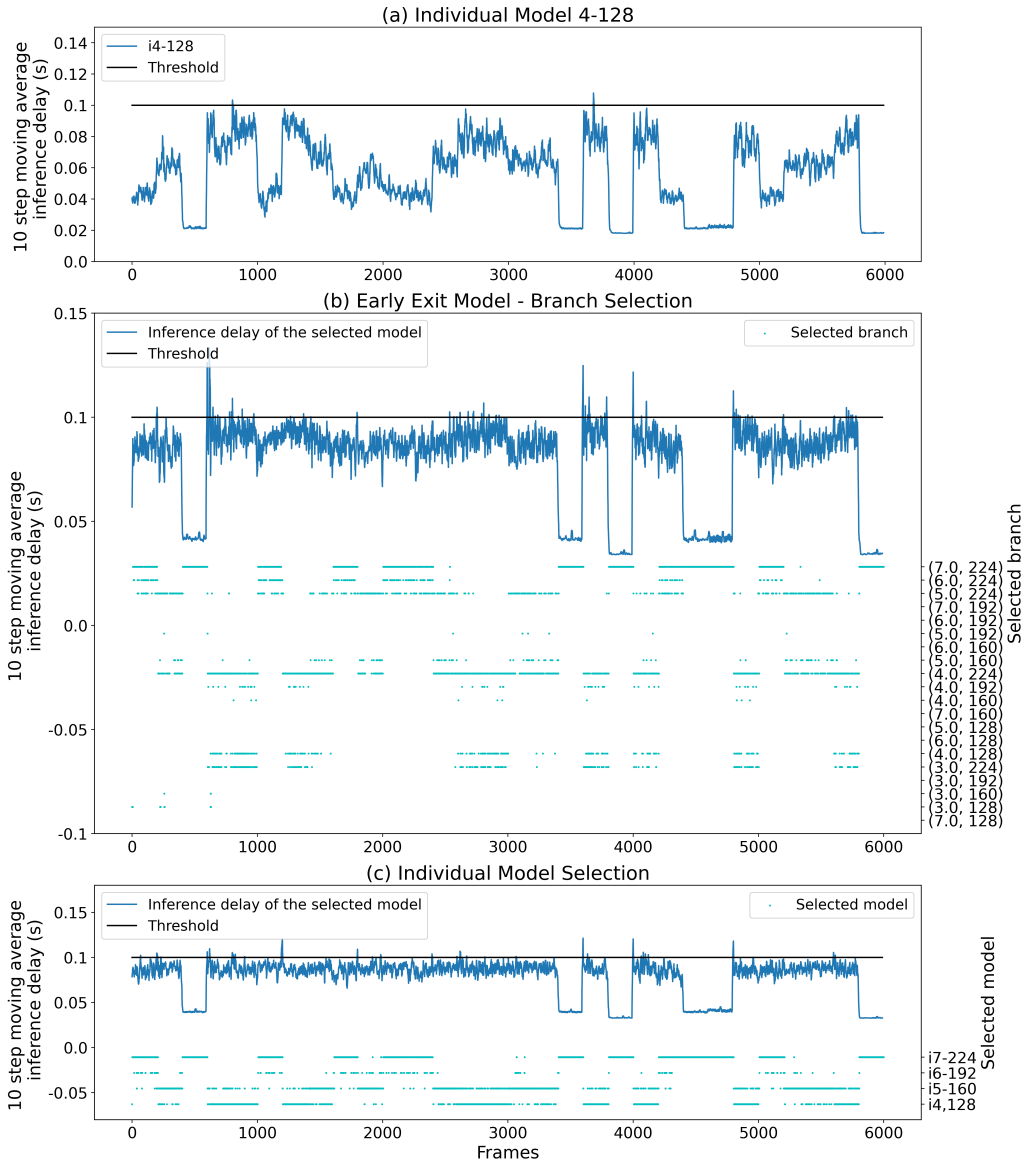


Figure 1.18. Temporal comparison of the smallest individual model, early exit, individual model selection under varying contention

overhead due to control logic. For example, early exit branch (7.0, 224) and individual model i7-224 are completely same in terms of architecture and input size. However, individual model runs slightly faster than the early exit branch. The same difference can be observed for the other individual models and their corresponding early exit branches.

Runtime Performance Comparison

We tested the early exit model and the individual model set on runtime with the contention profile in previous section. We used our regression model based selection methodology for early exit runtime branch selection and individual model set runtime model selection. The temporal comparison and the numerical results for the early exit, individual model set and the smallest individual model are shown in Figure 1.18 and Table 1.4. The individual model set has almost absolute 9% higher accuracy than the early exit model. Moreover, the delay violation of the individual model set is slightly less than the early exit model.

There are 20 early exit and input size combinations in the early exit model. However, this granularity is not used completely even though a large variety of contention levels (12 unique contention combinations as in Figure 1.7) are experienced. This is because one early exit-input size combination can be the optimal choice for more than one contention level, as in the case of individual models. On the other hand, this individual model set of 4 models can achieve slightly less delay violation and much better accuracy than the early exit model. Therefore, we can say that the high level of granularity of early exit networks is not helpful even in the presence of frequently changing contention. The accuracy of early exit model is only comparable to the smallest individual model which has a significantly lower delay violation of 5.91%.

In the early exit architecture, convolution layer parameters are shared among early exit branches. Therefore, the architecture aims to achieve less parameters than the total parameters of multiple individual models. However, a large part of the parameters in convolutional neural networks are coming from the fully connected layers at the classifier. The recent and successful EfficientNet architectures can be example for this. EfficientNetB0 has 5,330,571 parameters in total of which 1,281,000 parameters are from the fully connected layer at the classifier. Therefore, whenever we add a branch, we add a fully connected layer and a large number of parameters. As a result, the size of the early exit model is 44 MB, whereas the total size of our individual models is 55 MB (7+9+18+21). The early exit model, of course, would have much less parameters

Table 1.4. Early exit model and individual model set comparison - Dataset: ImageNetV2

Model	Accuracy (%)	Delay Violations (%)	Memory Consumption (MB)
i4-128	34.60	05.91	3070
Early exit model	37.03	21.90	3300
Individual models	45.87	20.81	3390

.

compared to the total of 20 individual models which correspond to each early exit branch-input size combination. However, as we discussed earlier, we do not need that many models for effectively adapting to contention. Moreover, our contention grading and model set pruning framework allows us to decrease the total number of models by intelligently selecting optimal models for the system contention and user requirement.

One drawback of using early exit models at runtime is the switching cost of the branches. Whenever the output of the model is changed to a different early exit branch, an additional time is required for the execution graph. We examined these switch costs. Some switching combinations take more time than the other ones but we did not observe a certain pattern. The average switching cost is 5.58 ms. The inference delays of early exit branches are ranging from 18 ms to 35 ms under no contention. Therefore, the average switching cost can be up to 31% of the inference delay whenever a switching occurs.

In summary, we believe that the proposed approach, which comprises of selecting individual models for contention adaptation is more effective in terms of accuracy, inference latency and runtime overheads compared to early exit based methods.

1.5.6 Comparison with Slimmable Network Based Method

Even though the original work of slimmable networks [92] does not consider runtime in the presence of contention, it is possible to use them in this context. Slimmable networks use less parameters to create sub-networks in the same backbone architecture. Since slimming is similar to early exit in the sense that they are both dynamic neural network methods and utilize weight sharing, we implement a slimmable neural network and compare our method with it.

We designed a slimmable neural network with 4 switches (0.25x, 0.50x, 0.75x, 1.0x) where the backbone architecture is EfficientNetB0. We use the same individual models that are used in the previous section. Therefore, the architectures of our biggest individual model and 1.0x switch of our slimmable network are exactly the same. The same training configuration is used as specified in the previous section.

The Effect of Batch Size on Slimmable Networks

Slimming operation is basically using less number of filter channels in convolution operations. Therefore, slimming reduces the FLOPs. However, this does not always translate to speedup. Every operation has an arithmetic intensity value which can be calculated by the ratio of number of FLOPS to number of byte accesses. Similarly, every processor has an ops to byte ratio that can be calculated by the ratio of math bandwidth to memory bandwidth. If the arithmetic intensity of an operation is smaller than the ops to byte ratio of the processor, the operation is limited by the memory. Conversely, if the arithmetic intensity of an operation is larger than the ops to byte ratio of the processor, the operation is limited by math (arithmetic). Finally, if neither of the math and memory pipelines of the processor are saturated by the operation, the operation is limited by the latency. The latency limitation happens when parallelism of the operation is not enough to saturate the processor's capabilities. While it is possible to calculate the arithmetic intensity of each operation in a neural network, it is not practical to do so since we are using very deep neural networks. Moreover, theoretical arithmetic intensity calculation is only a first-order approximation. Therefore, we provided empirical results with different batch sizes in Figure 1.19 to show the saturation points of GPU pipelines where the slimming operation becomes useful. It is important to note that these results are specific to a given combination of neural network and GPU. Using a better GPU in all aspects or using a neural network with smaller width would result in requiring larger batch size to make slimming useful. Figure 1.19 shows that the minimum batch size of 8 is required to achieve a speedup at every slimming point. However, using batch inferences in embedded systems is not useful. Since embedded systems

are usually used in real-time applications, waiting for new data for batches and then running batch inference may not be practical. Moreover, since embedded systems are already resource constrained systems, running inference with large batch size takes too much time and results in missing deadlines of many points in the batch data.

Even if we find a very specific scenario where inference with large batch size in an embedded system is required, our proposed method using individual models is still superior compared to the use of slimmable network switches in terms of accuracy. This is because training a slimmable network is a multi objective optimization like early exit since the weights of a slimmable network are shared among different switches. The comparison of individual models and slimmable networks for different batch sizes are shown in Figure 1.20. Slimmable network switches do not provide a tradeoff in batch size 1 and provide only a partial tradeoff in batch size 4. It starts to provide a tradeoff in batch size 8. However, the individual models provide a tradeoff in all batch sizes and have better accuracy than slimmable network switches. Note that i7-224 individual model and 1.0x slimmable network switch have the exact same architecture. The other individual models do not have the exact same architectures with slimmable switches but since they have similar inference delays for high batch sizes, their accuracies can be fairly compared.

All models in our experiments are implemented in graph execution instead of eager execution. The graph execution requires models to be statically compiled. As a result, it is much faster than eager execution. Since the slimmable networks are dynamic networks, implementing them in graph mode requires different approaches and additional logic. We noticed these implementation differences result in 1-4 ms deviation in inference delay. However, when the batch size is increased, this deviation becomes negligible.

Runtime Performance Comparison

Even though our framework is designed for real-time systems and large batch sizes are not preferred in real-time systems, we compare our methodology with slimmable networks by

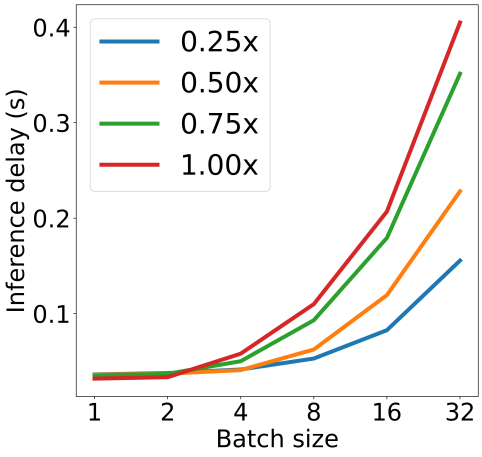
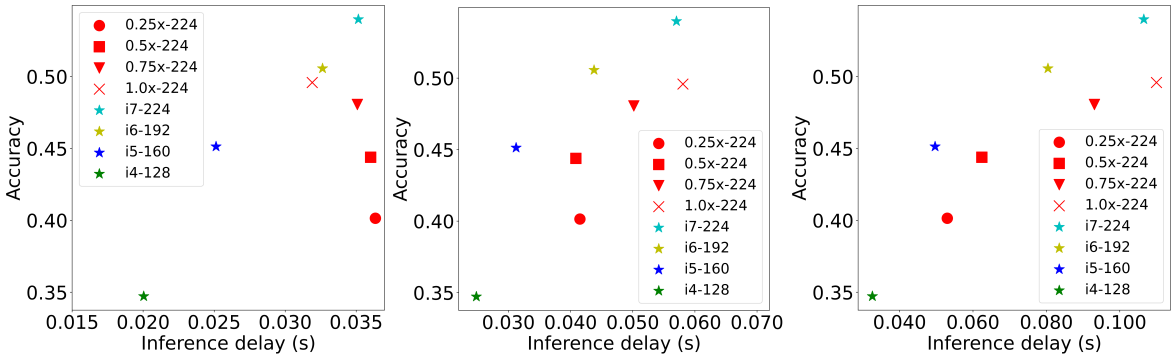


Figure 1.19. The effect of batch size on inference delays of switches of a slimmable network



(a) Batch size 1 **(b)** Batch size 4 **(c)** Batch size 8

Figure 1.20. Accuracy - inference delay plots of individual models and slimmable network switches under different batch sizes

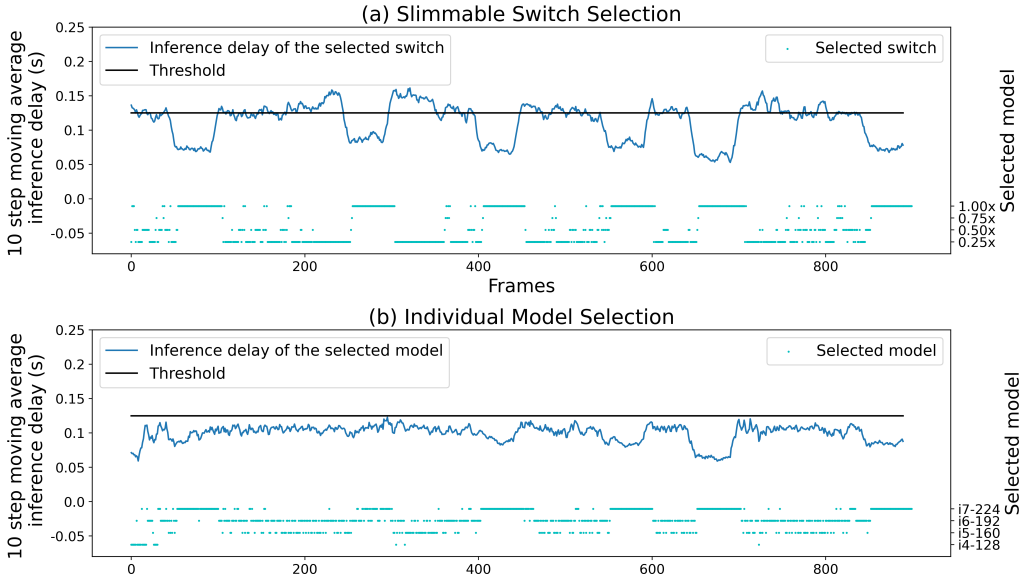


Figure 1.21. Temporal comparison of slimmable model and individual model selection under varying contention

Table 1.5. Slimmable model and individual model set comparison - Dataset: ImageNetV2.

Model	Accuracy (%)	Delay Violations (%)
Slimmable model	44.12	42.44
Individual models	51.25	16.55

increasing batch size for the sake of comparison. We used a batch size of 8 and decreased the intensity of contention compared to previous sections in order to keep the inference delays in a reasonable range. Therefore, the metrics in this section are not comparable to the ones in previous sections.

The temporal comparison and the numerical results for the slimmable model and individual model set are shown in Figure 1.21 and Table 1.5. The threshold is determined by the maximum delay of the heaviest model under no contention. This is a fair selection of threshold since the heaviest model is the same in both methods. Table 1.5 shows the accuracy achieved by our proposed individual model selection method is better than using the slimmable model as expected from previous analysis. The delay violation of the individual model set is also significantly better than slimmable model. These results are expected since the individual model

set provides a better tradeoff than the slimmable model.

The slimmable networks are similar to the early exit networks in the sense that they share weights for different switches. Therefore, they use less memory compared to individual model set that has same number of models as the switches in the slimmable network. However, as we discussed in the previous sections, our model set pruning technique reduces the number of models for a given system and applications, and therefore achieves efficient memory consumption.

In the end, we believe our proposed approach is more effective in terms of accuracy and inference latency than the slimmable neural network based method. Moreover, our approach does not have limitations such as minimum batch size depending on the neural network architecture and the hardware as slimmable neural networks do.

1.6 Conclusion

In this paper, we proposed a two stage framework to enable contention-aware adaptive image classification model selection. Our framework takes a deep learning model set, a user requirement and the system with contention and creates a contention-aware application that runs on the system. In the first stage, we define Contention Impact Profiler (CIP) that can profile system contention effect to our application. Then we analyze the profile with kernel density estimation to find the system contention levels. We define Artificial Contention Units (ACU) to regenerate these contention level in a controlled environment. Then, we run 3-stage model pruning on the given model set to select optimal models for the system contention and the user requirement. In the second stage, we define a runtime framework to use previously found models to adapt to changing contention. Our runtime framework employs linear regression models to predict future inference of the models and selects the optimal model for the existing contention. The experimental results show that our predictive model selection outperforms the average of individual models in both accuracy and inference delay violation. Predictive model selection also outperforms the reactive model selection methods and early exit method. We demonstrated

our technique using image classification while the contention is created by fusion, RADAR and LiDAR tasks to model an autonomous car environment. However, our framework can work with any neural network based primary application along with any contention applications. For example, the primary application can be object detection while contention can be created by data communication and data encryption. Alternatively, the primary application can be neural network based speech processing and video game graphics can create contention in a mobile system.

Chapter 1, in full, is a reprint of the material as it appears in ACM Transactions on Embedded Computing Systems 2022, Basar Kutukcu, Sabur Baidya, Anand Raghunathan, Sujit Dey. The dissertation author was the primary investigator and author of this paper.

Chapter 2

SLEXNet: Adaptive Inference Using Slimmable Early Exit Neural Networks

2.1 Introduction

Deep learning has transformed many research areas including computer vision [10], natural language processing [99], speech recognition [63] and many more. However, deep learning's superior performance comes with a cost, which is its dependency on high computation power. The training of a deep learning model requires more computation power compared to just using it for the inference. But the inference cost of a deep learning model is still considerably significant for resource constrained systems. Therefore, there have been many research efforts in recent times to enable efficient deep learning inference on resource constrained systems. Additionally, over the years, the use of resource constrained embedded systems have exponentially increased with the rapid emergence of cyber-physical systems in the era of the Internet-of-Things (IoT) applications [72]. For this reason, the resource constrained systems have a variable performance demands as they frequently interact with external factors such as people and physical environment, unlike servers which work in an isolated environment with well-defined workloads most of the time. As a result, implementing efficient inference of deep learning models is extremely challenging for resource constrained systems where a number of variable external factors exist.

The research on dynamic neural network architectures has garnered significant attention

in recent years, as it can create multiple models to support various system requirements and enable adaptive inference of deep learning models [30]. Many techniques are developed to create dynamic neural networks including early exiting [79], layer skipping [82], slimming [92], dynamic routing [58] and many more. Each of these methods has its advantages and disadvantages for different tasks. However, the advantage of being dynamic usually comes with a tradeoff between accuracy and complexity of the dynamic neural networks. Moreover, it is harder to train these models as they require custom training strategies. Additionally, it takes longer time to train these models as they are trained for multiple sub-architectures within one main architecture.

In this paper, we propose a new dynamic neural network architecture, *SLEXNet* that is designed for resource constrained systems with dynamic external factors. SLEXNet combines early exiting and slimming to create more sub-architectures than each of these techniques individually can offer. Moreover, early exiting and slimming have their own advantages in different situations. SLEXNet not only utilizes the advantages from these techniques, but also outperforms them due to using the combination of the techniques which provides more optimization knobs that the dynamic model can tune to. We also propose a scheduling algorithm that searches and finds out the best SLEXNet sub-architecture given the external factors and requirements.

The main contributions of this work are as follows:

- A new dynamic neural network architecture, called SLEXNet that combines early exiting and slimming to create a wide range of speed-power-accuracy characteristics in one architecture.
- A runtime scheduling algorithm that evaluates the current external factors and requirements, and searches within the SLEXNet sub-architectures efficiently in constant time to find the best sub-architecture that satisfies the given requirements.
- Demonstration of the performance of the SLEXNet and the Runtime Scheduling on a wide

range of experiments and comparison with other dynamic neural network techniques such as early exiting and slimming.

We implement SLEXNet with TensorFlow and conduct the experiments on Nvidia Jetson Orin. We investigate SLEXNet's capability in terms of adapting varying data rate and power consumption, considering processing time deadline and power budget as thresholds. We compare our algorithm with early exiting and slimming and show advantages of SLEXNet.

2.2 Related Work

Efficient and Scalable Neural Network Architectures: Some works have proposed efficient and scalable neural network architectures to satisfy different efficiency requirements. In [77], an efficient neural network architecture is proposed. This architecture is scaled in terms of depth, width and resolution to create bigger architectures step by step. Even though each of these models provide a different point in complexity, they are all individual models and do not have the dynamic network capacity. In [9], one efficient and scalable architecture is proposed. In this work, one model is trained and many sub-models are made available for deployment. Unlike our work, the runtime switching is not examined. There are ways to create more efficient models using the existing neural network architectures, such as quantization [28] and pruning [7]. Quantization can be done during training [13, 76] or after training [4, 57]. Pruning can be unstructured [29] or structured [55]. However, all the quantization and pruning methods are used to make an existing neural network more efficient. They do not target to make them dynamic.

Dynamic Neural Network Architectures: There are many works focusing on dynamic neural networks [30]. One main way to achieve a dynamic architecture is using multiple neural networks and activating some of them based on a logic. In [62], two different sized models are used. The big one is executed based on the softmax results of the little one. In [8], more than 2 models are used, and an additional logic is implemented to decide to use which model is enough for the given input. In [34], multiple models are created to use for inputs with different resolutions.

These works [8, 34, 62] do not modify the neural network architecture as our work does.

Another way to achieve a dynamic architecture is early exiting. In [79], one main network is created where multiple classifiers are added to intermediate positions in the main network. In [93], a methodology is proposed to convert the static models to dynamic models. In [82], instead of early exiting, a layer skipping idea is proposed. In [56], early exiting strategies are examined. These works are different from ours since they do not examine slimming and also runtime scheduling with power consumption as our work does.

Early exiting creates a dynamic depth in an architecture. It is also possible to create dynamic width by modifying the number of channels in convolution layers. In [92], a convolutional neural network architecture with 4 different levels of dynamic width is proposed. In [91], the previous architecture is extended to have an arbitrary level of width instead of 4 levels. In [11], an additional neural network is developed to activate the selected channels of convolution layers in the main prediction neural network. In [47], a gate module is used to activate the number of channels in each stage of a neural network. These works do not consider dynamic depth of the architecture.

Instead of creating flexibility with dynamic width and depth, some works defined various subnetworks and used dynamic routing, where the routing decisions are taken by reinforcement learning agents [52, 58]. The architecture of these works are different from ours.

Some works [5, 83, 85] have developed architectures to combine dynamic depth and width through gating. Unlike our work, gating requires additional computation to calculate which layers/block/channels will be skipped.

Dynamic Inference: The goal of having a dynamic architecture is to use it in dynamic inference. In [86], an early exit architecture is proposed and its response to contention in runtime is examined. This work do not consider dynamic width. In [44], a set of models are used to satisfy dynamic requirements of the applications with contention. In [78], a model is selected among a set of model by considering the complexity of the input on runtime. These works [44, 78] do not modify the neural network architecture. Instead, they use a set of models. One advantage of using

one dynamic neural network architecture (as in our work) instead of model selection is that only one model's weights need to be saved in the device compared to saving many different models' weights. This is especially important when resource constrained devices are used. Another advantage of our work is that it can create many fine-grained steps in terms of accuracy, latency and power consumption, whereas it is hard to find many models that form fine-grained steps in terms of these metrics. Moreover, [78] considers only input complexity while selecting a model for inference. Our work, on the contrary, considers latency and power consumption requirements while trying to maximize accuracy.

2.3 Background and Motivation

There have been a significant ongoing research effort on the dynamic networks, because of their useful features such as efficiency, representation power, adaptiveness, compatibility, and generality as stated in [30]. The adaptiveness becomes especially important when we consider dynamic environments and resource constrained systems. Modern cyber-physical systems like drones, autonomous cars, and mobile phones can be shown as examples for such dynamic environments since they can have varying requirements depending on the external factors. These systems usually have constrained resources since they run on a battery, require being small, or cannot get too hot. In the end, dynamic neural networks are great tools for such systems because of their adaptability that static models lack.

Another source of dynamism in the cyber-physical system can be introduced by different rates of incoming data. For example, as shown in [48], drones can employ multiple cameras which can be enabled/disabled depending on the application. This can create a varying data rate for the neural network architecture. Such varying data rate problem can be solved by the adaptability of the neural network architecture. There can also be dynamic requirements for the neural network application. One example for a dynamic requirement is the power budget. As investigated in [59], the wind has an effect on drones' energy consumption, which can create

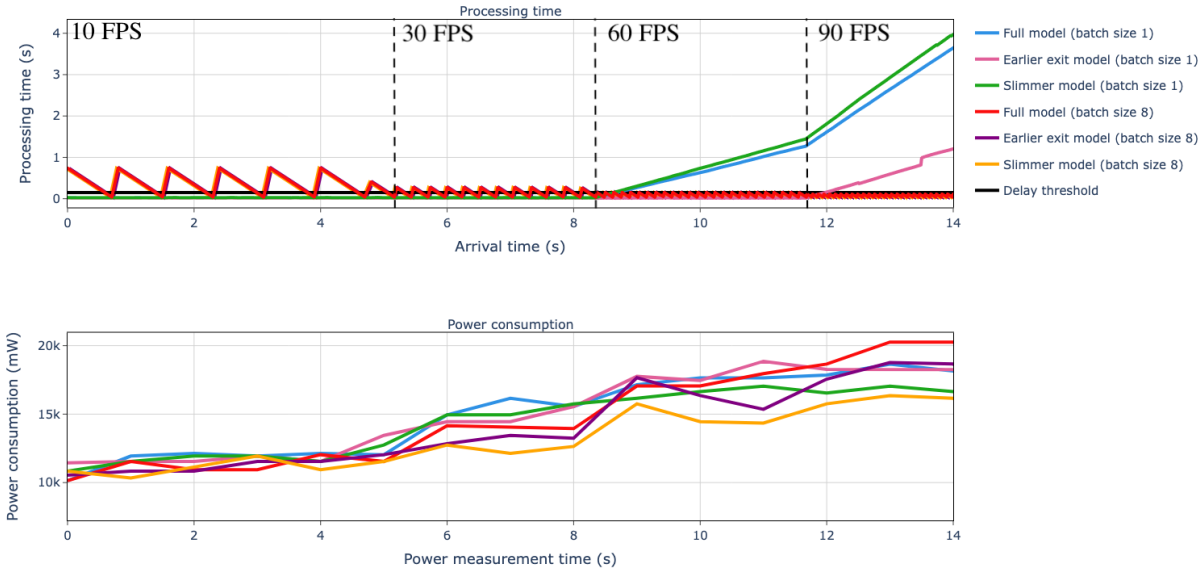


Figure 2.1. The adaptability comparison of static models, their early exit and slimmed versions

a dynamic power budget. Another example for dynamic requirements is processing time for neural network application. The drone communications have many factors that affect end-to-end latency [24, 31] which might require adaptability in time for other tasks on the drone.

The static models are not good candidates for dynamic environments, since they cannot adapt to varying conditions. A comparison of static models, along with their early exit and slimmed versions are given in Fig. 2.1 in terms of processing time and power consumption. In this example, the incoming data rate is changed every 100 frames. It starts with 10 FPS (Frame Per Second) and end with 90 FPS. The aim of the all models is to execute the incoming data without exceeding the delay threshold which is defined as 0.15 s in this particular example. The used static model is EfficientNetB0 [77]. When we examine the full model with batch size 1, we see that it can execute the incoming data with 10 and 30 FPS without any issues. However, after that point, it cannot keep up with the incoming data rate. This results in a queuing delay and therefore increased processing times for the following data. If we use the full model with batch size 8, we see high processing times in the slower incoming data rates. This happens since the model waits for batch to fill up. As a result, we see many delay threshold violations for this

model too.

Different dynamic architectures can provide adaptability to different conditions. Early exit [79] and slimming [92] techniques and their advantages can be seen in Fig. 2.1. The full model with batch size 1 starts to build a queuing delay after 60 FPS incoming data rate. On the other hand, the earlier exit version of this model can keep up with 60 FPS. Even though its buildup speed is less than the full model, it also starts to build a queuing delay after 90 FPS. When we compare the full model with batch size 8 and its slimmer version, processing times are similar. As explained in [44], slimming is not always effective in speeding up when small batch sizes are used in execution using GPUs. However, We notice a significant difference in power consumption. The slimmer version consumes much less power. This is useful when there is an adaptation requirement in power budget.

In this paper we present SLEXNet which combines the slimming and early exit techniques. It not only harnesses the advantages of each technique in one architecture, but also makes an improvement over each individual technique because of the hybrid approach. As a result, we achieve a hybrid architecture that can adapt to the conditions that early exit and slimming dynamic networks cannot. Moreover, we introduce a novel runtime scheduling algorithm that enables us to utilize the full potential of SLEXNet considering time and power constraints. Further explanations and comparisons are provided in section 2.5 with a wide range of experiments.

2.4 Methodology

2.4.1 SLEXNet

As mentioned earlier, the idea of SLEXNet is to combine early exiting and slimming in one architecture to create a more dynamic model. The early exiting method makes a model dynamic by changing the 'depth' of the model. On the other hand, the slimming method makes a model dynamic by changing the 'width' of the model. SLEXNet combines these two methods to create a more dynamic model with higher degree of adaptability.

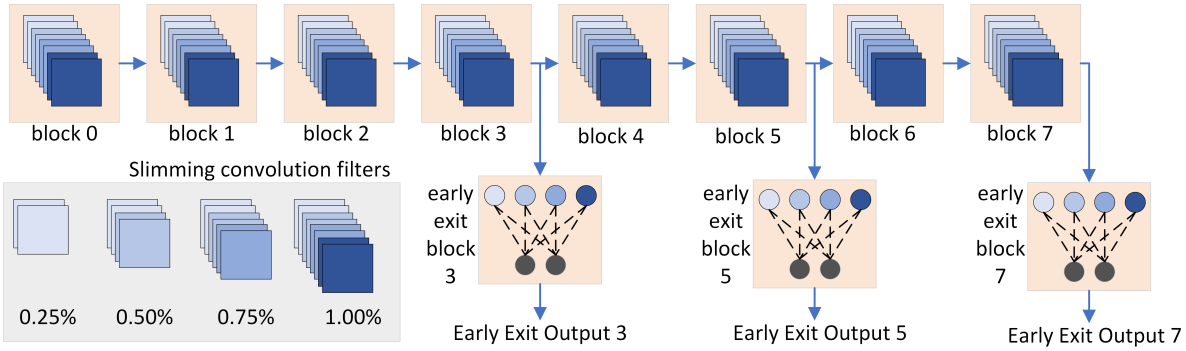


Figure 2.2. SLEXNet architecture implemented on EfficientNetB0

Architecture: We implement SLEXNet on EfficientNetB0 [77] architecture. The general overview of the SLEXNet architecture based on EfficientNetB0 is given in Fig. 2.2. EfficientNetB0 has 7 blocks, where each block can have sub-blocks. We replace the convolution and batch normalization layers with slimmable versions of them in each block. We use four slimming coefficients, which are 0.25x, 0.50x, 0.75x, 1.00x as described in [92]. We add three early exit blocks after blocks 3, 5, and 7. These early exit blocks are classifiers with another set of slimmable convolution and batch normalization, followed by global average pooling and fully connected layer.

Training: Since SLEXNet is a highly customized model, we use a customized training method, which is explained in Algorithm 1. We calculate and save the gradients for each early exit and slimming combination. Once we have all the gradients, we apply updates to the model for all gradients and then move to the next batch.

Knobs of SLEXNet for Adaptive Execution: In addition to the early exiting and slimming, we use batch execution as a knob in our runtime. Combining early exit and slimming with batch size enhances our ability to adapt to different runtime requirements and situations. Moreover, it creates more fine-grained steps to choose from.

Algorithm 1. Training of SLEXNet

```
model ← initialize SLEXNet
gradientList ← empty list
for epoch = 1,2,3...,TotalNumberOfEpochs do
    for Each batch in training set do
        for ee = 3,5,7 do
            for s = 0.25,0.50,0.75,1.00 do
                model.setEarlyExit(ee)
                model.setSlimming(s)
                logits ← model(batch)
                loss ← calculateLoss(logits)
                gradients ← calculateGradient(loss)
                add gradients to gradientList
            end for
        end for
    end for
    for Each gradients in gradientList do
        apply gradients to model
    end for
    gradientList ← empty list
end for
```

2.4.2 Runtime

SLEXNet is capable of satisfying various delay and power requirements by adjusting its knobs, however finding which knobs to use under different and varying conditions is another challenging task. In order to solve this, we developed a Runtime Scheduling algorithm which is used together with SLEXNet to provide an adaptive neural network execution system as illustrated in the Fig. 2.3. The runtime scheduler peeks at the data in the input queue and configures the SLEXNet for the next execution step using the previously benchmarked data, delay and power thresholds. In such systems, there is a constant flow of frames that are placed in input queue when they arrive. We execute the Runtime Scheduling algorithm before each execution of SLEXNet and put the results to the output queue.

The runtime scheduling algorithm is explained in Algorithm 2. It starts with calculating the average time difference (*avgTimeGap*) between arrival times of the frames that wait in the

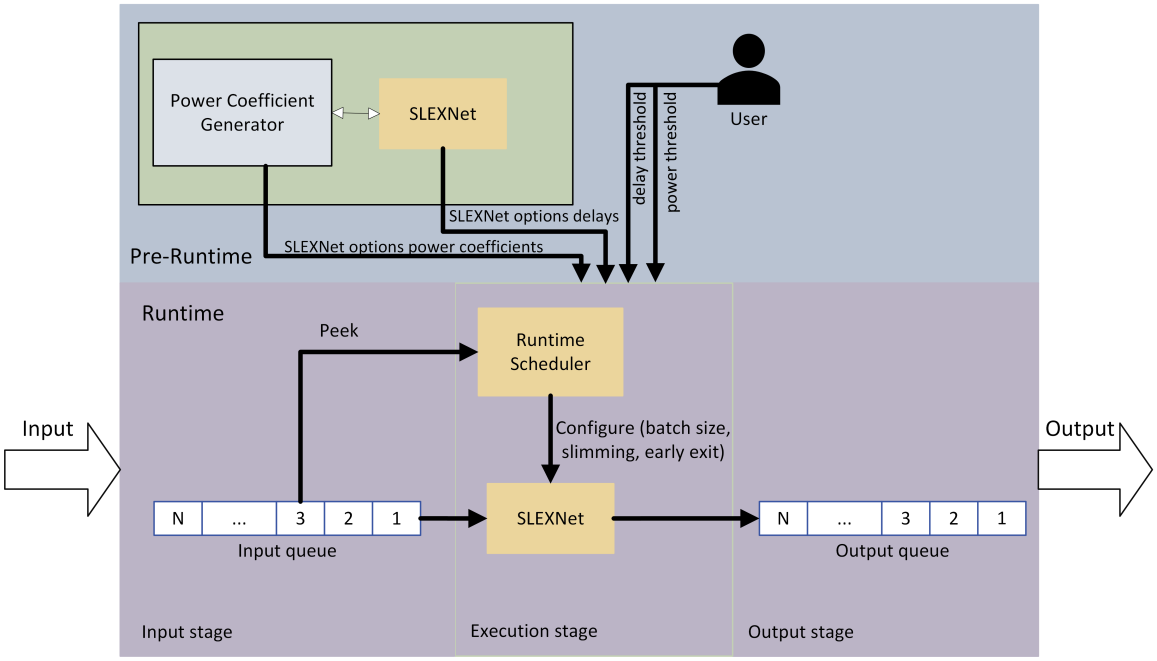


Figure 2.3. The overview of adaptive execution system using SLEXNet and Runtime Scheduling algorithm

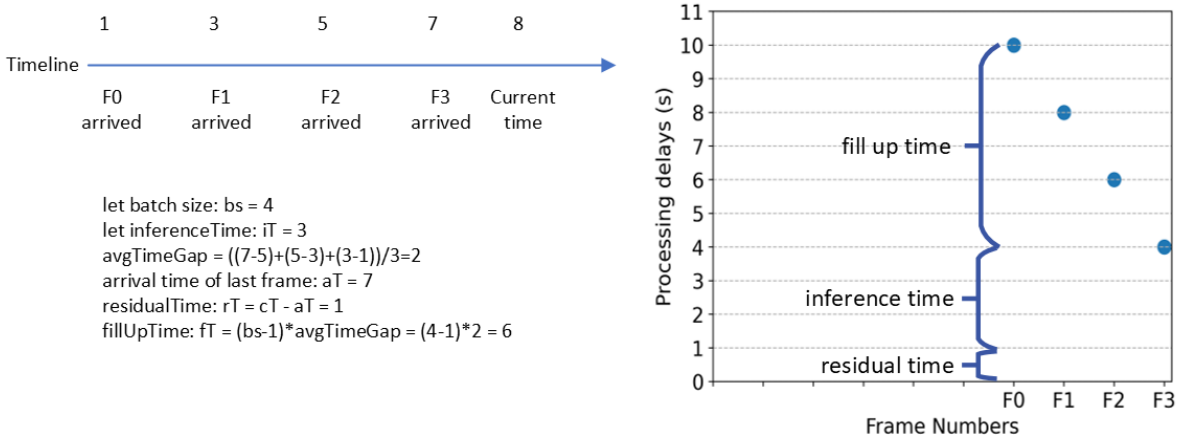


Figure 2.4. Simplified explanation of time estimation in runtime scheduling

Algorithm 2. Runtime scheduling of SLEXNet

Inputs: $SLEXNetDelays$, $SLEXNetPowerCoeffs$, $SLEXNetMaxPowers$, $delayThreshold$, $powerThreshold$, $inputQueue$, $SubNetworks$, $numOfWaitingFrames$: number of frames that wait in $inputQueue$

Output: $scheduledSLEXNetOption$

$avgTimeGap \leftarrow$ Mean of differences of arrival times of frames in $inputQueue$
$residualDelay \leftarrow currentTime -$ latest input's $arrivalTime$ in $inputQueue$ (if there's any)
for each $(s,e) \in SubNetworks$ **do** ▷ $SubNetworks$ has (slimming, early exit) tuples sorted by accuracy in decreasing order
    for $bs = 1,4,8$ **do**
        if $bs \geq numOfWaitingFrames$ **then**
            $SLEXNetOpt \leftarrow (bs,s,e)$
            $fillUpTime \leftarrow (bs-1) * avgTimeGap$
            $inferenceTime \leftarrow SLEXNetDelays[SLEXNetOpt]$
            $totalTime \leftarrow fillUpTime + inferenceTime + residualDelay$
            if $totalTime < delayThreshold$ **then**
                $powerCoefficient \leftarrow SLEXNetPowerCoeffs[SLEXNetOpt]$
                $maxPower \leftarrow SLEXNetMaxPowers[SLEXNetOpt]$
                $powerConsumption \leftarrow powerCoefficient/avgTimeGap$
                $powerConsumption \leftarrow min(powerConsumption, maxPower)$
                if $powerConsumption < powerThreshold$ **then**
                    Schedule $SLEXNetOpt$
                    $found \leftarrow True$
                    Break from all loops
                end if
            end if
        end if
    end for
end for
if not $found$ **then**
    $bs =$ highest batch size that is smaller than $numOfWaitingFrames$
    $s =$ the slimmest factor
    $e =$ the earliest exit
    $SLEXNetOpt \leftarrow (bs,s,e)$
    Schedule $SLEXNetOpt$
end if
if scheduled batch size $> numOfWaitingFrames$ **then**
    wait scheduled batch size to fill up
end if

input queue (Line 1). This value is used to estimate the total execution time later on. Then *residualDelay* is calculated. It is the waiting time of the latest frame in the input queue before the runtime scheduling and execution start. It is also used in the total execution time estimation. Then we iterate over the SLEXNet options, starting from the heaviest option. We do this by iterating *SubNetworks* which has (slimming factor, early exit point) tuples. These tuples are sorted by decreasing validation accuracy that is computed before the runtime algorithm. For example, the first element of *SubNetworks* is (1.00, 7) because it is the full network in terms of both the width and the depth. Similarly, the last element of *SubNetworks* is (0.25, 3) because it is the smallest capacity network among all the options and therefore has the worst validation accuracy. During consideration of each SLEXNet option, we first check if the batch size is equal to or larger than the number of inputs waiting in the input queue currently. Because if we pick a SLEXNet option that has a batch size smaller than the number of inputs waiting in the input queue, we leave some frames behind for this round of execution, which causes an increased *residualDelay* for the next cycle and consequently a hard case for finding a suitable SLEXNet option for the next cycle. If the batch size is larger than the number of inputs waiting in the input queue, we estimate the processing delay of the oldest frame in the input queue. A simplified example for this estimation is illustrated in Fig. 2.4. We first calculate the "fill up time" which is caused by the time difference between arrival times of the frames. In order to calculate this, we multiply (batch size - 1) with *avgTimeGap* that we calculate earlier. Then we get the inference time of the related SLEXNet option using our saved benchmark data. Then we add these two with *residualDelay* we calculate earlier to find the total time. This total time corresponds to the processing time of the oldest frame in the input queue if we use the related SLEXNet option. It is important to note that we can use a batch size that is larger than the number of inputs waiting in the input queue. If we decide to use such a batch size, the runtime algorithm waits for new frames to come after scheduling. This waiting time is actually projected in fill up time calculation. So we assume that the average time gap that current inputs have will remain similar for the incoming new frames. If the total time is less than the delay threshold, we move on to the

power consumption estimation.

The power consumption estimation requires some pre-runtime processing to extract "power coefficients". The average power consumption depends on two factors: The used SLEXNet option and the incoming data rate. The used SLEXNet option defines the maximum power, as it is directly related to how much of the available resources the system is using at a given execution time. The incoming data rate scales the maximum power of a SLEXNet option. Because if the incoming data rate is less than the current SLEXNet option's capability, the system stays idle while waiting for new data to come. It can be explained using the duty cycle idea as illustrated in Fig. 2.5. In this example, let 100 FPS be the maximum data rate the related SLEXNet option can handle. In other words, if the data comes with 100 FPS, the system executes the incoming data batch after batch without staying idle in between. In this case, the SLEXNet option consumes its maximum power. If the data comes with 10 FPS, then the average power consumption from execution drops to 10% of the maximum power, since the system waits idle 90% of the time in this case. So there's no execution for the 90% of the time. Similarly, when the data comes with 50 FPS, the average power consumption from execution drops to 50% of the maximum power. The duty cycle plot can be interpreted as following: it is 1 when the SLEXNet option executes the data with its maximum instantaneous power. It is 0 when it waits in idle state for new data to arrive (0 SLEXNet option power consumption). When the data comes faster than the SLEXNet option can handle (120 FPS in the figure), the power consumption does not change since the system is still executing data continuously, as in 100 FPS case. In this case, the data starts to queue up, however this does not change the power consumption of the system.

When we remove the base power from the measured power consumption, there's actually a constant power consumption to data rate ratio for each SLEXNet option. In other words, the ratio of average power consumption of any SLEXNet option under data rate of X FPS to X is constant for all X. We can define this ratio as power coefficient which is represented as the

following:

$$powerCoefficient = \frac{power_X}{FPS_X}$$

where X is a data rate smaller than the maximum capability of the given SLEXNet option. For example, this coefficient corresponds to $p/100$ in our example in Fig. 2.5. So, this SLEXNet option's power consumption would be $2p/10$ when the data comes with 20 FPS.

Since we calculate the *avgTimeGap* at the start of our algorithm, we actually have an estimate of incoming data rate in FPS, which is 1/*avgTimeGap*. We can use this number to estimate the power consumption as following:

$$estimatedPower = powerCoefficient * FPS = \frac{powerCoefficient}{avgTimeGap}$$

To be able to use this method, we need to calculate power coefficients for all SLEXNet options before runtime. In order to do that, we run all SLEXNet options under a small FPS (20 FPS is used in our case) and measure the power consumption. Then we remove the base power value from these numbers. Base power is measured when no execution is happening on the system. Then we divide these power consumption values to 20 (the used FPS) to calculate power coefficients. Since this ratio (*powerCoefficient*) only holds when the power consumption is equal to or less than the maximum power consumption, we also measure the maximum power consumption of all SLEXNet options to handle these cases.

Power estimation during runtime is also described in Algorithm 2. We load power coefficient and maximum power of the related SLEXNet option at lines 12 and 13, respectively. Then we estimate the power consumption using power coefficient and the average time gap at line 14. Lastly, we check if the estimated power is larger than the maximum power and if this is the case, we use the maximum power as the power consumption as power coefficient method is only for estimating power consumption that is less than the maximum power.

If the runtime algorithm cannot find a feasible SLEXNet option to schedule after iterating

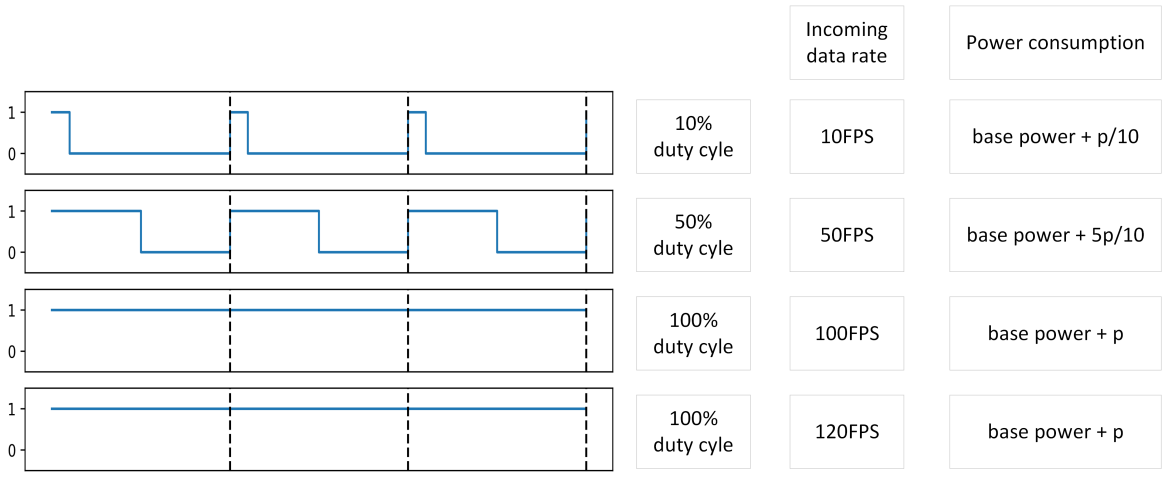


Figure 2.5. The effect of incoming data rate to the power consumption of a SLEXNet option

over all SLEXNet options, it still needs to schedule a SLEXNet option to keep executing incoming data. Since there's no feasible SLEXNet option, the scheduled one will fail the time threshold and/or the power consumption threshold. This means the system is in a state where the threshold failure is inevitable. This case is handled at line 26 in Algorithm 2. The algorithm schedules the lightest SLEXNet option (the slimmest and the earliest exit) to quickly get out of this state so that future inputs can be processed within the time and power thresholds. It also uses the highest possible batch size since batch execution is efficient and therefore provides a better chance to leave this state quickly. The system can get into this state due to queued data and can leave this state only if it executes the queued data faster than the incoming data rate. The fastest way of doing this is the SLEXNet option of the highest batch size, the slimmest factor and the earliest exit.

2.4.3 Challenges for SLEXNet

Architectural challenges

Implementing a slimmable early exit method on neural networks comes with certain challenges. We explain these challenges and how we approach them in this section.

Deciding on early exit points and slimming coefficients: The early exit points and slimming

coefficients are the main factors that define the performance of SLEXNet. In theory, an early exit point can be after any layer in the neural network and a slimming coefficient can be anything between 0 and 1. However, the following constraints need to be addressed.

1. A key constraint is the number of early exit point-slimming coefficient combinations, depending on the accuracy and runtime requirements. The subnetworks defined by early exit points and slimming coefficients are sharing the weights which are limited by the backbone architecture. That means, if too many early exit points and slimming coefficients are used, the accuracy of all subnetworks drops. On the other hand, if very small numbers of early exit points and slimming coefficients are used, there will be fewer options to pick from during runtime, which will result in worse runtime accuracy, and more time and power requirement failures. Therefore, it is important to find a middle ground depending on the backbone architecture and the runtime task.

2. Another additional constraint is the range and sparsity of early exit point and slimming coefficient parameters. If we define these parameters too close to their neighbors (i.e. slimming coefficients 0.20 and 0.25), we cannot see any significant difference in accuracy, time and power consumption between SLEXNet options. In the end, we decide to use evenly spaced parameters for early exit points and slimming coefficients, while ensuring an adequate space in between parameters.

3. Another one of these constraints is the backbone architecture. Most of the modern convolutional neural network architectures are based on building blocks that are repeated [32, 70, 77]. When defining an early exit, it is important to not cut these building blocks. Because if an early exit point cuts a building block, the defined subnetwork consists of an incomplete block, resulting in bad performance. Therefore, we define early exit points at the end of building blocks. However, this is not the only backbone architecture constraint to decide on early exit points. The effect of slimming coefficients on the backbone architecture needs to be considered to pick early exit points as well. The combination

of early exit points and slimming coefficients define the subnetworks of the backbone architecture. If the smaller subnetworks don't have enough parameters to learn the features of the dataset, they do not provide viable subnetwork options and unnecessarily hurt the performance of other subnetworks due to weight sharing.

Slimmable early exit classifiers: The size of a slimmable convolutional kernel is determined by the slimming coefficient. The full size of the convolution kernel is

$$[kernelsize,$$
$$kernelsize,$$
$$(inputChannelNumber) \times (slimmingCoefficient),$$
$$(outputChannelNumber) \times (slimmingCoefficient)]$$

This means the channel number of output tensors of a slimmable convolutional layer depends on the slimming coefficient. The early exit classifiers need to be slimmable as well, since the sizes of tensors from intermediate blocks change based on the slimming coefficient. The early exit classifiers consist of a slimmable convolution layer, a global pooling layer and a final fully connected layer. A fully connected layer expects a constant input size. Most modern neural network architectures handle this by using global pooling layers [32, 70, 77] before the fully connected layer. Global pooling layer gets the input tensor with size (batch size, spatial size, spatial size, channel number) and decreases it to a tensor with size (batch size, channel number). This is a constant size for neural networks without slimming, since the channel numbers of convolution filters without slimming are constant. However, if we use a slimmable convolution layer before the global pooling layer, the channel number is scaled by the slimming coefficient which cannot be handled by the last fully connected layer. Therefore, we use "half" slimmable convolutional layer in the slimmable early exit classifiers. These layers have filters that are slimmable in the input but constant in the output. The full size of the convolution kernel of these layers is [$kernelsize$, $kernelsize$, $(inputChannelNumber)$ x

$(slimmingCoefficient)$, $(outputChannelNumber)$]. So the output tensors of these layers have a constant number of channels and are independent of the slimming coefficient, and therefore can be used by the final fully connected layers. The output size of these slimmable convolutional layers is an architectural choice and potentially affects the performance of SLEXNet.

SLEXNet also introduces some new parameters to the backbone architecture due to the additional classifiers (at early exit 3 and early exit 5 for our implementation shown in Fig. 2.2). Note that early exit 7 is the normal output of the backbone architecture. The introduced parameters come from the slimmable convolution layer and the fully connected layer at the early exit classifiers. The filter size of the slimmable convolution layer is (kernel size, kernel size, input channel number, 4 x input channel number). The weight size of the fully connected layer is (4 x input channel number) x (number of output classes). Kernel sizes are 1 in the convolution layer in the early exit classifiers. The input channel numbers are 40 and 112 for early exit classifiers 3 and 5, respectively. So, the total number of parameters of early exit classifier 3 is (1 x 1 x 40 x 160) + (160 x 5) = 7200. Similarly, the total number of parameters of early exit classifier 5 is (1 x 1 x 112 x 448) + (448 x 5) = 52416. The total number of parameters of EfficientNetB0 is 5.3M [77]. Therefore, the parameters introduced by SLEXNet is around 1.2% of the backbone architecture.

Training of SLEXNet

SLEXNet consists of many subnetworks which are defined by early exit points and slimming factors. For example, early exit point 3 and slimming factor 0.75 define a subnetwork (=Subnetwork(3, 0.75)) together. Similarly, early exit 7 and slimming factor 0.25 define another one (=Subnetwork(7, 0.25)). All the subnetworks share the weights of the backbone architecture. This means the same set of weights has different purposes for different subnetworks. For example, the weights of block 2 in Fig. 2.2 works as high level feature extraction for Subnetwork(3, 0.75) because it is close to the early exit 3 classifier. However, the same weights of block 2 work as low level feature extraction for Subnetwork(7, 0.25) because block 2 is positioned in the initial

part of the 7-block subnetwork. Therefore, the weights of block 2 need to learn different features for different subnetworks. The training needs to be done in a way that all the subnetworks are trained equally and none of them is given a precedence over another subnetwork. If some subnetworks are trained before the other ones in each batch, it might make convergence harder for the subnetworks that are trained later due to shared weights. Therefore, we run forward (line 9 in Algorithm 1) and backward (line 11) passes for all subnetworks and calculate the gradients for each subnetwork. However, we apply the gradients to the weights after gradients of all subnetworks are calculated (line 17).

Runtime scheduling

The runtime scheduling problem is very challenging to solve optimally due to two main reasons. The first reason is that the search space is too big to solve during runtime before each execution. In the first glance, it may seem as if search space is small and constant since there are only 36 options (3 batch sizes * 3 earl exits points * 4 slimming coefficients) to pick at a given time. However, search space grows quickly based on the number of inputs in the input queue. Let's take a specific example where there are 6 frames in the input queue, and we want to execute them optimally. Optimal execution is maximizing accuracy while satisfying processing delay and power thresholds for all the frames. Our available batch sizes are $1, 4, 8$ and we have 12 subnetworks (3 early exits * 4 slimming coefficients). In one batch execution scenario, 6 frames can be executed in 3 batches. We can execute 4 of them first, then we can execute the remaining frames one by one. This means execution batches will be 4-1-1. Each of these batch executions can use one of the 12 subnetworks. Since there are 3 batch executions, there are 12^3 options just for 4-1-1 execution order. Alternative batch execution scenarios can be 1-4-1 (12^3 total options) or 1-1-4 (12^3 total options) or 1-1-1-1-1-1 (12^6 total options). Each batch execution has an effect on the inputs that are waiting in the input queue. Because those inputs have a smaller time window to satisfy the processing delay threshold since they spend time in the input queue waiting for their execution. This means the first execution option has an effect

on the last input waiting in the input queue. These numbers are only for 6 frames waiting in the input queue. If there are, for example, 7 frames in the input queue, the search space grows exponentially. Solving this problem during runtime takes long and more importantly unknown time since it depends on the number of inputs in the input queue.

The second reason for the diffulty in solving the runtime scheduling algorithm is that the environment is dynamic. This means frames keep coming to the input queue while we try to solve the scheduling problem. For example, while we are solving the 6-input case, another input can arrive in the input queue. In the end, our solution for the 6-input case might cause the 7th input to miss its processing delay threshold. This would make the solution suboptimal.

Our average time gap idea estimates the incoming data rate, which makes the problem solvable in almost constant and very short time. It is also used to accurately estimate end to end processing delay and power consumption. Therefore, it enables us to pick the most suitable SLEXNet option. This method is practical and highly effective. Therefore, it introduces a novelty to this kind of search problem where the environment is dynamic, and a fast solution is required.

2.5 Experiments

We use a specific notation to denote the SLEXNet options in the figures throughout this section. As explained in the previous section, the knobs of SLEXNet are the batch size, the slimming factor and the early exit point. We use the naming convention of *bsX_sY_eeZ* where X is the used batch size, Y is the slimming factor and Z is the early exit point. For example, bs4_s0.25_ee5 means batch size 4, slimming factor 0.25, early exit 5.

The runtime experiments are conducted on Nvidia Jetson Orin platform.

2.5.1 Training and Dataset Details

We train SLEXNet on AIDER dataset [46]. It is an aerial image dataset for emergency response applications. It includes images of emergency situations such as fire, collapsed buildings, floods, traffic incidents as well as normal images captured by drones. An example of each

Figure 2.6. Example images from AIDER dataset [46]

category in the dataset can be seen in Fig. 2.6. This dataset is relevant to our use case since it can be used in resource constrained systems such as drones where various runtime constraints can be imposed depending on the external conditions.

In addition to the customized training of SLEXNet shown in Algorithm 1, we use some general image based training methodologies. We first resize all the images to 224 by 224 pixels. Then, we use image augmentation provided by imgaug library [40]. The imgaug library applies random image augmentation techniques to each image in the training pipeline. We use Adam optimizer [41] with 0.01 learning rate. We use a batch size of 128 during training.

We split the dataset into a training set ($\sim$80%), validation set ($\sim$10%) and test set ($\sim$10%). We check the validation accuracy at every epoch during the training set and stop the training when there is no improvement for 50 epochs. Then we save the best validation accuracy model and use it in our runtime experiments in the following sections. The test set is used in our runtime experiments. The test accuracies of the SLEXNet branches after training are given in Fig. 2.7.

2.5.2 Offline Performance Evaluation of SLEXNet

In this section, we manually impose various runtime requirements on a real-time image classification problem and showcase SLEXNet's adaptability. This manual and offline evaluation shows SLEXNet's capabilities in full extent and is also helpful to understand the other experiments in the following sections. In this section, we consider two main metrics as requirements – speed and power. We use a different set of requirements for each of the metrics and use SLEXNet's adaptability to satisfy each of these requirements.

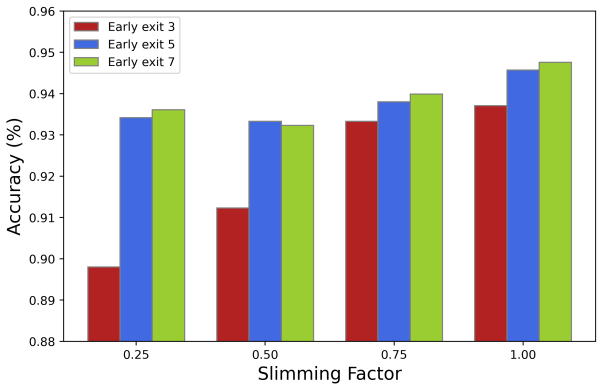
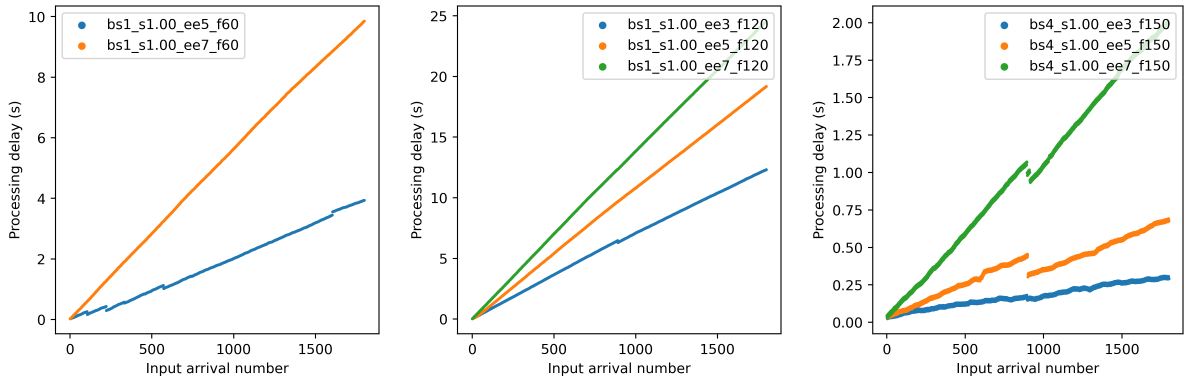


Figure 2.7. The accuracies of SLEXNet branches with different early exit points and slimming factors

Our aim is to find and select a SLEXNet option (combination of batch size, slimming factor and early exit branch) that satisfies the speed and power requirements best, while also sacrificing the accuracy least. Changing slimming factor and early exit branch results in the change in accuracy as shown in Fig. 2.7. For example, a slimmer model is less accurate. Similarly, if we use an earlier branch to exit, the model is less accurate. The other SLEXNet option is the batch size which however, does not have any effect on accuracy. But it has combined effects with other SLEXNet options, and also it is required for certain requirement cases.

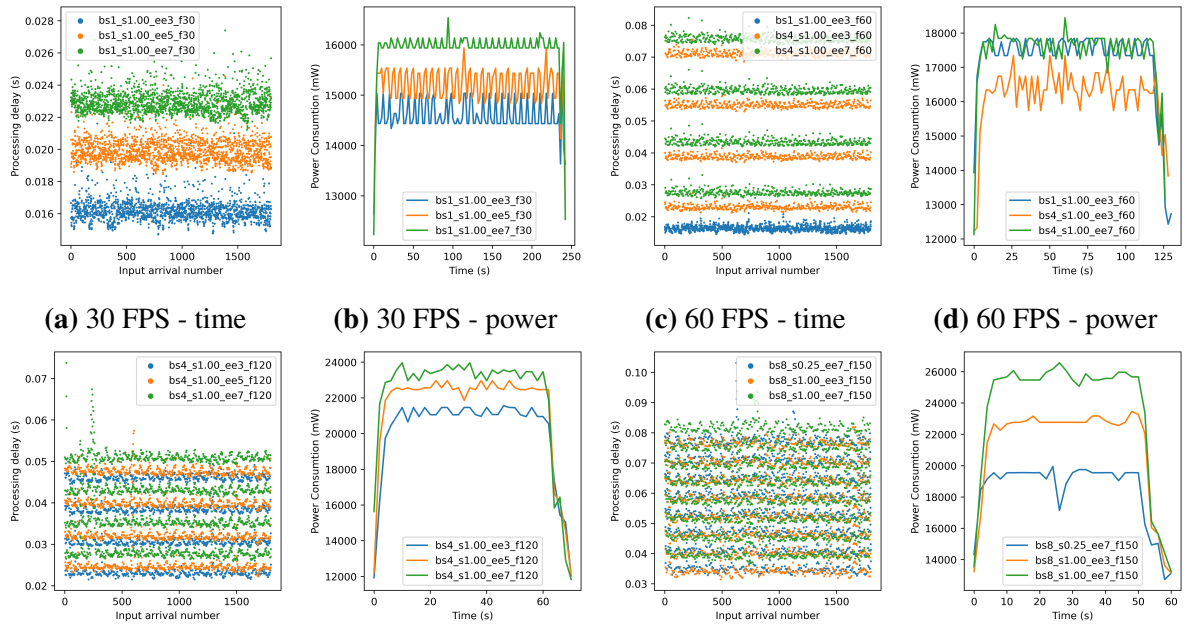
Since, in this offline evaluation, we keep the incoming FPS constant for each experiment, in section, we add the incoming FPS to the legend naming. So we also add $\_fW$ to our general naming convention. f means the frame rate and W is the placeholder for the frame rate value in FPS. For example, bs4_s0.25_ee5_f60 means batch size 4, slimming factor 0.25, early exit 5 under 60 FPS incoming data.

We change the incoming data FPS (Frames per second) rate and use different SLEXNet options to keep up with the incoming data. In the plots, we use the term '*processing delay*' which is defined as an input's serving time that includes the queuing delay and SLEXNet's inference delay. An input is captured with the defined FPS rate and put into the SLEXNet's input queue, then its queuing delay starts. The queuing delay ends when the input is taken from input queue by SLEXNet and the inference delay starts. When the input is processed by SLEXNet and put to



(a) 60 FPS **(b)** 120 FPS **(c)** 150 FPS

Figure 2.8. Some failing SLEXNet options for varying FPS incoming data rates (Accuracies of the SLEXNet options can be seen in Fig. 2.7)



(a) 30 FPS - time **(b)** 30 FPS - power **(c)** 60 FPS - time **(d)** 60 FPS - power

(e) 120 FPS - time **(f)** 120 FPS - power **(g)** 150 FPS - time **(h)** 150 FPS - power

Figure 2.9. Some working SLEXNet options for varying FPS incoming data rates (Accuracies of the SLEXNet options can be seen in Fig. 2.7)

the output queue, the inference delay ends. In the end, the processing delay is the sum of the queuing delay and the inference delay for each of the inputs.

Now, as the typical camera data can be of 30 FPS or 60 FPS and there can be multiple of them simultaneously stream the data, we may get even higher frame rates that are multiple of 30 FPS. Herein, we consider individual cases of different frame rates and show their offline performance below.

1) 30 FPS incoming data case:

30 FPS incoming data is slow enough to be satisfied (i.e., processing delay is below 1/30 s) by using batch size 1. However, when we do not use batches in execution, slimming is not effective in terms of speed-up. Therefore, we only demonstrate the early exit variations of the none slimmed (slimming factor 1.0) model in this use case in Fig. 2.9a. As we can see from the figure, all the used SLEXNet options can satisfy 30 FPS speed requirements, i.e. there is no increasing queuing delay that violates the processing delay requirement, while the inputs keep coming. However, early exiting still affects the processing delay of each input. When we consider the power consumption numbers in Fig. 2.9b, we can see that early exiting affects power. For example, if we don't have a power consumption requirement, we can select the SLEXNet option – {batch size 1, slimming factor 1.0, early exit point 7}, since it can catch up with the speed requirement and has the better accuracy compared to others which can be seen in Fig. 2.7. However, if we have a power consumption requirement of 15000 mW, we have to select {batch size 1, slimming factor 1.0, early exit point 3} among the three options because it can keep up with the speed requirement and also satisfies the power requirement.

2) 60 FPS incoming data case:

When we increase the incoming data speed to 60 FPS, it starts to be too fast for some SLEXNet options. For example, when we use the batch size of 1, the early exit 7 and 5 cannot keep up with the incoming data and a queuing delay is accumulated as shown in Fig. 2.8a. In this case, if we want to keep up with the incoming data, we can decrease the early exit parameter even further to early exit point 3. However, using early exit point 3 also decreases the accuracy as

seen in Fig. 2.7. If we want to keep the accuracy high while satisfying the speed the requirement, we can use batch sizes greater than 1. In this case, using a batch size of 4 allows us to select the SLEXNet option (early exit 7 and slimming 1.00) that yields the maximum accuracy while satisfying the speed requirement. The processing delay comparison of some working SLEXNet options is shown in Fig. 2.9c. As we can see from the figure, using batch size 1 and early exit 3 is the fastest. Using batch size 4 increases the minimum processing delay and also shows a 4-step processing delay pattern since 4 consecutive inputs are executed together and put to the output queue at the same time. Using an earlier exit on batch size 4 case still gives an improvement on speed. In the end, all three SLEXNet options can satisfy the speed requirement of this case. When we examine the power numbers shown in Fig. 2.9d, we see that all options are close to each other in terms of power consumption. When we compare bs1_s1.00_ee3_f60 and bs4_s1.00_ee7_f60, we see that the increase in batch size and decrease in early exiting balanced each other in terms of power consumption. On the other hand, bs4_s1.00_ee3_f60 has a considerably lower power consumption compared to the other two options. Therefore, if there would be a power consumption requirement around 17000 mW, bs4_s1.00_ee3_f60 would be the only feasible option among these three.

3) 120 FPS incoming data case:

If we increase the incoming data rate to 120 FPS, batch size 1 becomes infeasible even with the most efficient SLEXNet options as shown by monotonic increase in processing delay in Fig. 2.8b. Therefore, we have to use SLEXNet options with higher batch sizes to keep up with this incoming data rate. If we use batch size 4, we can keep up with the incoming data. Then we can still change other parameters to tradeoff accuracy with speed, as shown in Fig. 2.9e. In the power comparison plot in Fig. 2.9f, we see a similar pattern to the 30 FPS case. Depending on the power consumption requirement and accuracy preference, different SLEXNet options can be selected among these options.

4) 150 FPS incoming data case:

In the extreme speed requirements, even batch size 4 options become infeasible, as

shown in Fig. 2.8c. In this case, using batch size 8 with different combinations can keep up with data while providing different speed and power characteristics. 3 different SLEXNet options that can keep up with the incoming data rate are shown in Fig. 2.9g. Examining the plot, we see that bs8_s1.00_ee7_f150 has the largest processing delay, and the other two are similar. Decreasing slimming factor from 1.00 to 0.25 and decreasing exiting point from 7 to 3 seem to have similar effects on processing delay. However, there are significant differences in power consumption, shown in Fig. 2.9h. As expected, bs8_s1.00_ee7_f150 has the largest power consumption. However, bs8_s0.25_ee7_f150 has significantly less power consumption compared to bs8_s1.00_ee3_f150. Therefore, we can reach to a conclusion that slimming is more power efficient than early exiting when everything else remains the same and relatively large batch sizes are used.

2.5.3 Online SLEXNet Performance with Adaptive Scheduling

As shown in the previous section, some SLEXNet options are not feasible, or not optimal for a given power threshold and incoming data rate. In this section, we demonstrate how SLEXNet can adapt by switching options during runtime, which means selecting the right SLEXNet option for a given the current condition of the system and the requirements. We also compare SLEXNet with early exit and slimming methods to emphasize the advantages of our architecture.

Performance Metrics: In the experiments, we impose two requirements, one is the processing delay threshold and the other one is power consumption threshold. We measure the performance of the different techniques (architecture and scheduling algorithm) by their compliance with these thresholds in different scenarios. In addition to the various plots showcasing the delay and power performance on temporal scale, we use 3 main metrics to summarize and compare the performance of different techniques. These are named as on-time accuracy (OTA), time fail rate (TFR), and power fail rate (PFR). On-time accuracy is defined as the accuracy of the model when it satisfies the processing delay threshold. If the technique fails to satisfy the processing

delay threshold for some inputs, they are classified as false prediction. In other words, the true predictions are the ones that are predicted correctly by the model and processed under the processing delay threshold. Time fail rate is the ratio of the number of frames that violated the delay threshold to total number of frames. Power fail rate (PFR) is the ratio of the number of measurement points that violate the power threshold to the total number of points. These three metrics give a good idea about the performance of the different architectures and scheduling algorithms under different requirements.

Increasing FPS Scenario

In this section, we use a constant power threshold and increase the incoming data rate gradually in steps, and observe the response of SLEXNet and our switching algorithm. We start the data rate at 30 FPS and increase it by 5 after every 200 data points, until we reach 125 FPS. We put a processing delay threshold of 0.15 s and power consumption threshold of 18000 mW for the detailed experiment in this section. However, we also share the summary of results for combinations of processing delay threshold and power consumption threshold.

The processing delay of each frame and the used SLEXNet option to execute that frame can be seen in Fig. 2.10 when SLEXNet with our scheduling algorithm is used. This plot is useful to observe the adaptability of our methods. Since it is an increasing data rate scenario, it starts using batch size 1 and then moves to higher batch sizes to support the higher data rates. It also uses its other adaptive parameters to adapt to the increasing data rate. It starts using early exit 7 which is the full model and then switches to earlier exit points such as 5 and 3. Similarly, it starts to use slimming factor 1.0 which is the full model and then switches to slimmer versions. As a result, it can stay below the processing delay threshold all the time. Furthermore, it also satisfies the power threshold all the time, as shown in Fig. 2.13a.

To understand the scheduling mechanism better, we can examine the SLEXNet options that are rejected by our scheduling algorithm before it selects the current one. For example, let's take frame #1500 where bs4_s0.75_ee5 is used. When the scheduling algorithm starts

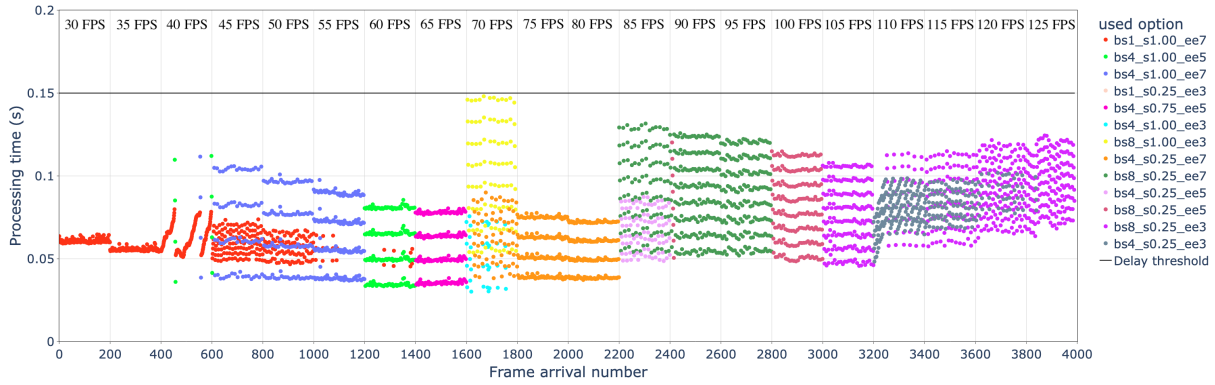


Figure 2.10. Processing delay of each frame by their arrival numbers when **SLEXNet** is used with our scheduling algorithm during increasing FPS scenario

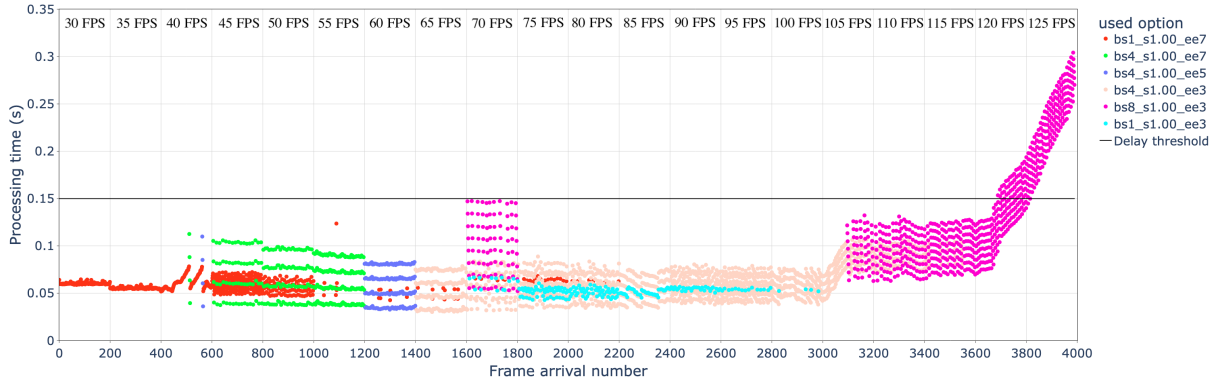


Figure 2.11. Processing delay of each frame by their arrival numbers when batch size and **early exit** are used during increasing FPS scenario

to search the SLEXNet options to schedule, there were already 2 frames waiting in the input queue. Therefore, it rejects all bs1 options. The rejected options in the rejection order and their reasons are as follows: bs1_s1.00_ee7 - bs is smaller than the backlog, bs4_s1.00_ee7 - power is estimated to be larger than power threshold (18903mW > 18000mW), bs8_s1.00_ee7 - delay is estimated to be larger than delay threshold (0.1547 > 0.15), bs1_s1.00_ee5 - batch size, bs4_s1.00_ee5 - power estimation (18194mW), bs8_s1.00_ee5 - time estimation (0.1533), bs1_s0.75_ee7 - batch size, bs4_s0.75_ee7 - power estimation (18580mW), bs8_s0.75_ee7 - time estimation (0.1544), bs1_s0.75_ee5 - batch size. Then it selects the option bs4_s0.75_ee5 eventually.

We also compare the performance of SLEXNet with the early exit and slimming tech-

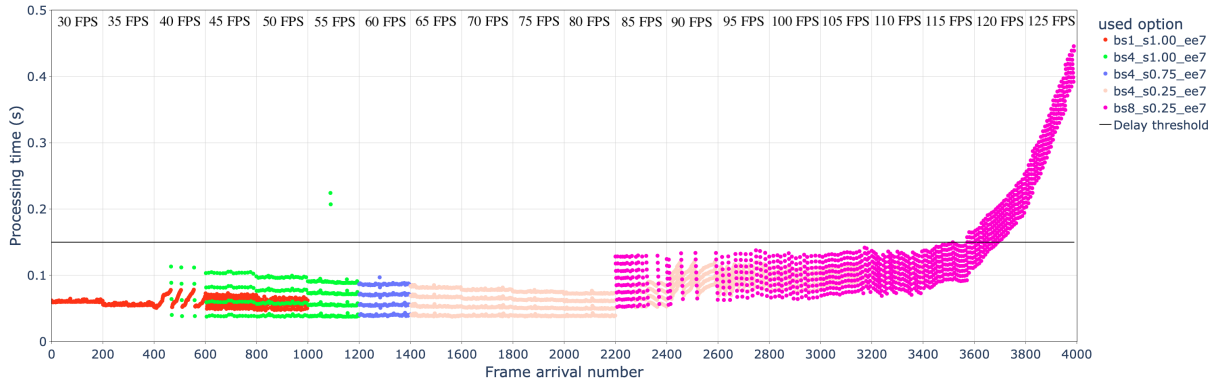
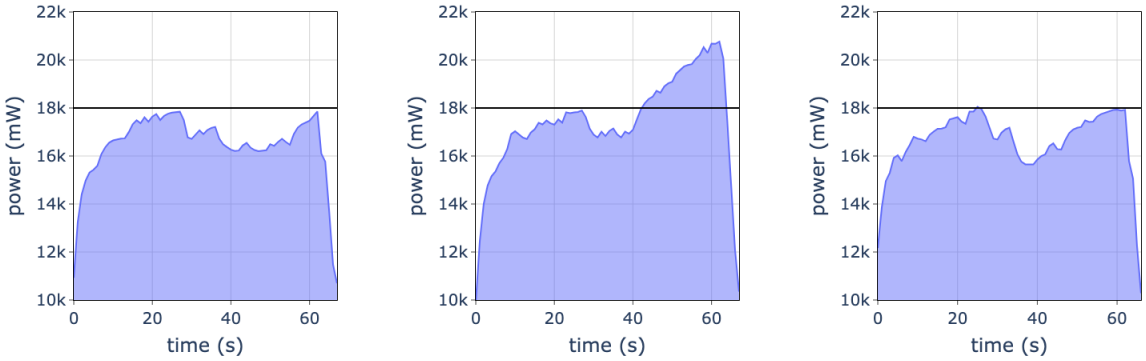


Figure 2.12. Processing delay of each frame by their arrival numbers when batch size and **slimming** are used during increasing FPS scenario

niques. The processing delay plot of early exit and slimming can be seen in Fig. 2.11 and Fig. 2.12, respectively. Neither of them can keep up with the increasing FPS after a point. As a result, they violate the delay threshold. Moreover, we can see the power consumption values for these techniques in Fig. 2.13b and Fig. 2.13c, respectively. Both of the techniques violate the power threshold at some points. Since slimming is a more power efficient method, its violation is rarer.

SLEXNet has many more options to select from compared to early exit and slimming. As a result, we can see that SLEXNet is using many options and combine them in different incoming data rate conditions. If we look at the legends of processing time plots, we see that SLEXNet is using 13 different options, while early exit and slimming are using only 6 and 5 options, respectively. Therefore, SLEXNet's increased flexibility is actually put in use in the adaptive scheduling.

These temporal plots were only based on two requirements, a constant processing delay threshold of 0.15 s and a power threshold of 18000 mW. Here we run SLEXNet with our scheduling algorithm under a range of time and power requirements and summarize the results in terms of the defined metrics in the Table 2.1. We also compare SLEXNet with early exit and slimming in the table. The table shows that SLEXNet has better on-time accuracy than early exit and slimming methods in all combinations of time and power requirements. Because SLEXNet violates the delay threshold less than other methods. Furthermore, it has more options to select



(a) SLEXNet **(b)** Early exit **(c)** Slimming

Figure 2.13. Power consumption values of different architecture during increasing FPS scenario

Table 2.1. Summary of increasing data rate results using on-time accuracy (OTA), time fail rate (TFR) and power fail rate (PFR)

Processing delay threshold(s)	Technique	Power threshold (mW)								
		18000			20000			22000		
		OTA(%)	TFR(%)	PFR(%)	OTA(%)	TFR(%)	PFR(%)	OTA(%)	TFR(%)	PFR(%)
0.10	SLEXNet	**86.4**	**5.5**	**0**	**88.2**	**4.4**	**0**	**88.5**	**3.5**	**0**
	Early Exit	77.2	16.5	32.4	78.8	15.1	11.8	79.8	14.1	0
	Slimming	70.7	23.6	0	72.9	21.3	0	71.7	22.7	0
0.15	SLEXNet	**91.8**	**0**	**0**	**92.5**	**0**	**0**	**92.3**	**0**	**0**
	Early Exit	87	6	30.9	86.9	6.4	11.6	87.4	5.8	0
	Slimming	83.8	9.2	1.5	85.7	7.4	0	85.3	7.9	0
0.20	SLEXNet	**91.9**	**0**	**1.5**	**92.5**	**0.1**	**0**	**92.2**	**0.1**	**0**
	Early Exit	90	2.8	30.4	89.8	3.3	12.9	91.9	0.9	0
	Slimming	88.1	4.7	8.6	89.4	3.6	0	89.5	3.4	0

from, and therefore it can tradeoff accuracy less than the other methods when it is applicable. We also notice that on-time accuracy decreases for all techniques when we impose more strict requirements. This happens because of two reasons. The first one is that when we impose a lower processing delay threshold, it is naturally harder to satisfy it. Since the classification of frames that do not satisfy the processing delay are considered false in on-time accuracy, the overall on-time accuracy decreases. The second reason is that when we impose lower processing delay and power thresholds, we limit the SLEXNet options that Runtime Scheduling algorithm can choose from. Runtime Scheduling algorithm is forced to select less accurate SLEXNet options to satisfy lower thresholds. In the end, the on-time accuracy drops again.

Randomly Varying FPS Scenario

Although increasing data rate scenario is a good way to examine the step by step response of the SLEXNet and Runtime Scheduling algorithm, in practical use cases, it is expected to see both increasing and decreasing data rate variations. Therefore, we evaluate the SLEXNet using randomly changing data rate and duration in this section. Additionally, similar to the previous section, we also compare SLEXNet with early exit and slimming.

The same random FPS scenario is used for SLEXNet, early exit and slimming. The processing delay results for SLEXNet are shown in Fig. 2.14 in detail. In this figure, we use the batch execution number instead of arrival number. All frames in the same batch executed together get the same batch execution number, but they have different arrival numbers. We decide to use batch execution number since it looks similar to the original plot, but it is easier to examine when there are many frames since the x-axis is smaller compared to arrival number plot. In these detailed results, we can see that SLEXNet is using various options to achieve the processing delay under the threshold which is 0.12 s while power consumption threshold of 18000 mW is imposed.

In this and the following section, we combined processing times and power consumption in one plot for each technique, as shown in Fig. 2.15. These results are for the case of processing delay threshold 0.12 s and power consumption threshold 18000 mW. So, Fig. 2.14 is the detailed version of the time trace of Fig. 2.15a. The used options are not shown in these figures to simplify it so that other aspects can be focused on better. In Fig. 2.15, the black horizontal line is the threshold for both processing time and power consumption. It is aligned at 0.12 s in the processing time axis and at 18000 mW in the power axis. The dark yellow trace is the processing time and the blue trace is the power consumption. The background is colored red whenever a processing time or power consumption violation happens. The results are similar to the increasing FPS scenario as SLEXNet achieves to stay below the processing delay threshold under changing conditions, while the early exit and slimming techniques violate the processing

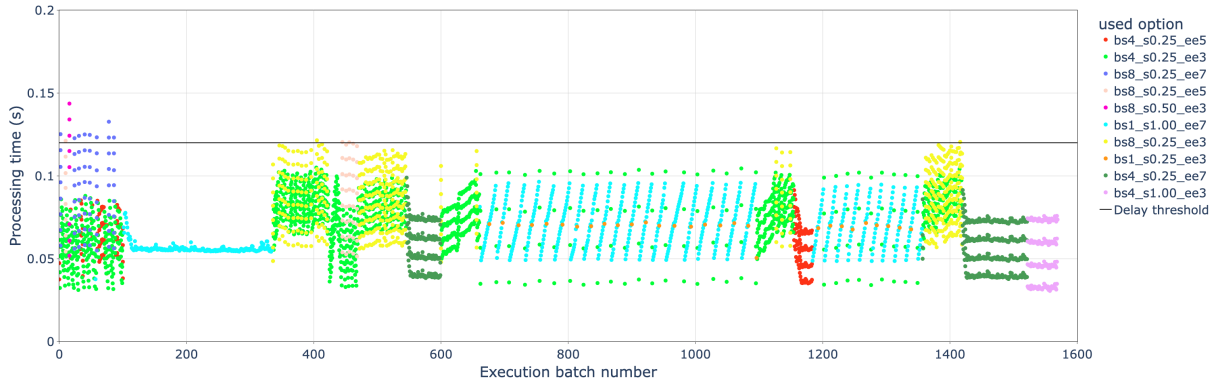
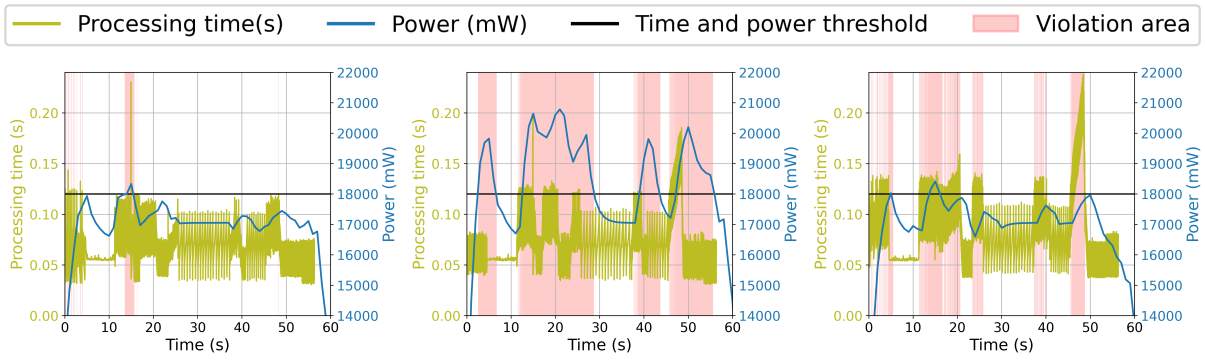


Figure 2.14. Processing delay of each frame by their execution batch numbers when **SLEXNet** is used with our scheduling algorithm during random FPS scenario

delay threshold frequently. In some high data rate regions, even though early exit is failing to satisfy the processing delay threshold, it is still better than slimming since slimming fails to satisfy the processing delay threshold for more number of frames in the same regions. On the other hand, the early exit case struggles to stay below power threshold. Whenever the incoming data rate increases, early exit starts exceeding the power threshold and stays there until incoming data rate drops to a level that early exit can manage. So we can say that early exit cannot adapt to increased data rate in respect to power consumption. Early exit's power consumption depends on the external factors such as incoming data rate because early exit method is not capable enough to decrease the power consumption when it is necessary. Slimming performs better than early exit. We can see that when the incoming data rate increases, the slimming's power consumption goes over the threshold slightly. But, since slimming technique is capable of decreasing the power consumption, it can recover from these points. In the end, slimming has much less power consumption fail rate than early exit, but still has some. SLEXNet is clearly better than the other two methods in terms of staying under the power threshold. When we examine the red highlighted background to investigate the combined performance in terms of processing time and power consumption, we can see that SLEXNet can satisfy both thresholds almost all the time and outperforms the other two methods.

The summary of results is presented in Table 2.2 with different processing delay threshold



(a) SLEXNet **(b)** Early exit **(c)** Slimming

Figure 2.15. Processing time and power consumption values of different techniques during changing FPS scenario - black line (—) is 0.12s processing time and 18000mW power consumption thresholds

Table 2.2. Summary of random data rate results using on-time accuracy (OTA), time fail rate (TFR) and power fail rate (PFR)

Processing delay threshold(s)	Technique	Power threshold (mW)								
		18000			20000			22000		
		OTA(%)	TFR(%)	PFR(%)	OTA(%)	TFR(%)	PFR(%)	OTA(%)	TFR(%)	PFR(%)
0.10	SLEXNet	**86.8**	**3.8**	**3.2**	**85.8**	**6.5**	**0**	87.3	4.4	0
	Early Exit	69.8	24.6	54	71.9	22.8	17.2	73.9	20	0
	Slimming	55.6	40.2	7.9	58.7	36.8	0	63.7	31.3	0
0.12	SLEXNet	**90.1**	**0.5**	**3.1**	**91**	**0.5**	**0**	91.4	0.3	0
	Early Exit	85.3	7.8	52.4	85.3	8	14.1	86.2	7.1	0
	Slimming	77.4	16.5	6.3	72.1	22.3	0	77.7	16.4	0
0.15	SLEXNet	**90.8**	**0.5**	**1.5**	**92.7**	0.2	**0**	92.6	**0**	**0**
	Early Exit	90.2	2.6	50.7	92.6	**0**	14.1	**92.7**	0.1	0
	Slimming	91.3	1.4	14.9	91.2	1.5	0	90	3.1	0

and power threshold combinations. Similar to the previous section, we see that SLEXNet outperforms early exit and slimming in all combinations. As we discussed earlier, early exit is better than slimming in terms of satisfying time requirements and slimming is better than early exit in terms of satisfying power consumption requirements. SLEXNet is better than both techniques in terms of satisfying time and power requirements. Because even though one technique is worse than the other for a task, it still creates some steps in that task. For example, even though early exit is worse than slimming for managing power consumption, it can still provide options with different power consumption. Similarly, slimming can provide options with different inference delays. When the two techniques are combined, SLEXNet has more options with fine-grained time and power profiles than the two techniques alone.

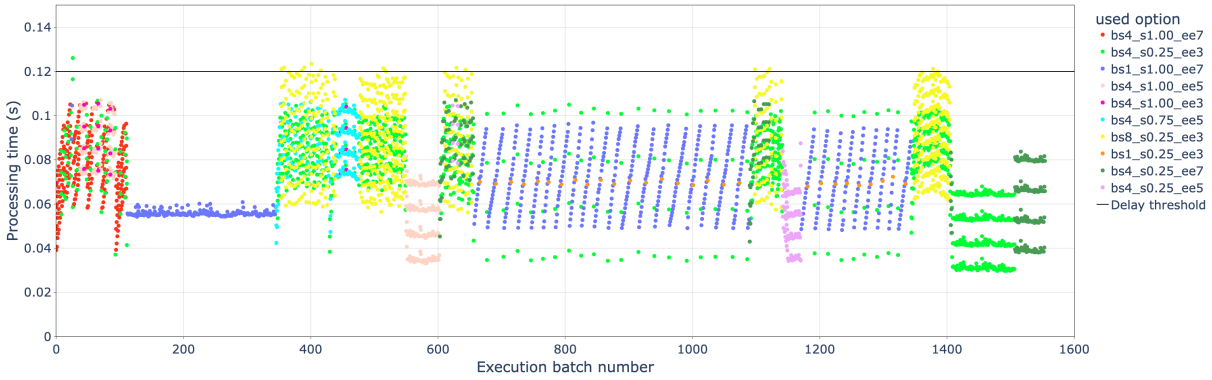
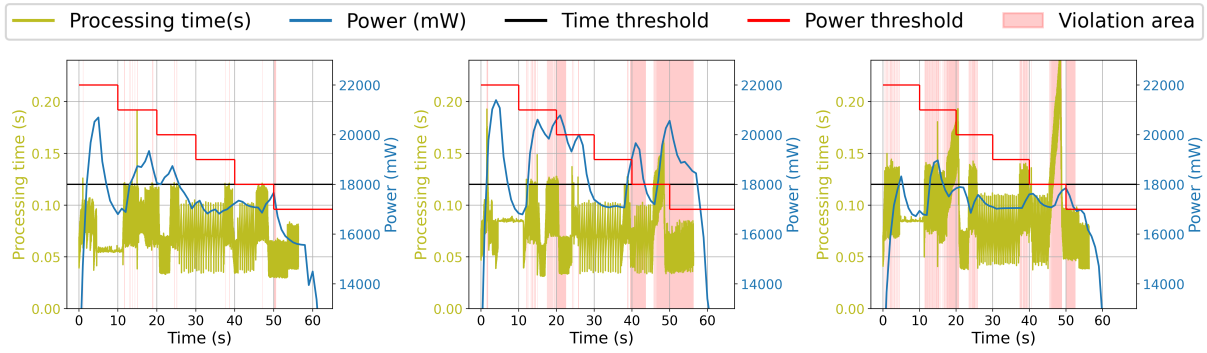


Figure 2.16. Processing delay of each frame by their execution batch numbers when **SLEXNet** is used with our scheduling algorithm during variable power threshold scenario



(a) SLEXNet **(b)** Early exit **(c)** Slimming

Figure 2.17. Processing time and power consumption values of different techniques during changing FPS and power threshold scenarios. The black line (—) indicates 0.12s processing time deadline and the red line (—) shows variable power consumption thresholds

Simultaneously Varying Power threshold and Data rate

In the previous sections, we show the response of SLEXNet under constant power threshold and changing data rate. SLEXNet and the scheduling algorithm can work with varying power threshold as well. In this section, we define a varying power threshold scenario on top of the previous randomly changing FPS scenario. We change power threshold from 22000 mW to 17000 mW by decreasing it 1000 mW every 10 seconds.

We share and discuss the detailed results and plots for 0.12 s processing delay threshold. The detailed processing delay plot is shown for SLEXNet in Fig 2.16. The combined processing

delay and power consumption plots are given in Fig. 2.17. In these figures, the black line is the time threshold and the variable red line is the power consumption threshold. As in the previous sections, SLEXNet achieves to stay under the processing delay threshold while incoming data rate and power threshold are varying throughout the experiment. On the other hand, we see that both early exit and slimming techniques violate the processing delay threshold in many occasions. Moreover, SLEXNet perfectly adapts to changing power threshold and achieves to stay under the requirement almost all time. However, early exit technique cannot satisfy the lower power thresholds as the model itself is not capable to perform for those levels of power threshold. Slimming performs well in terms of satisfying the power threshold, but its performance in terms of satisfying the processing delay is worse than early exit technique. The summary of the results with different processing delay thresholds is given in Table 2.3. By looking at the summary of the results and the plots, we can see that SLEXNet gets the advantage of each technique and in the end performs better than both early exit and slimming. Early exit technique fails the power threshold around 30% for all processing delay thresholds. Even though slimming's power fail rate is changing between 3% to 6.2%, its time fail rate is getting as high as 37%. On the other hand, SLEXNet achieves the minimal fail rate for both time and power. We can also see these results visually by examining the red background coloring in Fig. 2.17. As explained in the previous section, the red background coloring is done whenever a threshold (time or power) is violated.

We can also examine the SLEXNet's response to the additional varying power threshold by comparing Fig. 2.14 and Fig. 2.16. In both of these experiments, the same random data rate scenario is used. However, the power threshold is kept constant at 18000 mW in the first one and the power threshold is varied in the second one. For example, we can look at the last section in both of these plots. In Fig. 2.14, Runtime Scheduling algorithm selects the option {batch size 4, slimming factor 1.00x, early exit point 3}. This option is picked because it is estimated to satisfy the 0.12 s processing delay threshold and, 18000 mW power threshold. However, in Fig. 2.16, a different option is picked for the same last region. In that region, the processing delay is the

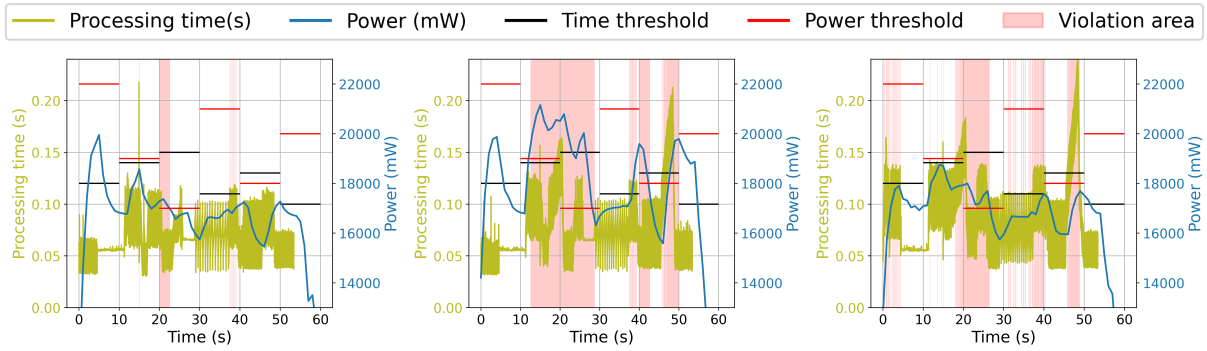
Table 2.3. Summary of changing power threshold results under various processing delay thresholds using on-time accuracy (OTA), time fail rate (TFR) and power fail rate (PFR)

Processing delay threshold(s)	Technique	OTA(%)	TFR(%)	PFR(%)
0.10	SLEXNet	**85.1**	**6.3**	**1.5**
	Early Exit	76.7	17.4	31.2
	Slimming	58.3	37	6.2
0.12	SLEXNet	**90.5**	**0.4**	**1.6**
	Early Exit	87.3	5.8	29.2
	Slimming	72.1	22	4.5
0.15	SLEXNet	**92.2**	**0.1**	**1.5**
	Early Exit	89.8	3	32.8
	Slimming	86.2	7	3

same as 0.12 s, but the power threshold is changed to 17000 mW. So the difference between the two experiments is the power threshold. Runtime Scheduling algorithm selects {batch size 4, slimming factor 0.25x, early exit point 5} for the lower power threshold case. So it decreased the slimming factor to satisfy the power threshold. As we found out earlier, the slimming is much more efficient to reduce the power consumption compared to early exit technique. We can also see this in our fail logs, where each SLEXNet option that is not scheduled is logged with its fail reason. In the variable power scenario, the SLEXNet option {batch size 4, slimming factor 1.00x, early exit point 3} is estimated to consume 17697 mW, which is higher than the 17000 mW power threshold at that section. Therefore, it is not picked for execution at that region where it is okay to use it with 18000 mW power threshold as in Fig. 2.14. When the slimming factor is decreased to 0.25x from 1.00x, early exit is increased to 5 from 3. Because the reduction in slimming factor caused a decrease in both power consumption and processing delay. So there was more room in processing delay to select a heavier model.

Simultaneously Varying Processing Delay Threshold, Power Threshold and Data rate

In this section, we vary the processing delay threshold on top of the experiment settings discussed in the previous section. The processing delay and power thresholds are randomly changed every 10 seconds. We use the random FPS scenario that is used in previous sections.



(a) SLEXNet **(b)** Early exit **(c)** Slimming

Figure 2.18. Processing time and power consumption values of different techniques during changing FPS, processing time and power thresholds scenarios.

Table 2.4. Summary of changing both time and power thresholds using on-time accuracy (OTA), time fail rate (TFR) and power fail rate (PFR)

Technique	OTA(%)	TFR(%)	PFR(%)
SLEXNet	**90.6**	**0.4**	**4.9**
Early Exit	81.0	8.2	33.9
Slimming	72.6	14.1	11.7

The time and power consumption values of SLEXNet, early exit and slimming methods are shared in Fig. 2.18. SLEXNet shows superior performance over early exit and slimming in this experiment as well. It minimizes the violation by using a correct SLEXNet option for different processing delay and power thresholds. The On-time accuracy, time fail rate and power fail rate values are also shared in Table 2.4 where SLEXNet achieves the highest on-time accuracy while achieving the smallest fail rates in terms of time and power.

2.5.4 Additional Analyses

Flexible Batch Size Analysis

In our experiments, we limit the runtime scheduling algorithm to select a batch size from a predefined constrained set. However, our algorithm is capable of selecting any batch size by evaluating the existing conditions and requirements. This can be done simply by changing the batch size set at line 4 of Algorithm 2. The batch size set can be changed to [1 to *max_batch_size*]

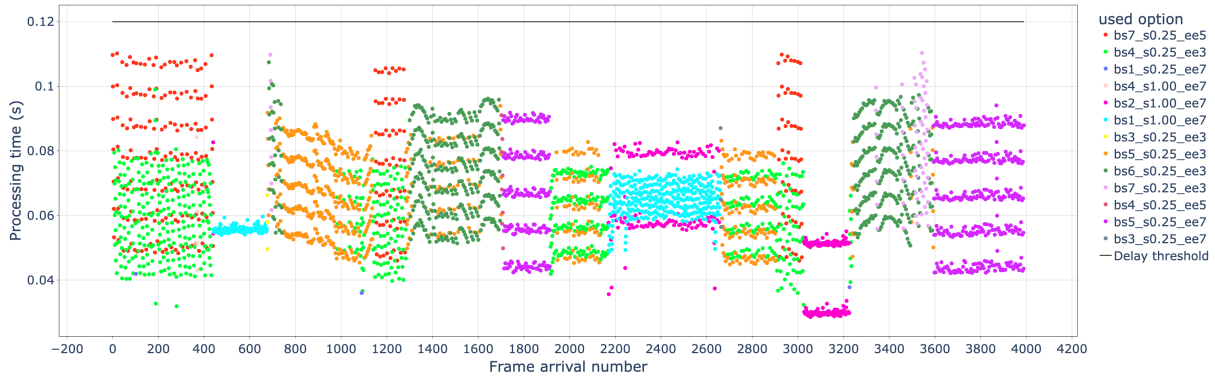


Figure 2.19. The processing time of each frame by their frame arrival number when **SLEXNet** with flexible batch sizes is used

Table 2.5. Summary of random data rate results with flexible batch size using on-time accuracy (OTA), time fail rate (TFR) and power fail rate (PFR)

Processing delay and power thresholds	Technique	OTA(%)	TFR(%)	PFR(%)
0.12s - 18000mW	SLEXNet	**90.5**	**0**	**0**
	Early Exit	89	3.9	55
	Slimming	76.8	17.2	6.6

where *max_batch_size* is the maximum batch size allowed. Even though this number can be any large integer number, it is unlikely that our runtime scheduling algorithm picks a large number for a batch size due to large $fillUpTime$ (line 7 at Algorithm 2) would violate $delayThreshold$ (line 10 at Algorithm 2). In this section, we set the available batch sizes from 1 to 8 and run the random FPS experiment that we used in Section 2.5.3.

We share the processing time results in Fig. 2.19 when processing delay and power thresholds are set to 0.12s and 18000mW, respectively. The runtime scheduling algorithm makes use of the flexible batch sizes and picks every batch size up until 7 at some point throughout the experiment. Flexible batch size actually slightly improves the performance of SLEXNet, while it deteriorates hurts the performance of the individual early exit and slimming techniques as shown in Table 2.5.

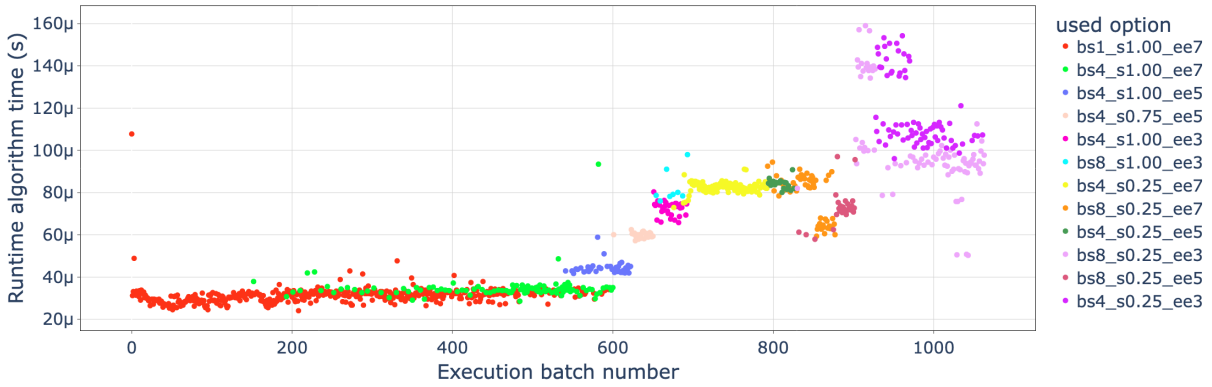
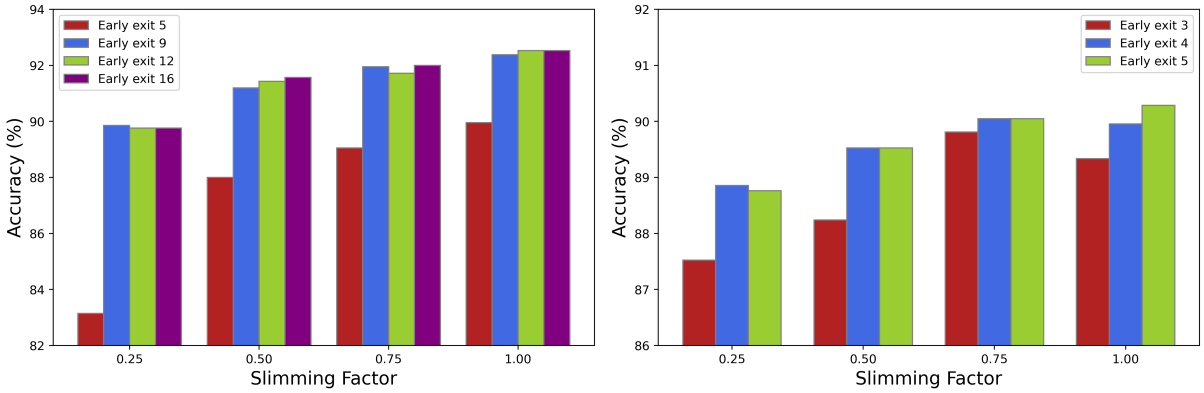


Figure 2.20. The running time of Runtime scheduling algorithm (Algorithm 2)

Runtime Scheduling Algorithm Time Analysis

The runtime scheduling algorithm (shown in Algorithm 2) needs to be fast and efficient since it is run for every execution batch. Therefore, it is important to analyze the overhead introduced by the runtime scheduling algorithm. We track its running time based on the experiment shown in Fig. 2.10, since in this experiment the FPS is increased regularly and it makes analyzing the runtime scheduling algorithm's running time clearer. The execution time of the runtime scheduling algorithm for every batch is shown in Fig. 2.20. It takes less time for more accurate subnetworks which have higher slimming coefficient and bigger early exit points. It also takes less time for smaller batch sizes. These numbers are aligned with Algorithm 2. Because the runtime scheduling algorithm iterates the SLEXNet options starting with the highest accuracy subnetwork and smallest batch size. For example, bs1_s1.00_ee7 takes the least time to be found by the algorithm, as it is the first option in the loops. Similarly, bs8_s0.25_ee3 takes the most time to be found by the algorithm, as it is the last option in the loops. In overall, the execution times are ranging between 30us and 160us. These numbers are very small and clearly negligible in our total processing delays, which are ranging between 0.1s to 0.2s. Therefore, we can conclude that the execution time of runtime scheduling algorithm is insignificant.



(a) SLEX MobileNetv2 **(b)** SLEX ResNet50v2

Figure 2.21. Accuracies of SLEX MobileNetv2 and ResNet50v2 branches

Scalability and Portability Analysis

Since our prototype SLEXNet is implemented on EfficientNetB0 backbone architecture, it is important to show its scalability and portability across models. To this aim, we implement SLEXNet on two more architectures to show scalability and portability of slimmable early exit models. We use MobileNetv2 [70] and ResNet50v2 [32] architectures as backbone in the new implementations.

MobileNetv2 consists of 16 inverted residual bottleneck blocks. We define the early exit points after 5th, 9th, 12th and 16th blocks. So this model has 4 early exit points, differing from EfficientNetB0 based one, which has 3 early exit points. We use the same 4 slimming coefficients which are 0.25, 0.50, 0.75 and 1.00. ResNetv2 consists of 5 stacked residual blocks where we define the early exits after the 3rd, 4th and 5th residual block. Here, we again use the same slimming coefficients. We train the new SLEXNets on the AIDER dataset. The accuracies of SLEXNet branches of these architectures can be seen in Fig. 2.21. This figure shows that slimmable early exit technique can be applied to different models with different architectures.

We also implement the runtime algorithm for these models. We use the same random FPS scenario which is used in section 2.5.3. We use the processing delay threshold 0.12s and 18000mW. The results of processing time by frame arrival number for SLEX MobileNetv2 and

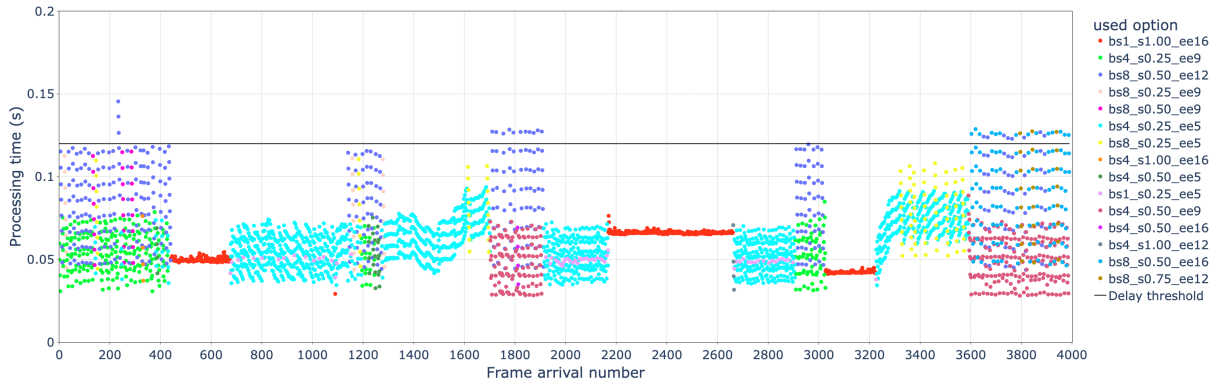


Figure 2.22. Processing delay of each frame by their frame arrival numbers when **SLEX MobileNetv2** is used with our scheduling algorithm during random FPS scenario

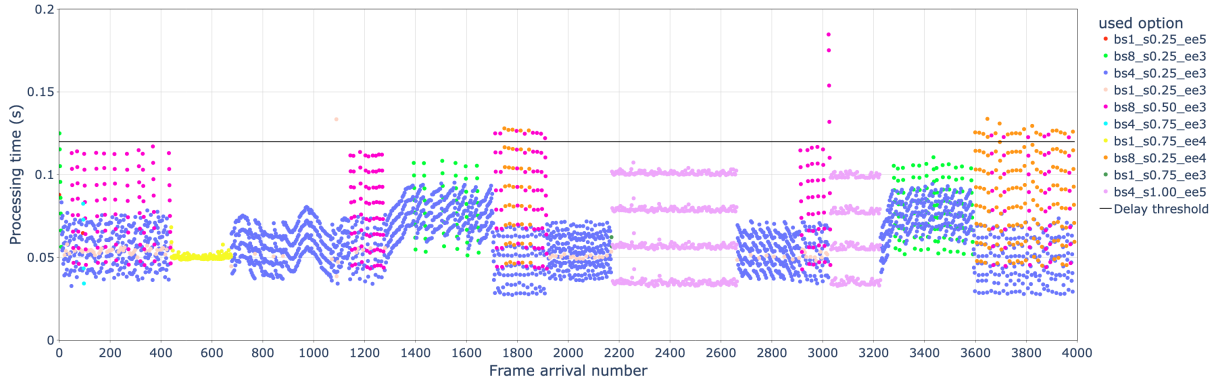
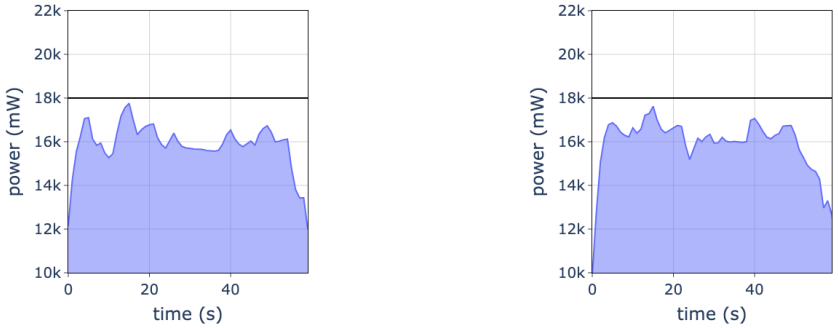


Figure 2.23. Processing delay of each frame by their frame arrival numbers when **SLEX ResNet50v2** is used with our scheduling algorithm during random FPS scenario



(a) SLEX MobileNetv2 **(b)** SLEX ResNet50v2

Figure 2.24. Power consumption of SLEX MobileNetv2 and SLEX ResNet50v2 during random FPS scenario

SLEX ResNet50v2 can be seen in Fig. 2.22 and Fig. 2.23, respectively. We also measure Time fail rates (TFR) of SLEX MobileNetv2 and ResNet50v2 as 1.0% and 1.1%, respectively. The power consumption values for these experiments can be seen in Fig. 2.24. It can be noticed that the power fail rates (PFR) are 0% for both architectures. These results show that slimmable early exit technique is not architecture dependent and can scale across different models.

2.6 Conclusion

In this paper, we proposed a dynamic neural network architecture – SLEXNet, and a Runtime Scheduling algorithm which enables utilizing SLEXNet with its full capacity during dynamic runtime environments. SLEXNet combines dynamic depth and width to adapt to varying time and power conditions better than the architectures that individually use each of these techniques. Runtime Scheduling algorithm estimates the inference time and power consumption of SLEXNet options accurately in different conditions. Then, it selects the most suitable SLEXNet option using these estimations. SLEXNet is implemented on Nvidia Jetson Orin and the experiments are conducted using an aerial drone image data. The dataset includes aerial images that require emergency response such as fire, flood and traffic incidents. We did a wide range of experiments by varying incoming FPS rates, processing delay thresholds, and power thresholds. We showed that SLEXNet outperforms early exit and slimming techniques in these experiments.

Chapter 2, in full, is a reprint of the material as it appears in ACM Transactions on Embedded Computing Systems 2024, Basar Kutukcu, Sabur Baidya, Sujit Dey. The dissertation author was the primary investigator and author of this paper.

Chapter 3

Fast and Scalable Design Space Exploration for Deep Learning on Embedded Systems

3.1 Introduction

Deep learning algorithms continue to grow their impacts on engineering, with substantial success in various fields including computer vision [10], natural language processing [99], speech recognition [63]. For real-world implementation of these algorithms on various mobile systems, embedded devices are essential. However, running the computation heavy algorithms such as deep learning inference on embedded devices has significant challenges [98]. Since embedded devices have small form factor and often need to be mobile, they have limited resources such as computation capacity and power supply. These resource constraints can significantly impact the application performance, e.g., latency, power consumption, application accuracy, memory consumption running etc. running on those embedded systems. Essentially, these performance metrics always form a tradeoff which is shaped by hardware knobs of the system and software knobs of the application. The combination of all these knobs can create an extremely large search space which makes it infeasible to exhaustively search and identify the Pareto frontier of these metrics. For efficient operation, using any set of configuration that is not on the Pareto frontier is undesirable, since it makes the systems perform less than its potential in one or more metrics.

Therefore, it is very important for the algorithms to identify the points that are on or close to the Pareto frontier within feasible search budget.

While the complexity of the computing systems has been rapidly increasing, more experts and their time are required to find close to optimal configurations. The design solutions found by experts could be good enough in some problems, However, it requires valuable engineering time. Moreover, the same engineering effort is required for every new problem even if design space is slightly changed. The engineering effort is limited by the expert's working hours, human introduced bugs, and a particular expert's capability. On the other hand, automation of the design space exploration does not have these drawbacks. It can work all the time; it is not limited by the working hours. Once developed as stable, no new bugs can be introduced, and it is not limited by a particular expert's capability. Therefore, we can justify the development of automation tools in design space exploration. However, an automation tool can be driven even by a random search which would result in relatively bad design solutions. Therefore, improving the accuracy of the search algorithms that drive the design space exploration is necessary. Therefore, we can say that it is important and necessary to develop design space exploration tools that are automatic and accurate in terms of finding close to optimal design solutions. Our proposed search algorithm is an effort to develop an automated, fast, and accurate design space exploration tool.

In search algorithms, the size of the total search space is an important factor for the algorithm design. Some search spaces are small enough for the algorithms to sample about $5\% - 30\%$ of the all search space [27, 104]. Scalability might not be an important design requirement for these algorithms. However, in this paper, our considered problem setting generates enormously large search spaces which we need to explore to solve the desired Pareto frontier, and thus, requires a scalable approach. In other words, approaches that require more resources with increasing number of iterations are not scalable and therefore bound to find suboptimal results in extremely large search spaces. To reinforce this premise, we explain how previous methods fail to scale for solving extremely large search spaces, and herein, propose a fast, accurate and scalable search algorithm that can efficiently find points close to the Pareto

frontier.

We experiment with a wide range of software applications including image classification, object detection, and large language models. We also use different hardware boards including Nvidia Jetson TX2, Nvidia Jetson Xavier and Nvidia Jetson Orin. We show that our algorithm outperforms the previous search algorithms both in terms of search time and search performance.

The main contributions of the paper are as follows:

- We introduce a novel region based search algorithm, called DivCon that can learn different structures in different regions in the same search space.
- We propose a search algorithm that is scalable, and therefore, can search extremely large search spaces without having time bottlenecks that the existing methods suffer from while performing the search.
- We address the complex search problems in real-world embedded systems, e.g., design space exploration by simultaneously considering the hardware and software parameters. We solve the problem of finding the optimal parameters for the Pareto frontier of performance, and demonstrate the advantage of our algorithm in solving such large-scale search problems.

The remainder of the paper is organized as follows. The related work is explained in Section 3.2. Then, the background for the problem, the shortcomings of the previous methods and our motivation is explained in Section 3.3. This is followed by the detailed explanation of DivCon in Section 3.4. Then, we show the experiments and results of comparing our method with other state-of-the-art methods in Section 3.5. Lastly, we conclude the paper in Section 3.6.

3.2 Related work

Bayesian optimization [25] has been a powerful tool to optimize problems with expensive objective functions. There are many works built on top of it. In [27], a model selection

algorithm is designed for the surrogate model in Bayesian optimization to solve FPGA synthesis problems. In [2], Bayesian optimization with a custom acquisition function and expert knowledge initialization is used to search microarchitecture parameters for RISC-V CPU. In [96], a new acquisition function is proposed to enable asynchronous batch evaluation during Bayesian optimization for solving analog circuit synthesis. In [74], search space is pruned within the Bayesian optimization for solving the FPGA design problems. All these works try to solve hardware design problems which are different from our problem. Also, they don't use the region based search as we do.

Excepted hypervolume improvement is a common choice of acquisition functions for Bayesian optimization methods that try to find Pareto front [12, 87]. Some works [15, 16] improved hypervolume calculation to speed up Bayesian optimization iterations. Our work differs from these, as our algorithm does not use hypervolume in the sampling process.

In [104, 105], an active learning methodology is used to sample points close to the Pareto front while discarding points that are not likely to be on the Pareto front. These works are different from our work as they are not scalable.

Many evolutionary algorithms are proposed to solve multi objective optimization problems [17, 60, 95]. In [1], a genetic algorithm is parallelized to accelerate its convergence for searching FPGA problems. In [45], the problem of mapping different neural network operations to different computation units is solved using an evolutionary algorithm. This work also focuses on searching hardware configurations to satisfy user requirements. Evolutionary algorithms use different sampling methods from our approach.

[71] also identified scalability issue of Gaussian Processes in Bayesian optimization and proposed to use deep neural networks instead of Gaussian Processes, allowing a higher degree of parallelization. Another work [20] that focuses on scalable Bayesian Optimization proposes to use local regions that move considering the best solution so far. This work's use of regions is different from ours. Moreover, this work focuses on the single optimal solution but not the Pareto frontier.

In [100], the input space is partitioned based on dominance number of each point, which is the number of points they Pareto-dominate in the output space. After partitioning, the sampling is done from the regions that are closer to the Pareto front. This work is different from ours as we use the regions in the output space, and we have a unique sampling methodology. [81] also use partitioning in the search but focuses on single optimal solutions but not Pareto frontier.

Some methods try to learn Pareto front instead of trying to find points on the pareto front [50, 51, 68]. In [51], the work of [95] is generalized to use models and learn the Pareto frontier.

Scalarization is also a common method in multi-objective optimization [42, 61]. Multiple objectives are combined with a secularization and optimized as if it is a single objective problem. We use each objective separately in the output space, hence scalarization is not used in our work.

3.3 Problem Formulation, Background and Motivation

3.3.1 Problem Formulation

In context of our considered problem, we want to search the Pareto frontier through all possible combinations of the hardware configurations of the embedded system. We can then define the component of the search space as follows:

Configuration Variables: Let c_i be a configuration variable that can take n_i different values, such that

$$c_i \in \{1, 2, ..., n_i\} \tag{3.1}$$

and let there are m different configuration variables in a system such that

$$i \in \{1, 2, ..., m\} \tag{3.2}$$

Configurations: Let **x** be the configuration, which is the collection of all configuration variables.

$\mathbf{x}$ can be written as

$$\mathbf{x} = (c_1, c_2, ..., c_m)$$

Configuration Search Space: Let $\mathbf{X}$ be the search space, a set of vectors, which contains configurations (vectors). Since there are m different configuration variables and each can have n_i different values, the total size of the configuration search space can be calculated as

$$s = \prod_{i=1}^{m} n_i$$

Therefore, $\mathbf{X}$ can be defined as

$$\mathbf{X} = \{\mathbf{x_1}, \mathbf{x_2}, ..., \mathbf{x_s}\}$$

This space is also called input space throughout the paper.

Metrics: We can test a given configuration on a system and measure the output metrics. Let $\mathbf{g}(\mathbf{x})$ be the function of configuration to output transformation. If there are p different metrics to measure in the system, there is a transformation function for each of them such that

$$\mathbf{g}(\mathbf{x}) = (g_1(\mathbf{x}), g_2(\mathbf{x}), ..., g_p(\mathbf{x}))$$

The space created by the metrics is also called output space throughout the paper.

Pareto Optimality: In the search space $\mathbf{X}$, only a subset of $\mathbf{X}$ is useful in terms of the best performance tradeoff, and all the other configurations in $\mathbf{X}$ are inferior to the configurations in this subset. The configurations in this set are called Pareto optimal. A configuration is considered as Pareto optimal if it has at least one metric that is optimal given constraints are satisfied in other metrics. If a configuration is Pareto optimal, then it is the best configuration for at least some requirements in the performance metric space. Therefore, for fast and accurate search, it is important to identify the Pareto optimal subset of $\mathbf{X}$. Our goal is to efficiently identify a subset

of $\mathbf{X}$ that is as close as possible to the Pareto optimal subset of $\mathbf{X}$.

We can give examples from one of our search spaces to explain the theoretical definitions given in this section better. For example, there are 4 hardware parameters that can have different values in one of our search spaces. These hardware parameters are # of CPU cores, CPU frequency, GPU frequency and EMC frequency. In our notation, each of these are considered as a "configuration variable". For example, GPU frequency is a configuration variable and can have 14 different values ranging from 114 MHz to 1.4 GHz. A tuple of values of configuration variables is called a "configuration". For example, # of CPU cores is 4, CPU frequency is the 2 (the number indicates index 2 in the list of actual values of CPU frequencies, ranging from 115 MHz to 2.3 GHz); similarly, GPU frequency is 5 and EMC frequency is 3. Hence, this tuple of (4,2,5,3) is a configuration. The collection of all configurations is called "configuration search space" or "input space". When we test a configuration on an actual system, we measure outcomes such as latency and power which are the "metrics".

3.3.2 Background and Motivation

Searching the best configurations for embedded systems that execute deep learning algorithms imposes multiple challenges. These challenges put certain optimization methods practically realizable, whereas make some other methods infeasible. The first challenge is that there is no analytical form of the function $\mathbf{g}(\mathbf{x})$ that we are trying to optimize. This prevents us from using many optimization algorithms that rely on derivatives of the objective function. The second challenge is that evaluating a point in the search space takes relatively long time, because there is no analytical form of the objective function and we can only evaluate an input configuration by actually running it on a real computing board. Now, running with one configuration on actual hardware can take up to a minute depending on the specific configuration. This limits the realization of these traditional optimization algorithms that relies on empirical measurements, as the extremely large search space with large measurement time can explode the optimization latency.

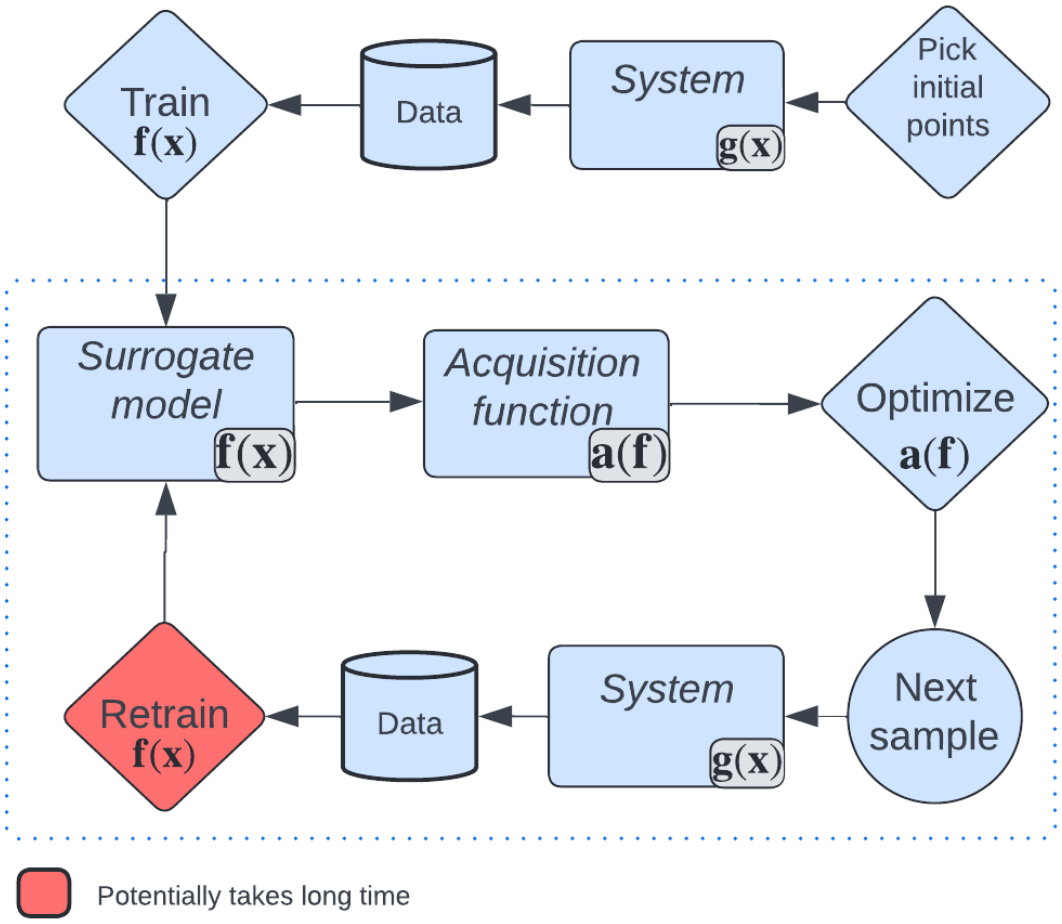
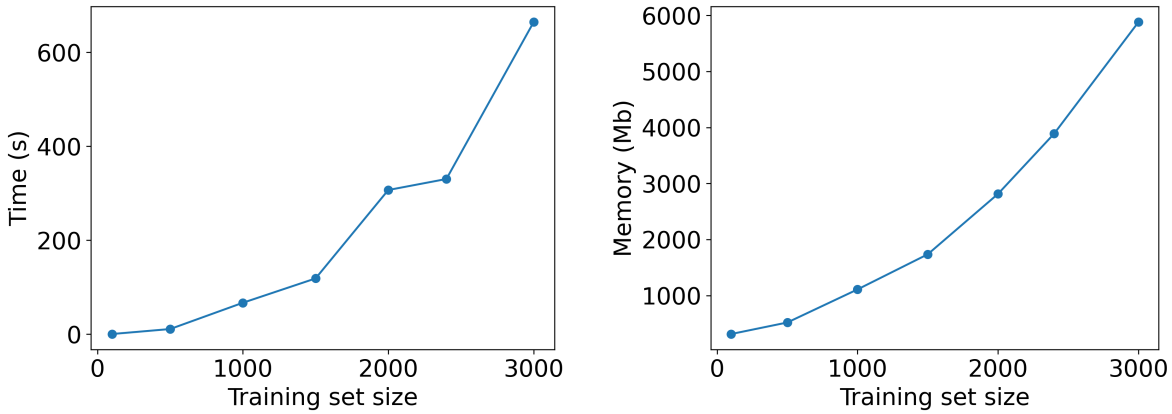


Figure 3.1. Bayesian Optimization summary

These challenges naturally point to certain optimization algorithms, e.g., Bayesian Optimization and Evolutionary algorithms which are suitable in tackling similar problems. However, they have their own drawbacks. In this section, we categorize these algorithms into 3 main methods. We explain how they work and describe their drawbacks in solving problems that we are focusing on in this paper.

Bayesian Optimization

Due to the aforementioned challenges of optimizing the function $\mathbf{g}(\mathbf{x})$ that finds the best configuration of the system, Bayesian optimization uses a *surrogate model* to model $\mathbf{g}(\mathbf{x})$. Let this



(a) Time

(b) Memory

Figure 3.2. The effect of training set size for training Gaussian process

surrogate model be $\mathbf{f}(\mathbf{x})$. While $\mathbf{f}(\mathbf{x})$ cannot model $\mathbf{g}(\mathbf{x})$ perfectly, it has a known analytical form, and therefore it can be evaluated quickly and optimized. The goal of the Bayesian optimization is to improve the accuracy of $\mathbf{f}(\mathbf{x})$ and find configurations that yield close to optimal results. It still needs to use $\mathbf{g}(\mathbf{x})$ that essentially increases the number of training points and hence, improves the accuracy of $\mathbf{f}(\mathbf{x})$, and it also checks for the optimality of the results. In the end, an iterative optimization framework is defined where $\mathbf{f}(\mathbf{x})$ is used to select the next best data point, $\mathbf{g}(\mathbf{x})$ is used to evaluate this data point, and then the evaluated results are used to improve the accuracy of $\mathbf{f}(\mathbf{x})$. An *acquisition function*, $\mathbf{a}(\mathbf{f})$ is used to select the next best data point from $\mathbf{f}(\mathbf{x})$. There are many different acquisition functions [84]. This iteration based optimization is illustrated in Figure 3.1.

The surrogate model is initialized and trained with some initial configurations. Then, the selected acquisition function is created based on the surrogate model. The acquisition function is optimized to decide on the next sample to test on the system. The next sample is then executed, and the results together with previous configurations and results, are used to retrain the surrogate model. However, this retraining process, shown as red in Figure 3.1, is not very scalable [20, 71]. Especially, when this optimization runs for larger iterations, the training set becomes larger and the retrain process takes longer time and therefore becomes infeasible. Gaussian processes [64]

are common choices for the surrogate model in Bayesian optimization due to their capability to work on different data and provide uncertainty in estimates. In Figure 3.2, we show the required resources in terms of time and memory for training Gaussian processes with increasing training set size. Both the required time and memory increase rapidly. This shows that training the surrogate model is not scalable.

[15, 16] are examples of this technique.

Pareto-focused Bayesian Optimization

Pareto-focused Bayesian optimization is actually a subcategory of the Bayesian optimization but it can have further advantages and disadvantages compared to the traditional Bayesian optimization. Similar to the traditional Bayesian optimization, it uses a surrogate model to estimate the actual system's function. The summary of the optimization loop of Pareto-focused Bayesian optimization is shown in Figure 3.3. First, the surrogate model is initialized and trained with some initial samples. Then the surrogate model is used to estimate all search space. The estimations, together with uncertainty values provided by the surrogate model, are used to select the next samples. There are different algorithms to decide how to select the next samples. The next sample is tested on the system and added to the collection of tested samples, which are used to retrain the surrogate model.

Since Pareto-focused Bayesian optimization is a subcategory of the Bayesian optimization, it inherits some disadvantages of Bayesian optimization. Retraining the surrogate model is the same process, and therefore it is still an obstacle for higher number of iterations. Furthermore, if the search space is extremely large, estimating results of every point in the search space and running an algorithm on all of these results to select a new sample are time-consuming. These 3 processes are shown in red in Figure 3.3.

[27, 104, 105] are examples of this technique.

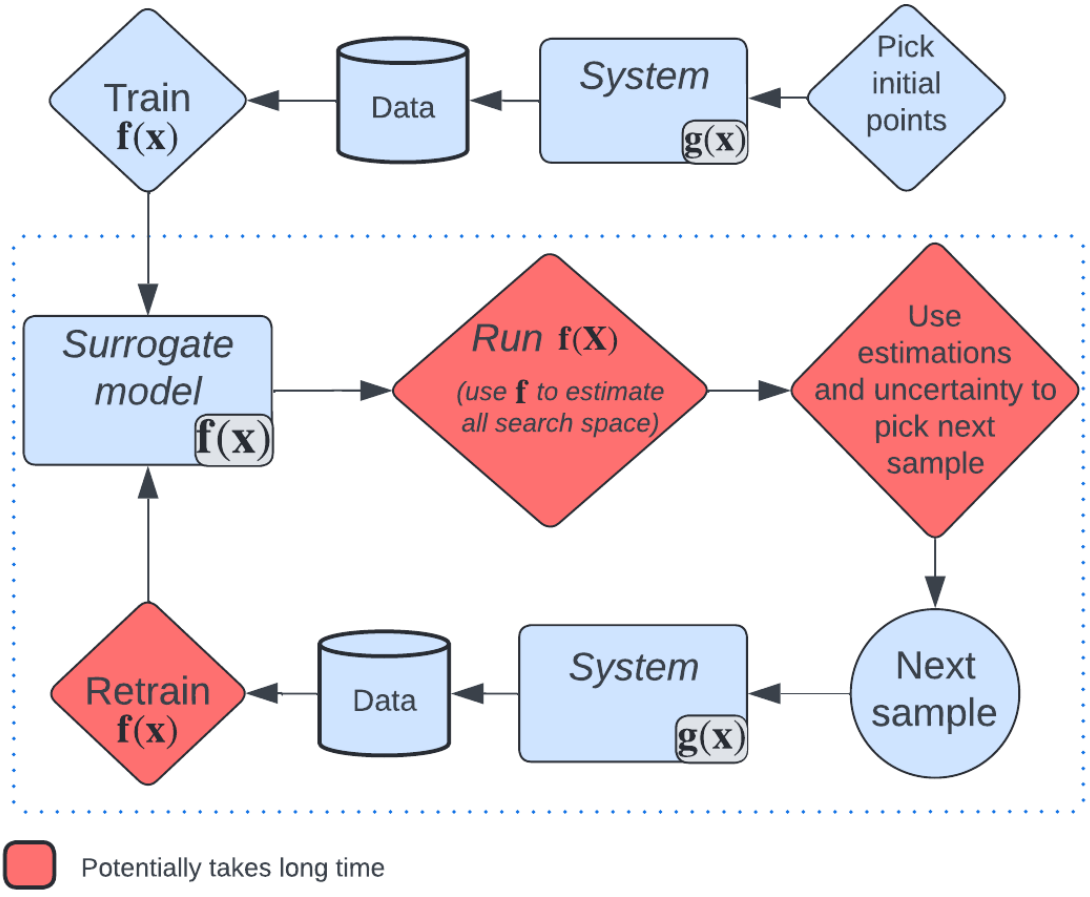


Figure 3.3. Pareto-focused Bayesian Optimization summary

Evolutionary Algorithms

Evolutionary algorithms have been very successful when applied on various search problems. There are many different versions of it, but the general framework usually follows the same principles. The algorithm maintains a population that is selected from the search space, consisting of individuals that are configurations. The individuals in the population enters a tournament where their results are compared and the winners are used to create new individuals. The creation of new individuals includes crossing over the features of the parents and mutations. The new individuals are then added to the population and the oldest individuals are removed from the population to encourage exploration.

The evolutionary algorithms are extremely fast to propose the next sample. Therefore, they don't have bottlenecks that previous methods have. However, they are heuristic based and considered sample inefficient compared to the Bayesian optimization methods [100]. As shown in [54], evolutionary algorithms may converge to suboptimal results compared to Bayesian optimization algorithms.

[17, 45, 60] are examples of this technique.

As explained in the previous sections, each group of methods has drawbacks for searching large search spaces. Therefore, to solve similar problems as ours, we need a more efficient search method that can quickly and accurately search large search spaces.

3.4 Proposed Approach: DivCon

To mitigate the shortcomings of the aforementioned approaches, we propose an efficient search algorithm, called DivCon , which is based on dividing the output space into regions, and then converging to the global results. In this section, we explain DivCon algorithm in detail and explain how it overcomes the limitations that the previous methods have, and therefore is much faster and scalable compared to them.

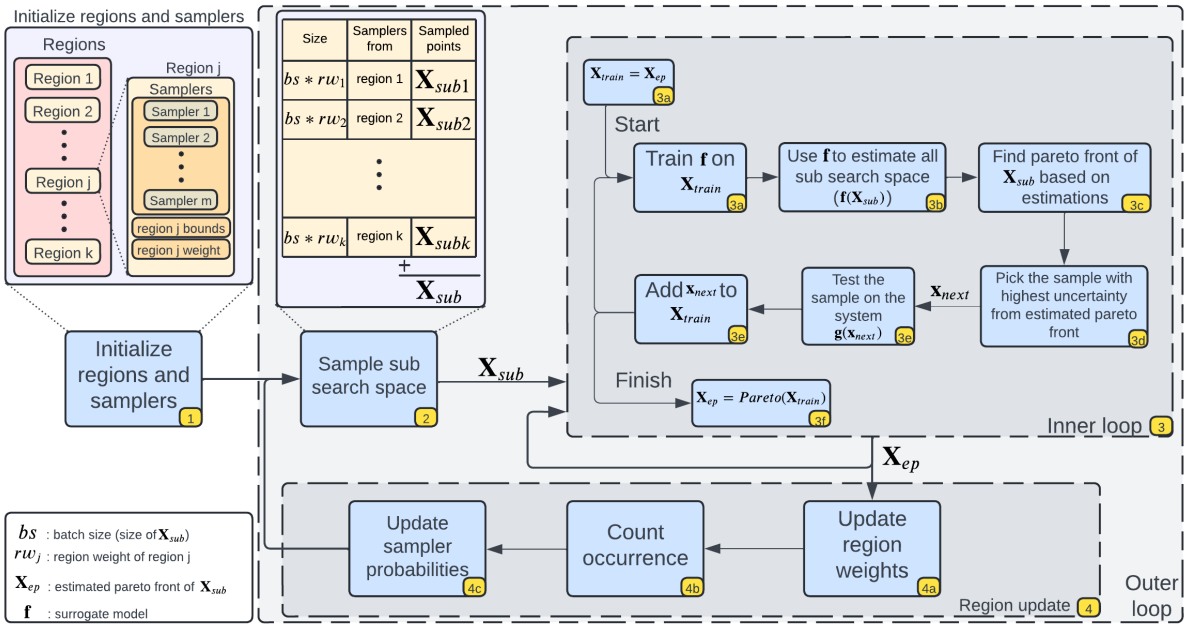


Figure 3.4. DivCon Overview

3.4.1 The Algorithm

The detailed overview of DivCon can be seen in Figure 3.4. In summary, DivCon divides the output space into regions. Each region has a set of samplers that are used to sample a sub search space. There's a sampler for each configuration variable in each region. The sampled sub-search space is searched for its own Pareto frontier in the inner loop. Then the results of the inner loop is used to update the sampler probabilities and the region weights in order to sample a better sub search space in the next outer loop iteration. The goal of the outer loop is to sample a sub-search space that is closer to the global Pareto frontier.

Each block in Figure 3.4 is labeled with yellow boxes on the right bottom corner. We explain in each stage of our algorithm using the same labels below.

1. Initialization: In this starting phase, first, the regions are created. Each region has region bounds, a region weight, and a set of samplers. The total number of regions (k in the Figure 3.4), is inferred from the hyperparameters of the DivCon which are shown in Table 3.1. For the ease of explanation, let us consider the 2D output space of a hypothetical problem shown in Figure

Table 3.1. DivCon Hyperparameters

Hyperparameter name
Metric limits (for each metric)
Metric space divider (for each metric)
bs (Batch size)
ssb (Sub search budget)
β (region weight update strength)
α (probability update strength)
Number of outer loop iterations

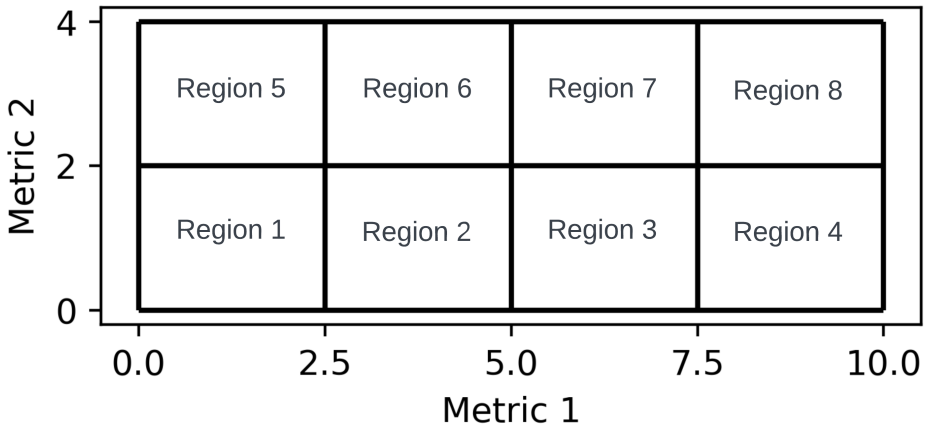


Figure 3.5. Regions illustration in a hypothetical 2D output space

3.5 that create a tradeoff space between two performance metrics. The value of metric 1 ranges between 0 and 10 and that of metric 2 ranges between 0 and 4. Also, let us assume that the range of metric 1 and 2 can be divided into 4 and 2 regions, respectively. As a result, there are 8 regions for these hyperparameters for this output space. The region bounds are also calculated from these parameters. For example, the region bounds for the region 7 in the Figure 3.5 are 5.0 to 7.5 in metric 1 space and 2 to 4 in metric 2 space. The value of the region weight *rw* is determined using the total number of regions (k). It is set to $1/k$ for all regions at the start. The sum of all region weights is always 1 such that

$$\sum_{j=1}^{k} rw_j = 1 \tag{3.3}$$

The use of regions and region weights is explained in the latter part of this section. The last component of the region is the set of samplers. A set of samplers includes a sampler for each configuration variable. Since there are m configuration variables as shown in Equation 3.2, there are exactly m samplers in the set of samplers as shown in Figure 3.4. Each sampler is used to sample a configuration variable, and the outputs of all samplers are collected to create a configuration. The samplers' probability distribution is initialized as a discrete uniform distribution where the ith sampler's probability mass function (PMF) is $1/n_i$ following the Equation 3.1. These probability distributions are updated in later stages of the algorithm to sample the points closer to the Pareto frontier.

2. Sampling: The sampling is a probabilistic process. For example, if a sampler has a available values set such as {1,2,3,4}, the sum of sampling probabilities for the values in this set is 1. For example, the probabilities can be {0.1, 0.2, 0.5, 0.2}. This means 50% of the time, this sampler samples the value 3.

The sampling process for one region happens in the following way. Each sampler samples a value for its corresponding configuration variable. So, Sampler 1 samples a value from the configuration variable 1 values set. Similarly, Sampler 2 samples a value from the configuration variable 2 values set. When all samplers sample a value for their corresponding configuration variable, the sampled points are combined, and it becomes a configuration. The number of configurations that are going to be sampled by a set of samplers of a region is determined by $bs * rw$ where bs is the batch size and rw is the region weight. So, the region with a higher region weight samples more configuration than a region with lower region weight. Since the sum of region weights is always 1 (Equation 3.3), the sum of the all sampled points is always equal to bs which is a hyperparameter as shown in Table 3.1. After each region samples configurations in the amount of their corresponding size, the configurations are collected in a sub-search space, which is shown as $\mathbf{X}_{sub}$ in Figure 3.4.

3. Inner loop: This is where DivCon estimates the Pareto front of the sub-search space. DivCon works in iterations, where one configuration is proposed to test on the system at every cycle. The

number of iterations is determined by a hyperparameter which is called *ssb* (sub search budget) as shown in Table 3.1.

3a. Training the surrogate model: DivCon uses Gaussian Process for the surrogate model. The surrogate model is trained on the training set in this stage. When the inner loop starts its iterations, the training set is the estimated Pareto frontier from the previous iteration of the outer loop. For the very first iteration of the outer loop, then the training set is chosen as a random set of configurations. The training set gets bigger at every inner loop iteration, so training of the surrogate model becomes more accurate at every inner loop cycle.

3b. Estimating sub search space with the surrogate model: In this stage, the surrogate model is used to estimate the results of all sub search space. It estimates the metrics together with the uncertainty values.

3c. Calculating the Pareto front of the estimations: In this stage, the Pareto front of the $\mathbf{X}_{sub}$ is calculated based on the estimations provided by the previous stage. The points found in this stage are not the true Pareto front of the $\mathbf{X}_{sub}$, since they are calculated based on the estimations of the metrics. The estimated Pareto front gets closer to the real Pareto front when we have a more accurate surrogate model.

3d. Picking the new sample: DivCon picks a new sample to test on the actual system at every iteration of the inner loop. DivCon picks the configuration that has the highest uncertainty among the points that are on the estimated Pareto front.

3e. Testing the new sample: The new sample is tested on the system and ground truth metrics are collected for the new sample. Then, the new sample and its ground truth metrics are added to the training set.

3f. Ending the inner loop: Once the number of iterations of the inner loop reaches *ssb*, the inner loop is terminated. As the last operation, we calculate the actual Pareto frontier of $\mathbf{X}_{train}$. Since every point in $\mathbf{X}_{train}$ is actually tested on the system, all the metrics are ground truth values. Therefore, $\mathbf{X}_{ep}$ is the actual Pareto front of $\mathbf{X}_{train}$. However, it is still the estimated Pareto front of $\mathbf{X}_{sub}$ since we do not test every configuration in the sub search space.

4. Region update: In this stage, the regions are updated so that the sampling stage at the next outer loop iteration can sample a sub search space closer to the global Pareto front.

4a. Updating the region weights: In this stage, DivCon updates the region weights using $\mathbf{X}_{ep}$. Each configuration falls in one of the regions, depending on the results of the configuration and region bounds. For example, if a configuration has the value of 8 and 1 in metric 1 and 2, respectively, then this configuration belongs to the region 4 for the hypothetical problem shown in Figure 3.5. Once DivCon puts every configuration to the regions they belong to, it counts the number of configurations in each region. Let cc_j be the number of counted configurations in region j. Then we divide all of these numbers by the sum of them to normalize, such that

$$normcc_j = \frac{cc_j}{\sum_{i=1}^{k} cc_i} \; \forall j \in 1,2,...,k \tag{3.4}$$

This results in all $normcc_j$ being between 0 and 1. Also, their sum is 1 as

$$\sum_{j=1}^{k} normcc_j = 1 \tag{3.5}$$

Then, we update the region weights using the following formula

$$rw_j = rw_j + \beta * (normcc_j - rw_j) \; \forall j \in 1,2,...,k \tag{3.6}$$

In Equation 3.6, the idea is that if a region has more Pareto optimal points in it, we should increase its region weight so that its samplers are used more in the sampling stage in the next iteration. Similarly, if there are not many Pareto optimal points in a region, its region weight should be decreased. β is called region weight update strength parameter. It is a hyperparameter that determines how aggressive the updates are. It can have any number between 0 and 1. If it is 0, no updates are made. If it is 1, the region weights are updated to the *normcc* values so they completely reflect the configurations in the $\mathbf{X}_{ep}$ in this iteration.

4b. Counting occurrences: In this stage, DivCon counts the occurrences of values for

each configuration variable in the estimated Pareto front $\mathbf{X}_{ep}$ for each region. $\mathbf{X}_{ep}$ is already divided into regions in the previous stage. For the ease of explanation, let us consider a configuration variable that can have the values in the set {1,2,3,4}. DivCon checks every configuration from $\mathbf{X}_{ep}$ in a region and counts the occurrences of the values {1,2,3,4}. For example, let's say there are 10 configurations put into region 4 in Figure 3.5. So if five of them has the value 1, one of them has the value 2, four of them has the value 3, and none of them has the value 4 for this configuration variable. The counts for this configuration variable for the region 4 will be {5,1,4,0}. Since there are m different configuration variables (Equation 3.2) and k different regions in the system, this process will be done for $m * k$ times. In other words, we will have the counts for each configuration variable in each region.

4c. Updating the sampler probabilities: There is a sampler for each configuration in each region, as shown in Figure 3.4. We use the counts found in the previous stage to update the sampler probabilities. Updating the sampler probabilities follow a logic that is similar to the updating the region weight. Let $\mathbf{prob}_{i,j}$ be the set of probabilities of the sampler i in region j. This sampler is used to sample configuration variable i for region j. Let $\mathbf{counts}_{i,j}$ be the count of occurrences for the configuration variable i in region j that are found in the previous stage. Both $\mathbf{prob}_{i,j}$ and $\mathbf{counts}_{i,j}$ are vectors with the length n_i from Equation 3.1. n_i is the number of different values that ith configuration variable can get. We start the probability updating process by dividing $\mathbf{counts}_{i,j}$ by the sum of the counts as

$$\mathbf{normcounts}_{i,j} = \frac{\mathbf{counts}_{i,j}}{sum(\mathbf{counts}_{i,j})} \quad \begin{matrix} \forall i \in 1,2,...,m, \\ \forall j \in 1,2,...,k \end{matrix} \tag{3.7}$$

All elements of $\mathbf{normcounts}_{i,j}$ are between 0 and 1. Their sum also equals to 1. Then, we use $\mathbf{normcounts}_{i,j}$ to update samplers' probabilities as following:

$$\mathbf{prob}_{i,j} = \mathbf{prob}_{i,j} + \alpha * (\mathbf{normcounts}_{i,j} - \mathbf{prob}_{i,j})$$
$$\forall i \in 1,2,...,m, \tag{3.8}$$
$$\forall j \in 1,2,...,k$$

In Equation 3.8, the aim is that if configurations with certain values of a configuration variable dominates the other configurations on the Pareto front, we should increase the probabilities for sampling these values for that configuration variable. α, similar to β, is a hyperparameter as shown in Table 3.1. It is called probability update strength parameter. It can have values between 0 and 1. If it is 0, no updates are made to probabilities. If it is 1, the probabilities are updated to the probability distribution created by the occurrence counts.

Equation 3.8 updates the probabilities of the samplers in a way that samplers tend to sample configuration variables and hence configurations that are close to the estimated Pareto front. If the Pareto front is estimated accurately, it is close to the global Pareto front and therefore the samplers essentially tend to sample configurations close to the global Pareto front.

3.4.2 Advantages over other methods

Unlike previously described methods in Figure 3.1 and Figure 3.3, DivCon does not take more time for larger iterations. The common latency bottleneck for approaches in Figure 3.1 and Figure 3.3 is due to the time-consuming retraining of the surrogate model. Hence, if the training set becomes too large, the training of the surrogate model takes a long time. Although DivCon has the same process of retraining the surrogate model at 3a (see Figure 3.4) and $\mathbf{X}_{train}$ gets bigger at every iteration of the inner loop, however, once the inner loop reaches *ssb* iterations and terminates, most of the configurations in $\mathbf{X}_{train}$ are shaved off in 3f since they are not Pareto optimal. In the next cycle of the outer loop, once DivCon enters the inner loop, the surrogate model starts training with $\mathbf{X}_{ep}$ which does not have many configurations. So, our surrogate model only learns the features of Pareto optimal configurations. In summary, unlike previous methods, the training set does not accumulate training points at every iteration and therefore,

retraining the surrogate model is always efficient and fast.

Table 3.2. The details of the used search spaces

Search space	Configuration variables			Total search space size		Metrics
	Names	Downsized # of values (range)	Full # of values (range)	Downsized Range	Full range	
ic_ss	# of CPU cores - Cluster 1	4 (1-4)	4 (1-4)	119,808	119,808	Latency Power
	# of CPU cores - Cluster 2	2 (1-2)	2 (1-2)			
	Cluster 1 CPU frequency	12 (345MHz-2GHz)	12 (345MHz-2GHz)			
	Cluster 2 CPU frequency	12 (345MHz-2GHz)	12 (345MHz-2GHz)			
	GPU frequency	13 (114MHz-1.3GHz)	13 (114MHz-1.3GHz)			
	EMC frequency	8 (40MHz-1.9GHz)	8 (40MHz-1.9GHz)			
od_ss	# of CPU cores	8 (1-8)	8 (1-8)	6,912	16,240	
	CPU Frequency	24 (115MHz-2.3GHz)	29 (115MHz-2.3GHz)			
	GPU frequency	9 (114MHz-1.4GHz)	14 (114MHz-1.4GHz)			
	EMC frequency	4 (800MHz-2.1GHz)	5 (204MHz-2.1GHz)			
llm_ss	# of CPU cores - Cluster 1	2 (1-2)	4 (1-4)	20,000	68,679,424	
	# of CPU cores - Cluster 2	2 (1-2)	4 (1-4)			
	# of CPU cores - Cluster 3	2 (1-2)	4 (1-4)			
	Cluster 1 CPU frequency	5 (422MHz-2.2GHz)	29 (115MHz-2.2GHz)			
	Cluster 2 CPU frequency	5 (422MHz-2.2GHz)	29 (115MHz-2.2GHz)			
	Cluster 3 CPU frequency	5 (422MHz-2.2GHz)	29 (115MHz-2.2GHz)			
	GPU frequency	5 (306MHz-1.3GHz)	11 (306MHz-1.3GHz)			
	EMC frequency	4 (204MHz-3.2GHz)	4 (204MHz-3.2GHz)			
multimodel	LLM model architecture	4 (1-4)	4 (1-4)	20,736	1,098,870,784	Latency Power Accuracy Memory
	LLM model precision	4 (1-4)	4 (1-4)			
	# of CPU cores - Cluster 1	2 (1-2)	4 (1-4)			
	# of CPU cores - Cluster 2	2 (1-2)	4 (1-4)			
	# of CPU cores - Cluster 3	2 (1-2)	4 (1-4)			
	Cluster 1 CPU frequency	3 (422MHz-2.2GHz)	29 (115MHz-2.2GHz)			
	Cluster 2 CPU frequency	3 (422MHz-2.2GHz)	29 (115MHz-2.2GHz)			
	Cluster 3 CPU frequency	3 (422MHz-2.2GHz)	29 (115MHz-2.2GHz)			
	GPU frequency	3 (306MHz-1.3GHz)	11 (306MHz-1.3GHz)			
	EMC frequency	2 (2.1GHz-3.2GHz)	4 (204MHz-3.2GHz)			

Other bottlenecks for Pareto-focused Bayesian optimization (Figure 3.3) include estimating all search space and using the estimations to select a new sample. Although DivCon also has similar operations, it estimates only $\mathbf{X}_{sub}$ which has a constant size of *bs* (batch size) at every iteration. *bs* is only a fraction of the all search space. Therefore, both the estimations and using estimations to pick a new sample does not take long time.

3.5 Experiments

3.5.1 Search Spaces

We have tested DivCon extensively on a wide range of search spaces. We have 4 search spaces that are combinations of different software applications and different hardware systems. The details of each search space can be seen in Table 3.2. Some of the search spaces are downsized so that we can find the actual Pareto front to compare performances of the algorithms in terms of how close they are estimating the actual Pareto front.

Image classification application: The first search space, ic_ss, is for image classification model EfficientNetB0 [77] running on Nvidia Jetson TX2. This search space has 6 configuration variables. When we combine all possible values of all configuration variables, there are 119,808 configurations in the search space. The output space include 2 metrics, namely latency and power.

Object detection application: The following search space, od_ss, is for object detection model YoloV8s [39] running on Nvidia Jetson Xavier. This search space has 4 configuration variables. However, we have downsized the ranges of the configuration variables. For example, there are 29 available CPU frequencies on the system, but we limited the search to use only 24 of them. The downsized search space size is 6,912. The full and downsized ranges of the configuration variables of each search space are shown in Table 3.2.

Large language model application: This search space, llm_ss, is for large language model Pythia 70m [6] running on Nvidia Jetson Orin. This search space has 8 configuration variables

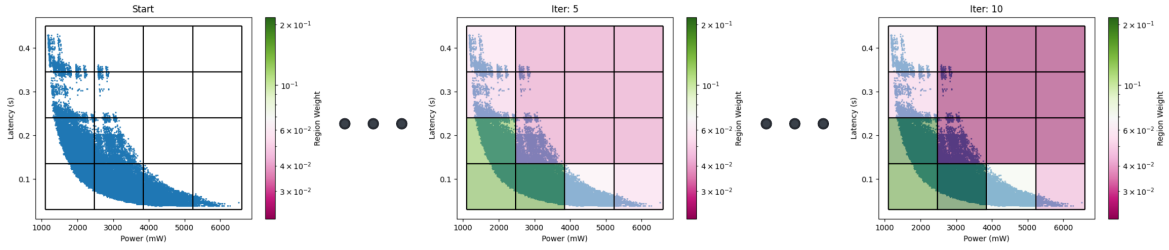


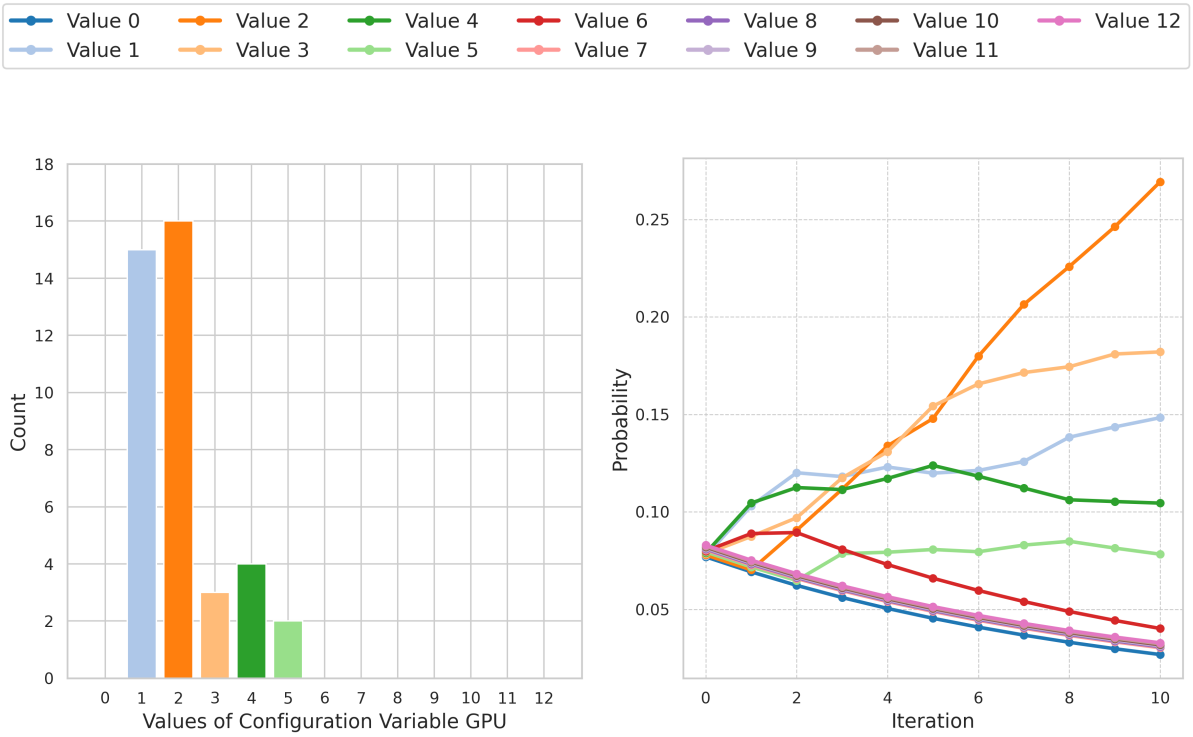
Figure 3.6. Change of region weights by outer loop iterations for ic_ss and 16 regions

yielding 68,679,424 configurations. We have downsized the configuration variables of this search space to have 20,000 configurations.

Multimodel large language model application: The last search space, multimodel_ss, is for searching LLM models, model precisions and Nvidia Orin hardware parameters. There are 4 Pythia models [6], namely Pythia 70m, Pythia 160m, Pythia 410m, and Pythia 1b. There are also 4 model precisions for these models including fp32, fp16, int8 and fp4. The quantization is done by using [19]. There are 10 configuration variables yielding 1,098,870,784 configurations. The downsized search space has 20,736 configurations. The output space is 4D where the metrics are latency, power, accuracy, and memory consumption. The accuracies of LLM models are benchmarked using multiple datasets available at [26].

3.5.2 Visualization of DivCon Working Mechanism

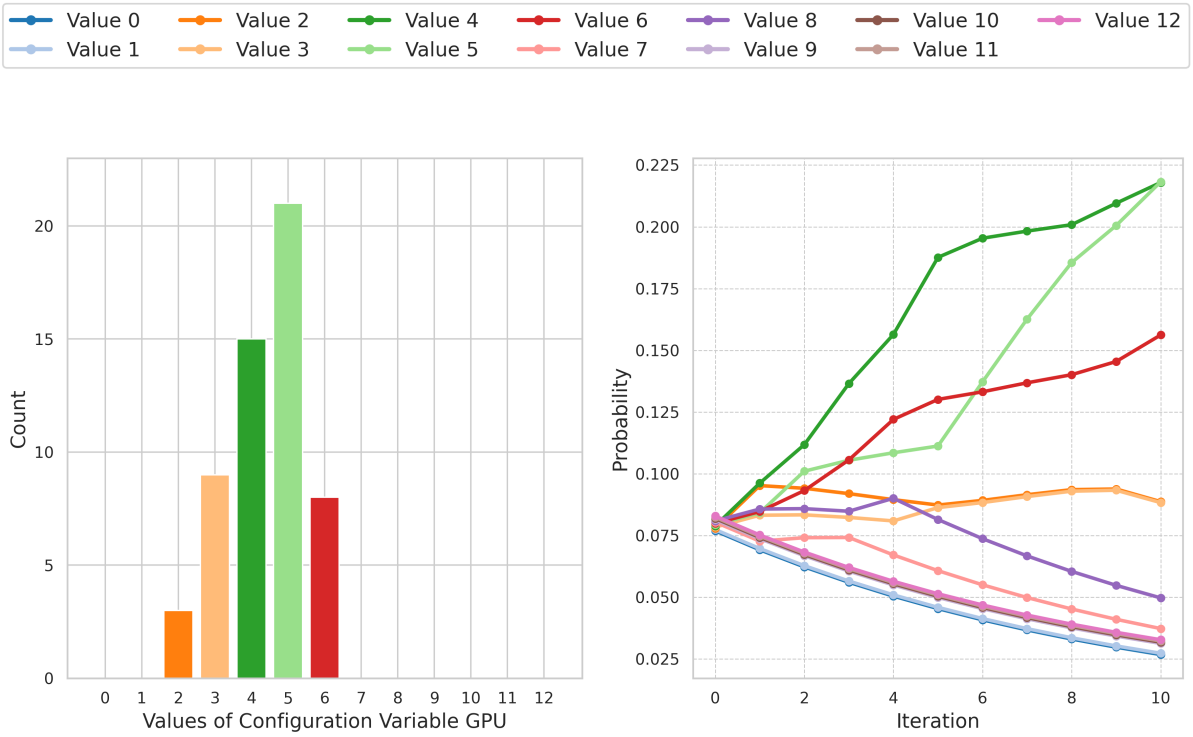
In this section, we provide visualizations and explanations to give insights about how DivCon works. The visualizations are based on the search of DivCon on ic_ss. Power and latency values of all configurations of ic_ss are shown in Figure 3.6. 16 regions are initialized, where the search space is divided equally by regions. The change of region weights are shown in Figure 3.6. The regions are used to find out which parts of the output space contain Pareto optimal points. The weights of the regions that contain Pareto optimal points increase, and the weights of those that do not contain Pareto optimal points decrease in every iteration. Since the sum of the region weights needs to be 1 all the time, the weights of the regions with more Pareto optimal points increase more compared to regions with less Pareto optimal points. We can see



(a) Counts for actual Pareto front

(b) Probability change

Figure 3.7. The values of GPU frequency configuration variable in ic_ss show up in the actual Pareto front in the region [1100mW-2475mW] - [0.135s-0.24s] and DivCon sampler's probability change for the GPU configuration variable in the same region in 10 iterations



(a) Counts for actual Pareto front

(b) Probability change

Figure 3.8. The values of GPU frequency configuration variable in ic_ss show up in the actual Pareto front in the region [2475mW-3850mW] - [0.03s-0.135s] and DivCon sampler's probability change for the GPU configuration variable in the same region in 10 iterations

this process in Figure 3.6. The weights of all regions are the same at the start. At iteration 5, we see that the regions that do not contain any Pareto optimal points have decreased weights. On the other hand, the regions at the bottom left have increased weights. The weights of the regions that contain the other parts of the Pareto front are smaller than the bottom left part but higher than the non-Pareto regions. This is because most of the Pareto optimal points lie in the regions at the bottom left. At Iteration 10, we see that these weights get even further from each other. As a result, DivCon promotes exploration and learning in earlier iterations and exploitation of the information in the later iterations, since these weights are used to decide how many points are sampled by each region's samplers.

We can examine the change of the sampler probabilities following the regions shown in

3.6. Let us consider the configuration variable GPU frequency. This configuration variable has 13 values that it can take, as shown in Table 3.2. The values of GPU frequency that show up in on the Pareto front in the region with region bounds [1100mW-2475mW] - [0.135s-0.24s] are shown in 3.7a. Since this region is a low power one, we see lower GPU frequency values more commonly on the Pareto front. The most common values are 2 and 1 which are followed by 4,3,5. The change of probabilities of the sampler for GPU frequency in this region is shown in 3.7b. The sampler is initialized with uniform probability. At each iteration, it learns to increase the probabilities of the values 1 to 5 and decrease the probability of sampling the other values. As a result, it samples configurations with GPU frequency value 1 to 5 more. Similarly, if we look at the region with region bounds [2475mW-3850mW] - [0.03s-0.135s], we see different values for the same configuration variable as shown in 3.8. Since this region is a higher power one, the higher values for the GPU frequency are observed on the Pareto front. The values 5 and 4 are the most common ones and 6,3,2 follow them as shown in Figure 3.8a. DivCon learns to increase the probabilities of sampling these points in later iterations as shown in Figure 3.8b.

The regions and samplers work together to learn the structure of the search space. The regions learn the output space while samplers learn the input space. Combining the two connects the information of the input and output spaces.

3.5.3 Comparison with other methods

In this section, we compare our algorithm with a quasi-random (Sobol sequence) method, EHVI [16] and ParEGO [42]. These algorithms are implemented using Ax (version 0.4.0) which is a software package built on BoTorch [3].

The execution time of search algorithms collected on a server with 11th Gen Intel i7-11700 with 16GB of RAM and Nvidia GeForce GTX 1660 Ti.

We use two metrics to compare the algorithms' performances. The first metric of the comparison is the hypervolume. Hypervolume is the enclosed area between a set of Pareto optimal points and a reference point. An example of calculating the hypervolume for a 2D space

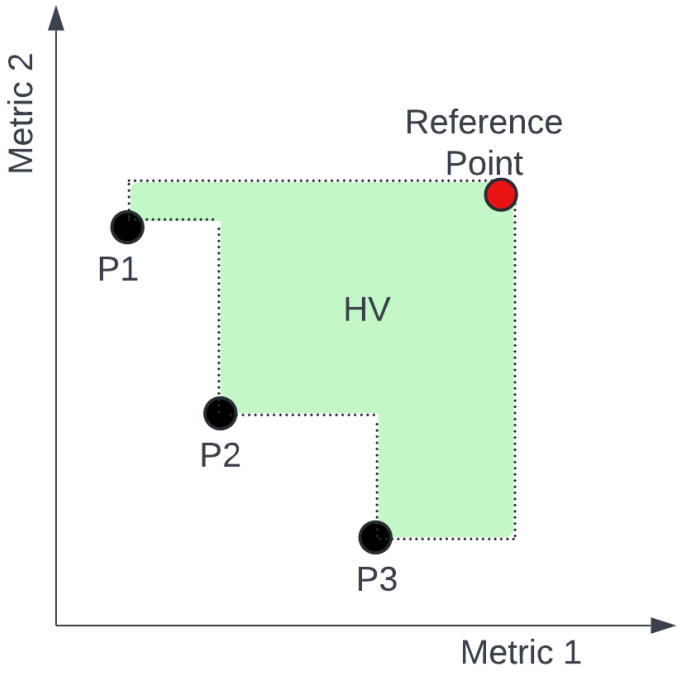


Figure 3.9. Hypervolume calculation illustration in a hypothetical 2D output space

is shown in Figure 3.9. If the global Pareto front is known, hypervolume log difference is defined as

$$HV\ log\ difference = log_{10}(HV(GPF) - HV(EPF))$$

where GPF is global Pareto front and EPF is estimated Pareto front. HV log difference is commonly used in multi-objective optimization community to compare algorithms. The lower the value of HV log difference, the better the algorithm is for estimating the Pareto frontier. We also use it for comparisons in this section.

The other metric of the comparison is the AUC (Area under the curve) for hypervolume log difference during search. Each algorithm proposes a configuration to test on the real system at every iteration. Therefore, the hypervolume is either improved or stayed at the same value at every iteration. AUC is a good way to understand the convergence speed of the algorithms. Since it adds up all the hypervolume log differences throughout the iterations, the lower is better. If an

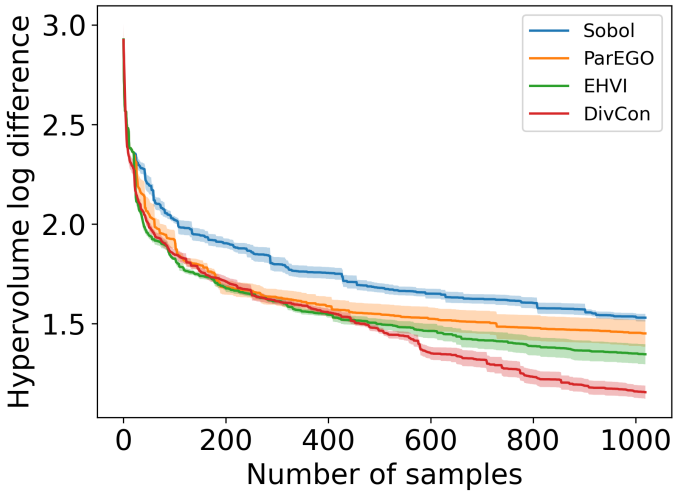
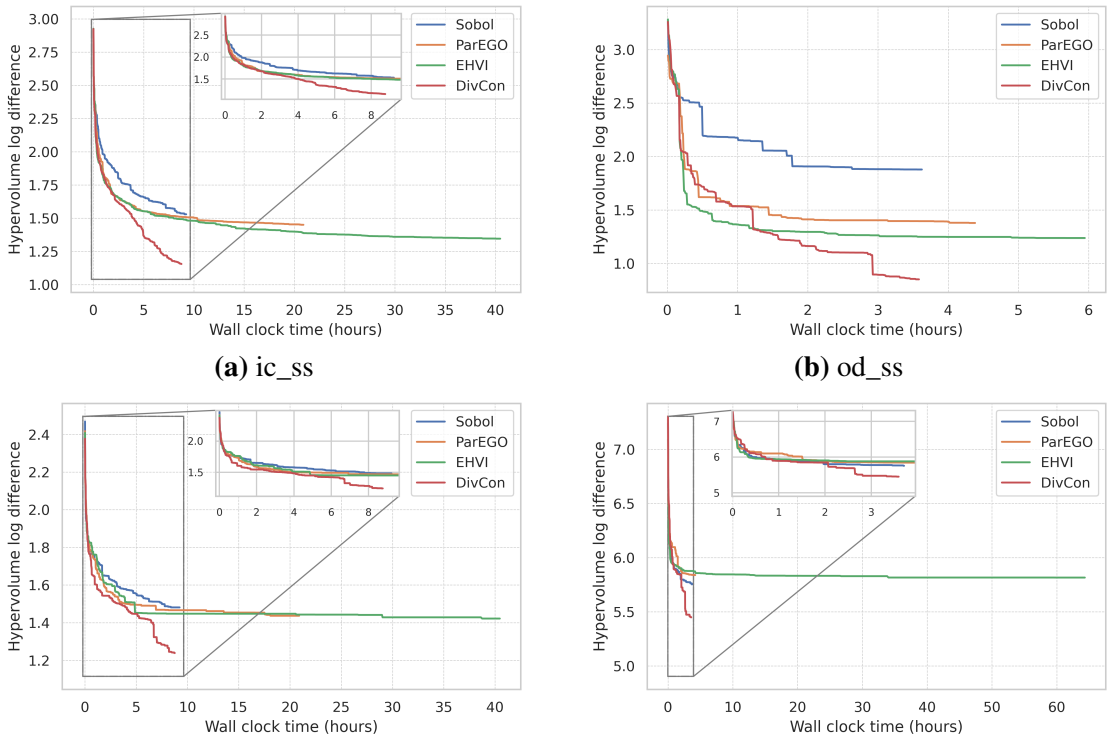


Figure 3.10. The Hypervolume log difference results of search algorithms on search space ic_ss. Each of the methods is run for 5 times. Solid line is the mean of the 5 runs. Shaded area is 1 standard deviation.

Table 3.3. Comparison of the methods

Search space	Method	Best HV log diff		AUC	
		Val	Ratio	Val	Ratio
ic_ss	Sobol	1.54	x1.00	1755.20	x1.00
	ParEGO	1.45	x0.95	1621.80	x0.92
	EHVI	1.35	x0.88	1560.05	x0.89
	DivCon	**1.16**	**x0.76**	**1502.74**	**x0.86**
od_ss	Sobol	1.88	x1.00	834.42	x1.00
	ParEGO	1.38	x0.73	623.19	x0.75
	EHVI	1.24	x0.66	560.93	x0.67
	DivCon	**0.86**	**x0.46**	**547.62**	**x0.66**
llm_ss	Sobol	1.48	x1.00	1599.24	x1.00
	ParEGO	1.44	x0.97	1541.32	x0.96
	EHVI	1.43	x0.96	1530.20	x0.96
	DivCon	**1.24**	**x0.84**	**1476.21**	**x0.92**
multimodel_ss	Sobol	5.76	x1.00	2362.22	x1.00
	ParEGO	5.84	x1.01	2393.45	x1.01
	EHVI	5.82	x1.01	2356.15	x1.00
	DivCon	**5.46**	**x0.95**	**2335.27**	**x0.99**

algorithm quickly gets to a low value, it has a smaller AUC value. On the other hand, if another algorithm gets to the same low value but only in late iterations, it has a larger AUC value.



(a) ic_ss **(b)** od_ss

(c) llm_ss **(d)** multimodel_ss

Figure 3.11. The Hypervolume log difference by algorithm run time. The mean of 5 runs for each algorithm is shown.

We show the hypervolume log difference results of DivCon and other algorithms for search space ic_ss in Figure 3.10. In this figure, we can see that Sobol, being the baseline algorithm, achieves the worst result. EHVI is a successful algorithm in terms of achieving low hypervolume log difference and therefore beats Sobol and ParEGO, expectedly. We can clearly see that DivCon outperforms other methods and reaches a set of configurations that yield lower hypervolume log difference compared to other methods at the end of the search. AUC for DivCon is also smaller than other algorithms, since it converges quickly. The best hypervolume log difference and AUC values for each search space are shown in Table 3.3. We also show the ratio to Sobol baseline of the best hypervolume log difference and AUC. Each algorithm is allowed to sample the same number of configurations in each search for a fair comparison. A budget of 1000 samples is allowed for the search spaces ic_ss and llm_ss. A budget of 400 samples is allowed for od_ss and multimodel_ss. The reason for reduced budget for od_ss is

that it has a smaller search space as shown in Figure 3.2. The reason for reduced budget for multimodel_ss is that higher number of iterations are not feasible in terms of time for EHVI due to higher dimensions of output space. The summary of the results in Table 3.3 shows that the ranking between the algorithms stay more or less similar both in terms of best hypervolume log difference and AUC while DivCon achieves significantly better results than other algorithms in all the search spaces.

These comparisons are made by considering the number of samples each algorithm inquiries. So these comparisons do not reflect the time required by the algorithms. However, as explained earlier, the algorithm speed is the main motivation point for developing DivCon . Therefore, we also show the hypervolume log difference with wall clock time for all search spaces in Figure 3.11. Wall clock time includes the time that algorithm requires to decide its next sample and the time that is required to run the sample on the system for each iteration. In these plots, we also show the zoomed in versions, considering the time DivCon finishes sampling. It can be clearly seen that DivCon finishes the search extremely quickly compared to other methods. Depending on the search space and the number of iterations, DivCon 's search time can vary 3 to 8 hours, while other algorithms require significantly more time. Even though Bayesian optimization methods such has EHVI [16] is better than other previous methods in terms of finding a better Pareto front, they become infeasible in terms of required time when the dimension of output space increases, as shown in Figure 3.11d. DivCon can find better Pareto front in much less time compared to what other algorithms take.

DivCon outperforms other methods in hypervolume log difference and AUC considering both number of samples and algorithm run time as shown in Table 3.3 and Figure 3.11.

3.6 Conclusion and Future Work

In this paper, we proposed a fast and scalable search algorithm for Pareto optimal design space exploration for deep learning algorithms on embedded systems. Our approach

learns the structure of the data because of our proposed region based search technique, and regional samplers. It does not have the bottlenecks which the Bayesian optimization methods suffer from. Our algorithm is especially superior in problems where testing a specific sample configuration is not very expensive but still it is impossible to exhaustively search through all possible configuration combinations. For example our multimodel_ss search space has $1B$ data point. Even though, testing a point takes 30 seconds, exhaustive search still requires very large time and in more complex setup becomes practically infeasible.

It is critical for efficient running of deep learning algorithms on embedded systems, to find the Pareto frontier of performance metric by actually running those applications. Hence, our proposed method can provide an efficient and scalable solution for this kind of search problem which is highly essential for implementing those on real-world systems. The experimental results on real-world embedded systems and applications, reinforces the advantage of our proposed approach over other existing methods.

DivCon currently works with integer input space, since our target problem is the hardware and software configurations of deep learning models on embedded systems. In the future, we plan to enable continuous input space for DivCon and evaluate our algorithm with synthetic problems such as ZDT [102] and DTLZ [18] which are popular benchmarks in multi objective optimization community.

Chapter 3, in full, is a reprint of the material as it appears in IEEE Access 2024, Basar Kutukcu, Sabur Baidya, Sujit Dey. The dissertation author was the primary investigator and author of this paper.

Chapter 4

Conclusion

This thesis has presented several methodologies to address problems and bottlenecks of enabling deep learning models on resource constrained devices. The presented methodologies addressed challenges in the impact of contention in the system, adapting to the changing conditions and requirements during runtime, and exploring vast search spaces of configurations for deep learning models and resource constrained systems.

Chapter 1 has presented a model selection framework that makes a deep learning application contention agnostic. In other words, the aim of the framework is to deliver results within a known and acceptable time, independent of the contention on the system. To achieve this, the framework grades the contention in the system and prepares a set of deep learning models. Then, using the selection algorithm, it picks the appropriate model from the model set considering the contention and requirements at the time. Our framework has advantages such as rapid model switching and flexibility of choosing different range of models. Analysis based on our experiments shows that our framework can achieve to mitigate the impact of the contention by sacrificing the accuracy minimally.

Chapter 2 has presented a dynamic neural network architecture and a branch selection algorithm to adapt the changing conditions and requirements of a resource constrained mobile system. Dynamic neural networks are useful tools since they contain only one set of weights, however, can improve latency by sacrificing accuracy by changing architecture. We have

presented a neural network architecture that combines slimming and early exit techniques to create a new architecture that carries the strength of both. Our analysis shows that our hybrid architecture overperforms both slimming and early exit techniques. Moreover, we have presented a branch selection algorithm that is fast and accurate. The accuracy of the branch selection algorithm is verified with the extensive experiments showing the high satisfaction rate for both time and power requirements,

Chapter 3 has presented a design space exploration algorithm for deep learning models on resource constrained devices. This problem is proven to be difficult due to vast search space size, the challenges of testing a sample and the lack of the analytical form of the objective function. Our algorithm uses a sampling-based approach to mitigate the scalability problem of Bayesian Optimization. The extensive analysis, including multiple deep learning architectures and hardware, showed that our approach can find better Pareto frontier in much less time compared to existing state-of-the-art Bayesian Optimization based methods.

Bibliography

[1] Rizwan A. Ashraf, Francis Luna, Damian Dechev, and Ronald F. DeMara. Designing digital circuits for fpgas using parallel genetic algorithms (wip). In *Proceedings of the 2012 Symposium on Theory of Modeling and Simulation - DEVS Integrative M&S Symposium*, TMS/DEVS '12, San Diego, CA, USA, 2012. Society for Computer Simulation International.

[2] Chen Bai, Qi Sun, Jianwang Zhai, Yuzhe Ma, Bei Yu, and Martin D.F. Wong. Boomexplorer: Risc-v boom microarchitecture design space exploration framework. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pages 1–9, 2021.

[3] Maximilian Balandat, Brian Karrer, Daniel R. Jiang, Samuel Daulton, Benjamin Letham, Andrew Gordon Wilson, and Eytan Bakshy. BoTorch: A Framework for Efficient Monte-Carlo Bayesian Optimization. In *Advances in Neural Information Processing Systems 33*, 2020.

[4] Ron Banner, Yury Nahshan, and Daniel Soudry. Post training 4-bit quantization of convolutional networks for rapid-deployment. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pages 7948–7956, 2019.

[5] Ali Ehteshami Bejnordi and Ralf Krestel. Dynamic channel and layer gating in convolutional neural networks. In Ute Schmid, Franziska Klügl, and Diedrich Wolter, editors, *KI 2020: Advances in Artificial Intelligence - 43rd German Conference on AI, Bamberg, Germany, September 21-25, 2020, Proceedings*, volume 12325 of *Lecture Notes in Computer Science*, pages 33–45. Springer, 2020.

[6] Stella Biderman, Hailey Schoelkopf, Quentin Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, USVSN Sai Prashanth, Edward Raff, Aviya Skowron, Lintang Sutawika, and Oskar van der Wal. Pythia: A suite for analyzing large language models across training and scaling, 2023.

[7] Davis W. Blalock, Jose Javier Gonzalez Ortiz, Jonathan Frankle, and John V. Guttag. What is the state of neural network pruning? In Inderjit S. Dhillon, Dimitris S. Papailiopoulos,

and Vivienne Sze, editors, *Proceedings of Machine Learning and Systems 2020, MLSys 2020, Austin, TX, USA, March 2-4, 2020*. mlsys.org, 2020.

[8] Tolga Bolukbasi, Joseph Wang, Ofer Dekel, and Venkatesh Saligrama. Adaptive neural networks for efficient inference. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 527–536. PMLR, 2017.

[9] Han Cai, Chuang Gan, Tianzhe Wang, Zhekai Zhang, and Song Han. Once-for-all: Train one network and specialize it for efficient deployment. In *8th International Conference on Learning Representations, ICLR 2020, Addis Ababa, Ethiopia, April 26-30, 2020*. OpenReview.net, 2020.

[10] Junyi Chai, Hao Zeng, Anming Li, and Eric W.T. Ngai. Deep learning in computer vision: A critical review of emerging techniques and application scenarios. *Machine Learning with Applications*, 6:100134, 2021.

[11] Zhourong Chen, Yang Li, Samy Bengio, and Si Si. You look twice: Gaternet for dynamic filter selection in cnns. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, pages 9172–9180. Computer Vision Foundation / IEEE, 2019.

[12] Ivo Couckuyt, Dirk Deschrijver, and Tom Dhaene. Fast calculation of multiobjective probability of improvement and expected improvement criteria for pareto optimization. *J. Glob. Optim.*, 60(3):575–594, 2014.

[13] Matthieu Courbariaux and Yoshua Bengio. Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR*, abs/1602.02830, 2016.

[14] Ekin D. Cubuk, Barret Zoph, Dandelion Mané, Vijay Vasudevan, and Quoc V. Le. Autoaugment: Learning augmentation strategies from data. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, pages 113–123. Computer Vision Foundation / IEEE, 2019.

[15] Samuel Daulton, Maximilian Balandat, and Eytan Bakshy. Differentiable expected hypervolume improvement for parallel multi-objective bayesian optimization. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[16] Samuel Daulton, Maximilian Balandat, and Eytan Bakshy. Parallel bayesian optimization of multiple noisy objectives with expected hypervolume improvement. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman

Vaughan, editors, *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pages 2187–2200, 2021.

[17] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T. Meyarivan. A fast elitist non-dominated sorting genetic algorithm for multi-objective optimisation: NSGA-II. In Marc Schoenauer, Kalyanmoy Deb, Günter Rudolph, Xin Yao, Evelyne Lutton, Juan Julián Merelo Guervós, and Hans-Paul Schwefel, editors, *Parallel Problem Solving from Nature - PPSN VI, 6th International Conference, Paris, France, September 18-20, 2000, Proceedings*, volume 1917 of *Lecture Notes in Computer Science*, pages 849–858. Springer, 2000.

[18] Kalyanmoy Deb, Lothar Thiele, Marco Laumanns, and Eckart Zitzler. Scalable multi-objective optimization test problems. In *Proceedings of the 2002 Congress on Evolutionary Computation, CEC 2002, Honolulu, HI, USA, May 12-17, 2002*, pages 825–830. IEEE, 2002.

[19] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *Advances in Neural Information Processing Systems*, 36, 2024.

[20] David Eriksson, Michael Pearce, Jacob R Gardner, Ryan Turner, and Matthias Poloczek. *Scalable global optimization via local Bayesian optimization*. Curran Associates Inc., Red Hook, NY, USA, 2019.

[21] Amir Erfan Eshratifar and Massoud Pedram. Runtime deep model multiplexing for reduced latency and energy consumption inference. In *38th IEEE International Conference on Computer Design, ICCD 2020, Hartford, CT, USA, October 18-21, 2020*, pages 263–270. IEEE, 2020.

[22] Jamil Fayyad, Mohammad A. Jaradat, Dominique Gruyer, and Homayoun Najjaran. Deep learning sensor fusion for autonomous vehicle perception and localization: A review. *Sensors*, 20(15):4220, 2020.

[23] Boyuan Feng, Kun Wan, Shu Yang, and Yufei Ding. SECS: efficient deep stream processing via class skew dichotomy. *CoRR*, abs/1809.06691, 2018.

[24] Rostand A. K. Fezeu, Eman Ramadan, Wei Ye, Benjamin Minneci, Jack Xie, Arvind Narayanan, Ahmad Hassan, Feng Qian, Zhi-Li Zhang, Jaideep Chandrashekar, and Myungjin Lee. An in-depth measurement analysis of 5g mmwave PHY latency and its impact on end-to-end delay. In Anna Brunström, Marcel Flores, and Marco Fiore, editors, *Passive and Active Measurement - 24th International Conference, PAM 2023, Virtual Event, March 21-23, 2023, Proceedings*, volume 13882 of *Lecture Notes in Computer Science*, pages 284–312. Springer, 2023.

[25] Peter I. Frazier. A tutorial on bayesian optimization. *CoRR*, abs/1807.02811, 2018.

[26] Leo Gao, Jonathan Tow, Baber Abbasi, Stella Biderman, Sid Black, Anthony DiPofi, Charles Foster, Laurence Golding, Jeffrey Hsu, Alain Le Noac'h, Haonan Li, Kyle McDonell, Niklas Muennighoff, Chris Ociepa, Jason Phang, Laria Reynolds, Hailey Schoelkopf, Aviya Skowron, Lintang Sutawika, Eric Tang, Anish Thite, Ben Wang, Kevin Wang, and Andy Zou. A framework for few-shot language model evaluation, 12 2023.

[27] Quentin Gautier, Alric Althoff, Christopher L. Crutchfield, and Ryan Kastner. Sherlock: A multi-objective design space exploration framework. *ACM Trans. Des. Autom. Electron. Syst.*, 27(4), mar 2022.

[28] Amir Gholami, Sehoon Kim, Zhen Dong, Zhewei Yao, Michael W. Mahoney, and Kurt Keutzer. A survey of quantization methods for efficient neural network inference. *CoRR*, abs/2103.13630, 2021.

[29] Song Han, Jeff Pool, John Tran, and William J. Dally. Learning both weights and connections for efficient neural network. In Corinna Cortes, Neil D. Lawrence, Daniel D. Lee, Masashi Sugiyama, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 28: Annual Conference on Neural Information Processing Systems 2015, December 7-12, 2015, Montreal, Quebec, Canada*, pages 1135–1143, 2015.

[30] Yizeng Han, Gao Huang, Shiji Song, Le Yang, Honghui Wang, and Yulin Wang. Dynamic neural networks: A survey. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 44(11):7436–7456, 2022.

[31] Vikas Hassija, Vinay Chamola, Adhar Agrawal, Adit Goyal, Nguyen Cong Luong, Dusit Niyato, Fei Richard Yu, and Mohsen Guizani. Fast, reliable, and secure drone communication: A comprehensive survey. *IEEE Communications Surveys & Tutorials*, 23(4):2802–2832, 2021.

[32] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Identity mappings in deep residual networks. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part IV*, volume 9908 of *Lecture Notes in Computer Science*, pages 630–645. Springer, 2016.

[33] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *CoRR*, abs/1704.04861, 2017.

[34] Gao Huang, Danlu Chen, Tianhong Li, Felix Wu, Laurens van der Maaten, and Kilian Q. Weinberger. Multi-scale dense networks for resource efficient image classification. In *6th International Conference on Learning Representations, ICLR 2018, Vancouver, BC, Canada, April 30 - May 3, 2018, Conference Track Proceedings*. OpenReview.net, 2018.

[35] Gao Huang, Zhuang Liu, Laurens van der Maaten, and Kilian Q. Weinberger. Densely connected convolutional networks. In *2017 IEEE Conference on Computer Vision and*

Pattern Recognition, CVPR 2017, Honolulu, HI, USA, July 21-26, 2017, pages 2261–2269. IEEE Computer Society, 2017.

[36] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and <1mb model size. *CoRR*, abs/1602.07360, 2016.

[37] Forrest N. Iandola, Matthew W. Moskewicz, Sergey Karayev, Ross B. Girshick, Trevor Darrell, and Kurt Keutzer. Densenet: Implementing efficient convnet descriptor pyramids. *CoRR*, abs/1404.1869, 2014.

[38] Babak Shahian Jahromi, Theja Tulabandhula, and Sabri Cetin. Real-time hybrid multi-sensor fusion framework for perception in autonomous vehicles. *Sensors*, 19(20):4357, 2019.

[39] Glenn Jocher, Ayush Chaurasia, and Jing Qiu. Ultralytics YOLO, jan 2023.

[40] Alexander B. Jung, Kentaro Wada, Jon Crall, Satoshi Tanaka, Jake Graving, Christoph Reinders, Sarthak Yadav, Joy Banerjee, Gábor Vecsei, Adam Kraft, Zheng Rui, Jirka Borovec, Christian Vallentin, Semen Zhydenko, Kilian Pfeiffer, Ben Cook, Ismael Fernández, François-Michel De Rainville, Chi-Hung Weng, Abner Ayala-Acevedo, Raphael Meudec, Matias Laporte, et al. imgaug. https://github.com/aleju/imgaug, 2020. Online; accessed 01-Feb-2020.

[41] Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun, editors, *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015.

[42] J. Knowles. Parego: a hybrid algorithm with on-line landscape approximation for expensive multiobjective optimization problems. *IEEE Transactions on Evolutionary Computation*, 10(1):50–66, 2006.

[43] Basar Kutukcu, Sabur Baidya, Anand Raghunathan, and Sujit Dey. Contention-aware adaptive model selection for machine vision in embedded systems. In *3rd IEEE International Conference on Artificial Intelligence Circuits and Systems, AICAS 2021, Washington, DC, USA, June 6-9, 2021*, pages 1–4. IEEE, 2021.

[44] Basar Kutukcu, Sabur Baidya, Anand Raghunathan, and Sujit Dey. Contention grading and adaptive model selection for machine vision in embedded systems. *ACM Trans. Embed. Comput. Syst.*, 21(5):55:1–55:29, 2022.

[45] Basar Kutukcu, Sabur Baidya, Anand Raghunathan, and Sujit Dey. Evosh: Evolutionary search with shaving to enable power-latency tradeoff in deep learning computing on embedded systems. In *2023 IEEE 36th International System-on-Chip Conference (SOCC)*, pages 1–6, 2023.

[46] Christos Kyrkou and Theocharis Theocharides. Emergencynet: Efficient aerial image classification for drone-based emergency monitoring using atrous convolutional feature fusion. *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, 13:1687–1699, 2020.

[47] Changlin Li, Guangrun Wang, Bing Wang, Xiaodan Liang, Zhihui Li, and Xiaojun Chang. Dynamic slimmable network. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2021, virtual, June 19-25, 2021*, pages 8607–8617. Computer Vision Foundation / IEEE, 2021.

[48] Shun Li, Linhan Qiao, Youmin Zhang, and Jun Yan. An early forest fire detection system based on dji m300 drone and h20t camera. In *2022 International Conference on Unmanned Aircraft Systems (ICUAS)*, pages 932–937, 2022.

[49] Yue Li, Devesh K. Jha, Asok Ray, and Thomas A. Wettergren. Feature level sensor fusion for target detection in dynamic environments. In *American Control Conference, ACC 2015, Chicago, IL, USA, July 1-3, 2015*, pages 2433–2438. IEEE, 2015.

[50] Xi Lin, Zhiyuan Yang, Qingfu Zhang, and Sam Kwong. Controllable pareto multi-task learning. *CoRR*, abs/2010.06313, 2020.

[51] Xi Lin, Zhiyuan Yang, Xiaoyuan Zhang, and Qingfu Zhang. Pareto set learning for expensive multi-objective optimization. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022.

[52] Lanlan Liu and Jia Deng. Dynamic deep neural networks: Optimizing accuracy-efficiency trade-offs by selective execution. In Sheila A. McIlraith and Kilian Q. Weinberger, editors, *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, (AAAI-18), the 30th innovative Applications of Artificial Intelligence (IAAI-18), and the 8th AAAI Symposium on Educational Advances in Artificial Intelligence (EAAI-18), New Orleans, Louisiana, USA, February 2-7, 2018*, pages 3675–3682. AAAI Press, 2018.

[53] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott E. Reed, Cheng-Yang Fu, and Alexander C. Berg. SSD: single shot multibox detector. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part I*, volume 9905 of *Lecture Notes in Computer Science*, pages 21–37. Springer, 2016.

[54] Andre K.Y. Low, Eleonore Vissol-Gaudin, Yee-Fun Lim, and Kedar Hippalgaonkar. Mapping pareto fronts for efficient multi-objective materials discovery. *Journal of Materials Informatics*, 3(2), 2023.

[55] Jian-Hao Luo, Jianxin Wu, and Weiyao Lin. Thinet: A filter level pruning method for deep neural network compression. In *IEEE International Conference on Computer*

Vision, ICCV 2017, Venice, Italy, October 22-29, 2017, pages 5068–5076. IEEE Computer Society, 2017.

[56] Mason McGill and Pietro Perona. Deciding how to decide: Dynamic routing in artificial neural networks. In Doina Precup and Yee Whye Teh, editors, *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, volume 70 of *Proceedings of Machine Learning Research*, pages 2363–2372. PMLR, 2017.

[57] Eldad Meller, Alexander Finkelstein, Uri Almog, and Mark Grobman. Same, same but different: Recovering neural network quantization error through weight factorization. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 4486–4495. PMLR, 2019.

[58] Augustus Odena, Dieterich Lawson, and Christopher Olah. Changing model behavior at test-time using reinforcement learning. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Workshop Track Proceedings*. OpenReview.net, 2017.

[59] Lorenzo Palazzetti. Routing drones being aware of wind conditions: a case study. In *2021 17th International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pages 343–350, 2021.

[60] Annibale Panichella. An adaptive evolutionary algorithm based on non-euclidean geometry for many-objective optimization. In Anne Auger and Thomas Stützle, editors, *Proceedings of the Genetic and Evolutionary Computation Conference, GECCO 2019, Prague, Czech Republic, July 13-17, 2019*, pages 595–603. ACM, 2019.

[61] Biswajit Paria, Kirthevasan Kandasamy, and Barnabás Póczos. A flexible framework for multi-objective bayesian optimization using random scalarizations. In Amir Globerson and Ricardo Silva, editors, *Proceedings of the Thirty-Fifth Conference on Uncertainty in Artificial Intelligence, UAI 2019, Tel Aviv, Israel, July 22-25, 2019*, volume 115 of *Proceedings of Machine Learning Research*, pages 766–776. AUAI Press, 2019.

[62] Eunhyeok Park, Dongyoung Kim, Soobeom Kim, Yong-Deok Kim, Gunhee Kim, Sungroh Yoon, and Sungjoo Yoo. Big/little deep neural network for ultra low power inference. In Gabriela Nicolescu and Andreas Gerstlauer, editors, *2015 International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2015, Amsterdam, Netherlands, October 4-9, 2015*, pages 124–132. IEEE, 2015.

[63] Rohit Prabhavalkar, Takaaki Hori, Tara N Sainath, Ralf Schlüter, and Shinji Watanabe. End-to-end speech recognition: A survey. *arXiv preprint arXiv:2303.03329*, 2023.

[64] Carl Edward Rasmussen. Gaussian processes in machine learning. In Olivier Bousquet, Ulrike von Luxburg, and Gunnar Rätsch, editors, *Advanced Lectures on Machine Learning,*

ML Summer Schools 2003, Canberra, Australia, February 2-14, 2003, Tübingen, Germany, August 4-16, 2003, Revised Lectures, volume 3176 of *Lecture Notes in Computer Science*, pages 63–71. Springer, 2003.

[65] Mohammad Rastegari, Vicente Ordonez, Joseph Redmon, and Ali Farhadi. Xnor-net: Imagenet classification using binary convolutional neural networks. In Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling, editors, *Computer Vision - ECCV 2016 - 14th European Conference, Amsterdam, The Netherlands, October 11-14, 2016, Proceedings, Part IV*, volume 9908 of *Lecture Notes in Computer Science*, pages 525–542. Springer, 2016.

[66] Benjamin Recht, Rebecca Roelofs, Ludwig Schmidt, and Vaishaal Shankar. Do imagenet classifiers generalize to imagenet? In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 5389–5400. PMLR, 2019.

[67] Joseph Redmon and Ali Farhadi. Yolov3: An incremental improvement. *CoRR*, abs/1804.02767, 2018.

[68] Michael Ruchte and Josif Grabocka. Scalable pareto front approximation for deep multi-objective learning. In James Bailey, Pauli Miettinen, Yun Sing Koh, Dacheng Tao, and Xindong Wu, editors, *IEEE International Conference on Data Mining, ICDM 2021, Auckland, New Zealand, December 7-10, 2021*, pages 1306–1311. IEEE, 2021.

[69] Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael S. Bernstein, Alexander C. Berg, and Li Fei-Fei. Imagenet large scale visual recognition challenge. *Int. J. Comput. Vis.*, 115(3):211–252, 2015.

[70] Mark Sandler, Andrew G. Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. Mobilenetv2: Inverted residuals and linear bottlenecks. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 4510–4520. Computer Vision Foundation / IEEE Computer Society, 2018.

[71] Jasper Snoek, Oren Rippel, Kevin Swersky, Ryan Kiros, Nadathur Satish, Narayanan Sundaram, Md. Mostofa Ali Patwary, Prabhat, and Ryan P. Adams. Scalable bayesian optimization using deep neural networks, 2015.

[72] Elijah Spicer and Sabur Baidya. Performance tradeoff in dnn-based coexisting applications in resource-constrained cyber-physical systems. In *2023 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 219–221. IEEE, 2023.

[73] Nitish Srivastava, Geoffrey E. Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. Dropout: a simple way to prevent neural networks from overfitting. *J. Mach. Learn. Res.*, 15(1):1929–1958, 2014.

[74] Qi Sun, Tinghuan Chen, Siting Liu, Jin Miao, Jianli Chen, Hao Yu, and Bei Yu. Correlated multi-objective multi-fidelity optimization for hls directives design. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 46–51, 2021.

[75] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 2818–2826. IEEE Computer Society, 2016.

[76] Shyam Anil Tailor, Javier Fernández-Marqués, and Nicholas Donald Lane. Degree-quant: Quantization-aware training for graph neural networks. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021.

[77] Mingxing Tan and Quoc V. Le. Efficientnet: Rethinking model scaling for convolutional neural networks. In Kamalika Chaudhuri and Ruslan Salakhutdinov, editors, *Proceedings of the 36th International Conference on Machine Learning, ICML 2019, 9-15 June 2019, Long Beach, California, USA*, volume 97 of *Proceedings of Machine Learning Research*, pages 6105–6114. PMLR, 2019.

[78] Ben Taylor, Vicent Sanz Marco, Willy Wolff, Yehia Elkhatib, and Zheng Wang. Adaptive deep learning model selection on embedded systems. In Zheng Zhang and Christophe Dubach, editors, *Proceedings of the 19th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems, LCTES 2018, Philadelphia, PA, USA, June 19-20, 2018*, pages 31–43. ACM, 2018.

[79] 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, ICPR 2016, Cancún, Mexico, December 4-8, 2016*, pages 2464–2469. IEEE, 2016.

[80] Jun Wang, Tanner A. Bohn, and Charles X. Ling. Pelee: A real-time object detection system on mobile devices. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett, editors, *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pages 1967–1976, 2018.

[81] Linnan Wang, Rodrigo Fonseca, and Yuandong Tian. Learning search space partition for black-box optimization using monte carlo tree search. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin, editors, *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020.

[82] Xin Wang, Fisher Yu, Zi-Yi Dou, Trevor Darrell, and Joseph E. Gonzalez. Skipnet: Learning dynamic routing in convolutional networks. In Vittorio Ferrari, Martial Hebert,

Cristian Sminchisescu, and Yair Weiss, editors, *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part XIII*, volume 11217 of *Lecture Notes in Computer Science*, pages 420–436. Springer, 2018.

[83] Yue Wang, Jianghao Shen, Ting-Kuei Hu, Pengfei Xu, Tan M. Nguyen, Richard G. Baraniuk, Zhangyang Wang, and Yingyan Lin. Dual dynamic inference: Enabling more efficient, adaptive, and controllable deep inference. *IEEE J. Sel. Top. Signal Process.*, 14(4):623–633, 2020.

[84] James T. Wilson, Frank Hutter, and Marc Peter Deisenroth. Maximizing acquisition functions for bayesian optimization. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*, NIPS'18, page 9906–9917, Red Hook, NY, USA, 2018. Curran Associates Inc.

[85] Wenhan Xia, Hongxu Yin, Xiaoliang Dai, and Niraj K. Jha. Fully dynamic inference with deep neural networks. *IEEE Trans. Emerg. Top. Comput.*, 10(2):962–972, 2022.

[86] Ran Xu, Jinkyu Koo, Rakesh Kumar, Peter Bai, Subrata Mitra, Ganga Maghanath, and Saurabh Bagchi. Approxnet: Content and contention aware video analytics system for the edge. *CoRR*, abs/1909.02068, 2019.

[87] Kaifeng Yang, Michael Emmerich, André H. Deutz, and Thomas Bäck. Multi-objective bayesian global optimization using expected hypervolume improvement gradient. *Swarm Evol. Comput.*, 44:945–956, 2019.

[88] De Jong Yeong, Gustavo Adolfo Velasco-Hernández, John Barry, and Joseph Walsh. Sensor and sensor fusion technology in autonomous vehicles: A review. *Sensors*, 21(6):2140, 2021.

[89] Jin Hyeok Yoo, Yecheol Kim, Ji Song Kim, and Jun Won Choi. 3d-cvf: Generating joint camera and lidar features using cross-view spatial feature fusion for 3d object detection. In Andrea Vedaldi, Horst Bischof, Thomas Brox, and Jan-Michael Frahm, editors, *Computer Vision - ECCV 2020 - 16th European Conference, Glasgow, UK, August 23-28, 2020, Proceedings, Part XXVII*, volume 12372 of *Lecture Notes in Computer Science*, pages 720–736. Springer, 2020.

[90] Jiahui Yu and Thomas S. Huang. Network slimming by slimmable networks: Towards one-shot architecture search for channel numbers. *CoRR*, abs/1903.11728, 2019.

[91] Jiahui Yu and Thomas S. Huang. Universally slimmable networks and improved training techniques. In *2019 IEEE/CVF International Conference on Computer Vision, ICCV 2019, Seoul, Korea (South), October 27 - November 2, 2019*, pages 1803–1811. IEEE, 2019.

[92] Jiahui Yu, Linjie Yang, Ning Xu, Jianchao Yang, and Thomas S. Huang. Slimmable neural networks. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[93] Zhihang Yuan, Bingzhe Wu, Guangyu Sun, Zheng Liang, Shiwan Zhao, and Weichen Bi. S2DNAS: transforming static CNN model for dynamic inference via neural architecture search. In Andrea Vedaldi, Horst Bischof, Thomas Brox, and Jan-Michael Frahm, editors, *Computer Vision - ECCV 2020 - 16th European Conference, Glasgow, UK, August 23-28, 2020, Proceedings, Part II*, volume 12347 of *Lecture Notes in Computer Science*, pages 175–192. Springer, 2020.

[94] Jeff Zhang, Sameh Elnikety, Shuayb Zarar, Atul Gupta, and Siddharth Garg. Model-switching: Dealing with fluctuating workloads in machine-learning-as-a-service systems. In Amar Phanishayee and Ryan Stutsman, editors, *12th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2020, July 13-14, 2020*. USENIX Association, 2020.

[95] Qingfu Zhang and Hui Li. Moea/d: A multiobjective evolutionary algorithm based on decomposition. *IEEE Transactions on Evolutionary Computation*, 11(6):712–731, 2007.

[96] Shuhan Zhang, Fan Yang, Dian Zhou, and Xuan Zeng. An efficient asynchronous batch bayesian optimization approach for analog circuit synthesis. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

[97] Xiangyu Zhang, Xinyu Zhou, Mengxiao Lin, and Jian Sun. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In *2018 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018*, pages 6848–6856. Computer Vision Foundation / IEEE Computer Society, 2018.

[98] Tianming Zhao, Yucheng Xie, Yan Wang, Jerry Cheng, Xiaonan Guo, Bin Hu, and Yingying Chen. A survey of deep learning on mobile devices: Applications, optimizations, challenges, and research opportunities. *Proceedings of the IEEE*, 110(3):334–354, 2022.

[99] Wayne Xin Zhao, Kun Zhou, Junyi Li, Tianyi Tang, Xiaolei Wang, Yupeng Hou, Yingqian Min, Beichen Zhang, Junjie Zhang, Zican Dong, Yifan Du, Chen Yang, Yushuo Chen, Zhipeng Chen, Jinhao Jiang, Ruiyang Ren, Yifan Li, Xinyu Tang, Zikang Liu, Peiyu Liu, Jian-Yun Nie, and Ji-Rong Wen. A survey of large language models. *CoRR*, abs/2303.18223, 2023.

[100] Yiyang Zhao, Linnan Wang, Kevin Yang, Tianjun Zhang, Tian Guo, and Yuandong Tian. Multi-objective optimization by learning space partitions. *CoRR*, abs/2110.03173, 2021.

[101] Chenzhuo Zhu, Song Han, Huizi Mao, and William J. Dally. Trained ternary quantization. In *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017.

[102] Eckart Zitzler, Kalyanmoy Deb, and Lothar Thiele. Comparison of multiobjective evolutionary algorithms: Empirical results. *Evol. Comput.*, 8(2):173–195, 2000.

[103] Barret Zoph, Vijay Vasudevan, Jonathon Shlens, and Quoc V. Le. Learning transferable architectures for scalable image recognition. In *2018 IEEE Conference on Computer*

Vision and Pattern Recognition, CVPR 2018, Salt Lake City, UT, USA, June 18-22, 2018, pages 8697–8710. Computer Vision Foundation / IEEE Computer Society, 2018.

[104] Marcela Zuluaga, Andreas Krause, and Markus Püschel. e-pal: An active learning approach to the multi-objective optimization problem. *Journal of Machine Learning Research*, 17(104):1–32, 2016.

[105] Marcela Zuluaga, Guillaume Sergent, Andreas Krause, and Markus Püschel. Active learning for multi-objective optimization. In *Proceedings of the 30th International Conference on Machine Learning, ICML 2013, Atlanta, GA, USA, 16-21 June 2013*, volume 28 of *JMLR Workshop and Conference Proceedings*, pages 462–470. JMLR.org, 2013.