AI at the Edge: Efficient Deep Learning for Resource-Constrained Environments

By

Marina Neseem

B.S., Ain Shams University, Cairo, Egypt, 2017

M.S., Brown University, Providence, RI, 2021

Thesis

Submitted in partial fulfillment of the requirements for the Degree of Doctor of Philosophy in the School of Engineering at Brown University

PROVIDENCE, RHODE ISLAND

May 2024

This dissertation by Marina Neseem is accepted in its present form by the School of Engineering as satisfying the dissertation requirement for the degree of Doctor of Philosophy.

Date ______________ ____
Sherief Reda, Advisor

Recommended to the Graduate Council

Date ______________ ____
Sherief Reda, Advisor

Date ______________ ____
Jacob Rosenstein, Reader

Date ______________ ____
James Tompkin, Reader

Approved by the Graduate Council

Date ______________ ____
Thomas A. Lewis, Dean of the Graduate School

Vitae

Marina Neseem was born and raised in Cairo, Egypt. She received her B.Sc. in Computer and Systems Engineering from Ain Shams University, Cairo, Egypt in 2017. She received her M.Sc. in Electrical and Computer Engineering from Brown University in 2021 during her studies in the Ph.D. program. Her main areas of research include edge intelligence, efficient machine learning, and hardware-software co-design.

marina_neseem@brown.edu
https://marinaneseem.me/

Brown University, RI, USA

Selected Publications

1. **Neseem, M.**, McCullough, C., Hsin, R., Leichner, C., Li, S., Chong, I., Howard, A., Lew, L., Reda, S., Rautio, V., and Moro, D., 2024, June. PikeLPN: Mitigating Overlooked Inefficiencies of Low-Precision Neural Networks. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR 2024).
2. Agiza, A.[1], **Neseem, M.**[1], and Reda, S., 2024, June. MTLoRA: A Low-Rank Adaptation Approach for Efficient Multi-Task Learning. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR 2024).
3. **Neseem, M.**[1] , Agiza, A.[1], and Reda, S., 2023, AdaMTL: Adaptive Input-dependent Inference for Efficient Multi-Task Learning. In Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition Workshops (CVPRW 2023).
4. **Neseem, M.** and Reda, S., 2021, November. AdaCon: Adaptive Context-Aware Object Detection for Resource-Constrained Embedded Devices. In Proceedings of IEEE/ACM International Conference On Computer Aided Design (ICCAD 2021).
5. Hosny, A.[1], **Neseem, M.**[1] and Reda, S., 2021, December. Sparse Bitmap Compression for Memory-Efficient Training on the Edge. In Proceedings of IEEE/ACM

[1]The first two authors contributed equally to this work.

Symposium on Edge Computing (SEC 2021).

6. **Neseem, M.**, Nelson, J. and Reda, S., 2020, July. AdaSense: adaptive low-power sensing and activity recognition for wearable devices. In 57th ACM/IEEE Design Automation Conference (DAC 2020).

7. Ajayi, T., Chhabria, V., Fogaça, M., Hashemi, V., Hosny, A., Kahng, A., Kim, M., Lee, J., Mallappa, U., **Neseem, M.**, et al., 2019, "Toward an open-source digital flow: First learnings from the openroad project." In Proceedings of the 56th Annual Design Automation Conference (DAC 2019).

Acknowledgments

This thesis would not have been possible without the constant support, guidance and inspirations of many kind individuals. First and foremost, I would like to express my immense gratitude to my advisor and mentor, Prof. Sherief Reda, whose guidance, support, and valuable insights during the course of my research has made this thesis possible. I would also like to thank Prof. Jacob Rosenstein and Prof. James Tompkin for being on my defense committee and taking the time to review my thesis.

I am profoundly grateful for the fruitful collaborations with my co-authors: Abdelrahman Hosny, Ahmed Agiza, Jon Nelson, Daniele Moro, Ville-Mikko Rautio, In Suk Chong, Shan Li, Conor McCullough, Lukasz Lew, Andrew Howard, Randy Hsin and Chas Leichner.

I am grateful to the AI Compiler team at Microsoft Research for giving me two internship opportunities. A special thanks to Mason Remy, whose mentorship and guidance were invaluable. I also wish to express my gratitude to the Argos Codec team at Google for offering me an internship opportunity, with a particular mention of Daniele Moro for his mentorship. Additionally, my thanks go to Ville-Mikko Rautio for his guidance and support during my internship and thereafter.

Words fall short of expressing my profound appreciation for my loving husband, Mark Sidhom, whose exceptional care and support have been pivotal throughout the ups and downs of my PhD journey. Additionally, I extend my deepest gratitude to my family for their constant support and love. I am particularly thankful to my parents, Hesham Wasfy and Verseen Shafik, whose support has been foundational to my accomplishments. My brother, Michael Hesham, deserves special thanks for his everlasting encouragement.

Last but not least, I am grateful to my fellow graduate students and friends in Prof. Reda's group at Brown, including Dr. Mostafa Said, Soheil Hashimi, Hokchhay Tan, Abdelrahman Hosny, Farnaz Nouraei, Abdelrahman Hussein, Ahmed Agiza, Jon Nelson, Jingxiao Ma, Manar Abdelatty, Mahdi Boulila and many others who have made the past five years unforgettable. My heartfelt thanks also go to my friends Yostina Farid, Reza Esfandiarpoor, and Miriam George for their companionship during my PhD years.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

1.1 Why AI on the Edge?

Artificial Intelligence (AI) for edge applications represents a paradigm shift in how data is processed and insights are generated in computing systems. Traditionally, machine learning models have relied heavily on cloud-based infrastructures, where data collected by edge devices (e.g., smartphones, IoT devices) is transmitted to centralized servers for processing and analysis. However, this model faces challenges such as high latency, bandwidth limitations, and concerns over data privacy and security. To address these issues, there is a growing trend towards deploying machine learning models directly on edge devices, enabling local data processing and real-time decision-making without the need to constantly communicate with the cloud.

Deploying Artificial Intelligence (AI) on edge devices presents several challenges that stem from the inherent constraints and operational demands of edge computing environments. Primarily, edge devices often have limited computational power, memory, and energy resources, which restricts the complexity of AI models that can be effectively run on such devices. This necessitates the development of lightweight, efficient models and algorithms that can deliver reliable performance without overtaxing the device's capabilities. Furthermore, ensuring the privacy and security of data processed at the edge is critically important, especially as these devices frequently handle sensitive information in an increasingly interconnected landscape. This challenge is compounded by the distributed nature of edge devices, which can complicate efforts to provide consistent updates and patches across all devices. Additionally, achieving low latency in AI applications is crucial for real-time decision-making processes, yet the varying network conditions and bandwidth limitations associated with edge computing can hinder this goal. Lastly, managing and maintaining a vast network of AI-enabled edge devices raises logistical issues, requiring robust frameworks for remote management, diagnostics, and firmware updates to ensure smooth and secure operations across diverse and often geographically dispersed devices. These challenges necessitate a multidisciplinary approach, combining advancements in machine learning, hardware engineering, cybersecurity, and network infrastructure to fully realize the potential of AI on edge devices.

One of the key benefits of machine learning on the edge is the ability to achieve lower latency in applications where real-time processing is crucial. For example, in autonomous vehicles, real-time data processing and decision-making are imperative for safety and performance. By processing data locally on the vehicle, the system can respond more quickly to dynamic road conditions compared to relying on cloud-based computations. Similarly, in healthcare, wearable devices can monitor vital signs and detect anomalies in real-time, offering opportunities for immediate intervention.

Another significant advantage is the enhancement of privacy and security. By processing data locally and minimizing the transfer of sensitive information to the cloud, edge ML can mitigate the risks of data breaches and unauthorized access. This is particularly important in applications dealing with personal or sensitive data, such as in smart homes or personal health devices. Local processing ensures that only essential information, rather than raw data, may need to be sent to centralized servers, thus enhancing user privacy.

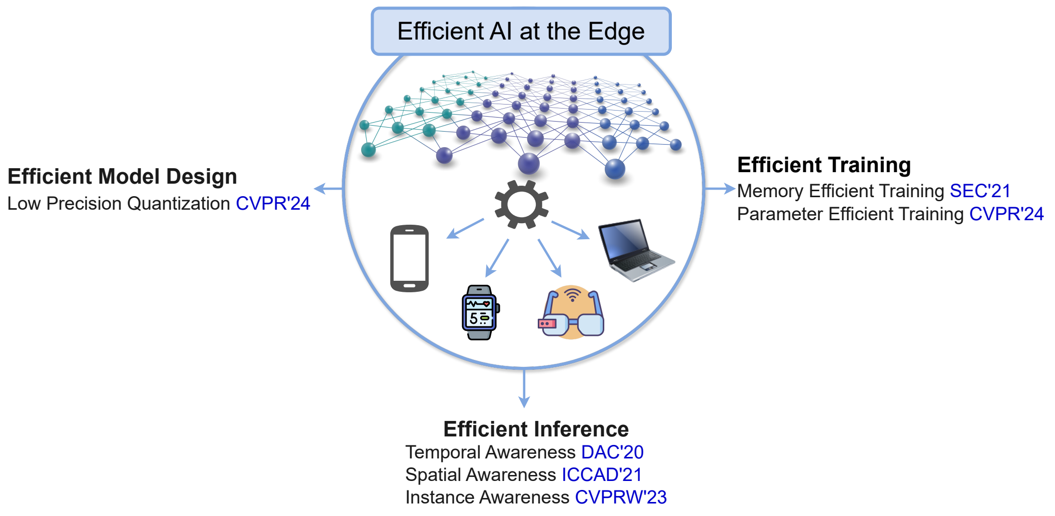

Efficient AI at the Edge
Efficient Model Design
Low Precision Quantization CVPR'24
Efficient Training
Memory Efficient Training SEC'21
Parameter Efficient Training CVPR'24
Efficient Inference
Temporal Awareness DAC'20
Spatial Awareness ICCAD'21
Instance Awareness CVPRW'23

Figure 1.1: Thesis Contributions

1.2 Stages for Deploying AI on the Edge

The development of AI models suitable for resource-constrained edge devices involves several critical steps. First, it is crucial to design highly efficient architectures tailored for these models. Additionally, effective training and fine-tuning are imperative to facilitate on-device personalization and to mitigate domain shifts. Lastly, ensuring efficient inference that aligns with the resource limitations of edge devices is essential to advance the capabilities of AI on edge devices.

This dissertation explores various challenges associated with all the three phases of AI model deployment on edge devices, as illustrated in Figure 1.1. Initially, the thesis investigates model compression and low-precision quantization, which are critical for deploying AI models on edge devices. It begins by examining commonly used low-precision cost metrics, uncovering a significant underestimation of inference costs. To rectify this, the thesis suggests enhancements to existing cost metrics to more accurately represent these overlooked costs, thus ensuring the models are suitable for the constraints of edge devices. Additionally, informed by the novel cost metric, it introduces a new family of low-precision architectures that surpass existing models in terms of efficiency and performance.

Subsequently, this thesis explores efficient training strategies to facilitate model personalization, guaranteeing alignment with the unique requirements of individual users. Our comprehensive investigation identifies memory limitations as the primary challenge in training deep learning models on edge devices. As a response, this thesis capitalizes on the inherent sparsity present in the feature maps to significantly lower memory demands during training. This approach allows for the training of more sophisticated models directly on edge devices.

Finally, this thesis presents innovative methods that utilize context-awareness to develop efficient inference strategies. These strategies allow the model's computational requirements to dynamically adjust in response to the characteristics of incoming data, optimizing performance under stringent resource constraints. To address scenarios with temporal data dependencies, this thesis proposes a framework for adaptive inference with temporal-awareness. This framework intelligently modulates compute resources during runtime, guided by patterns recognized in historical data. Building on adaptive frameworks, this thesis explores integrating spatial awareness to create efficient inference strategies. This method involves hierarchical decision-making, starting with an early, cost-effective assessment of the input's spatial characteristics to determine the most suitable specialized downstream model. Such adaptive inference strategies significantly reduce latency and energy consumption with minimal impact on accuracy. Additionally, this thesis introduces data-driven adaptive policies, enabling the model to learn from the input instances how to allocate computational resources based on the complexity of the input frame. This enhances the efficiency and accuracy of AI models and broadens the adaptability of adaptive inference across various tasks on edge devices.

1.3 Thesis Contributions

Efficient Low-Precision Model Design: In Chapter 3, we investigate compression techniques like quantization. Low-precision quantization is recognized for its efficacy in neural network optimization. Our analysis reveals that non-quantized elementwise operations which are prevalent in layers such as parameterized activation functions, batch normalization, and quantization scaling dominate the inference cost of low-precision models. These non-quantized elementwise operations are commonly overlooked in SOTA efficiency metrics such as Arithmetic Computation Effort (ACE) [177]. In this chapter, we propose ACE_{v2} - an extended version of ACE which offers a better alignment with the inference cost of quantized models and their energy consumption on ML hardware. Moreover, we introduce *PikeLPN*, a model that addresses these efficiency issues by applying quantization to both elementwise operations and multiply-accumulate operations. In particular, we present a novel quantization technique for batch normalization layers named *QuantNorm* which allows for quantizing the batch normalization parameters without compromising the model performance. Additionally, we propose applying *Double Quantization* where the quantization scaling parameters are quantized. Furthermore, we recognize and resolve the issue of distribution mismatch in Separable Convolution layers by introducing *Distribution-Heterogeneous Quantization* which enables quantizing them to low-precision. *PikeLPN* achieves Pareto-optimality in efficiency-accuracy trade-off with up to 3× efficiency improvement compared to SOTA low-precision models.

Leveraging Sparsity for Memory-Efficient Training on the Edge: In Chapter 4, we explore the feasibility of running training on edge devices. Training on the Edge enables neural networks to learn continuously from new data after deployment on memory-constrained edge devices. Our analysis shows that memory footprint from activations is the main bottleneck for training on the edge. Existing incremental training methods fine-tune the last few layers sacrificing accuracy gains from re-training the whole model. In this chapter, we investigate the memory footprint of training deep learning models, and use our observations to propose *BitTrain*. In *BitTrain*, we exploit activation sparsity and propose a novel bitmap compression technique that reduces the memory footprint during training. We save the activations in our proposed bitmap compression format during the forward pass of the training, and restore them during the backward pass for the optimizer computations. The proposed method can be integrated seamlessly in the computation graph of modern deep learning frameworks. Our implementation is safe by construction, and has no negative impact on the accuracy of model training. Experimental results show up to 34% reduction in the memory footprint at a sparsity level of 50%. Further pruning during training results in more than 70% sparsity, which can lead to up to 56% reduction in memory footprint. *BitTrain* advances the efforts towards bringing more machine learning capabilities to edge devices.

Adaptive Inference with Temporal Awareness: In Chapter 5, we examine the process of deriving context from historical data patterns to enhance our machine learning (ML) framework. By leveraging this context, this framework intelligently modulates compute resources during runtime, guided by patterns recognized in historical data To validate this concept, we implemented this approach within a human activity recognition ML framework, which is frequently utilized in wearable technology. Wearable devices have strict power and memory limitations. As a result, there is a need to optimize the power consumption on those devices without sacrificing the accuracy. This chapter presents *AdaSense* – a sensing, feature extraction and classification co-optimized framework for Human Activity Recognition. The proposed techniques reduce the power consumption by dynamically switching among different sensor configurations as a function of the patterns in user activity over time. The framework selects configurations that represent the pareto-frontier of the accuracy and energy trade-off. AdaSense also uses low-overhead processing and classification methodologies. The introduced approach achieves 69% reduction in the power consumption of the sensor with less than 1.5% decrease in the activity recognition accuracy.

Adaptive Inference with Spatial Awareness: In Chapter 6, we investigate integrating spatial context awareness into model architecture design to improve its efficiency. This method involves hierarchical decision-making, starting with an early, cost-effective assessment of the input's spatial characteristics to determine the most suitable specialized downstream mode. By harnessing this spatial information, we seek to dynamically enhance the efficiency of our ML model. To evaluate the efficacy of this strategy, we have implemented it in an object detection ML framework, which is extensively employed in applications such as surveillance cameras and augmented reality devices. Those object detection ML frameworks often have large computational and energy requirements that challenge their deployment on resource-constrained edge devices. Object detection takes an image as an input, and identifies the existing object classes as well as their locations in the image. In this chapter, we present *AdaCon*. *AdaCon* leverage the prior knowledge about the probabilities that different object categories can occur jointly in the same spatial context to increase the efficiency of object detection models. In particular, our technique clusters the object categories based on their spatial co-occurrence probability. Then, it uses those clusters to design an adaptive network. During runtime, a branch controller decides which part(s) of the network to execute based on the spatial context of the input frame. Our experiments using COCO dataset show that our adaptive object detection model achieves up to 45% reduction in the energy consumption, and up to 27% reduction in the latency, with a small loss in the average precision (AP) of object detection.

Adaptive Inference with Instance Awareness: In Chapter 7, we investigate the model's capability to make predictions based on learnt contextual information, and dynamically adapt its inference process based on this context. The goal is to design a generalizable context-aware ML model where we don't need to handcraft the context criteria. To test the applicability of this approach, we apply it to a vision transformer model that performs multiple tasks simultaneously. Those multi-task models are necessary for applications where we need to extract a lot of information from the input like augmented reality applications running on Augmeted Reality devices. Multi-task learning (MTL) models usually consist of a shared encoder to extract representative features from the input frame, followed by task-specific decoders to generate predictions for each task. Generally, the shared encoder in MTL models needs to have a large representational capacity in order to generalize well to various tasks and input data, which has a negative effect on the inference latency. In this chapter, we argue that due to the large variations in the complexity of the input frames, some computations might be unnecessary for the output. Therefore, we introduce *AdaMTL*, an adaptive framework that learns task-aware inference policies for the MTL models in an input-dependent manner. Specifically, we attach a task-aware lightweight policy network to the shared encoder and co-train it alongside the MTL model to recognize unnecessary computations. During runtime, our task-aware policy network decides which parts of the model to activate depending on the input frame and the target computational complexity. Extensive experiments on the PASCAL dataset demonstrate that AdaMTL reduces the computational complexity by 43% while improving the accuracy by 1.32% compared to single-task models. Combined with SOTA MTL methodologies, AdaMTL boosts the accuracy by 7.8% while improving the efficiency by 3.1×. When deployed on Vuzix M4000 smart glasses, AdaMTL reduces the inference latency and the energy consumption by up to 21.8% and 37.5%, respectively, compared to the static MTL model.

CHAPTER 2

Background

In this chapter, we provide the background and a concise overview of the relevant prior work related to the techniques proposed in this dissertation. We begin with a discussion of methods for efficient training, as detailed in Section 2.1. Following this, Section 2.2 explores prior work aimed at enhancing the efficiency of inference in deep learning models. Additional details regarding the related work for the specific techniques discussed in this dissertation are included in their respective chapters.

2.1 Efficient Training for Deep Learning Models

Transfer Learning. Deep learning models trained on large datasets (e.g. ImageNet [31]) can be widely used to retrain neural networks on the edge with local data. The idea is to keep the parameters of the feature extraction layers unchanged, and only train the last layers [123]. Transfer learning on the edge can be used for customization of mobile services as well as for offline retraining. This approach saves training memory because the intermediate activations for the feature extractor do not need to be stored. However, the accuracy can significantly drop, especially when the new data is coming from a distribution that is very far from the distribution of the data used during the initial training. To solve this issue, Cai *et al.* [17] proposed fine-tuning the both the final layers as well as the biases of the feature extractor (i.e intermediate activations are not needed to compute the gradients for the biases). This approach saves the memory footprint; however, fine-tuning all the layers significantly increase the ability of the model to adapt on the new data.

Low Precision Training. Micikevicius *et al.* [113] use half precision (16 bits) for weights, gradients, and activations. This reduces the memory footprint by a factor of 2×, and it can be complementary to any other low-memory training technique to maximize the savings. Courbariaux *et al.* [26] show that they can train models using 10-bits multiplications without severely affecting the accuracy. Jia *et al.* [79] increase the training throughput of a single GPU using a mixed-precision training method. Dipankar *et al.* [29] use fixed-point integer operations to train the models. They show that this can achieve competitive results to training with floating-point operations. All of these techniques use lower precision to reduce the memory footprint of training and possibly the number of operations needed, which compromises on the accuracy of the model.

Microbatching. Huang *et al.* [70] use microbatch-based training where they can sequentially send smaller subsets of the batch through the network, and accumulate the gradients until the whole batch is processed. Then, gradient update is executed once. This approach reduces the memory footprint without affecting the total number of operations performed. It is important to note that microbatching has a direct impact on the statistical characteristics of batch normalization layers. That is why it needs to be exercised carefully in order to avoid losing accuracy.

Rematerialization. Chen *et al.* [21] first proposed the idea of trading computation for memory. The idea is to discard saving the activations and recalculate them, layer-by-layer, upon backpropagation. Gruslys *et al.* [46] proposed a dynamic programming approach that balances between caching of intermediate results and re-computation. The interested reader is referred to [136] for a detailed technical report on combining some of the above techniques for training.

Parameter-Efficient Training. Parameter-efficient training (PEFT) has become increasingly important, especially when dealing with large-scale pre-trained models [66, 175, 40, 67] since traditional fine-tuning methods, which involve adjusting a significant portion of a model's parameters for specific tasks, can be resource-intensive. Two common techniques in this domain are adapters [175, 40] and Low-Rank Adaptation (LoRA) [66, 32]. *Adapters* are lightweight modules inserted between the layers of a pre-trained model, which allows for targeted modifications to the model's behavior without altering the original pre-trained weights. This approach is beneficial as it reduces the number of parameters that need to be fine-tuned, thus lowering the computational burden. Adapters have shown effectiveness in various tasks, providing a flexible and efficient way to adapt large models to specific tasks or datasets. However, one limitation of adapters is the additional parameters they introduce, which can lead to increased computational requirements during inference. On the other hand, *LoRA* offers a different approach to PEFT. LoRA involves modifying the weight matrices of a pre-trained model using low-rank decomposition. This method allows for fine-tuning the model's behavior while maintaining the original structure and size of the weight matrices. The key advantage of LoRA is that it does not introduce additional parameters during the model's runtime. Instead, it updates the pre-existing weights to enhance the model's performance on new tasks with minimal increase in computational requirements. LoRA has been successfully applied in various fields, including NLP [66, 32, 19, 22] and computer vision [61], demonstrating its versatility and effectiveness. Some recent studies have proposed new solutions to extend the benefits of PEFT for multi-task adaptation. One such approach is the Hypernetworks [108], which uses shared networks to generate adapter parameters for all layers conditioned on the task, thus allowing for the sharing of information across different tasks while enabling task-specific adaptation through task-specific adapters. Building on top of it, Polyhistor [100] explores PEFT in the domain of dense vision tasks, specifically on hierarchical vision transformers. Polyhistor proposes two ideas: decomposing hypernetworks into low-rank matrices and using custom kernels to scale fine-tuning parameters to the different transformer blocks.

2.2 Efficient Inference for Deep Learning Models

Compact Deep Learning Architectures. The optimization of compute and memory resources for model inference on edge devices has been a critical area of focus in recent research [20]. To address these challenges, researchers have hand-crafted compact models [65, 107, 162, 145]. SqueezeNet, for instance, is a parameter-efficient model designed for resource-constrained environments [71]. It achieves AlexNet-level accuracy on the ImageNet dataset with 50 times fewer parameters by employing strategies like replacing 3×3 convolutions with 1×1 ones, converting input channels to only 3×3 filters, and delaying downsampling for enhanced accuracy. Similarly, MobileNets [65] utilize spatially separable convolutions, which split a 3×3 convolution into two smaller operations – a 3×1 and a 1×3 convolution—thereby reducing the computational cost and number of parameters from 9 to 6, and consequently decreasing matrix multiplications. Building on these advancements, MobileOne [154] has further identified parameterized activation functions and skip connections as major sources of latency in edge devices. Leveraging these insights, MobileOne developed a model capable of operating within 1ms on mobile platforms, showcasing significant progress in reducing latency for edge computing applications.

Model Compression Techniques. In addition to crafting compact models, model compression techniques have been employed to facilitate running deep neural networks (DNNs) on small devices [50, 93]. There are three primary methods for reducing the size of networks: quantization, pruning, and knowledge distillation. *Quantization* involves converting the parameters of a DNN from floating-point representations to low-bit width numbers, which simplifies the computational demands by eliminating costly floating-point multiplications. Research in low-precision quantization demonstrates that networks can be effectively quantized to 4 bits with only minimal loss in accuracy [23, 83, 2, 124].

Additionally, some studies focus on power-of-two quantization, which is known for its hardware efficiency [147, 54, 91]. *Pruning* targets the removal of less critical parameters, such as those near zero, to reduce the computational demands of the model [14, 76, 171, 143]. For instance, Molchanov *et al.* [117] utilize a loss-approximating Taylor expansion as a gradient-based metric for identifying pruning candidates. Anwar *et al.* [5] and Yang *et al.* [165] select candidates through random evaluations and energy consumption weighting, respectively. Strategies such as early pruning [120] and dynamic pruning [48] aim to enhance integration with retraining processes, thus conserving time spent on retraining. *Knowledge distillation* involves constructing a smaller DNN that replicates the functionality of a larger, more complex one [62]. This process is executed by training the smaller network using the output predictions of the larger model, allowing the smaller network to approximate the learning function of its larger counterpart. Collectively, these approaches – quantization, pruning, and knowledge distillation – can be applied separately or in combination to optimize DNNs for operation on constrained devices [50].

CHAPTER 3

Efficient Model Design

3.1 Introduction

Quantization has long been established as a method to improve the efficiency of deep neural networks, resulting in smaller models and accelerated processing [41]. Recent studies have shown impressive results in image classification tasks, making the use of low-precision quantization (i.e., 4 bits or fewer) increasingly popular [124, 177, 104, 128]. In these compact models, convolutional and fully connected layers are typically constrained to 4-bit precision or even less, while precision is maintained at higher levels in other layers of the network. For example, the state-of-the-art (SOTA) binary network PokeBNN [177] binarizes the convolutional layers of ResNet-50 [57], and to avoid accuracy loss, they incorporate extra skip connections, extra batch normalization layers, and parameterized activation functions (DPReLU) that are executed in high precision.

We analyze the key efficiency bottlenecks in low-precision models uncovering a fundamental limitation of the efficiency metrics in literature, ACE [177], CPU64 [110, 104], Unit-gate model [181] and FA-count [133]. Those metrics exclude the elementwise operations in arithmetic calculations, a sentiment grounded in the belief that their contribution to the total computation cost is negligible compared to MAC operations. Optimizing for those metrics drives researchers to prioritize the reduction of computational precision in Convolutional and Dense layers, yet they overlook the quantization of elementwise operations. As a result, operations such as batch normalization, activation functions, and quantization scaling multiplications, are often performed at full precision. Moreover, SOTA low-precision models tend to rely extensively on mechanisms like branching [68] and skip connections [57], which significantly increase energy costs associated with memory reads and writes. To overcome this issue, we propose ACE_{v2} which extends the efficiency metric ACE to account for all arithmetic operations in quantized neural networks including both elementwise and MAC operations. This would help guide researchers' choices when designing low-precision models.

Guided by our ACE_{v2} metric, we design *PikeLPN* – a novel family of efficient low-precision models. *PikeLPN* quantizes both elementwise and MAC operations. Remarkably, *PikeLPN* not only achieves a 3.5× cost reduction compared to SOTA binary models [104, 177], it also achieves competitive accuracy levels on ImageNet [30].

Our contributions can be summarized as follows:

- We identify and analyze the overlooked cost of non-quantized elementwise operations in SOTA low-precision models. Our analysis shows that the non-quantized elementwise operations used in parameterized activation functions, batch normalization, and quantization scaling dominate the inference cost of low-precision models.

- We propose ACE_{v2} – an extension to the existing hardware-agnostic cost metric *ACE*. ACE_{v2} offers a better alignment with the cost of the low-precision models and their energy consumption on ML hardware by accounting for all arithmetic operations during inference.

- We propose *PikeLPN* – a novel family of low-precision architectures, which improves the efficiency of low-precision models by quantizing both elementwise and multiply-accumulate operations. Specifically, we propose (a) *QuantNorm* for effective batch normalization quantization, (b) *Double Quantization* where quantization parameters are also quantized, and (c) *Distribution-Heterogeneous Quantization* for Separable Convolution layers to tackle their distribution mismatch problem.

The rest of the chapter is organized as follows. We review the related work in Section 3.2. In Section 3.3, we propose ACE_{v2} providing detailed analysis to the overlooked efficiency bottlenecks by previous cost metrics. Then, guided by the new cost metric, we propose our efficient *PikeLPN* model. Next, we compare PikeLPN to SOTA low-precision models in Section 3.4. Finally, we conclude in Section 3.5.

3.2 Related Work

Low-precision Quantization: A substantial body of work exists in the realm of low-precision quantization, exemplified by studies that indicate that architectures can be quantized to 4 bits with minimal impact on accuracy [23, 83, 2, 124]. Others perform logarithmic quantization methods known for their hardware efficiency [147, 54, 91]. In addition, there are attempts to push the boundaries by introducing predominantly binary models where some of the convolution layers are quantized to 1 bit while other layers are maintained at a higher precision [177, 104, 127]. Some researchers have also developed automated strategies for mixed-precision modeling to dynamically choose the optimal precision for each layer, contingent upon a predetermined efficiency metric [84]. However, existing approaches primarily focus on the quantization of multiply-accumulate (MAC) operations in convolution and dense layers. They commonly neglect elementwise operations such as those in batch normalization layers and activation functions. Our empirical findings show that this assumption becomes invalid for low-precision models, specifically 4 bits or below.

Architectural Approaches to Low-precision Models: Several studies have adopted architectural modifications to enhance the performance of low-precision models.

Many such modifications involve the integration of modules consisting solely of elementwise operations, aiming to minimize computational and parameter overhead. For instance, the channelwise real-valued rescaling of binarized tensors has been proposed as an effective means to reduce quantization error [130]. This approach incorporates elementwise floating-point multiplications for each channel. Additional methods, as suggested in [28], advocate for per-vector quantization, which results in multiple elementwise multiplications per channel. Studies like FracBNN [178] and PokeBNN [177] include extra Batch Normalization layers in their predominantly binary models to expedite the training convergence. Moreover, the use of parameterized activation functions, such as PReLU [59] and DPReLU [177], has become a standard practice for improving the performance of low-precision models [104, 103]. All these modifications necessitate elementwise floating-point multiplications and additions. Moreover, the introduction of skip connections has proven beneficial in enhancing low-precision model quality. Notably, ReActNet [104] and PokeBNN [177] are designed with 4 and 3 parallel branches, respectively. Although skip connections only involve elementwise additions, they contribute to an increased memory access during inference to store multiple activations increasing the inference cost [78].

Cost Metrics for Efficiency Evaluation: MAC operations have been recognized in literature as the principal contributors to inference cost of deep learning models. As a result, efficiency metrics have predominantly focused on these specific operations. The *CPU64* metric [103, 104, 102] has been used to gauge the efficiency of mixed-precision neural networks when running on CPUs. With the growing utilization of specialized machine learning hardware and accelerators, a newer metric named *ACE* has been introduced [177]. *ACE*, an acronym for *Arithmetic Computation Effort*, is formulated as the product of the number of MAC operations and the bitwidth of the two operands involved, which is directly proportional to the number of active hardware bit-adders required. The Unit-gate model [181] and FA-count [133] correlate very well with ACE and differ only by a small constant factor [1]. All these metrics do not consider elementwise operations. Thus, in

[1]They do not account for carry-save format for local accumulator representations typically used in

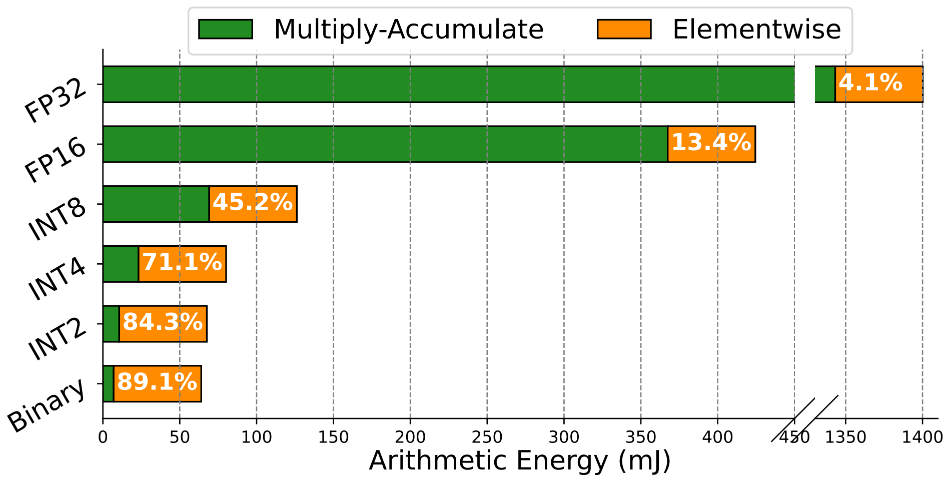


Figure 3.1: Arithmetic Energy on 45nm CMOS technology by multiply-accumulate operations versus non-quantized elementwise operations for MobileNetV2. Energy costs are calculated using Table 3.1. The figure reveals that elementwise operations are a substantial contributor to the overall cost in low-precision models.

this chapter, we extend the ACE metric introducing ACE_{v2}, and this extension should generalize to other metrics as well. All these metrics, including the extended ACE, are technology node independent.

3.3 Designing Highly Efficient Low-Precision Models

In this section, we identify previously overlooked costs in state-of-the-art (SOTA) cost metrics. Additionally, we propose extending the *Arithmetic Computational Effort* (ACE) metric [177] to provide a more accurate representation of the inference cost of low-precision models. Subsequently, we assess the impact of various design alternatives in low-precision models on the cost of inference. Finally, we present *PikeLPN* – a novel family of low-precision models.

3.3.1 Cost Metrics for Low Precision Models

The prevalent notion is that multiply-accumulate operations in the convolution and dense layers are the sole substantial contributors to inference cost in deep learning models

systollic arrays.

[124, 177, 104]. This viewpoint stems from the observation that for full precision models the energy cost of those layers is more than 95% of the total model operations as shown in Figure 3.1. Consequently, commonly used efficiency metrics for quantized neural networks, such as *CPU64* [103, 104, 102] and *ACE* [177], are tailored to exclusively account for multiply-accumulate operations in these specified layers. Optimization in accordance with these metrics drive researchers to prioritize reducing the precision of multiply-accumulate operations in convolution and dense layers while maintaining high precision for all other elementwise operations. Moreover, they re-parameterize the models adding layers that only have elementwise operations to compensate for any accuracy losses by low-precision quantization [177, 104]. However, our analysis reveals that these non-quantized elementwise operations substantially contributes to the arithmetic cost during inference of low-precision models (i.e., 8 bits and lower), thereby challenging the prevailing assumptions.

Figure 3.1 illustrates the relative contributions of low-precision multiply-accumulate operations and non-quantized elementwise operations to the total energy consumption by arithmetic computations at various precisions. The data reveals a notable trend: the proportion of energy consumed by elementwise operations becomes more significant as the precision decreases. For example, in binary-quantized models, those non-quantized elementwise operations account for up to 89% of the total cost. This observation highlights the limitations of existing metrics in accurately gauging the efficiency of quantized models. Consequently, we propose ACE_{v2} which extends the ACE metric [177] to account for both multiply-accumulate operations as well as elementwise operations. We anticipate that our comprehensive ACE_{v2} metric will enable more informed optimization choices within the research community.

Table 3.1: Cost under 45nm CMOS technology [168, 64] [2]. $f(i,j)$ refers to the formula used to calculate the ACE_{v2} cost where i and j are the precisions of the two operands. $c_a = 6$ and $c_s = 5$. The correlation coefficient between ACE_{v2} and the independently measured arithmetic energy consumption is 0.991.

	MULTIPLY		**ADD**		**SHIFT**	
	Energy (pJ)	$\boldsymbol{ACE_{v2}}$	**Energy** (pJ)	$\boldsymbol{ACE_{v2}}$	**Energy** (pJ)	$\boldsymbol{ACE_{v2}}$
FP32	3.7	992	0.9	192	-	-
FP16	1.1	240	0.4	96	-	-
$f(i,j)$	$i \cdot j$ - $max(i,j)$		$c_a \cdot max(i,j)$		-	
INT32	3.1	992	0.1	32	0.13	32
INT16	-	240	-	16	0.057	12.8
INT8	0.2	56	0.03	8	0.024	4.8
INT4	-	12	-	4	-	1.6
INT2	-	2	-	2	-	0.4
Binary	-	-	-	1	-	-
$f(i,j)$	$i \cdot j$ - $max(i,j)$		$max(i,j)$		$i \cdot log_2(j)/c_s$	

3.3.2 Introducing ACE_{v2}

ACE has been used to estimate the cost of inference on idealized ML hardware implemented with CMOS methodology [177]. ACE is defined by its authors as the number of bitadders (i.e., digital circuit adding 3 bits to form a 2 bit number – carry and sum) required to perform every multiply-accumulate operation. The authors justify that definition by showing a high correlation coefficient (i.e., 0.946) between the number of bitadders and the independently measured energy consumption on 45nm CMOS technology. While ACE provides a hardware-agnostic method to evaluate the efficiency of quantized neural networks, it fails to include the elementwise operations which can be the dominating cost factor in low precision models as shown in Figure 3.1. Moreover, ACE does not provide a way to estimate the cost of shift operations which are required to implement non-linear base-2 logarithmic quantization [169, 168]. We propose ACE_{v2} which improves ACE by extending it to include elementwise multiplication, elementwise addition, and shift operations. We establish the ACE_{v2} formulas for the previously discussed operations as shown in Table 3.1.

[2]Energy costs for low-precision operations can be extrapolated linearly for addition and quadratically for multiplication [24].

Elementwise Multiplications:

Using established methods for constructing multipliers, such as adder trees proposed by Wallace and Dadda [159, 27], we calculated the number of adders needed to multiply an i-bit number by a j-bit number as $i \cdot j - max(i, j)$. This formula exactly matches the optimal number of adders for $1 <= i, j <= 64$. Detailed derivation in Appendix A.

Elementwise Additions:

Fixed-point numbers added using established adders [3] activate an upper bound of $max(i, j)$ bit adders to add i-bit and j-bit numbers. Floating-point adders additionally require exponent alignment, significand addition, and normalization steps [134], resulting in a much higher energy consumption compared to fixed-point adders as shown in Table 3.1. We analyze the operations needed in floating point adders [134] and come to an ACE_{v2} cost of 6× the cost of a fixed-point adder. Therefore, we derive ACE_{v2} for floating point adders using $c_a \cdot max(i, j)$ with $c_a = 6$. Detailed derivation in Appendix A.

Shift Operations:

A Barrel Shifter is an established method to shift and rotate i-bit numbers by j locations in modern processors [55]. The barrel shifter is implemented as a cascade of $i \log_2(j)$ *2:1* multiplexers. Therefore, we derive ACE_{v2} for a shift operation as $i \log_2(j)/c_s$ where c_s is the ratio of the cost of a *2:1* multiplexer compared to a full adder. Since a full adder can be efficiently implemented using five *2:1* multiplexers based on [82], we assign $c_s = 5$.

To verify the correctness of our ACE_{v2} metric, Table 3.1 shows a 0.991 correlation coefficient between the *independently* measured energy consumption of various arithmetic units on the 45nm CMOS technology and its ACE_{v2} cost, a notable improvement compared to the 0.946 correlation coefficient in ACE [177]. Using those definitions, we estimate a

[3]While there are many methods for constructing adders, such as Carry Lookahead Adder [122] and Ripple Carry Adder [6], the particular implementation has a limited effect on the energy use.

Table 3.2: The contribution of non-quantized Batch Normalization Layers to the overall ACE_{v2} cost.

Model	**BN Adds** (*Million*)	**BN Mults** (*Million*)	**BN ACE_{v2}** (%)
MobileNetV2 $(4W, 4A)$	6.67	6.67	**41.87**
ResNet50 $(1W, 1A)$	10.58	10.58	**41.38**

more accurate arithmetic cost for any quantized model.

3.3.3 Overlooked Efficiency Bottlenecks

Batch Normalization:

Batch normalization layers, which necessitate elementwise multiplications and additions, typically retain parameters in floating-point format during deep neural network quantization to maintain training stability and prevent accuracy loss [177, 104, 127]. Consequently, these operations are performed using floating-point (FP32) arithmetic, with a single FP32 operation consuming approximately 18× more energy than an INT8 multiplication, as detailed in Table 3.1. Assessing the impact of these non-quantized batch normalization layers in Table 3.2 reveals that they can account for as much as 42% of the total ACE_{v2} cost in various low-precision models. This substantial contribution shows the importance of considering the cost of these operations and potentially quantizing its parameters.

Activation Layers:

In recent literature, low-precision models have increasingly replaced ReLU [3] activation functions with parameterized activation functions such as PReLU [59] and DPReLU [177] to improve performance and training stability of quantized models [104, 128]. The dynamic parameterized rectified linear unit (DPReLU), for instance, is defined by the following

Table 3.3: The cost overhead when replacing a ReLU activation function with non-quantized parameterized alternatives such as PReLU and DPReLU on a 4-bit MobileNetv2 model.

Activation	**Adds** $(Million)$	**Mults** $(Million)$	$\boldsymbol{ACE_{v2}}$ $(\times 10^9)$	**Overhead** (%)
ReLU [3]	0	0	20.44	-
PReLU [59]	0	6.1	26.5	**+29.6%**
DPReLU [177]	6.1	6.1	27.67	**+35.3%**

piecewise function:

$$DPReLU(x) = \begin{cases} \eta(x-\alpha)-\beta & \text{if } x-\alpha>0 \\ \gamma(x-\alpha)-\beta & \text{otherwise} \end{cases} \tag{3.1}$$

Here, the parameters η, α, β, and γ are represented in floating-point format. Consequently, the computation of DPReLU necessitates both elementwise floating-point multiplications and additions. Our study, detailed in Table 3.3, assesses the impact of these elementwise operations on the ACE_{v2} cost. We find that in a 4-bit MobileNetV2 model, the incorporation of different activation functions — namely ReLU, PReLU, and DPReLU — significantly influences the cost. Specifically, the use of PReLU and DPReLU, despite their benefits on accuracy, introduces up to 35% increase in the overall inference cost. This finding highlights the need to balance the benefits of parameterized activation functions with their computational demands.

Skip Connections:

Skip connections are regarded as zero-cost operations in terms of arithmetic computation. Consequently, previous work overused them to improve the model performance without having any measurable effect on the cost [177, 104, 127]. For instance, ReActNet [104] incorporated four parallel branches, quadrupling its memory footprint compared to a single-path model. PokeBNN [177] followed a similar design, incorporating three parallel branches. However, such branching necessitates the concatenation of feature maps

Table 3.4: Arithmetic Intensity computed according to Equation (3) for a ResNet-50 model with various number of branches.

Arithmetic Intensity (Ops/Element ↑)		
2 Branches	3 Branches	4 Branches
73.5	49.66	36.75

from previous layers, leading to an increase in the amount of data concurrently stored in memory. That increase the required memory reads and writes which have significant costs. As an example, in a processor with a 32KB cache designed using 45nm CMOS technology, moving an 8-bit element from the cache consumes approximately $2.5pJ$ of energy. This is about 12× the energy needed for an INT8 multiplication operation, which requires only around $0.2pJ$ as shown in Table 3.1. This disparity becomes even more profound when data must be transfered from DRAM, where the energy requirement balloon to $162.5pJ$ – 810× higher than the INT8 multiplication [64]. Quantifying this overhead in a hardware-agnostic manner is challenging since it is influenced by a multitude of factors including the underlying hardware architecture, memory location, and model size. Yet, understanding its impact remains crucial to design efficient models. We advocate for the adoption of *Arithmetic Intensity* as a practical metric to measure memory reads and writes during inference [78]. Arithmetic Intensity (AI_c) is defined as the ratio of the arithmetic operations (M_c) to the amount of data, including both Weights (W) and Activations (A), required to execute these operations as shown in Equation 3.2.

$$AI_c = \frac{M_c}{W + A} \tag{3.2}$$

Consequently, Arithmetic Intensity serves as an indicator of the amount of memory reads and writes to perform computational operations. Adding branches lead to a substantial increase in the amount of data that must be loaded to execute a relatively small number of operations; hence decreasing the arithmetic intensity as shown in Table 3.4.

Table 3.5: ACE_{v2} of a 4-bit MobileNetV2 and a Binary ResNet50 model with various quantization granularities. The *Overhead* represents the percentage of cost required by the extra FP operations due to quantization (i.e. quantization scaling).

Quantization Granularity	**Mults** $(Million)$	$\boldsymbol{ACE_{v2}}$ $(\times 10^9)\downarrow$ Total	Overhead (%)
MobileNetV2 - $< 4W, 4A >$			
Layerwise [41]	6.67	20.44	**32.52%**
Channelwise [41]	6.67	20.44	**32.52%**
Sub-Channelwise [28]	13.35	27.06	**48.97%**
ResNet50 - $< 1W, 1A >$			
Layerwise [41]	10.63	28.13	**32.03%**
Channelwise [41]	10.63	28.13	**32.03%**
Sub-Channelwise [28]	32.75	50.08	**63.55%**

Quantization Granularity Overhead:

Uniform quantization, a widely adopted technique in SOTA low-precision models [127, 124, 177], transforms discrete integer values, q, into continuous real values, r through the affine relation

$$r = S(q - Z) \tag{3.3}$$

where S is a scale factor. S is a critical component of quantization which is typically learned as an arbitrary floating-point value during training. In the inference phase, this necessitates an elementwise multiplication by S, contributing to computational overhead [74]. Proper scaling is crucial in quantization to mitigate quantization error enabling quantized models to maintain high accuracy. Quantization granularity dictates the level at which scaling factors are applied in a model [41]. For example, Layerwise quantization assigns a single scale factor based on all weights within a layer. Channelwise quantization, widely adopted in state-of-the-art low-precision models, allocates a unique scaling factor to each channel, catering to the varying distributions of weights and potentially enhancing model accuracy. Sub-Channelwise quantization takes this further by assigning several scaling factors within each channel, allowing for even finer adjustments at the expense of increased computational cost [28]. All quantization granularities add one or more elementwise multiplications per channel. Table 3.5 compares the ACE_{v2} cost of such

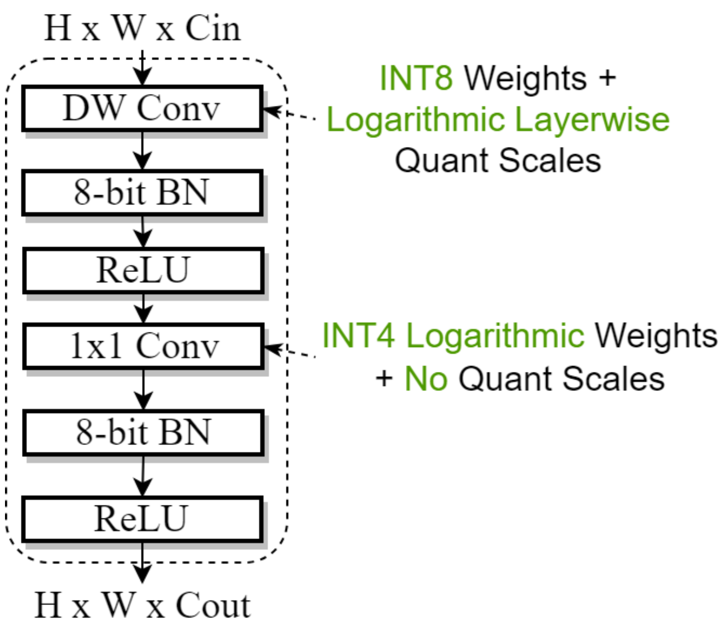

H x W x Cin
DW Conv
8-bit BN
ReLU
1x1 Conv
8-bit BN
ReLU
H x W x Cout
INT8 Weights + Logarithmic Layerwise Quant Scales
INT4 Logarithmic Weights + No Quant Scales

Figure 3.2: *PikeLPN* building block architecture.

quantization granularities. In the popular Channelwise quantization, the overhead from elementwise multiplications is 32% of the total cost.

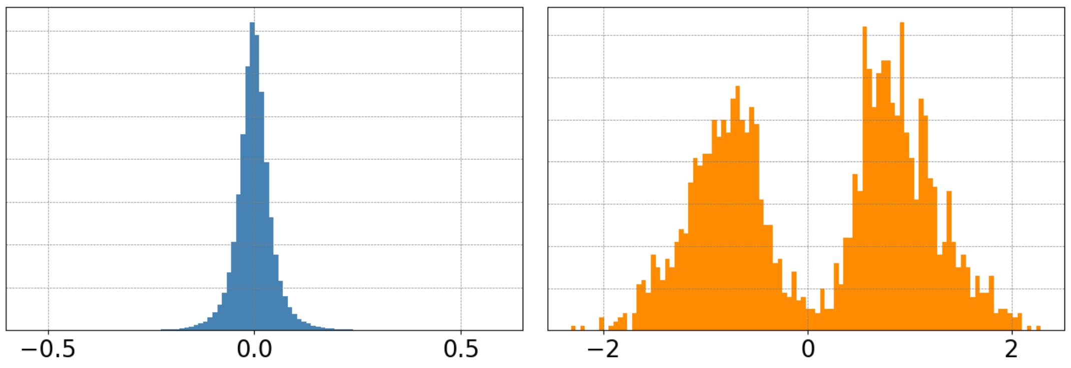
3.3.4 PikeLPN Architecture

Based on our comprehensive analysis, we introduce *PikeLPN*, a novel architecture engineered to mitigate the inefficiencies of SOTA low-precision models. This section introduces the basic block of our proposed *PikeLPN* model, explores quantization strategies for the different layers, and proposes a novel method for quantizing batch normalization layers without compromising the model's accuracy.

PikeLPN Basic Block: To engineer an effective low-precision model, we first design the baseline architecture with building blocks that are inherently efficient. With this principle in mind, our architecture adopts separable convolutional layers, subdivided into depthwise and pointwise convolutions, in line with the framework established by MobileNetV1 [65]. Those layers are widely recognized for their computational efficiency and have been integrated into SOTA efficient ConvNets [144, 154]. Figure 3.2 illustrates the building block for *PikeLPN*. To maximize computational efficiency, the used architecture deliberately

Table 3.6: Top-1 Accuracy on ImageNet vs ACE_{v2} cost of PikeLPN using various quantizers for the Depthwise and Pointwise Layers. PW-Convolution layers contribute to 95% of the number of multiply-accumulate operations in the model, that is why we lower the precision of the PW Conv layers to 4 bits while we keep the DW Conv layers at 8-bits.

Pointwise Conv.		**Depthwise Convs**		**Top-1**	$\boldsymbol{ACE_{v2}}$
Weights	Q-Params	Weights	Q-Params	(%)	($\times 10^9$)
Linear-4	Arbitrary	Linear-8	Arbitrary	68.50	20.91
Linear-4	PoT	Linear-8	PoT	68.41	15.93
PoT-4	-	PoT-8	-	64.50	10.05
PoT-4	-	Linear-8	Arbitrary	67.60	12.86
PoT-4	-	Linear-8	PoT	**67.55**	**10.95**



(a) (b)

Figure 3.3: Weights distribution of pre-trained PW and DW Convolution layers in PikeLPN where (a) Sample Pointwise layer weights (b) Sample Depthwise layer weights.

avoids parameterized activation functions and skip connections that are likely to increase computational cost as explained in Subsection 3.3.3. Finally, our model uses the first and last blocks from the MobileNetV1 architecture due to their proven effectiveness and reliability.

Quantizing Separable Convolution Layers: Linear quantizers results in a set of equally spaced values since they use affine mapping as shown in Equation 3.3. Non-uniform quantizers have different constraints. For example, Power-of-two (PoT) [115] restrict quantization levels to be powers-of-two values. They can be used to increase the representational density of small values, furthermore, they have the benefit of replacing the multiplication operations during inference with shifts which are significantly cheaper as shown in Table 3.1. However, using PoT quantizers for both pointwise (PW) and depthwise (DW) convolution operations in the separable convolution block leads to significant accuracy degradation as shown in the third row of Table 3.6. To get some insights, we analyze the distribution of the full-precision weights of PikeLPN when pre-trained on ImageNet. Figures 3.3(a) and 3.3(b) visualize the distributions of a sample PW and DW weights respectively. Interestingly, the majority of the weights of the PW layer lie around ± 0.1, while the weights in the DW layer are distributed around ± 2. This mismatch in weights distribution across different layers makes low-precision quantization for the separable convolution blocks challenging because the used values fail to capture both distributions. To address this problem, we propose using *Distribution Heterogeneous Quantization* where the pointwise weights use the more efficient PoT quantizer while the depthwise weights use a linear quantizer. It is important to note that pointwise convolutions contribute to 95% of the number of multiply-accumulate operations in *PikeLPN*; hence using the PoT quantizer in pointwise layers only improves the model's efficiency by 50% as shown in Table 3.6.

Double Quantization: Quantization requires extra elementwise multiplications by a floating-point scaling factor which add significant overhead as shown in Table 3.5. While we can not completely remove the scale factor, we can reduce the overhead from quantization scale multiplications by quantizing those quantization parameters. We refer to quantizing the quantization parameters as *Double Quantization*. We consider using a PoT scale for the linear depthwise quantizer in *PikeLPN* which can potentially reduce the elementwise operation from $3.7mJ$ to $0.13mJ$ based on Table 3.1. Our experiments indicates negligible effect on accuracy when applying *Double Quantization* as shown in Table 3.6.

Quantizing Batch Norm Layers: Batch normalization layers are used in most modern deep learning models to stabilize the training and improve their performance [72]. Batch normalization is computed as follows:

$$batchnorm(x) = \frac{(x - \mu) * \gamma}{\sqrt{\sigma^2 + \epsilon}} + \beta \tag{3.4}$$

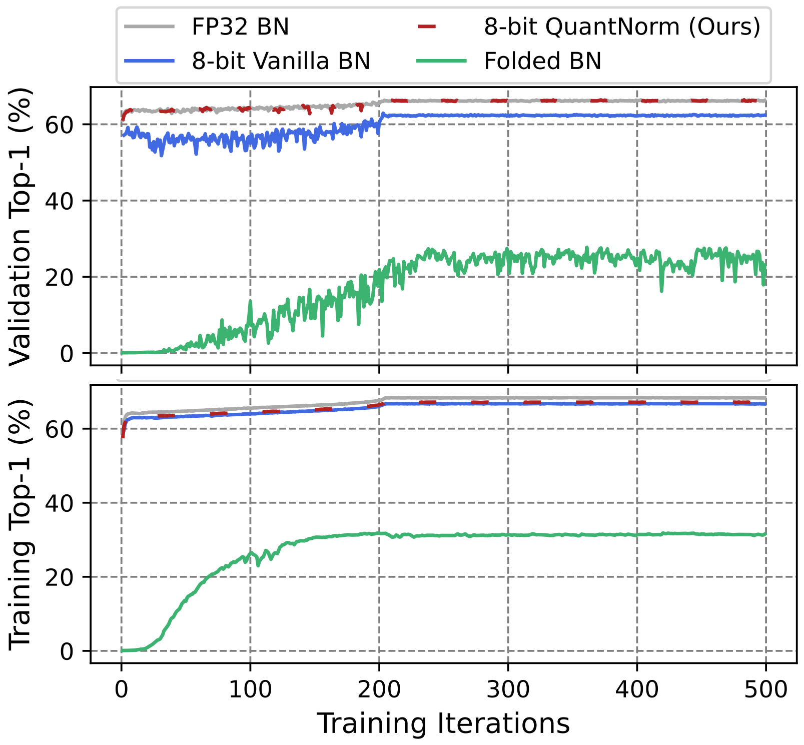


Figure 3.4: Validation (Top) and Training (Bottom) Top-1 Accuracy during QAT of PikeLPN-1× on ImageNet for different Batch Norm Quantization techniques.

Where x is the input feature map and the batch norm parameters μ, γ, σ, β are represented as floating-point values. To avoid performing floating point multiplications and additions, those parameters need to be quantized as follows:

$$Qbatchnorm(x) = \frac{(x - Q(\mu)) * Q(\gamma)}{\sqrt{Q(\sigma)^2 + \epsilon}} + Q(\beta) \tag{3.5}$$

Computation folding is a commonly used approach to reduce the overhead of batch normalization operations in quantized models (i.e., mainly in 8 bit models) [74]. However, the batch normalization parameters (i.e., μ, γ, σ, and β) have to be quantized to the same precision of the preceding convolution layers to enable folding. Doing that in low-precision models (i.e., 4 bits or lower) leads to a significant loss in accuracy as shown in Figure 3.4. That is why previous low-precision model research [177, 124, 127] excluded batch normalization layers from the quantization process, where they keep the batch norm

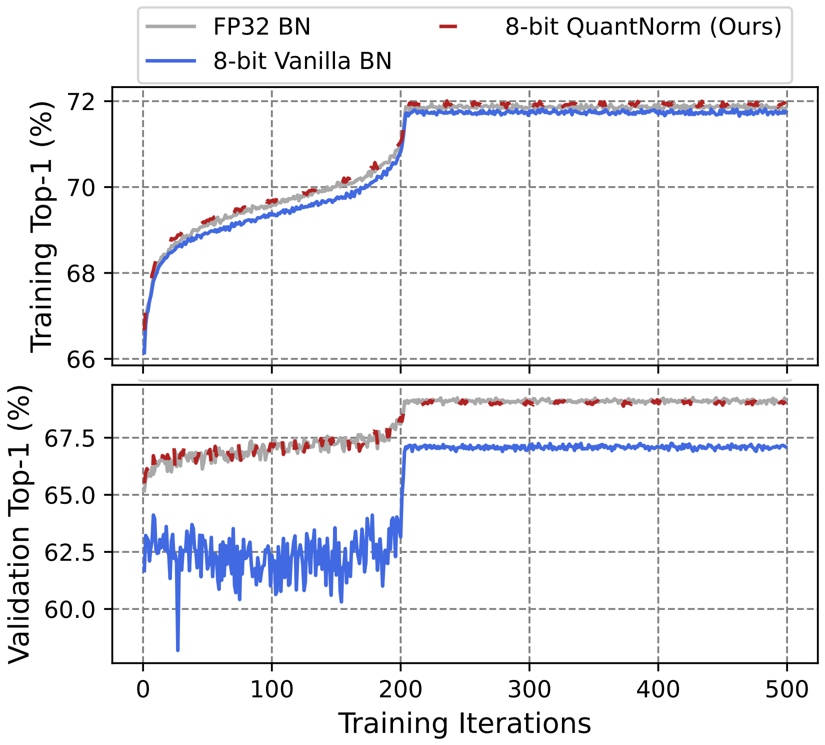


Figure 3.5: Validation (Top) and Training (Bottom) Top-1 Accuracy during QAT of PikeLPN-2× on ImageNet for different Batch Norm Quantization techniques.

parameters as floating point numbers. However, as we showed earlier in Table 3.2, the non-quantized batch normalization operations can add up to 40% overhead to the model's ACE_{v2} cost.

Another solution is to quantize the batch normalization parameters at a higher precision. Figure 3.4 shows the validation accuracy curve during training when batch normalization parameters are represented as INT8 values (denoted as *8-bit Vanilla BN*). Although the accuracy is better than the folded batch norm, we can still notice some degradation in accuracy compared to non-quantized batch norm layers. To minimize the accuracy loss, we propose a novel *QuantNorm* layer. In our *QuantNorm* layer, we re-write the batch norm quantization operation as shown in Equation 3.6 where we first multiply by a quantized scale s, then add a quantized bias b. s is represented as the quantized division between the γ and σ parameters as shown in Equation 3.7. Using *QuantNorm* helps reduce

Table 3.7: Comparison of PikeLPN variants' training parameters, with dropout rates calibrated to mitigate overfitting. Each model's training duration and learning rate strategy are customized according to its complexity. They are initialized with weights from an floating point PikeLPN model, employing a consistent learning rate of 10^{-12} during the tail period to enhance stability and validation accuracy, crucial for smaller models.

PikeLPN Size	**1×**	**2×**	**3×**	**6×**
$\boldsymbol{ACE_{v2}}$ ($\times 10^9$)	8.68	15.74	33.97	59.10
Channel Multiplier	1.0	1.0	1.5	2.0
Activation precision (int bits, frac bits, sign bit)	(6, 1, 1)	(8, 7, 1)	(8, 7, 1)	(8, 7, 1)
Removal of BN layers between depthwise and pointwise convolutions	Yes	No	No	No
Constant learning rate tail period (epochs)	300	300	20	50
Training epochs	500	1500	1000	1000
Dropout rate	1e-3	1e-3	0.5	0.7

quantization error by allowing high precision division in the scale s computation during training. As shown in Figures 3.4 and 3.5, our *QuantNorm* layer maintains close-to-FP accuracy without any extra costs compared to vanilla quantization for batch norm layer. After training, we pre-compute s to avoid high precision division during inference.

$$Qbatchnorm(x)_{improved} = x * s - b \tag{3.6}$$

$$s = Q(\frac{\gamma}{\sqrt{\sigma^2 + \epsilon}}) \tag{3.7}$$

$$b = Q(\beta) - Q(\mu) * s \tag{3.8}$$

Model Scaling: To generate a Pareto family of models, we scale the number of output channels as practiced in the MobileNetV1 model [65]. We also scale the precision of the input activation to the pointwise convolution layers in the PikeLPN block. We present four versions of *PikeLPN* arachitecture: PikeLPN-1×, 2×, 3×, and 6×. The details for each model are described in Table 3.7. For nomenclature, the scale factor represents the ACE_{v2} cost of the scaled model compared to that of the smallest model. For example, PikeLPN-3× has approximately 3 times the ACE_{v2} cost of PikeLPN-1×.

3.4 Experiments

3.4.1 Implementation and Training

All models are implemented using QKeras [25], then we performed Quantization-aware training (QAT) [74]. We train and evaluate the *PikeLPN* family of models on the ILSVRC12 ImageNet classification dataset [30]. To train our low-precision models, we follow a multi-phase training approach. We first train the full-precision model, then we quantize the model as explained previously in Subsection 3.3.4, and train for another 500 epochs. All Models are trained with an effective batch size of 256 using an *AdamW* optimizer and a Cosine Decay schedule. We use label smoothing regularization with cross-entropy loss and a smoothing factor of 0.1 for all models. The initial learning rate is $1e-4$ and annealed using a cosine schedule to $1e-12$. Detailed training parameters are shown in Table 3.7. We use standard augmentation techniques like resizing, cropping, and flipping. At test time, all *PikeLPN* models are evaluated on images of resolution 224×224.

3.4.2 Results

To evaluate the accuracy-efficiency trade-off by *PikeLPN*, we compare its performance to state-of-the-art low-precision models. Figures 3.6 and 3.7 show that *PikeLPN* establishes the SOTA Pareto frontier for low-precision and binary models in terms of arithmetic energy consumption and ACE_{v2} cost respectively. Table 7.3 compares *PikeLPN* to SOTA low-precision models in terms of Top-1 Accuracy on ImageNet, Energy consumption

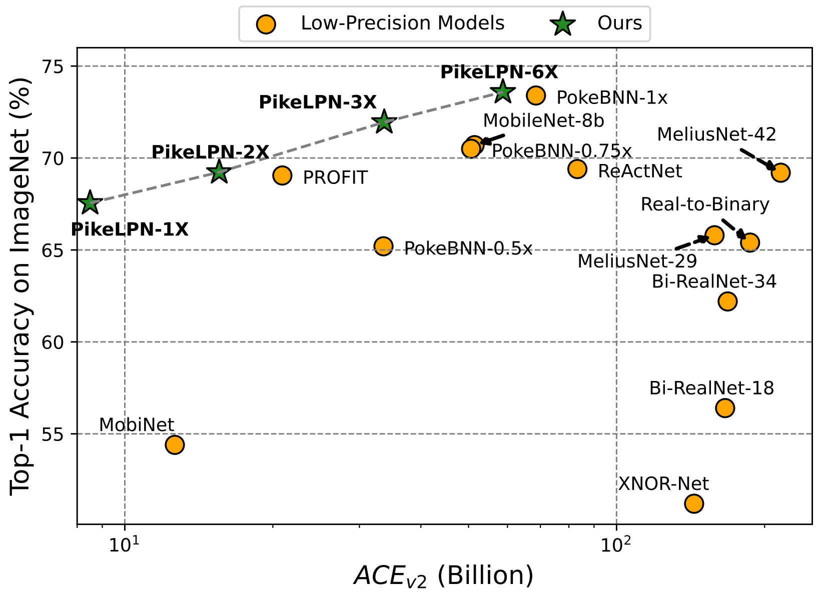


Figure 3.6: Accuracy vs ACE_{v2} of PikeLPN and SOTA low-precision neural networks. ACE_{v2} is an efficiency metric that estimates the cost of arithmetic operations during inference.

in *millijoules*, ACE_{v2}, and Arithmetic Intensity. We clearly see how the elementwise operations dominate (i.e., 31 up to 93%) the ACE_{v2} cost for other low-precision models. On the other hand, *PikeLPN* carefully quantizes the elementwise operations reducing their contribution to the total energy consumption to less than 5%. Additionally, *PikeLPN*-1× is 1.5× more efficient in terms of both ACE_{v2} and arithmetic energy consumption compared to *MobiNet* [127] (i.e., A binary version of MobileNetV1 with added skip connections) while achieving **13.2%** higher Top-1 Accuracy on ImageNet. Moreover, *PikeLPN*-3× achieves 1.5% higher Top-1 accuracy than PokeBNN-0.75× [177] (i.e., A binary ResNet-50 with parameterized activation functions) while being 35% more efficient. In terms of arithmetic intensity, *PikeLPN* shows a much higher arithmetic intensity when compared to other low-precision models, this is mainly due to the absence of any skip connections. As mentioned earlier in Section 3.3.3, high arithmetic intensity is advantageous as it suggests a greater proportion of computational operations per data element, which can lead to reducing the memory reads and writes by the model; hence reducing the overall energy consumption during inference.

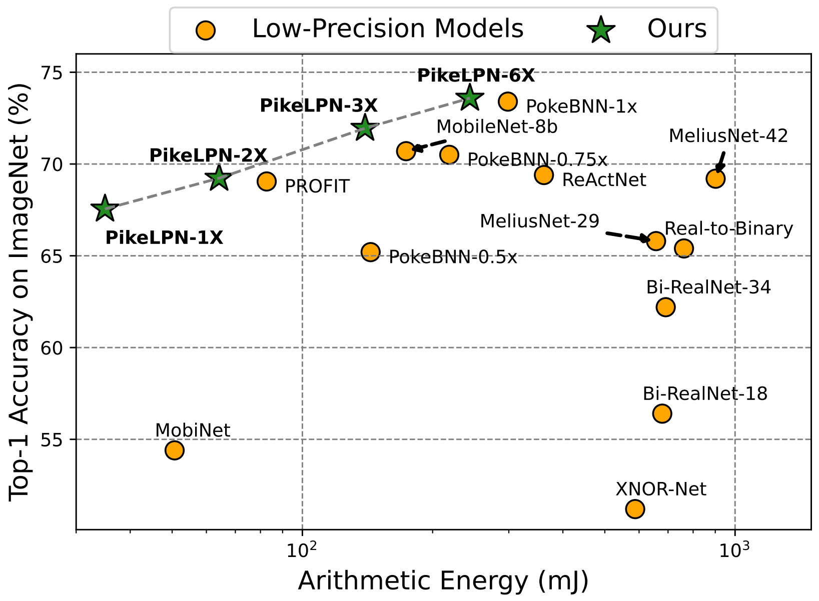


Figure 3.7:]
Accuracy vs Energy Consumption by the arithmetic operations of our *PikeLPN* and SOTA low-precision neural networks.

Moreover, to highlight the efficiency improvements in *PikeLPN*, Figure 3.8 illustrates the contribution of MAC versus elementwise operations to ACE_{v2} for PikeLPN-1× and PokeBNN-0.5× [177]. Since, PokeBNN-0.5× quantize the convolution layers to 1-bit reducing the cost of multiply-accumulate (MAC) operations, it shifts the energy burden to the elementwise operations within these remaining high-precision layers. Although there are fewer of these elementwise operations, they use more energy because they are still in high precision.

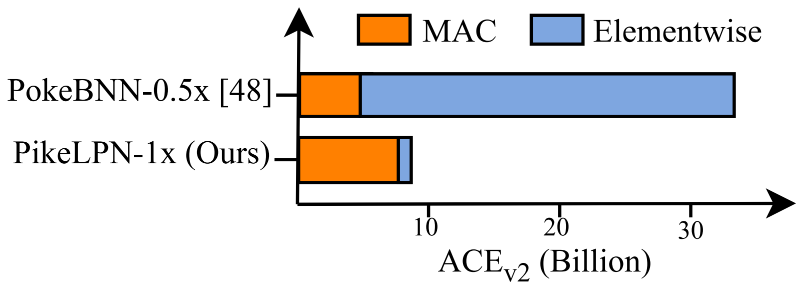


Figure 3.8: Contribution of multiply-accumulate (MAC) versus elementwise operations to the efficiency metric ACE_{v2} for PikeLPN-1X and PokeBNN-0.5X [177]. *PikeLPN* selectively increases the precision of MAC operations which allows for effectively quantizing elementwise operations, achieving 3× more efficiency while being 2% more accurate on ImageNet.

Table 3.8: Results – *PikeLPN* versus SOTA low-precision models in terms of Accuracy and Efficiency Metrics. ACE_{v2} is measured according to the definition in Section 3.3.2. The fourth and fifth columns show the contribution to the overall ACE_{v2} cost by multiply-accumulate and elementwise operations respectively. *Energy* represents the arithmetic energy according to $45nm$ CMOS technology according to table 3.1. *Arithmetic Intensity* is an indication for the memory reads and writes required by the model as explained in Section 3.3.3. *Used Precisions* represent the the precision of the various operations in the mixed-precision models.

Model	**Accuracy**	**Arithmetic Computational Effort (ACE_{v2})**			**Energy**	**Arithmetic Intensity**	**Used**
	(%)	Total ($\times 10^9 \downarrow$)	MAC (%)	Elementwise (%)	($mJ \downarrow$)	(Ops/Element $\uparrow$)	**Precisions**
XNOR-Net [130]	51.2	143.78	-	-	587.69	-	32, 1
MobiNet [128]	54.4	12.64	13.17	86.83	50.66	28	-
Bi-RealNet-18 [102]	56.4	166.26	-	-	678.75	-	32, 1
Bi-RealNet-34 [102]	62.2	168.11	-	-	691.47	-	32, 1
MobileNet (8W, 4A) [85]	64.0	33.8	68.96	31.04	118.54	39.57	32, 8, 4
MobileNet (4W, 8A) [85]	65.0	33.8	68.96	31.04	118.54	39.57	32, 8, 4
Real-to-Binary Net [110]	65.4	186.85	-	-	762.24	-	32, 1
MeliusNet-29 [10]	65.8	158.21	-	-	656.81	-	32, 1
PokeBNN-0.5x [177]	65.2	33.58	4.18	95.81	143.78	24.5	32, 8, 4, 1
PikeLPN-1× (Ours)	**67.55**	**8.50**	96.38	3.62	**34.98**	39.57	8, 4
PROFIT [124]	69.05	20.91	47.51	52.49	82.70	39.57	32, 4
MeliusNet-42 [10]	69.20	215.71	-	-	901.82	-	32, 1
PikeLPN-2× (Ours)	**69.23**	**15.56**	97.87	2.13	**64.20**	39.57	16, 8, 4
ReActNet [104]	69.4	83.24	26.78	73.22	361.63	36.75	32, 1
PokeBNN-0.75x [177]	70.5	50.61	5.11	94.88	218.51	40.48	32, 8, 4, 1
MobileNet (8bit) [85]	70.7	51.44	79.61	20.39	173.68	39.57	32, 8
PikeLPN-3× (Ours)	**71.95**	**33.70**	98.52	1.48	**139.59**	**52.66**	16, 8, 4
PokeBNN-1x [177]	73.4	68.56	6.16	93.83	298.44	40.48	32, 8, 4, 1
PikeLPN-6× (Ours)	**73.59**	**58.74**	98.87	1.13	**243.85**	**63.38**	16, 8, 4

3.5 Conclusion

Our investigation into SOTA low-precision models uncovered overlooked efficiency bottlenecks, particularly noting that operations traditionally considered negligible—such as elementwise operations in activation functions, batch normalization, and quantization scaling can contribute up to 90% of the inference cost. Addressing these challenges, we proposed ACE_{v2} which extends the efficiency metric ACE to better reflect the inference cost of low-precision models. Moreover, we introduced *PikeLPN*, a novel family of models that quantizes both elementwise and multiply-accumulate operations. Specifically, we propose (a) a novel *QuantNorm* layer for effective batch normalization quantization, (b) *Double Quantization* where quantization parameters are also quantized, and (c) *Distribution-Heterogeneous Quantization* for Separable Convolution layers to tackle their distribution mismatch problem. *PikeLPN* achieves up to a threefold reduction in inference cost over existing low-precision models while improving the Top-1 accuracy in ImageNet dataset.

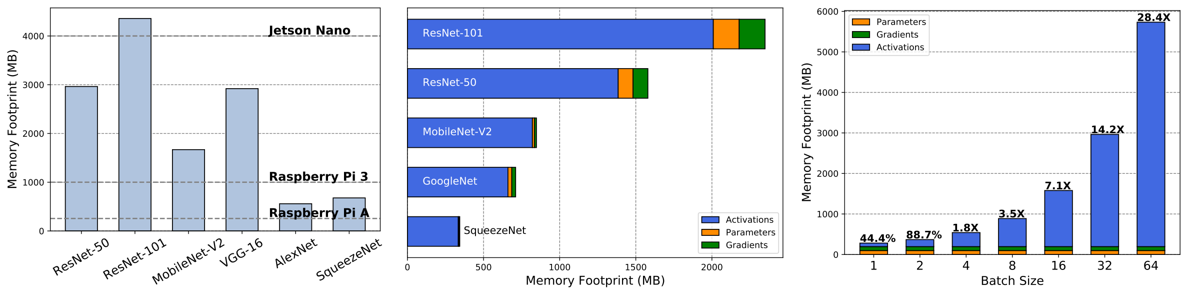
CHAPTER 4

Efficient Training

4.1 Introduction

Over the past decade, deep learning has achieved unprecedented successes in various domains. Researchers have realized the benefits of deploying deep learning models on edge devices; therefore, they started to develop techniques to make them more resource efficient [51]. However, only deploying the pre-trained models on edge devices is not sufficient. Edge devices are continuously collecting rich and sensitive data. This new data can be used to fine-tune those models which would significantly improve their performance and their adaptive capability to new environments.

A prime example for on-device learning is model personalization. With advances in digital services, large tech companies strive to make their services as unique to each of their users as possible. For example, personal assistants such as Siri (Apple), Alexa (Amazon), Cortana (Microsoft), and Google Assistant recognize the voice and the accent of their owner, and learn to not recognize other voices after their initial setup. Human activity recognition [94], health applications [141] and smart home appliances [81] also demand model personalization to improve user satisfaction. Model personalization defeats the purposes of *generalizability*, which is the primary metric for the performance of deep



(a) (b) (c)

Figure 4.1: Memory Footprint. (a) Training Memory Footprint at Batch Size 32. (b) Memory Footprint Components for MobileNet-v2, ResNet-50, and Resnet-101 with Batch Size 16. (c) Memory footprint using different batch sizes for ResNet-50.

learning models. Training on the edge not only makes model personalization more feasible, but also keeps data private and safe.

Current approaches send the data to cloud servers in order to execute training epochs to fine-tune the models. After that, updated versions of the models are deployed on the edge devices. However, this approach risks the privacy of the data which could be sensitive, such as medical data that are protected by HIPAA regulations [156]. Moreover, continuously syncing data requires a huge network bandwidth. For example, traffic surveillance cameras deployed at every street intersection in a city would require sending gigabytes of data to the cloud everyday, which is extremely expensive. Furthermore, it can be even unfeasible due to weak or limited internet connection as it is the case in remote agricultural lands [90], or even in space exploration missions (e.g. Mars Rovers) [137, 157]. That is the reason why on-device learning is essential to push the limits of edge capabilities.

On-device learning is significantly challenging due to the energy, compute, and memory constraints on the edge devices. Some work has started exploring training on the edge [51, 97]. Memory footprint is one of the main challenges for training on the edge. Figure 4.1-a shows the memory required for training some of the modern computer vision models. We observe that even the memory of a Jetson Nano board is insufficient for training an average state-of-the-art model [142]. During training, memory has three main components: (i) the model parameters, (ii) the activations for each layer computed during the forward pass, and (iii) the gradients computed during the backward pass. In Figure 4.1-(b), we see that the memory for activations is the dominant factor. In addition, the memory footprint increases significantly as the batch size increases as shown in Figure 4.1-(c). However, little work has been devoted in optimizing the activations memory with respect to the amount of work invested in optimizing the parameters memory. The main reason is that model optimization has been the main use case for deploying models for inference on the edge devices. To solve this problem, [89] proposed retraining the fully connected layers only, while Cai *et. al.* [17] suggest training the biases and the fully connected layers only, while freezing all the weights. In other words, the idea is to not save the activations because they are only needed to compute the weights gradients, which is partially discarded in their proposals. However, this limits the network's capacity to learn from the new data, and reduces the accuracy gains from retraining. Having the flexibility to tune all the weights of a model maximizes the benefits of on-device learning.

Other researchers have explored general techniques to reduce the memory footprint of training regardless of the used hardware. Model parameters can be sparsified throughout training, which reduces the number of model parameters and gradients, leaving the activations memory unaffected [118]. Using half precision [113] and reducing the batch size [70] have a direct impact on reducing the memory footprint of training, and improve parallelization. However, these techniques introduce a toll on the accuracy of the pre-trained models. Furthermore, checkpointing reduces the memory footprint during training by only storing the activations of a subset of layers, and recomputing the needed layers again during backpropagation [21]. This provides a trade-off between the memory footprint and the number of floating-point operations (FLOPs). Nonetheless, this method is targetted towards training deeper models on server-scale GPUs.

In this work, our goal is to reduce the memory footprint of model training on the edge without affecting the accuracy. The rationale is that we opt for training on the edge to improve the accuracy of the already trained models, while complying with constraints (e.g. data privacy, cloud connectivity, bandwidth). That raises a fundamental question “How much memory is needed for training?”, and more specifically “How much memory is needed to store the activations?”. By analyzing the activations, we found that activations by nature are sparse. More than 70% of the stored activations are zeros due to ReLU non-linearity which is used in most neural network models. We leverage this observation to make on-device learning more feasible.

In BitTrain, we propose to detach the activation storage from their involvement in computations. This allows us to compress the activations for later use during the backward pass. We also introduce activations pruning which can further increase the memory savings while producing a memory accuracy trade-off. Our contributions can be summarized as follows:

- In modern deep learning frameworks, we detach the activations storage from the *Tensor* representation in computation graphs[1], allowing us to address the memory footprint issues of neural network activations.
- We present BitTrain, a novel Bitmap Sparse Compression method to efficiently store the activations with negligible computational overhead, and with no change to the underlying computation graph. By construction, our compression is safe and has no negative impact on the model accuracy.
- We analyze the theoretical and empirical memory reduction by using our method. Experimental results show that we can achieve up to 34% memory saving at a sparsity level of 50% per convolution activation. Combining our method with existing work that increase sparsity, we can achieve up to 56% memory saving at a sparsity level of 75%.
- Since BitTrain is orthogonal to existing methods, we study the effect of combining our method with existing techniques, namely: low-precision training, activation

[1]Computation graphs are how modern deep learning frameworks represent neural networks for both training and inference.

pruning, and checkpointing. Then, we discuss how each combination affects the amount of memory required for training.

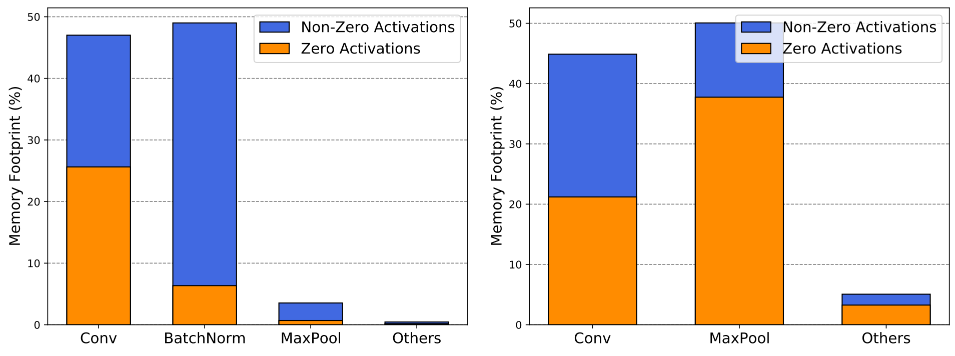
The rest of the chapter is organized as follows. In Section 4.2, we give a brief background on the most common sparse data formats in literature. Then, we present our methodology for reducing the memory footprint in Section 4.3. In Section 4.4, we present a detailed theoretical and empirical analysis of our method. Moreover, we investigate the gains from combining our method with existing techniques. Finally, we conclude our study in Section 4.5.

4.2 Background

Sparse Data Formats. For highly sparse matrices, storing the non-zero elements and their indices is more efficient than storing all the values in a dense format. The most popular format is the Coordinate list format (COO). It stores each non-zero value (floating-point) along with its n-dimensional indices (fixed-point). Modern deep learning frameworks offer efficient implementations for the COO format. However, this format is optimized for reducing the number of matrix operations, and is only memory-efficient if the matrix has a high sparsity ratio (i.e., 80%); otherwise, the dense format would consume less memory.

4.3 Leveraging Sparsity for Memory-Efficient Training

Deep Learning frameworks represent activations for Convolutional Neural Networks as four-dimensional matrices. In modern deep learning frameworks, they are represented in a dense matrix format of dimensions (*batch_size*, *num_channels*, *height*, *width*) where *batch_size* is the batch size used during either training or testing, *num_channels* is the



(a) ResNet-50 (b) AlexNet

Figure 4.2: Activations Memory footprint for different layers in ResNet-50 and AlexNet.

number of channels at any given layer, *height* and *width* are the height and the width of the activation maps respectively. The goal of BitTrain is to reduce the total memory footprint required for training a model. In the following, we derive the foundations of our method and describe the implementation details.

4.3.1 Activation Sparsity in Neural Networks

Exploiting Sparsity. In the modern deep learning frameworks (i.e. PyTorch and TensorFlow), the input activation for each layer is stored during the forward pass, then those activations are used for calculating the gradients during the backward pass. Storing the activations for all the layers creates a huge memory footprint during the training process as illustrated earlier in Figure 4.1b. However, most of modern deep learning models use the rectified linear activation function (ReLU) [3]. Using ReLU activation functions results in sparse activations in the successive layers. We can leverage this sparsity to compress the activations, and hence reduce the memory footprint for training.

In Figure 4.2, we analyze the contribution of the activations of various layers types (e.g. Convolution, Batch Normalization, Max Pooling) to the memory footprint, as well as the average activation sparsity for those layer types throughout the whole models. Figures 4.2a and 4.2b shows that the memory used to store the activations of the different layers types in ResNet-50 and AlexNet respectively. For ResNet-50, we notice that the

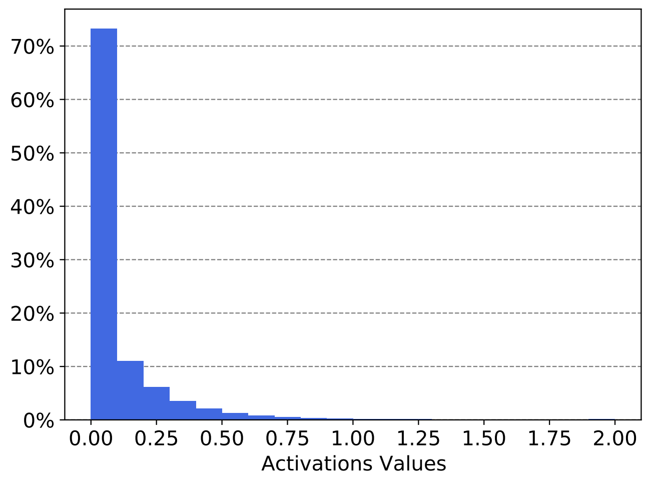


Figure 4.3: Activations values histogram for ResNet.

memory used to store the convolution and the batch normalization activations dominates the memory used by the other layers. While for AlexNet, the convolution and the max pooling layers activations dominate the memory. That means that we should direct our efforts towards reducing the memory used to store the activations of those three layers (convolution, batch normalization, max pooling).

From Figure 4.2, we can also notice that the activations sparsity is relatively high for the convolution and the max pooling layers. However, it is not a surprise that the activations sparsity for the batch normalization layer is very low. That means that using a sparse representation for the batch normalization activations will not offer any reduction in the memory footprint. To overcome this problem, we apply the double mask batch normalization introduced by Lieu *et al.* [97]. The idea is to use the sparse pattern of the the inputs to the normalization layer as a mask to apply for its outputs. In other words, it propagates sparsity through the batch normalization layer and helps our method reduce more memory.

Moreover, we analyze the activation values throughout the whole network. Figure 4.3 shows a histogram of the activation values for all the layers of a pre-trained ResNet model. We notice that more that 70% of the activations are close to zero. This implies that we can neglect storing those activations (i.e. we can assume that they are zeros), and hence increase the activations sparsity which would further increase the gain from our compression methodology.

Memory vs. Matrix Operations. In server-grade model training, a large GPU memory can accommodate model parameters as well as activations calculated during the forward pass. On edge devices, memory is not only a scarce resource, but may also be shared between the main CPU and the GPU (if exists). For example, the Nvidia Jetson Nano board houses an ARM Cortex-A57 MPCore processor that shares a 4GB memory with a Maxwell-based GPU that performs up to 4 floating-point operations per clock cycle [7]. External memory access has an interrupt latency of at least 200 clock cycles assuming zero wait state [167] – a figure that is empirically higher depending on the system load and the cache status. Due to the limited memory available, convolution activations from earlier layers will be offloaded to disk (using virtual memory pages), since they are the least recently used. If no swap memory is available, the training process will be killed by the operating system.

Building on the research done on checkpointing (where activations are not saved at all; instead re-calculated), we propose to trade expensive matrix multiplications for cheaper memory operations. This trade-off is analogous to *Memoization* in algorithmic contexts [9]. In essence, we take advantage of sparsity and save input activations in a compressed format that leaves more memory for the following dense operations to be performed. Although the compression and decompression processes add operations to the training loop, they save the disk access time resulting from memory swapping.

4.3.2 Sparse Bitmap Format

Computation Graphs. Modern deep learning frameworks (e.g. Tensorflow [1] and PyTorch [126]) have offered an adequate level of abstraction for training deep learning models. A developer can now imperatively describe the architecture of their neural

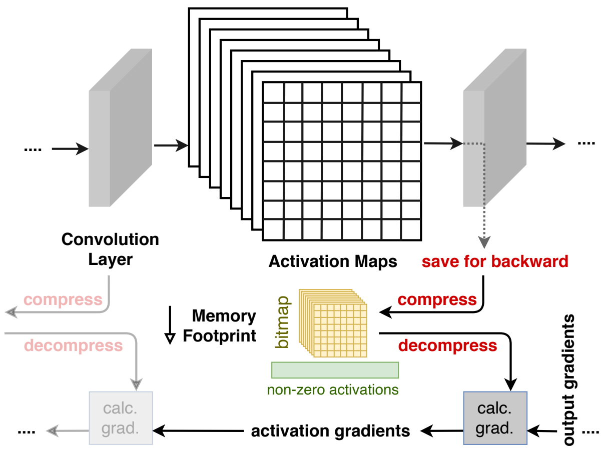


Figure 4.4: Compressing dense activations in a bitmap.

network, and the framework takes care of the compiling code to lower-level constructs that work efficiently with different hardware interfaces (especially GPUs). To make this happen seamlessly, these framework construct *a computation graph* that can be used to track and execute the necessary elements of the backpropagation algorithm (in a process called auto-differentiation [125]). The computation graph can be either static or dynamic. In a static computation graph, the neural network is constructed once in the beginning, and then gets attached to a training session. In this case, memory occupied by the sizes of its tensors (i.e. matrices) is reserved in the beginning. On the other hand, dynamic computation graphs get built dynamically, reserving memory for tensors immediately after declaring them, and releasing them when they go out of scope. This distinction is important in our work, since there is no memory management APIs offered by these frameworks in their Python interfaces. Optimizing memory has to be implemented at the lower level (using C++), which we describe later in Section 4.4.

Bitmap Format. As discussed in Section 4.2, there is a plethora of sparse matrix representation formats. These formats are mainly designed for both storage and operations. In other words, mathematical operations such as multiplication, division, and inverse are defined and computationally efficient. We adopt a bitmap format that is optimal for storage as shown in Figure 4.4. During the forward pass, input activations for a

Algorithm 1 Dense to Bitmap Matrix Compression

```
Input   : Dense Activation Matrix ($T$)
Output  : Bitmap Matrix ($B$)
$B$.shape = $T$.shape // deep copy
Flatten $T$
for $i$ in $0, .., length(T)$ do
    if $T[i] == 0$ then
        Push 0 bit to $B$.bitmap
    else
        Push $T[i]$ to $B$.values
        Push 1 bit to $B$.bitmap
    end
end
delete $T$ // free dense memory
```

given layer is compressed into: (i) a vector containing the non-zero elements, and (ii) a bitmap that sets a bit to 1 at the indices of the non-zero elements. In the backward pass, these activations are decompressed in order to calculate the gradients with respect to the activations as part of the backpropagation algorithm. The bitmap format represents the minimum perceivable memory required to store the information in a matrix; that is non-zero elements (represented as half-, single- or double-precision) and a single bit for each element index. We denote the memory footprint as M_d and M_b for stashing activations in a dense format and bitmap format respectively. For single-precision (FP32), the memory footprint (in bytes) would be calculated as:

$$M_d = 4\times \text{ total activations}$$
$$M_b = 4\times \text{ non-zero activations} + (1/8) \times \text{ total activations}$$

We also compare the bitmap format with the COO format, which represents indices as either integers (4 bytes) or longs (8 bytes). We denote the memory footprint (in bytes) of the COO format as M_c, and it can be calculated as:

$$M_c = (4 + [4|8]\times \text{ num-dimensions }) \times \text{ non-zero activations}$$

where 4 is the size of single-precision for saving the activation values, and $[4|8]$ are the sizes for either integer or long indices. For example, PyTorch and Tensorflow use long indices by default in their COO implementations. Figure 4.5 shows that the COO format is only

Algorithm 2 Bitmap to Dense Matrix Decompression

```
Input   : Bitmap Matrix ($B$)
Output : Dense Activation Matrix ($T$)
Construct 1-dimensional $T$; initialize $j = 0$
for $i$ in $0, .., length(B).bitmap$ do
    if $B.bitmap[i]$ is set then
        Push $B$.values[j] to $T$; increment j
    else
        Push 0.0 to $T$ // half-, single- or double-precision
    end
end
reshape $T$ to $B$.shape
delete $B$ // free bitmap memory
```

efficient if the activations have a sparsity of at least 80% (20% non-zero activations). We observe that the dense representation consistently maintains a low memory footprint. However, the proposed sparse bitmap format can reduce the memory footprint even if the activations matrix has low sparsity. Unlike other sparse matrix formats, the sparse bitmap format is used for stashing the activations until they are needed in the backward pass, and not for directly operating on them (e.g. multiplication).

4.3.3 Sparse Bitmap Compression Algorithm

In order to achieve empirical memory reductions, our algorithm avoids copying matrices in function calls. Compressing a dense matrix to a bitmap matrix is outlined in Algorithm 1. We first start by keeping a copy of the shape (dimension sizes) of the original matrix. Operating on a vector is also necessary to avoid complex index resolution operations, so we flatten the dense matrix. This is an in-place operation that does not move values to different memory locations. Lines 3 to 10 scans the vector, constructing the bitmap and stashing the non-zero elements. Finally, line 11 frees the memory used by the dense matrix to be used in the subsequent layers. Algorithm 2 performs the opposite operation in the backward pass. In particular, it maps the non-zero elements to a dense vector matching the bitmap, and then reshapes the data (in-place) according to the previously saved shape.

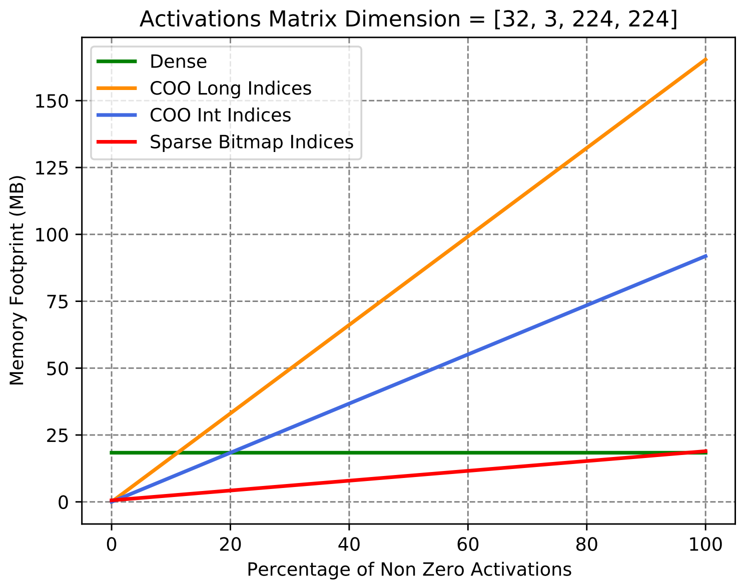


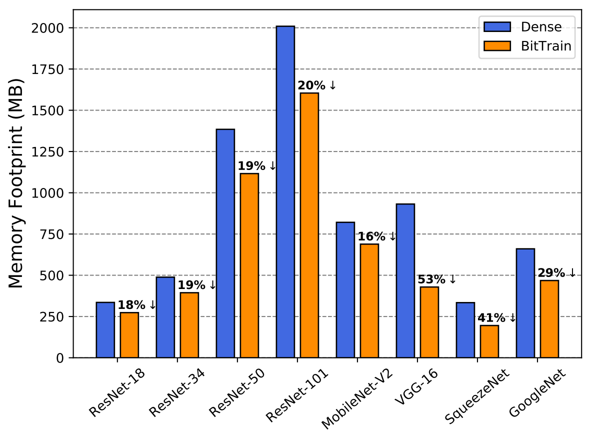
Figure 4.5: Memory footprint of our proposed bitmap format as compared to other sparse matrices.

Note that both algorithms are linear runtime. They take $O(n)$ time, where n is the total number of activations in a matrix. In the next section, we provide additional implementation details for a modern deep learning framework.

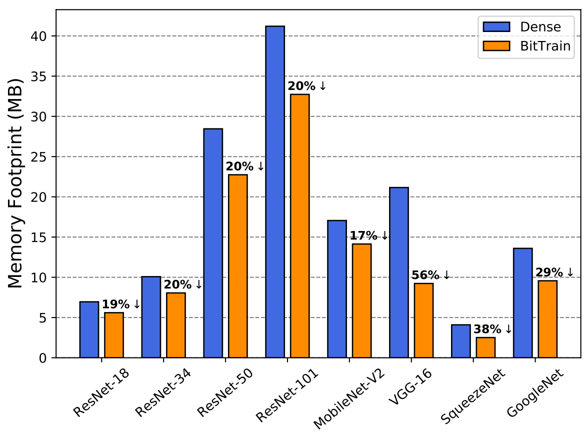
4.4 Experiments

Our experimental analysis tests the hypothesis of memory reduction using the sparse bitmap format both theoretically and empirically. First, we analyze the memory footprint reduction in state-of-the-art CNN-based architectures using both ImageNet [31] and CIFAR 10 [86] image datasets. Afterwards, we study the compound reduction in training memory footprint when combining BitTrain with other orthogonal training methods such as low precision training, activation pruning, and checkpointing. Finally, we present our own implementation of BitTrain, and analyze the on-board memory footprint reduction as well as the runtime overhead.

Setup. In our experiments, we use PyTorch version 1.7, and in the C++ implementa-



(a)



(b)

Figure 4.6: Activations Memory footprint for training different classification models on (a) ImageNet (b) Cifar10.

tion, we use libtorch version 1.7. We use Clang version 10.0.0 as the compiler, and compile using C++14 standards. Empirical memory footprint measurements are performed on Nvidia Jetson Nano board that has 4GB of memory. More details on the implementation and memory measurements is provided below.

Classification Models. Our sparse bitmap format reduces the memory footprint for storing activations with high sparsity as previously illustrated in Figure 4.5. To access the overall impact of using our proposed methodology on the memory footprint during training, we choose 8 state-of-the-art classification models that vary in size and complexity. We analyze the training memory footprint of those models for two different classification datasets: CIFAR-10 (Input resolution is 32×32), and ImageNet (Input resolution is 224×224). Figure 4.6 shows the memory footprint of BitTrain in comparison to classical training (referred to as *Dense*). Figures 4.6a and 4.6b represent training different classification models on ImageNet, and CIFAR-10 respectively. The results show that BitTrain reduces the memory footprint by up to 56%, this improvement is achieved by leveraging the activations sparsity that are naturally found in those models. We can notice that the improvements tends to be higher for VGG-16, SqueezeNet, and GoogleNet. The reason is that those models tend to have higher sparsity percentages because they do not have any batch normalization layers. In the following Section, we propose using activations pruning to increase the activations sparsity, and hence maximize the memory footprint reduction that could be achieved by our sparse bitmap compression technique.

Combining BitTrain with Low Precision. Neural network parameters are typically represented in single-precision (FP32) [IEEE 32-bit]. Using half-precision (FP16) [IEEE 16-bit] has shown to be sufficient for the general case of training neural network as discussed in Section 4.2. It reduces the memory footprint of all components (model parameters, activations, and optimizer gradients). Low precision training is supported in modern deep learning frameworks, and is as straightforward as specifying the $float16$ data type for all parameters and model inputs. Figure 4.7 shows the memory footprint reduction that can be achieved when combining our bitmap format for saving the activations with half-precision arithmetic. We observe that combining using half-precision activation values with our sparse bitmap compression offers a 55-75% saving in the memory required for storing the activations when compared to the dense full-precision baseline. This means that using our proposed technique achieves up to an additional 25% when used with half-precision than using half-precision alone. As for the accuracy, Sohoni *et al.* [136] show that training with half-precision does not meaningfully affect the final accuracy. It is important to note that half-precision requires native hardware support in order to achieve empirical results [106].

Combining BitTrain with Activation Pruning. As illustrated in Figure 4.3, more than 70% of the activations have values that are close to zero. This means that 70% of the activations are not effective during the training process. We leverage this fact to further increase the activations sparsity which would further improve the memory footprint when using our sparse bitmap compression. In our implementation, we only store the activations that exceed a certain pre-defined close-to-zero threshold. However, if the activation value is less than the threshold, we prune this value (i.e. set it to zero). In Figure 4.8, we analyze the effect of using activation pruning along with BitTrain on the

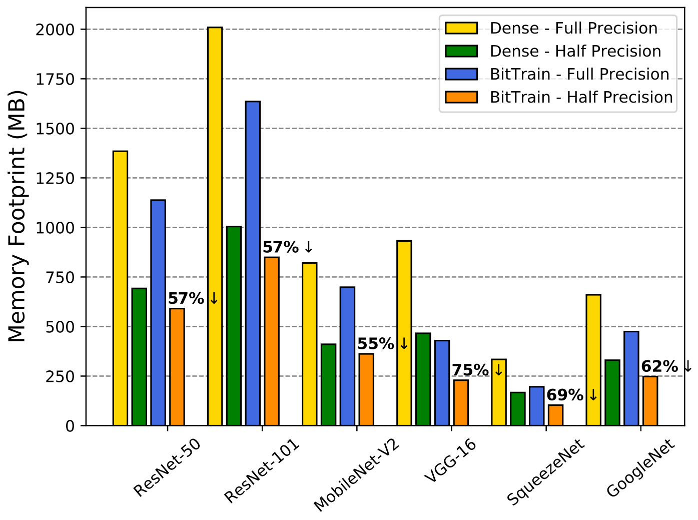
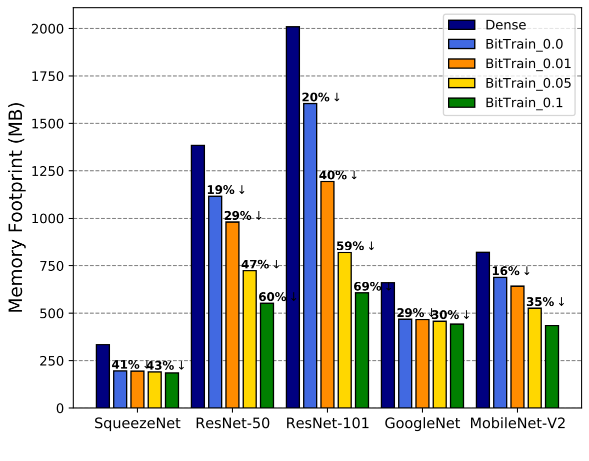


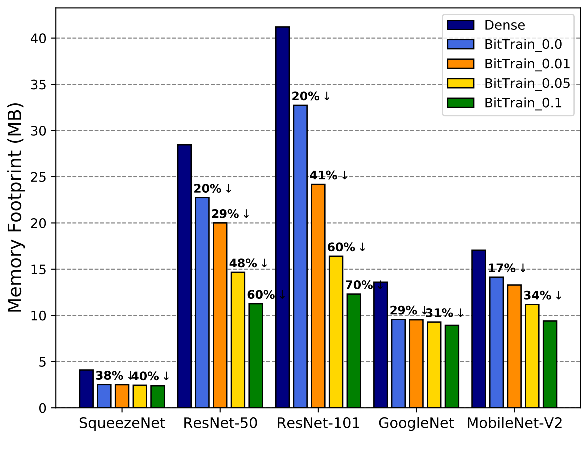
Figure 4.7: Activations Memory footprint for training on ImageNet when using half-precision (FP16) for storing the activations.

training memory footprint. Figures 4.8a and 4.8b show the training memory footprint for five different models on ImageNet, and CIFAR-10 respectively. We analyze the memory savings using different activation pruning thresholds (0, 0.01, 0.05, 0.1). The results shows that memory footprint reduction increases as the activation pruning threshold increases. This is expected because increasing the activation pruning threshold increases the percentage on zero elements in the model, which maximizes the gains from using our sparse bitmap compression technique. We can notice that the gains from using activation pruning varies from one model to another depending on the percentage of close-to-zero activations in the model. For example, using activation pruning with ResNet-50 and ResNet-101 achieves up to an additional 49% reduction in memory footprint, while it provide insignificant memory footprint reduction when applied to GoogleNet.

We also analyze the accuracy of using BitTrain along with the activation pruning on CIFAR-10 in Figure 4.9. We can see that the accuracy drop varies between different models. The accuracy drop depends on the significance of the pruned values, and how the model training adapts to the pruning. This creates a memory-accuracy trade-off. For some models like ResNet-50, it might be worth it to trade a negligible loss in accuracy for



(a)



(b)

Figure 4.8: Activations pruning analysis for classification models with different pruning thresholds. In all figures *Bitmap_x* denotes that our sparse bitmap compression is used along with activation pruning with threshold x (a) Training Memory footprint for ImageNet (b) Training Memory footprint for CIFAR-10.

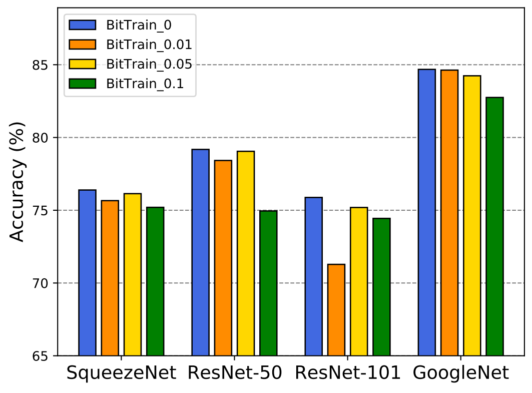


Figure 4.9: Classification Accuracy on Cifar-10 for different models when training under different activation pruning thresholds.

up to a 49% reduction in the memory footprint. However, it might not be worth it for models like GoogleNet, where using the activation pruning achieves a modest reduction in the memory footprint.

Combining BitTrain with Checkpointing. Checkpointing is used in literature to trade computations for memory. The idea is to only store some of the intermediate activations, and re-compute the others during backpropagation. In this section, we implement the checkpointing algorithm, then combine it with BitTrain to analyze the

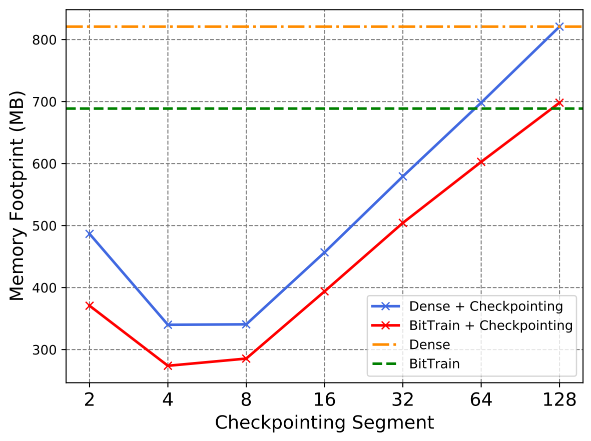


Figure 4.10: Memory footprint reduction for the activations when using checkpointing for partial storage of the activations for MobileNet-V2 when trained on ImageNet.

compound memory savings. We implement the *checkpoint-every-m* checkpointing strategy as it is the commonly-used approach for checkpointing [136]. In the *checkpoint-every-m* strategy, the input activations of every m layers are stored during the forward pass. For example, assume that we have a simple feedforward model with $m \times n$ layers. During the forward pass, we store the activations for one layer every m layers. This divides the model into n segments, where each segment has one layer with stored input activation (i.e., we store the input activations for n layers). During the backward pass, we recompute the activations of all the layers for each segment, and we store them temporarily in order to compute gradients with respect to the layers within the segment. After this, we discard the temporarily stored activations and proceed to the next segment. We combine the *checkpoint-every-m* checkpointing strategy with BitTrain, and analyze the compound effect on the memory footprint as shown in Figure 4.10. Using our sparse bitmap compression, we can achieve up to an extra 25% reduction in memory footprint compared to checkpointing alone.

Implementation Details. High-level languages used in the deep learning frameworks do not provide fine-grained memory management APIs. For example, Python depends on garbage collection techniques the frees up memory of a given object (i.e. tensor or matrix) when there is no references to it [151]. This leaves very little room to the developer in controlling how tensors are stored in memory. Moreover, all data types in Python are of type *PyObject*, which means that numbers, characters, strings, and bytes are actually Python objects that consumes more memory for object metadata in order to be tracked by the garbage collector. In other words, defining bits or bytes and expecting to get accurate memory measurements is infeasible. Therefore, we implemented BitTrain in C++, using *bitset* and *vector* data types from the C++ standard library for storing the bitmap and the non-zero activations respectively. Our implementation extends libtorch's C++ API [126], by defining a tensor that inherits from the default tensor implementation. We chose libtorch because its tensor definition separates the tensor storage from its definition in the computation graph, allowing us to implement Algorithms 1 and 2. Furthermore, it allows our tensor to integrate natively to its dynamic computation graph.

Measuring Memory Footprint. Advances in memory hierarchies (i.e., cache, memory, virtual memory) has made it challenging to measure the exact memory consumed by training a neural network. In addition, deep learning frameworks heavily depend on shared libraries on the host system that can be used by other processes. In BitTrain, we measure the memory footprint using the Unique Set Size (USS) of the running process. USS is the memory that is unique to the process and which would be freed if the process was to be terminated at the moment of measurement. On Linux, we calculate this value by parsing all the private blocks in */proc/pid/smaps*. We note that previous methods described in [17, 97, 118] do not provide implementations, and do not measure the actual memory footprint. Rather, they only present approximate calculations from the PyTorch APIs. Since BitTrain is mainly focusing on edge devices, we show how the implementation is compared to the theoretical estimations.

Activations Memory Reduction. Table 4.1 shows the memory reduction per convolution activations as compared to the calculated results. Although a batch size of 32 is more stable for training [135], we chose a batch size of 16 as a more realistic benchmark for training on the edge. We tested our compression against convolutional

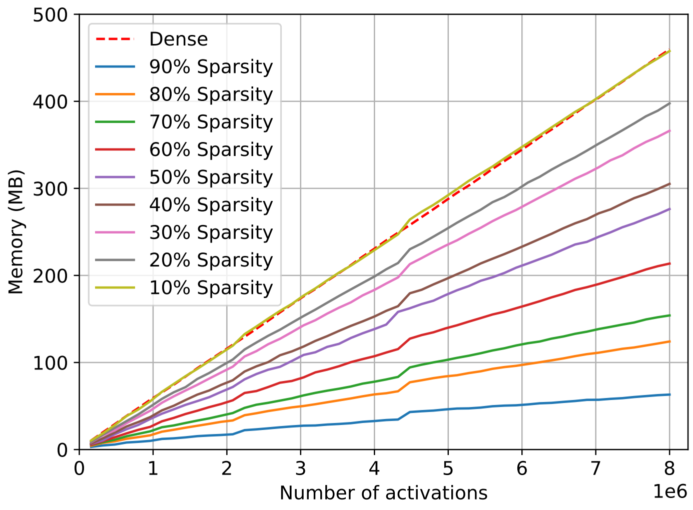


Figure 4.11: On-board (Jetson Nano) memory reduction as a function of the activation size and sparsity level (implementation using libtorch C++ API).

layer sizes (number of channels, width and height of activation maps) in ResNet, which can be representative of many convolution layer sizes in the literature. First, we observe that while the empirical gain deviates from the theoretical calculations, the implementation is still efficient at different sparsity levels. For example, at 50% sparsity, our method achieves up to 34% memory reduction. According to Figure 4.3 in Section 4.3, sparsity can be expected to be more than 70%. In this case, we save activations memory by up to 56% depending on the size of the activations.

Moreover, we analyzed how the bitmap format scales with the increasing number of activations. Figure 4.11 shows that it scales sub-linearly as compared to the saving activations in a dense format. We observe that memory savings is proportional to the number of activations and the sparsity level. This proves that BitTrain is a step towards to enabling training modern convolutional neural networks on edge devices.

Table 4.1: Memory footprint reduction when using our proposed bitmap format for storing convolution activations. Convolution sizes are chosen as they appear in order in the ResNet model. On-board is executed a Jetson Nano board.

Batch Size	**Channels**	**Width**	**Height**	**num elements**	**%non-zeros**	**Dense Tensor (MB)**		**Bitmap Tensor (MB)**			
						Theoretical	**On-board**	**Theoretical**	**Improv (%)**	**On-board**	**Improv (%)**
16	3	224	224	2,408,448	0%	9.19	10.53	0.29	96.88	2.50	76.26
					25%	9.19	10.77	2.58	71.88	5.14	52.27
					50%	9.19	10.83	4.88	46.88	7.47	31.02
					75%	9.19	10.87	7.18	21.88	9.71	10.67
					100%	9.19	10.76	9.47	-3.13	12.03	-11.80
16	7	112	112	1,404,928	0%	5.36	7.17	0.17	96.88	2.67	62.76
					25%	5.36	7.11	1.51	71.88	4.07	42.76
					50%	5.36	6.96	2.85	46.88	5.19	25.43
					75%	5.36	7.4	4.19	21.88	6.97	5.81
					100%	5.36	7.17	5.53	-3.13	8.11	-13.11
16	64	56	56	3,211,264	0%	12.25	13.95	0.38	96.88	3.03	78.28
					25%	12.25	13.95	3.45	71.88	6.1	56.27
					50%	12.25	13.86	6.51	46.88	9.12	34.20
					75%	12.25	13.48	9.57	21.88	11.77	12.69
					100%	12.25	13.85	12.63	-3.13	15.24	-10.04
16	128	28	28	1,605,632	0%	6.13	7.41	0.19	96.88	2.10	71.66
					25%	6.13	7.64	1.72	71.88	4.02	47.38
					50%	6.13	7.38	3.25	46.88	5.31	28.05
					75%	6.13	7.65	4.79	21.88	6.91	9.67
					100%	6.13	7.92	6.32	-3.13	8.83	-11.49
16	256	14	14	802,816	0%	3.06	4.80	0.10	96.88	2.36	50.83
					25%	3.06	4.61	0.86	71.88	3.16	31.45
					50%	3.06	4.99	1.63	46.88	4.39	12.02
					75%	3.06	4.95	2.39	21.88	5.11	-3.23
					100%	3.06	4.33	3.16	-3.13	5.24	-21.02

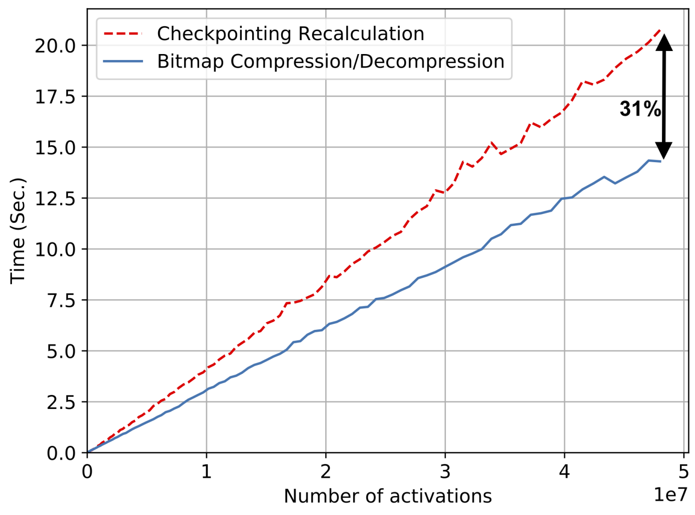


Figure 4.12: On-board (Jetson Nano) runtime reduction of bitmap compression/decompression vs. recalculating activations (checkpointing) at 50% sparsity.

Runtime. We analyzed the runtime savings that the memory operations (from bitmap compression and decompression) offer when compared to the expensive matrix multiplication operations for one layer convolution. Figure 4.12 shows that the bitmap compression saves up to 31% of runtime at the activation size of over 48m elements. While memory operations are slower than floating-point operations (in terms of clock cycles), in the edge training use case, we find that it is indeed faster to perform pure memory operations than floating-point calculations due to the fact that memory is the constraint. This is also desirable as it would reduce the total power consumed by the training. Therefore, our bitmap compression/decompression method is even more compute-efficient.

Summary. Using sparse bitmap compression is an efficient way to reduce the memory footprint for training deep learning models on the edge, with 18-53% overall training memory reduction of well-established image classification models. We have shown that memory reduction can indeed be measured empirically, achieving up to 34% memory reduction in storing convolution activations at a sparsity level of 50% (resulting from ReLU activations). Our method is orthogonal to existing methods in the literature, and further pushes down the memory footprint. For example, pruning increases sparsity to more than

75%, which can save up to 56% of the activations memory footprint, with negligible effect on the accuracy. Furthermore, low-precision can also double down memory consumption with up to 55-75% reduction. In addition, using bitmap compression for saving the activation outperforms classic checkpointing by eliminating the need for reproducing expensive matrix operations.

4.5 Conclusion

We propose BitTrain – a *Sparse Bitmap Compression* technique for memory-efficient training on the edge. Unlike previous methods that focus on saving memory to train deeper models on server-grade infrastructure, BitTrain directly optimizes the training memory footprint by addressing the most critical component of it – activations memory. We exploit activations sparsity and save them in a compressed format that scales sub-linearly with the total number of activations. BitTrain reduces the training memory footprint with no effect on the accuracy. Extensive experiments on benchmark datasets show that our method is orthogonal to existing work, and can be efficiently combined with them. BitTrain is a step further for efficient learning on the edge.

BitTrain is a first step towards enabling a new frontier in edge intelligence capabilities. In the future, BitTrain can be extended to further enable on-device learning. First, the compression and decompression processed can be integrated into the autograd libraries of the modern deep learning frameworks. The idea is to provide a seamless implementation similar to the checkpointing API. Second, defining native matrix operations on the bitmap format would make it more convenient for transfer learning on the edge. This will push the frontier of special hardware support for model "adapting" pre-trained models to new data locally on constrained devices. Third, combining all discussed methods in an integrated and efficient implementation would improve the empirical results, and push them closer to the theoretical calculations. This work democratize AI for low-resource settings and can also advance the state-of-the-art of privacy-sensitive AI applications.

CHAPTER 5

Efficient Inference with Temporal Awareness

5.1 Introduction

Recent advances in wearable devices show a strong potential to revolutionize applications in both health monitoring and patient diagnosis. For example, wearable devices have a big influence in post-op rehabilitation. A successful surgery depends on monitoring the patient's condition after the surgery, which is carried out through clinical visits. However, these visits are often insufficient and can be greatly enhanced by continuous monitoring using wearable devices. For example, it was recently shown that a wearable stretch sensor can be used to provide feedback for physical therapy or rehabilitation exercises [8, 4]. Also, special robotic devices can be used for post-stroke shoulder rehabilitation to identify misalignment [39]. Furthermore, a recent work uses wearable devices to detect hidden anxiety and depression in young children [111]. Activity recognition models have also been used to enhance health monitoring for the elderly. A case study uses wearable sensors and other environmentally placed sensors to predict health decline and critical health situations [174] and another one uses inertial wearable sensors and gait detection to provide useful digital bio-markers in dementia [43].

Wearable devices have very limited power and memory constraints. For example, Human Activity Recognition (HAR) platforms need to reach a satisfying activity recognition accuracy while consuming low power, resulting in a power-accuracy trade-off, which creates the need to co-optimize all the power consuming components of the device.

Usually wearable devices have two main energy consuming components: the sensors and the processing unit. In HAR platforms [11] [119], there are two main stages: first, collecting real time data from an accelerometer and/or a gyroscope, and second, preprocessing the data to extract some features before forwarding them to a classifier to analyze the user activity, such as walking, sitting, running, etc. Many recent works have presented optimized classification algorithms that are low in power and memory [87, 49, 88]. A few recent papers addressed methods to reduce the power consumption of the sensors by shutting down some of them [12] or by switching the sensors to a lower sampling frequencies with less intense user activities [119]. However, we observe that the *averaging window* is also a key factor affecting the sensor's energy consumption. Moreover, switching the sensors to different sampling rates introduces a classification challenge as each sampling rate provides a feature set of different size; this challenge was overlooked in previous works by retraining different classifiers for each used sampling rate creating a memory overhead. To tackle this challenge, we are the first to consider manipulating the feature extraction step to output the same feature size for data from different sensor configurations.

In AdaSense, we propose a novel low-power sensing technique that optimizes the power consumption of wearables while maintaining high activity recognition accuracy. We co-optimize the sensor, the feature extraction and the classifier to enhance the energy consumption of wearable devices. Our main contributions can be summarized as follows:

- **Sensor Configurations Design Space Exploration:** We provide a complete evaluation for the trade-off between activity recognition accuracy and power consumption under 16 different accelerometer sampling frequency and averaging window combinations to pick the optimal sensor configurations that trade off the accuracy with the power consumption in an optimal way.

- **Adaptive Low-Power Sensing technique:** Using the outcomes of the sensor configurations design space exploration, we introduce a novel technique that dynamically switches the sensor operation among these different configurations as a function of the user activity. We also provide detailed results for the recognition accuracy, memory and power consumption in comparison to the previously used techniques.
- **Features Extraction and Classification:** We propose a new feature extraction methodology which unifies the features set for heterogeneous data coming from various accelerometer configurations (i.e. sampling frequency and averaging window). This enables the usage of a single classifier that is capable of recognizing the human activity regardless the chosen configuration.

The rest of the chapter is organized as follows. We summarize the related work in Section 5.2. Then, we present our activity recognition framework in Section 5.3. In Section 5.4, we analyze the design space of the different sensor configurations, and we propose our adaptive low-power sensing technique. Next, we show the experimental setup and results in Section 5.5. Finally, we conclude in Section 5.6.

5.2 Related Work

Due to the tight energy constraints of wearable devices, recent work have focused on approaches to minimize power consumption while maintaining high performance (recognition accuracy) [119]. Researchers have determined that the sensor is responsible for a large fraction of the device's power consumption, reaching about 47% [12]. Therefore, reducing the sensor's power consumption significantly decreases the total energy requirement for the device. This can be done by switching the sensor to low-power mode, reducing the sampling frequency [119], turning off the sensor intermittently, shutting down one of the sensor's axes [12] or using compressed sensing techniques for selective sampling [105]. However, to the best of our knowledge there has not been any approach that considers the averaging window of the sensor to maximize power savings.

There are some approaches that consider using approximate circuits to achieve significant power savings without sacrificing much accuracy [53, 52], while others focus on minimizing the power consumption of the used machine learning algorithms [87, 49]. Feature extraction and data processing also require significant energy, so researchers have developed techniques to reduce the complexity of the extracted features in order to save power [180]. For example, statistical features are relatively simple to calculate, while the Fourier Transform and Discrete Wavelet Transform are more computationally complex, so we can dynamically choose which features to calculate based on the required power budget [12].

Once the relevant design points have been identified, various approaches attempt to determine the optimal strategy for switching between these design points at runtime. NK *et al.* [119] propose to switch to power saving design points when the user is doing low-intensity activities because these do not require as many data points to classify. Liu *et al.* [161] instead propose to use compressed sensing techniques to determine how many samples are needed for reconstruction and consequently sample as needed.

Finally, the machine learning classifier must be compatible with the dynamic sensor data. If the sensor data is acquired under settings that are different from the training data, then the accuracy can be significantly degraded. NK *et al.* [119] address this problem by retraining a separate neural network for each design point, while Liu *et al.* [161] use linear interpolation to normalize for variable sampling rate.

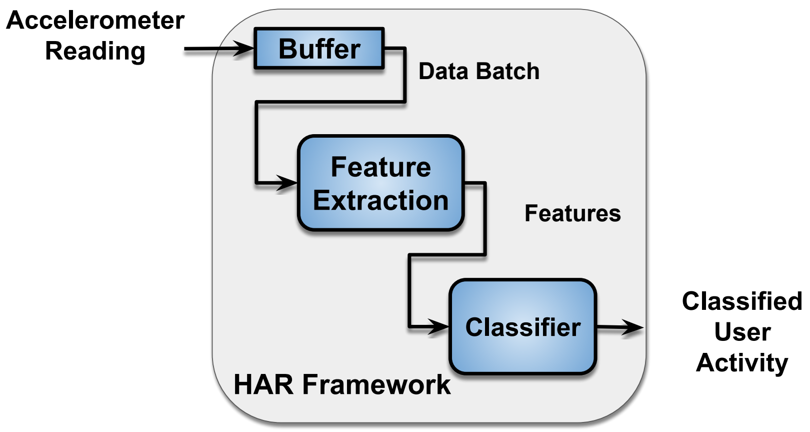


Figure 5.1: Human Activity Recognition Framework.

5.3 Adaptive Human Activity Recognition Framework

The ultimate goal of our HAR framework is to read an accelerometer's 3-axis data, and analyze them into one of six daily activities: sit, stand, walk, go upstairs, go downstairs and lie down. Two main components are needed to do that task: the feature extraction and the classification. Both tasks should be done on the wearable device, so the needed processing power and storage memory should be meticulously considered. Moreover as mentioned in Section 5.1, the HAR classifier should handle heterogeneous sensor data of different nature, e.g., different sampling frequencies and averaging windows, without creating a processing or a memory overhead. AdaSense tackles this heterogeneous data problem by using a new feature extraction technique that unifies the size of the features vector regardless the accelerometer's configurations. In this section, we first explain the main components of the HAR framework, then we present our new methodology for feature extraction as well as the used activity classification model.

5.3.1 Main Components

The main components of our HAR framework can be summarized in Fig. 5.1. The input is the data collected by the accelerometer, and the output is the user activity class. A batch of sensor data is needed to perform meaningful classification, so a buffer is added to control the classification frequency. The buffer stores the accelerometer data over two consecutive seconds. Then every one second, we push the collected data batch in the buffer through the rest of the pipeline, where we run feature extraction and classification to predict the user activity. We introduce one second overlap between the data batches to give the classifier some insightful information about the previous classification data batch. The challenge is that the data batch size depends on the sampling frequency, i.e., the number of samples stored in the buffer during one second period would be 100 with 100 Hz sampling frequency, and 50 when the sampling frequency is 50 Hz, making the job harder for the classifier. This issue would be handled during the feature extraction by constructing similar size feature set regardless the size of the processed data batch.

5.3.2 Feature Extraction

We can split the used features set into two categories: *statistical features* and *Fourier transform coefficients*. The statistical features include the mean and the standard deviation of the signal for the x, y and z coordinates. These statistical quantities capture the general structure of the accelerometer data over the selected batch. The Fourier transform coefficients capture frequency information for each activity. However, we noticed that we do not need to use all the Fourier transform components. For the sake of activity recognition, the first three coefficients in each coordinate, representing the frequency components up to 3 Hz, are enough to get around 97% recognition accuracy. The big advantage of using these features is that the feature vector size would be the same regardless of the size of the processed data batch. This solves part of the classifier's problem; however, the classifier still needs to adapt to the information in the features depending on the used sampling frequency and averaging window.

5.3.3 Human Activity Classifier

AdaSense's classifier must gracefully handle heterogeneous sensor data in order to maintain high accuracy. This task is not trivial because classification accuracy can degrade significantly if the sensor configurations of the test data are different from the configurations of the training data. A commonly used approach is to retrain a different model for each sensor configuration, which is guaranteed to provide high recognition accuracy but adds memory overhead to store multiple classifiers [119]. However, since we have unified the size of the feature set for different sampling rates, we can do the classification using one neural network with two layers: one hidden layer with RELU activation function and an output layer with 6 neurons and a softmax. By training this network on data from different sampling frequencies and averaging windows, we expect it to perform well using much less memory depending on the different sensor configurations used, i.e if there are four different sensors configurations, using our method will need one network instead of four different networks using 4× less memory to store the weights.

5.4 Low-Power Sensing

Wearable devices have two main power consuming components: the sensors and the processing unit. In Section 5.3, we described our HAR framework structure and the used methodologies to perform efficient processing and classification. In this section, we describe how AdaSense optimizes the power consumption of the sensor. We first analyze the accuracy and power trade-off for different sensor configurations. Next, we use the outcome of this analysis to design an adaptive controller that changes the sensor configurations according to the user activity.

Table 5.1: Acceleromenter Sampling Frequency and Averaging Window Combinations.

• 100Hz / 128 (F100_A128)	• 50Hz / 128 (F50_A128)
• 25Hz / 128 (F25_A128)	• 12.5Hz / 128 (F12.5_A128)
• 6.25Hz / 128 (F6.25_A128)	• 25Hz / 32 (F25_A32)
• 12.5Hz / 32 (F12.5_A32)	• 6.25Hz / 32 (F6.25_A32)
• 50Hz / 16 (F50_A16)	• 25Hz / 16 (F25_A16)
• 12.5Hz / 16 (F12.5_A16)	• 6.25Hz / 16 (F6.25_A16)
• 50Hz / 8 (F50_A8)	• 25Hz / 8 (F25_A8)
• 12.5Hz / 8 (F12.5_A8)	• 6.25Hz / 8 (F6.25_A8)

5.4.1 Sensor operation Modes

Usually sensors has two operation modes: *normal mode* and *low-power mode*. For the sensor to give less noisy readings, the output is not just the instantaneous reading at the required sampling point. Instead, it is the average of the collected samples over a certain window before the sampling point; the size of that window is called the *averaging window*. In normal mode, the sensor stays on all the time, so the averaging window does not affect the power consumption. However, in low-power mode, the sensor switches between normal and suspend modes, so both the sampling frequency and the averaging window determine the time in which the sensor has to be on; hence, significantly affecting the power consumption.

5.4.2 Sensor configurations Design Space Exploration

Using the setup mentioned in Section 5.3, we study the power consumption and the recognition accuracy at different sensor configurations. We choose 16 different sampling frequency and averaging window combinations given in Table 5.1, and we analyze the accuracy and power trade-off as illustrated in Fig. 5.2. Each point in the graph represents a different sensor configuration, and each configuration has a unique current-accuracy value which is a factor of the sampling rate and the noise due to using lower averaging windows. We observed that the four configurations {*F100_A128, F50_A16, F12.5_A16, F12.5_A8*}, highlighted by the diamond shape, dominate the others, and create a Pareto front that

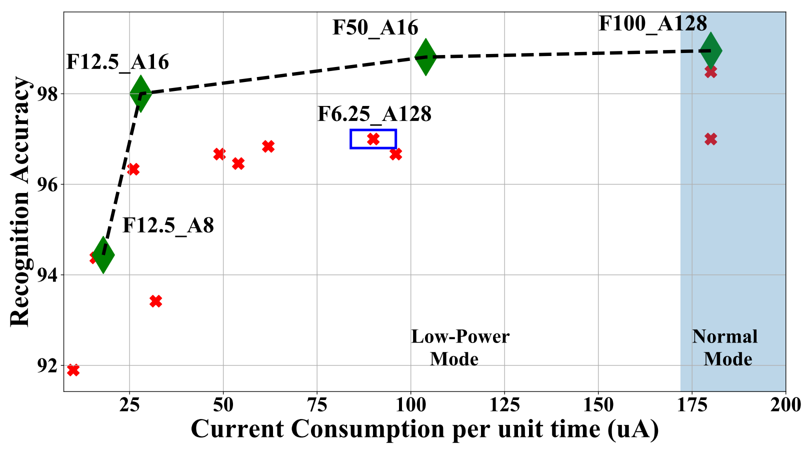


Figure 5.2: Accelerometer configurations accuracy and power trade-off.

optimally trades off accuracy with power. *F100_A128* has the highest accuracy and current consumption, whereas *F12.5_A8* has the lowest accuracy and current consumption. However, the other red points do not offer any benefits in the energy-accuracy trade-off; for example, the point *F6.25_A128* marked by the blue rectangle is dominated by the point *F12.5_A16* which has higher accuracy and lower current consumption.

5.4.3 Adaptive Low-Power Sensing Technique

To reduce the sensor's power consumption, AdaSense introduces a new technique that switches among different sampling frequencies and averaging windows. Our adaptive controller dynamically switches to a lower power configuration when the user activity is stable (i.e, the user has been doing the same activity for a long time), and switches back to the highest accuracy configuration when the user activity changes to capture the right one. In other words, if the user has been walking for a certain period of time, it means that this user is steadily doing the same activity and will probably continue doing it for a while, so we can lower the sampling frequency and averaging window to reduce the power consumption of the sensor. However, if the activity keeps changing rapidly, then the sensor needs to operate at its highest power to capture the correct activity. Fig. 5.3

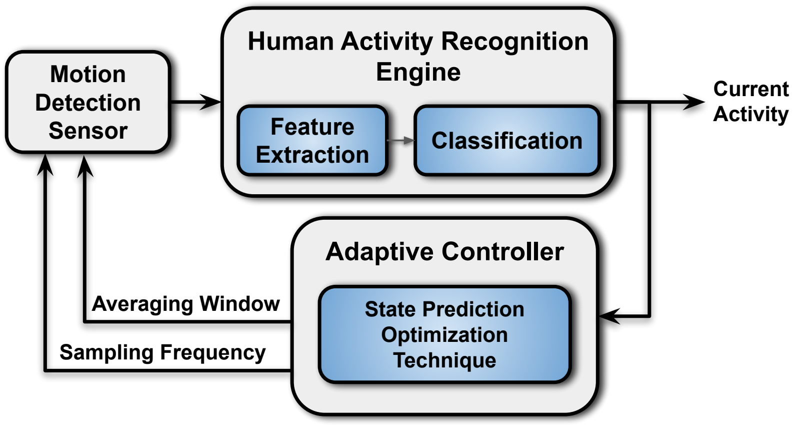


Figure 5.3: Low-Power Sensing HAR Framework.

shows the proposed framework for AdaSense. The sensor first operates at its high power configurations, then it forwards the collected data to the HAR framework. As explained in Section 5.3, the HAR framework classifies the data, and feeds its output to an adaptive controller which in turn adjusts the next episode's sensor configurations. Our controller uses a novel technique called state prediction optimization technique (SPOT) to take its decision.

5.4.4 The State Prediction Optimization Technique (SPOT)

The adaptive controller in AdaSense makes its decision regarding the sensor's configuration using our SPOT technique which is based on a simple finite state machine. First, we have four states representing the four optimal sensor configurations chosen during the design space exploration analysis earlier. These four states are sorted in descending order according to the power consumption. The accelerometer starts working at the first state *(F100_A128)*, and every one second it compares the current classification output with the previous classification output. When the activity stabilizes, i.e when the classifier output remains the same for few classification attempts called *stability threshold*, it moves to the lower power state. If the classified user activity changed at any state, the sensor returns back to the first high accuracy state.

For example, suppose we chose to operate on four different states named {*F100_A128, F50_A16, F12.5_A16, F12.5_A8*} as shown in Fig. 5.4, and *C1, C2, C3, C4* are the conditions that control the transition from one state to another such that:

- **C1:** Current Activity == Last Activity & Counter < stability threshold
- **C2:** Current Activity == Last Activity & Counter = stability threshold
- **C3:** Current Activity != Last Activity
- **C4:** Current Activity == Last Activity

SPOT starts by operating at *F100_A128*, and frequently classifies the output data from the sensor and increment a counter whenever the current activity matches the previous one. It continues to do so till the counter reaches the stability threshold. At that point, it switches to the next state in which it follows the same behaviour, successively switching to the next states till it reaches the last one and stays there. If at any state the current activity did not match the previous one, SPOT resets the counter, and switches immediately to the first high accuracy state, then the behaviour is repeated as long as the device is on.

5.4.5 The SPOT technique with confidence

The total power consumption depends on the time spent at each state. In SPOT, the decision to move from a lower power state to a higher power state is taken when the classifier reports a change in the human activity. The classifier reports that the activity is changed in two cases: when the activity actually changes, or when the classifier mispredicts due to some noise in the sensor's data. As a result, we introduce the *confidence* parameter in SPOT, which adds some tolerance to the noisy data. This confidence is the classifier's probability of the chosen output class; for example, if we have two output classes (walk, sit), and the softmax at the classifier's output layer gave the probabilities (0.8, 0.2), then the classification would be "walk" with confidence 0.8. In SPOT with confidence, the decision to move to a higher power state is taken when the classifier reports that the

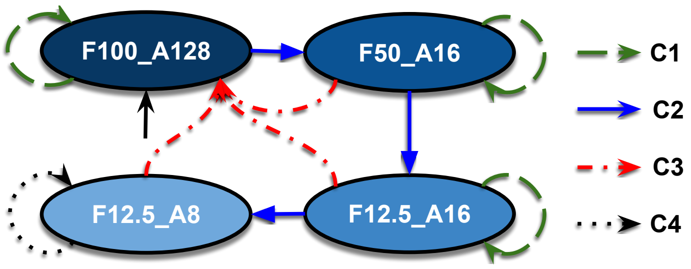


Figure 5.4: State Prediction Optimization technique FSM, states named as F(sampling frequency)_A(averaging window), while C1-C4 are the conditions directing the flow from the first to the last state as a function of the activity stability.

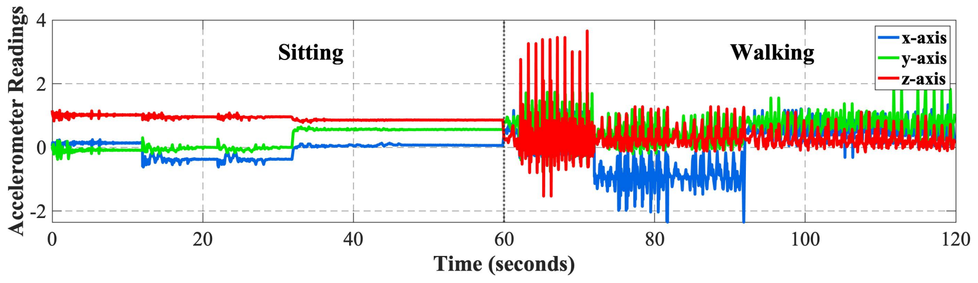
activity is changed with a confidence higher than a certain threshold called *confidence threshold*.

5.5 Experiments and Results

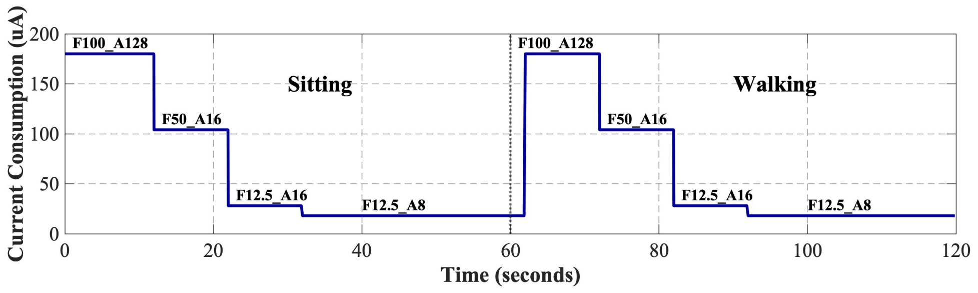
5.5.1 Experimental Setup

Hardware Used: We evaluate the proposed adaptive sensing technique using a Texas Instruments CC2640R2F MCU [149] integrated with a Bosch Sensortec BMI160 16-bit inertial measurement unit (IMU) [16]. We operated the IMU in both the normal mode and the low-power mode to set different sampling frequencies and averaging windows.

Data: We only enabled the IMU's accelerometer, and we collected data with the x, y and z sensor readings at different sampling frequencies and averaging windows. Then we used the collected data to evaluate our methodology. Using the setup mentioned in Section 5.3, we trained our neural network on an extensive data set of 7300 activity windows of the four optimal accelerometer configurations {*F100_A128, F50_A16, F12.5_A16, F12.5_A8*} analyzed in table 5.1. The data recorded 6 different activities: walk, sit, lie down, go upstairs, going downstairs, stand.



(a) 3-axes Accelerometer Readings with Time.

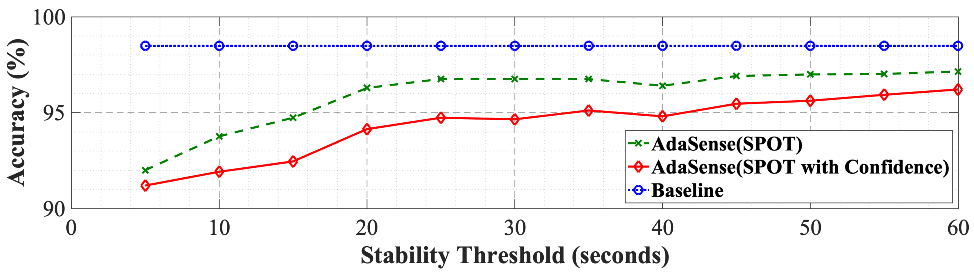


(b) Sensor Current Consumption per unit time.

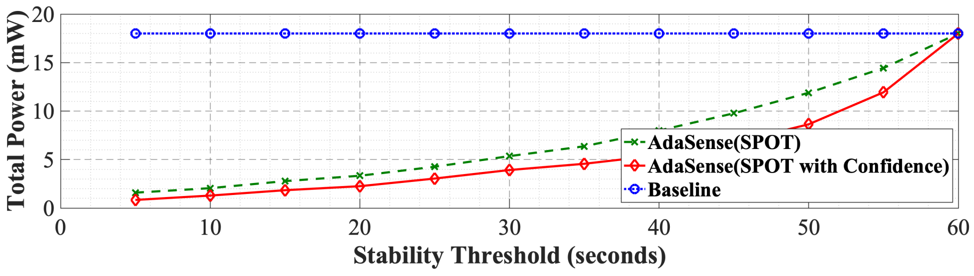
Figure 5.5: AdaSense Behavioural Analysis.

5.5.2 AdaSense Behavioural Analysis

As an illustration of AdaSense's performance, we analyze its inputs and outputs over a time interval of 120 seconds as shown in Fig. 5.5. We show a whole use case in which the user sits for the first 60 seconds, then the user changes the activity and starts to walk for another 60 seconds. Fig. 5.5a shows the inputs from the accelerometer over the chosen time interval where the y-axis is the x, y and z axes of the accelerometer reading. Fig. 5.5b shows the classification and power analysis, the y-axis is the current consumption in μA. We observe that AdaSense starts operating at the high power configuration of the sensor $F100\_A128$, and then it gradually switches to lower power configurations, until it reaches the minimum $F12.5\_A8$ after 28 seconds. It stays there till the activity changes at time 60 seconds when the sensor switches back to the highest power state again, and repeatedly does the same behaviour till it also reaches the minimum after another 28 seconds.



(a) Classification Accuracy with Stability Threshold.



(b) Total Power Consumption with Stability Threshold.

Figure 5.6: AdaSense Power and Accuracy analysis.

5.5.3 Power & Accuracy Analysis

We first analyze the impact of the proposed methodology to co-optimize the sensor, feature extraction and classification of the HAR framework on the activity recognition accuracy and the sensor's power consumption. Fig. 5.6a shows the accuracy of classification as we increase the stability threshold which determines the stability of the activity; hence, switch the sensor to a lower configuration. We compare the accuracy under three scenarios. In the first scenario, we prevented the controller from switching among different sensor configurations, i.e. the sensor operates on the high power configuration $\{F100\_A128\}$ all the time; we take this as our baseline to measure how switching between different configurations affects the accuracy. In the second and the third scenarios, we analyze AdaSense using our two different adaptive controllers: SPOT and SPOT with confidence of value 0.85 respectively to switch among the four sensor configurations when the user activity becomes stable. In the three scenarios, we trained a single neural network on data from the four different accelerometer configurations mentioned before, and we used that model to test the classification accuracy while changing the setup of the adaptive sensor configurations controller.

From the results of Fig. 5.6a, we observe that the accuracy increases as the value of the stability threshold increases. Specifically, as the stability threshold increases from zero to 20 seconds, the classification accuracy rapidly increases from 91% to 96.5%. Then, the accuracy saturates within a range of 1.5% below the baseline. This reduction in accuracy is expected because when the stability threshold is low (i.e., < 20 seconds), the adaptive controller promptly switches to low-power accelerometer configurations; however, since the user activity is not stable for a long time, it triggers changes among different sensor configurations which result in a lower recognition accuracy. When the stability threshold is high enough (i.e., > 20 seconds), the loss in accuracy is negligible when compared to the baseline.

Next, we compare the total power consumption for the sensor as a function of the stability threshold in seconds. As shown in Fig. 5.6b, the power consumption increases with the increase in the stability threshold; when the stability threshold is low, the sensor rapidly switches to a low-power configuration after few seconds, so more time is spent at the lower power configurations which minimizes the total power consumption by the sensor. However, as the stability threshold increases, the time spent on the low-power sensor configuration decreases, so the total power increases. Furthermore, at stability threshold of 60 seconds, the power consumption matches the baseline as the sensor spends all the time operating at the high power configuration. In average, the total power can be reduced by 60% using SPOT and 69% using SPOT with confidence.

5.5.4 Comparison to the previous work

This section compares the accuracy and the power consumption of AdaSense to the related work. NK *et al.* [119] use an activity intensity based approach to reduce the power consumption of the sensors; the sensors switch to low-power mode with low-intensity user activities (i.e. stand, sit, lie down), and operate at the normal mode with more intense activities (i.e. walk, go upstairs, go downstairs). NK *et al.* define the intensity of the

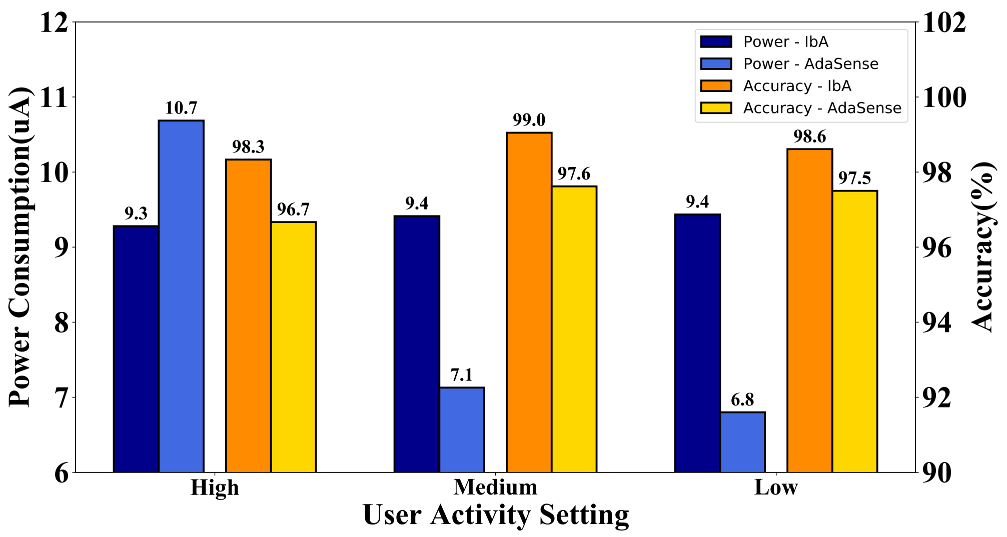


Figure 5.7: Comparison between AdaSense and Intensity Based Approach(IbA) [119] in terms of Accuracy and Power Consumption under different user activity settings.

activity using the first derivative of the accelerometer readings, and they retrain separate classifiers for each used sampling frequency. Fig. 5.7 summarizes the comparison, the x-axis shows three user activity settings {*High, Medium, Low*}, which differs in terms of the user activity change rate. *High* means that the user activity is not stable (i.e. changes every 10 seconds), while *Low* means that the user activity is quite stable (i.e. it takes the user at least 1 minute to change the activity). The left y-axis shows the power consumption of using AdaSense versus the activity intensity based technique. As expected, when the user activity setting is high, AdaSense spends most of the time at the high power sensor configuration; therefore, the power consumption is relatively high. However when the user activity starts to be more typical, the power consumption is reduced by at least 25% compared to the previous work. The right y-axis shows the recognition accuracy. The results show that AdaSense has has slightly lower recognition accuracy (i.e ranging from 1% to 1.5%) depending on the setting in comparison to the technique used by NK *et al.*. This loss in accuracy is acceptable in trade of the significant power and memory savings.

Memory Requirements: While NK *et al.* retrain different neural networks for the different sampling frequencies, AdaSense trains a single classifier on data from different sensor configurations, consuming 2$\times$ less memory to store the classifier(s) weights. This memory reduction is important for wearable devices as they only have few KBs of memory.

Data Processing Overhead: In AdaSense, we do not need to compute the derivative of the collected sensor data to switch among the different configurations. Therefore, we prevent computations overhead that might compromise the power savings from the sensor.

5.6 Conclusion

Wearable devices have many advantageous applications in health services, and with the advancement in the research done to reduce the power consumption on those wearable devices, their deployment in real applications would become more practical. This chapter presented a low-power sensing technique for activity recognition on wearable devices. Using an adaptive controller, we dynamically switched the sensor to lower sampling frequencies and averaging windows depending on the stability of the user-activity. We analyzed the trade-off between the accuracy and the power consumption for different sensor configurations. Then, we designed an adaptive controller that switches among the resulting optimal sensor configurations. We also co-optimized the features extraction and the classification achieving up to 69% reduction in total power consumption with less than 1.5% degradation in the recognition accuracy.

CHAPTER 6

Efficient Inference with Spatial Awareness

6.1 Motivation

Convolutional neural networks (CNNs) are used to achieve high accuracy in object detection tasks [131, 132]. In object detection, the input is an image, and the outputs are the object classes and their location in the image represented by the bounding box coordinates. CNN-based object detectors can be divided into two main categories: two-stage detectors [132, 60] and one-stage detectors [131, 95]. Two-stage detectors first identify some regions of interest (ROIs) as candidates, and then classify and regress those ROIs to identify the objects and their bounding boxes. One-stage detectors directly predict the object categories and the bounding boxes using some default anchors. One-stage detectors are less accurate than two-stage detectors, but they are more efficient; hence they are more suitable for edge devices.

In recent years, the deployment of CNNs on edge devices like mobile phones, smart glasses, and augmented reality devices has become essential for real-time response and for data privacy reasons. However, those networks have high memory and computational power demands, which challenge their deployment on resource-constrained embedded devices. These devices are battery-operated, so low energy consumption is crucial. They also have memory constraints, so the model should be compact. Finally, those models have real-time constraints when deployed on edge devices.

To reduce the memory and computational demands of CNNs, researchers have explored pruning and quantization of those models to create a trade-off among energy, latency, and accuracy [42]. Others use dynamic networks [148, 170, 176, 146] that can be reconfigured during runtime to accommodate the target accuracy and efficiency. Another effective approach is to use hierarchical networks [80]. Those networks have shown success in image classification tasks because each image has only one object, so the objects can be grouped based on their visual or semantic similarities [44]. However, for object detection, an image can have objects that are not visually or semantically similar. Therefore, the previously proposed methods would not work for object detection. For example, a scene with a parking meter next to a car is very likely to happen. None of the techniques based on visual or semantic similarity will group the car and the parking meter together. Another example is a scene with a couch and a television. However, in a spatial-context-based approach, a car and a parking meter will be grouped together, while a couch and a television will form another group. Thus, we propose using the spatial context to group the object classes, this would allow our adaptive model to efficiently complete the detection task by executing a single branch, making it appropriate for resource-constrained devices.

In this chapter, we propose a novel approach that leverages the information about the spatial context of the objects to design an efficient adaptive model for object detection. Our adaptive model reduces the energy consumption, the latency, and the memory footprint for object detection with a negligible loss in accuracy. Our contributions can be summarized as follows:

- To the best of our knowledge, AdaCon is the first work to introduce an adaptive methodology for one-stage object detectors handling images with multiple objects.

Our adaptive method enables efficient object detection on resource-constrained embedded devices.

- AdaCon leverages the information about the spatial-context of the object categories to construct a knowledge graph. Then, we use the constructed knowledge graph to design our adaptive context-aware object detection model.
- We introduce a simple yet effective and generalizable methodology that can be easily applied to design the neural network architecture of the detection branches for our adaptive model.
- We apply our adaptive methodology to two different state-of-the-art object detectors, and we compare different generated adaptive models representing different energy-accuracy trade-offs to the static baselines.
- We deploy our AdaCon models on an embedded Nvidia Jetson nano board, and analyze the accuracy, latency, and energy of the different architectures. Our adaptive model achieves up to 45% reduction in energy, and up to 27% reduction in the latency with small loss in average precision on the COCO dataset.

The rest of the chapter is organized as follows. We review the related work in Section 6.2. Then, we introduce our adaptive object detection technique in Section 6.3. Next, we show the experimental setup and results in Section 6.4. Finally, we conclude in Section 6.5.

6.2 Related Work

Object Detection: Object detection is a well-researched topic because it is essential in many applications like augmented reality, surveillance, and autonomous driving. However, most object detectors are not suitable for embedded or wearable devices because they are based on complex power-hungry DNNs with large memory footprint. One-stage detectors such as YOLOv3 [131] and RetinaNet [95] are faster than most two-stage detectors like

Faster R-CNN [132] and Mask R-CNN [60], so they are more suitable for real-time response on edge devices. Thus, we choose YOLOv3 [131], and RetinaNet [95] as a the baseline architecture on which we apply our method for adaptive context-aware object detection.

Efficient Neural Networks: Over the past few years, the need for efficient computing on edge devices increased, so researchers started designing compact networks [65, 107, 162]. Others used pruning and quantization [37, 42]. In this chapter, we consider a different research direction for optimization by exploring adaptive networks that can leverage the nature of the object detection task. Our approach is an orthogonal effort to build more efficient neural networks that are suitable for constrained embedded devices.

Dynamic/Adaptive Neural Networks: Adaptive neural networks have been adopted by many researchers to reduce the computational complexity needed for neural networks, hence reduce the latency and energy requirements without sacrificing much accuracy. Tann *et al.* [148, 146] and Yu *et al.* [170] trained a single network at different widths to permit adaptive accuracy-energy trade-offs at runtime. Others skip some layers [160], or deploy some early exit criteria [15] to reduce the computation complexity. Recently, Zhang *et al* [176] proposed building Domain-Aware networks that decide which part of the network to run based on the weather and the time of the day in which the device is operating. All these methods are generic for convolutional neural networks and image classification, but they have barely been applied to object detection. Our work is different because we leverage the information about the spatial context of the object categories. The spatial context implies the probability that different objects can occur jointly, and this is an essential information that can be used to further optimize object detection models.

Context-aware Object Detection: On another research thread, some researchers have been exploring the use of prior knowledge about the real world to improve the accuracy of object detection models. Fang *et al* [36], and Xu *et al* [164] integrate knowledge-graphs and adjacency-matrices to leverage the information about the co-occurrence and the

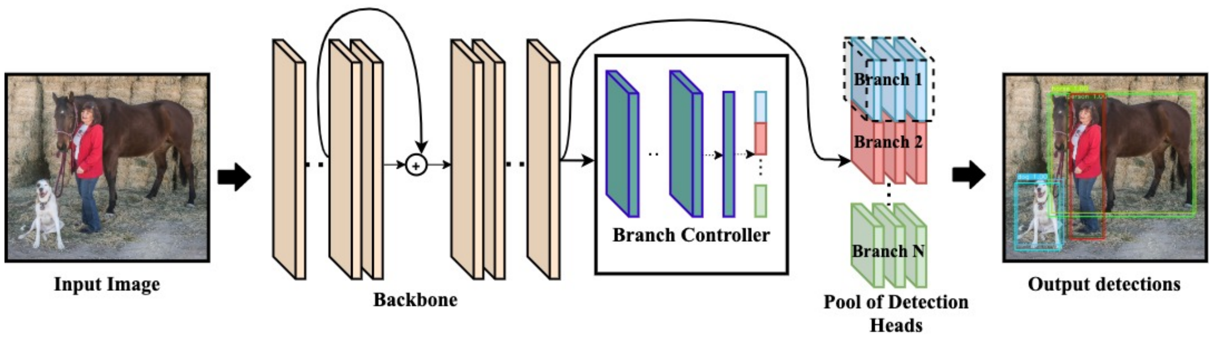


Figure 6.1: Illustration of our Adaptive Object Detection model. The backbone is first executed to extract features from the input image. Then, the branch controller takes the extracted features, and route them towards one or more of the downstream detection heads. Only the chosen head(s) are then executed to get the detected object categories and their bounding boxes.

locations of the objects for more accurate object detection. Their main goal is to increase the model accuracy, and they achieve this by adding more computational complexity, which further reduces the model efficiency. However, our criteria is different, because we exploit the idea of using the prior knowledge to build efficient adaptive object detection models. Our adaptive model architecture reduces the energy consumption, latency, and runtime memory requirements of object detectors, making them more appropriate for resource-constrained devices.

6.3 Method

Modern object detectors are typically composed of two main parts: a *backbone* which extracts the features from the input image, and a *head* which is used to predict the object categories and their bounding boxes. In this work, we propose an adaptive context-aware neural network architecture for object detection. We achieve this architecture by leveraging the information about the co-occurrence of objects in the spatial domain while designing our object detection model.

Our adaptive model consists of two main components: spatial-context-based clustering and a hierarchical object detection model. In the spatial-context-based clustering, we extract the information about the co-occurrence of object categories from the training data, and use this information to cluster the object categories, where each cluster has the objects that co-occur in the spatial domain with high probability. Then, we leverage the extracted information to design an adaptive object detection model.

As illustrated in Figure 6.1, our adaptive object detection model consists of three main components: *backbone*, *branch controller*, and a pool of *specialized branches*. The backbone is a CNN responsible for extracting the features from the input image. The branch controller is a regression model with a sigmoid activation function at its output. The branch controller has an input size similar to the output feature maps of the backbone, and the number of its outputs is equal to the number of clusters chosen while running the spatial-context-based clustering. Those outputs represent the *confidence score* that the input image belongs to the corresponding spatial context. Using the confidence scores in the branch controller, we enable two modes of operation: *single-branch execution* where only the branch with the highest score is selected, and *multi-branch execution* where all the branches with scores higher than a certain threshold are selected. The final component of our model is a pool of specialized detection heads (branches). Each specialized branch is responsible for detecting the objects that belong to one spatial context.

During run-time, the input image first passes through the backbone, then the output of the backbone is passed to the branch controller which classifies those feature maps to one or more spatial contexts. Receiving the decision from the branch controller, we execute the chosen branch(es), and concatenate their outputs to complete the detection task. In this Section, we are going to explain in detail the clustering method in Section 6.3.1. Section 6.3.2 shows our method for selecting the architecture of the specialized branches. Finally, we explain the training method for our adaptive object detection model in Section 6.3.3.

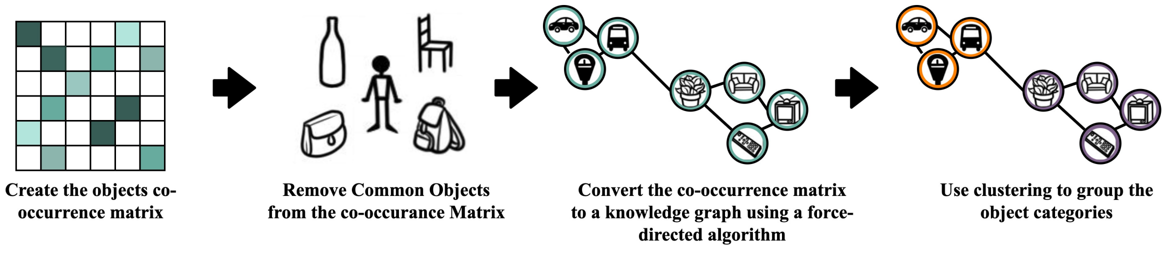

Create the objects co-occurrence matrix
Remove Common Objects from the co-occurance Matrix
Convert the co-occurrence matrix to a knowledge graph using a force-directed algorithm
Use clustering to group the object categories

Figure 6.2: Spatial-Context-based clustering for the object categories.

6.3.1 Spatial-Context based Clustering

The ultimate goal of our clustering technique is to group objects that can co-occur in the same scene. Then, we use those clusters to construct our hierarchical detection model where each cluster has a corresponding branch in the model. This would guarantee high performance and high efficiency because it would enable detecting most of the objects in a scene by executing only one branch of the detection model. Figure 6.2 shows our spatial-context-based clustering technique. First, we construct the co-occurrence matrix of the object categories where each value represents the frequency of the co-occurrence of the object categories in the same scene across all the training dataset. Then, we extract the common objects. Those are the object categories that have high probability of co-occurrence with more than 75% of other object categories. We remove those common objects from the co-occurrence matrix, and add them later to each cluster. Our intuition is that some objects such as a "person" or a "backpack" can co-occur with any other object in any spatial setup. Thus, it is better that our network considers the presence of those common objects all the time. Next, we convert the frequency-based co-occurrence matrix to a correlation matrix. Then, we use the correlation matrix to build a knowledge graph using Fruchterman-Reingold force-directed algorithm [38]. As shown in Figure 6.2, the nodes of the knowledge graph represent the object classes, and the edges represent the probability of the co-occurrence of the connected objects in the same scene. Afterwards, we use agglomerative clustering to group the nodes based on their location in the knowledge graph. This results in clusters of object classes where the inter-cluster objects have low probability of appearing together in the same scene, while the intra-cluster objects have a high likelihood of occurring jointly.

6.3.2 Detection Head Architecture

In our hierarchical adaptive model, each branch (i.e., detection head) is responsible for detecting a subset of the object classes. The number of object classes assigned to a certain branch depends on the total number of branches in the adaptive model, as well as the outcome of the spatial-context clustering as explained in Section 6.3.1. This means that for a given model with a certain number of branches, the object classes are not equally distributed among those branches. Therefore, the number of object classes should be considered while designing the branch architecture to guarantee that each branch has an appropriate representational capacity to make it efficient without sacrificing accuracy.

We propose a systematic approach to choose the detection head architecture. Our approach can be easily generalized to different object detection models. For a given model, we choose the head architecture of the static baseline model as our template. For each branch detection head, we define *a compression factor* equal to the number of object classes assigned to this branch, divided by the total number of object classes. Then for each layer in the template head architecture, we keep the same number of layers, but we compress the model by reducing the number of channels in each layer by the previously calculated compression factor.

For example, assuming that we use a two-branch AdaCon model to detect 30 object classes, and our spatial-context-based clustering assigned 18 and 12 objects to the first and the second branches, respectively. Then, the compression factor for the first branch is 18/30, while it is 12/30 for the second branch. Table 6.1 compares the number of parameters, and the number of multiply-and-accumulate (MAC) operations for the compressed branches, as well as the static baseline template for YOLOv3 [131] and RetinaNet [95]. We can notice that the total number of parameters and MAC operations for the branches are always smaller than the corresponding template. Moreover, the various branches would not be active at the same time. Therefore, the dynamic number of parameters and MAC operations at a time would be significantly smaller, which reduces the memory footprint as well as the latency and the energy. More analysis is presented in Section 6.4.

Table 6.1: Detection Head (Branches) Architectures for AdaCon models with various number of branches.

Model		**Branches Parameters** (M)	**Branches MACs** $(Gops)$
RetinaNet	Baseline	6.7	22.5
	2 Branches	3.0, 2.1	4.2, 2.9
	3 Branches	2.1, 1.4, 1.4	2.9, 1.9, 1.9
	4 Branches	1.7, 1.4, 1.4, 0.5	2.4, 1.9, 1.9, 0.7
	5 Branches	1.4, 1.3, 1.4, 0.5, 0.6	1.9, 1.8, 1.9, 0.7, 0.9
YOLOv3	Baseline	21.3	8.4
	2 Branches	6.0, 9.2	2.4, 3.6
	3 Branches	6.0, 3.8, 3.8	2.4, 1.5, 1.5
	4 Branches	5.0, 3.8, 3.8, 1.2	1.9, 1.5, 1.5, 0.5
	5 Branches	3.8, 3.6, 3.8, 1.2, 1.5	1.5, 1.4, 1.5, 0.5, 0.6

6.3.3 Training the Adaptive Object Detection Model

Each module of our adaptive network is trained separately with the correct pairs of inputs and labels depending on the task assigned to this module. We use a multi-stage training technique to train our adaptive model. In stage 1, we train the backbone. Then in stage 2, we freeze the backbone, and concurrently train our branch controller as well as the detection branches.

Training the backbone: In our implementation, we train the backbone as a part of the static model. We then take the pre-trained backbone and use it as the backbone of our adaptive model.

Training the branch controller: The branch controller is a regression model with few convolutional layers and a sigmoid activation layer at the output. It predicts the probability that the input image belongs to each spatial context. As shown in Figure 6.1, the inputs to the branch controller are the feature maps generated by the backbone for each image, and the labels are the dominant spatial-context on each image. To generate the labels for training the branch controller, we map the objects in every image to their spatial-context according to the output of the spatial-context-based clustering, and we label the image with the dominant spatial context.

Training the detection branches: The detection branches are trained concurrently on the relevant object categories according to the output of the spatial-context-based clustering proposed in Section 6.3.1. The input to each branch is also the feature maps generated by the backbone for each image, and the labels are the bounding boxes and the object categories in the image.

6.4 Results

6.4.1 Experimental Setup

Dataset: We evaluate our method using the Microsoft COCO dataset [96]. This dataset has over 120K training images with 80 different object categories. The 80 categories cover a wide range of indoor (i.e., bedroom, kitchen, bathroom, living room, office, etc.) and outdoor scenes (i.e., street, park, farm, zoo, etc.). The images in the COCO dataset are all collected from real world scenes with some occlusions, and under various lighting conditions. Those images are annotated with the bounding boxes for the different object categories.

Implementation: We implemented our proposed technique using PyTorch. We used our spatial-context-based clustering to cluster the 80 objects of COCO dataset into different number of clusters as mentioned in Section 6.3.1. We apply our adaptive context-aware object detection technique to two different state-of-the-art one-stage object detectors:

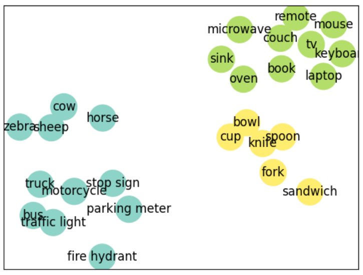
YOLOv3 [131], and RetinaNet [95]. In order to modify the static models into an adaptive context-aware network, we use a pre-trained backbone, and replace the detection head with our pool of specialized branches as well as our branch controller as explained in Section 6.3. We used Darknet-53 backbone with YOLOv3, and ResNet50 backbone with RetinaNet.

Hardware and Analysis Tools: We deploy our object detection model on an NVIDIA Jetson Nano board as a representative device of modern embedded systems [77]. This board has a low-power embedded GPU, a Quad-core ARM Cortex CPU with 4 GB of RAM, and it uses about 5 to 10 Watts depending on the workload. We analyzed the latency and the power consumption of our proposed technique on the Jetson Nano board. We used Nvidia's Tegrastats utility to measure the latency and the power consumption. We also used the COCO official APIs to evaluate the model accuracy.

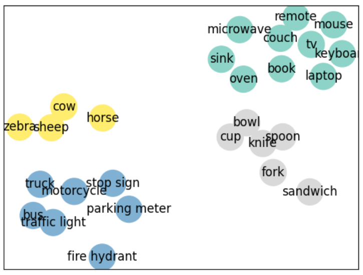
In this Section, we evaluate our spatial-context-based clustering by visualizing some of the formed clusters in Section 6.4.2. In Section 6.4.3, we validate the ability of the branch controller to detect the right spatial context for the input images. After that, we analyze the overall performance and efficiency of our AdaCon models in Section 6.4.4. Then, we show the effect of the introduced branch controller execution modes in Section 6.4.5. Finally, we analyze the accuracy-efficiency trade-off that our AdaCon models can achieve by choosing a different number of clusters, as well as different branch controller execution modes in Section 6.4.6.

6.4.2 Spatial-Context-Based Clustering Evaluation

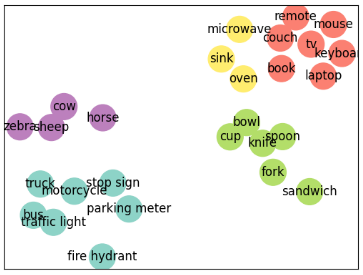
To evaluate our spatial-context-based clustering technique, we visualize the formed clusters for a subset of the COCO dataset. For the purpose of clear visualization only, we choose 30 representative object categories covering different spatial contexts (i.e., indoors, outdoors, street, farm, living room, kitchen, etc.). Figure 6.3 shows the result of our spatial-context-aware clustering technique when choosing different number of



(a) 3 Clusters



(b) 4 Clusters



(c) 5 Clusters

Figure 6.3: Our spatial-context-based clustering output for 30 object categories from the COCO dataset. The formed clusters become more fine-grained as the number of clusters increase.

clusters. Each node in Figure 6.3 represents a different object category, and the distance between any two objects represents the probability of their co-occurrence in the same scene (spatial context). We can notice that as the number of clusters increases, the clustering automatically becomes more fine-grained. For example, clustering the objects into two clusters results in an indoor objects cluster, and another one for the outdoor objects. Clustering the objects into four clusters, further categorizes the outdoor objects into objects that can be found on a farm, and others that can be found on a street, while categorizes the indoor objects to living room and kitchen objects.

6.4.3 Branch Controller Accuracy Evaluation

The branch controller takes the feature maps generated by the backbone for each image, and decides which branch(es) to execute based on the spatial context in the image (i.e., each spatial-context has a corresponding detection branch in our adaptive object detection model). The ultimate goal of the branch controller is to determine the dominant spatial context in the input image. The branch controller achieves that goal by assigning confidence scores that the input image belongs to each spatial context. Then, the dominant context would be the one with the highest confidence score.

The branch controller accuracy is crucial for the accuracy of our adaptive model. The reason is that as the accuracy of the branch controller decreases, its ability to take the

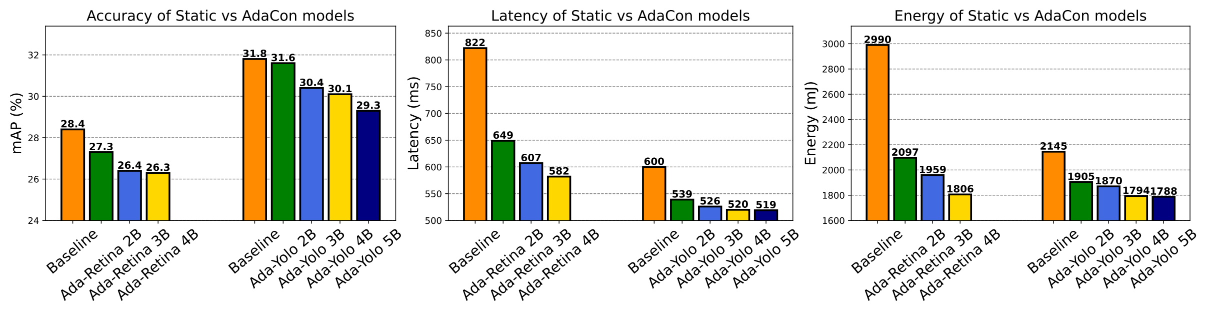
Table 6.2: Branch Controller accuracy in detecting the correct spatial-context for different AdaCon models.

Model name	**Backbone**	**Num. Branches**	**Accuracy (%)**	**Param** (M)	**MAC** ($Gops$)
Yolov3	Darknet53	2	95.3	2.53	0.4
Yolov3	Darknet53	3	93.3		
Yolov3	Darknet53	4	93.2		
Yolov3	Darknet53	5	91.7		
RetinaNet	Resnet50	2	93.3	0.55	0.17
RetinaNet	Resnet50	3	90.9		
RetinaNet	Resnet50	4	89.5		

right decision about which branch(es) to execute decreases, and this directly affects the performance of the adaptive model. To validate the decision-making capability of our branch controller, we analyze its accuracy in detecting the dominant spatial context on COCO dataset. Table 6.2 shows the accuracy of the branch controller for different AdaCon models with different number of branches. As illustrated in the Table 6.2, our branch controller is a light-weight model (i.e., the number of parameters and MAC operations are low) to prevent any memory, or compute overhead. The results show that the branch controller accuracy is higher with fewer number of branches. This is expected because its task becomes more complex as the number of branches increase.

6.4.4 AdaCon Performance and Efficiency Evaluation

To analyze the overall performance of our proposed adaptive technique, we show the average precision, the latency per inference, and the energy per inference of our AdaCon models with different number of branches compared to the static baselines in Figures 6.4a, 6.4b, and 6.4c, respectively. The x-axis shows the different static and adaptive models with different number of branches (i.e., Ada-YOLO 2B denotes an AdaCon-YOLO model with two branches operating in the single-branch execution mode). In Figure 6.4a, the



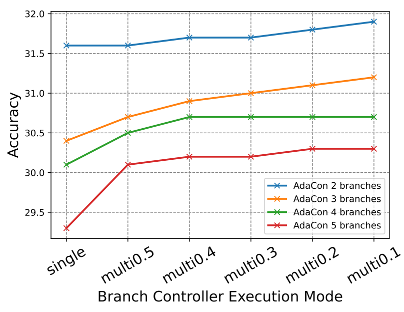
(a) (b) (c)

Figure 6.4: Evaluation of (a) Average Precision (b) Latency (c) Energy for AdaCon-RetinaNet and AdaCon-YOLO models with different number of branches under single-branch execution mode compared to the static baseline models. Input image resolution is 416 × 416.

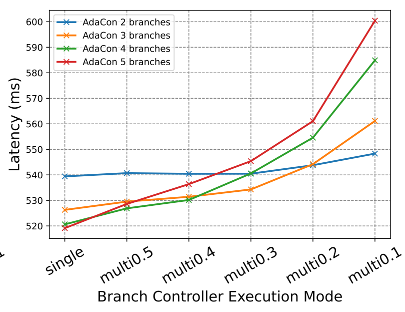
y-axis shows the mean average precision of the object detection. In Figures 6.4b and 6.4c, the y-axis represents the latency per inference in milliseconds, and the energy per inference in millijoules, respectively. The results show that the average precision of the detection models decreases as the number of branches increases. The reason is that the accuracy of the branch controller decreases as the number of branches increases, leading to more misses by the branch controller. On the other hand, the energy and the latency decrease as the number of branches increase. The reason is that for models with more branches, each branch is responsible for a smaller subset of the object classes, hence it is more compact as explained in Section 6.3.2.

6.4.5 AdaCon Evaluation for the Branch Controller Execution Modes

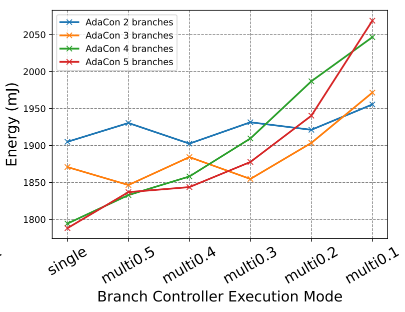
As explained in Section 6.3, our branch controller gives a confidence score for executing each branch based on the spatial context of the input image. We introduce two modes of operation: *single-branch execution (single)* mode where only the branch with the highest confidence score gets executed, and *multi-branch execution mode (multi)*. In the *multi-branch* execution, all the branches with a confidence score higher than a certain threshold are executed. These execution modes not only boost the accuracy, but they also



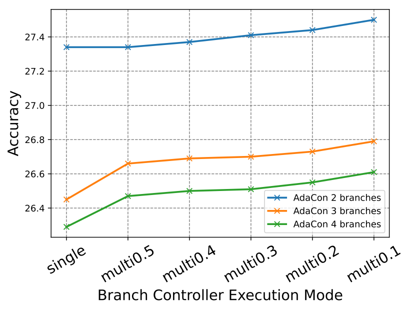
(a) Average Precision of AdaCon-YOLO



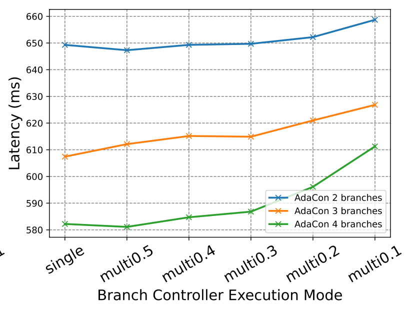
(b) Latency of AdaCon-YOLO



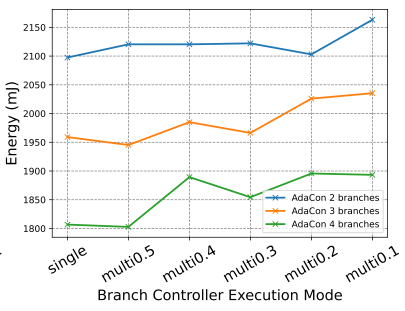
(c) Energy of AdaCon-YOLO



(d) Average Precision of AdaCon-RetinaNet



(e) Latency of AdaCon-RetinaNet



(f) Energy of AdaCon-RetinaNet

Figure 6.5: Evaluation of the Average Precision, the Latency and the Energy for AdaCon-YOLO and AdaCon-RetinaNet models under different branch controller execution modes. *single* represents single-branch execution mode, while *multix* represents multi-branch execution mode where x represents the confidence score threshold used by the branch controller. Input image resolution is 416×416.

add some flexibility during runtime because they enable a trade-off between the efficiency and the accuracy without the need to use a different model, or even re-train the existing model.

To analyze the effects of the different branch controller execution modes, we compare the accuracy, energy, and latency of the adaptive models under different branch controller execution modes as shown in Figure 6.5. The x-axis gives the different branch controller execution modes. *single* represents single-branch execution mode, while *multix* represents multi-branch execution mode where x represents the confidence score threshold used by the branch controller. The y-axis shows the mean average precision, the latency per inference in milliseconds, and the energy consumption per inference in millijoules in Figures 6.5a, 6.5b, 6.5c, respectively for AdaCon-YOLO, and in Figures 6.5d, 6.5e, and 6.5f, respectively for AdaCon-RetinaNet. We can notice that as the branch controller threshold decreases, more branches are executed, boosting the accuracy. On the other hand, executing more branches adds an overhead to the latency and the energy of our adaptive model. Figures 6.5b, and 6.5c show that multi-branch execution for AdaCon models with a larger number of branches can have a bigger overhead on the latency and energy. This is reasonable because if the threshold for multi-branch execution is low, more branches get executed, adding a bigger computational overhead when compared to AdaCon models with a fewer number of branches.

6.4.6 Pareto-Frontier analysis for AdaCon

As mentioned in Section 6.4.3, and 6.4.5, AdaCon has two different knobs that can be tuned to achieve the required performance and efficiency trade-off. These knobs are the number of branches, and the branch controller execution mode. We combine the settings from the two knobs, and generate 30 different adaptive models. We name our adaptive models as (*nB-mode*), where n refers to the number of branches in the AdaCon model, and *mode* refers to the branch controller execution mode (i.e., *multi* or *single*), along with the branch controller threshold in case of *multi* execution. In Figure 6.6, we analyze the energy/accuracy and the latency/accuracy trade-offs for AdaCon-RetinaNet, and AdaCon-YOLO along with the static baselines. In this experiment, we use input resolution 416×416.

Figures 6.6a, and 6.6c show the energy/accuracy trade-off for AdaCon-RetinaNet, and AdaCon-YOLOv3, respectively. The x-axis shows the energy per inference in millijoules, while the y-axis shows the average precision. The red circles represent the static baselines, and the green stars represent the Pareto-frontier adaptive models. The Pareto-frontier models maximize the accuracy, and minimize the energy. Finally, the dominated models (i.e., the models with lower accuracy, and higher energy consumption than other models) are shown as blue triangles. Similarly, in Figures 6.6b, and 6.6d, the green stars represent

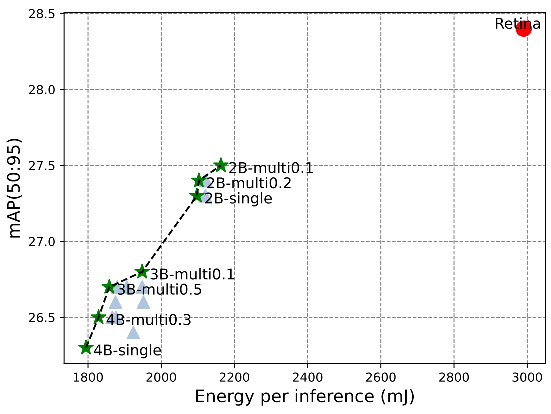
Table 6.3: Results - AdaCon vs Static Models Efficiency Metrics - Sparam represents the total number of parameters for the model. Dparam represents the number of parameters used by the adaptive model at a time. MACs is the number of multiply-accumulate operations.

Model	**Image**	**Sparam**	**Dparam**		**MACs**	
	Size	(M)	(M)	**(%)**	($Gops$)	**(%)**
RetinaNet	416	32.44	0.00	-	41.08	-
AdaCon 2B-multi 0.1	416	32.97	30.15	**-7.1%**	26.50	**-35.5%**
AdaCon 3B-multi 0.1	416	32.76	29.13	**-10.2%**	20.98	**-48.9%**
RetinaNet	320	32.44	0.00	-	24.27	-
AdaCon 2B-multi 0.3	320	32.97	30.15	**-7.1%**	15.49	**-36.2%**
AdaCon 3B-multi 0.5	32.76	29.13	**-10.2%**	12.19	**-49.8%**	
AdaCon 4B-single	320	32.97	28.74	**-11.4%**	12.06	**-50.3%**
YOLOv3	416	61.92	61.92	-	33.01	-
AdaCon 2B-multi0.1	416	58.31	51.53	**-16.8%**	28.34	**-14.1%**
AdaCon 3B-multi0.3	416	56.75	48.44	**-21.8%**	27.11	**-17.9%**
AdaCon 5B-single	416	56.95	45.88	**-25.9%**	26.10	**-20.9%**
YOLOv3	320	61.92	61.92	-	19.53	-
AdaCon 2B-multi0.2	320	58.31	51.59	**-16.7%**	16.78	**-14.1%**
AdaCon 3B-multi0.5	320	56.75	47.88	**-22.7%**	15.91	**-18.5%**
AdaCon 4B-single	320	56.90	46.56	**-24.8%**	15.60	**-20.1%**

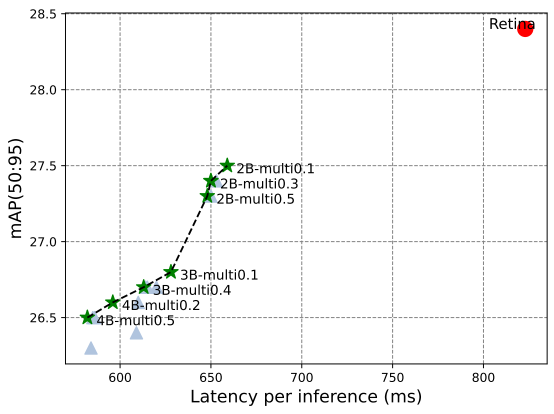
the pareto-frontier models, which maximize the accuracy, and minimize the latency. We can notice that the Pareto-frontier includes AdaCon models with a different number of branches, and different branch execution modes. The Pareto-frontier models cover a range of efficiency-performance trade-off, and provide run-time flexibility for our AdaCon models.

For RetinaNet, our AdaCon Pareto-frontier models achieve around 27% to 45% reduction in the energy consumption, and 20% to 27% reduction in latency, while losing less than two points of average precision. Similarly, for YOLOv3, our adaptive Pareto-frontier models achieve around 8% to 17% reduction in the energy consumption, and 8% to 13% reduction in latency, while sometimes slightly increasing the accuracy.

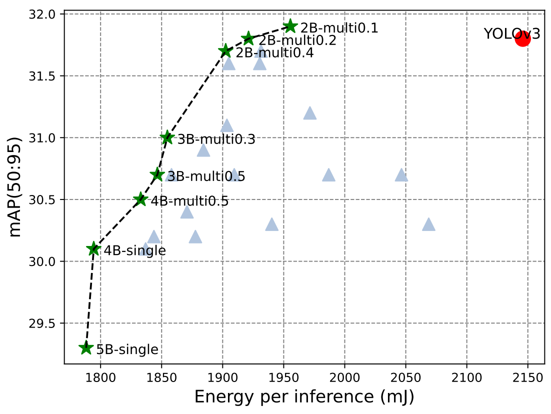
To provide more insights on the performance of our adaptive object detection models, we show the detailed analysis for the static baseline, as well as some of the Pareto-frontier



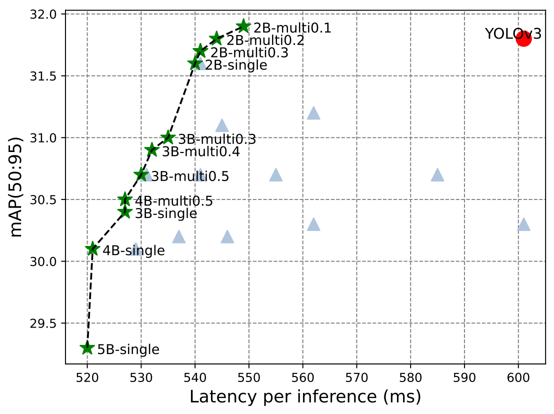
(a) Energy/Accuracy trade-off for static and AdaCon-RetinaNet models



(b) Latency/Accuracy trade-off for static and AdaCon-RetinaNet models



(c) Energy/Accuracy trade-off for static and AdaCon-YOLOv3 models



(d) Latency/Accuracy trade-off for static and AdaCon-YOLOv3 models

Figure 6.6: Energy, Accuracy and Latency Trade-offs for the different adaptive models. The adaptive models are named as (nB-*mode*), where n refers to the number of branches in the AdaCon model, and *mode* refers to the branch controller operation mode (i.e., *single* or *multi*), and the branch controller threshold in case of *multi* execution.

AdaCon models in Tables 6.4 and 6.3. In Table 6.4, we analyze three different standard metrics for the model accuracy: AP_{50}, AP_{75}, and $AP_{50:95}$. AP_{50} and AP_{75} represent the average precision with intersection over union (IOU) > 0.5 and > 0.75, respectively. $AP_{50:95}$ is the mean of the average precision for IOU ranging from 0.5 to 0.95 with a step size of 0.05, so it is the most representative to the overall accuracy of the model. We also analyze some efficiency metrics: the latency per inference in milliseconds, and energy per inference in millijoules. In Table 6.3, *Static parameters (Sparam)* represents the total number of parameters for the model. *Dynamic parameters (Dparam)* represents

Table 6.4: Results - AdaCon vs Static Models Accuracy and Efficiency Metrics - Latency and Energy are measured per inference. Efficiency is *Accuracy*/(*Energy* × *Latency*)

Model	**Image Size**	**Accuracy** $AP_{50:95}$	(%)	AP_{50}	AP_{75}	**Latency** (ms)	(%)	**Energy** (mJ)	(%)	**Efficiency** (%)
RetinaNet	416	28.4	-	44.7	29.6	823.0	-	2990.4	-	-
AdaCon 2B-multi 0.1	416	27.5	**-3.2%**	43.5	28.6	659.0	**-19.9%**	2163.2	**-27.7%**	**+67.2%**
AdaCon 3B-multi 0.1	416	26.8	**-5.6%**	42.8	27.8	628.0	**-23.7%**	1947.9	**-34.9%**	**+89.9%**
RetinaNet	320	26.1	-	41.3	27.1	627.0	-	2349.7	-	-
AdaCon 2B-multi 0.3	320	25.1	**-3.8%**	40.1	26.1	504.0	**-19.6%**	1521.5	**-35.2%**	**+84.8%**
AdaCon 3B-multi 0.5	320	24.4	**-6.5%**	39.2	25.4	483.0	**-23.0%**	1351.2	**-42.5%**	**+111.0%**
AdaCon 4B-single	320	24.0	**-8.0%**	38.5	24.7	458.0	**-27.0%**	1273.1	**-45.8%**	**+132.3%**
YOLOv3	416	31.8	-	55.3	33.1	601.0	-	2145.9	-	-
AdaCon 2B-multi0.1	416	31.9	**+0.3%**	53.2	33.6	549.0	**-8.7%**	1955.5	**-8.9%**	**+20.5%**
AdaCon 3B-multi0.3	416	31.0	**-2.5%**	52.1	32.6	535.0	**-11.0%**	1854.7	**-13.6%**	**+26.7%**
AdaCon 5B-single	416	29.3	**-7.9%**	49.7	30.5	520.0	**-13.5%**	1788.2	**-16.7%**	**+27.8%**
YOLOv3	320	29.3	-	52.0	30.4	414.0	-	1451.1	-	-
AdaCon 2B-multi0.2	320	29.1	**-0.7%**	49.1	30.6	380.0	**-8.2%**	1256.3	**-13.4%**	**+25.0%**
AdaCon 3B-multi0.5	320	28.0	**-4.4%**	47.6	29.1	366.0	**-11.6%**	1206.3	**-16.9%**	**+30.0%**
AdaCon 4B-single	320	27.3	**-6.8%**	46.6	28.3	359.0	**-13.3%**	1205.4	**-16.9%**	**+29.4%**

the number of parameters used by our adaptive model at a time. We used *Dparam* as an estimate of the memory footprint of our adaptive model. MACs is the number of multiply-accumulate operations measured in Giga operations.

In general, we want to increase the accuracy, and reduce the energy as well as the latency of our models. That is why we define the *Efficiency* metric as *Accuracy*/(*Energy* × *Latency*) to compare the overall quality of the static versus the adaptive models. Our Pareto-frontier AdaCon models can achieve up to 1.32×, and 30% improvement in *Efficiency* over the static baseline for RetinaNet, and YOLOv3, respectively. The results show that our AdaCon technique achieves higher efficiency for RetinaNet compared to

YOLOv3. The reason is that the reductions in latency and energy are directly proportional to the reduction in the number of MAC operations as shown in Tables 6.4 and 6.3. Our technique is mainly focused on the detection head component of the object detection model. The detection head represents around 62% and 25% of the total number of MAC operations for RetinaNet and YOLOv3, respectively. That is why the improvements are more significant for RetinaNet.

6.5 Conclusion

In this chapter, we present a novel spatial-context aware adaptive network for object detection. Using the prior knowledge about the co-occurrence of objects in the real world scenes, we categorize the objects according to their probability to occur jointly. Then, we design an energy-efficient adaptive model that consists of three parts: a backbone to extract the features in the images, a branch controller that route the output of the backbone to the right specialized branch(es) according to the spatial context of the input image, and a pool of context-specialized branches. Our method improves the overall model efficiency by up to 30% for YOLOv3 and 132% for RetinaNet. AdaCon reduces the energy consumption by 8% to 45%, the latency by 8% to 27%, with a small loss in the accuracy.

CHAPTER 7

Efficient Inference with Sample Awareness

7.1 Introduction

Modern computer-vision-based applications require solving multiple tasks simultaneously in order to form a complete perception of the surrounding visual environment. For example, a simple augmented reality application might need to determine the surface normals, estimate the depths, detect an object of interest, and track it. Similarly, an autonomous vehicle should be able to detect both static and dynamic objects in the scene, determine their proximity and track them [73, 116]. Moreover, all that complex processing needs to be performed on every input frame in real-time, which is extremely challenging, especially since those applications usually run on resource-constrained devices with strict compute, memory, and energy budgets. That is why enabling efficient computation models where these tasks can be performed simultaneously is crucial to make those applications practical.

In recent years, Multi-task learning (MTL) has been used to learn related vision tasks simultaneously [138, 98, 173]. On the one hand, MTL models leverage shared representations and inter-task interactions to improve performance on each task, potentially outperforming single-task models. On the other hand, compared to single-task models, MTL models can reduce both the memory footprint, the energy consumption, and the latency during inference since they avoid recomputing the features in the shared layers. Different approaches have been proposed to determine which layers should be shared across tasks and which layers should be task-specific in MTL models [114, 163, 140].

One common MTL approach is to have a shared encoder that extracts the critical features from the input scene, followed by some task-specific decoders that predict the output of the corresponding task [153, 152, 163]. Generally, the shared encoder in MTL models needs to have a large representational capacity in order to generalize well to various tasks and input data from different complexities. Vision Transformers have proven to be a powerful tool for extracting strong feature representations from the inputs, improving the performance of the downstream tasks [101, 45, 150]. Hence, few recent works have explored using them as a shared encoder for feature extraction in MTL models [13, 166]. Although these approaches achieve impressive performance outcomes, their computational complexity and memory footprint are usually huge, making them impractical for real-time processing. That's why, in this work, we focus our efforts on improving the computational complexity of transformer-based Multi-task Learning models.

Real-world visual scenes have large variations in complexity. For example, a scene with a single object and an open background would be easier to process than one with an occluded object and a cluttered background. We argue that treating all input frames equally regarding processing complexity can be wasteful. Therefore, we propose using an adaptive inference policy to reduce the computational complexity of MTL models based on the input complexity. Dynamic Input-dependent inference policies have only been explored for single-task image classification problems [172, 112]. However, MTL models are more complex due to inter-task dependencies and complex architectures.

To this end, we propose an adaptive MTL framework that recognizes the unnecessary computations in the model depending on the input complexity. We achieve this by learning an adaptive task-aware policy network that ultimately decides on which parts of the MTL model to activate during runtime. Our contributions can be summarized as follows:

- We propose an adaptive Multi-task Learning framework that optimizes Transformer-based MTL models depending on the input complexity.
- We introduce a task-aware policy network, and we show that it is more effective in recognizing the unnecessary computations in MTL models compared to task-agnostic policy networks.
- To prove that our policy network can be easily plugged to improve MTL models' efficiency, we combine AdaMTL with a SOTA MTL model [152] and show that AdaMTL boosts the accuracy by 7.8% while improving the efficiency by $3.1\times$.
- We deploy AdaMTL on the Vuzix M4000 AR glasses [158], reducing the inference latency and the energy consumption by up to 21.8% and 37.5%, respectively, compared to the static MTL model.

The rest of the chapter is organized as follows. We review the related work in Section 7.2. Then, we introduce our adaptive input-dependent MTL framework in Section 7.3. Next, we show the qualitative and quantitative analysis of our methodology as well as the ablation study in Section 7.4. Finally, we conclude in Section 7.5.

7.2 Related Work

Deep Multi-task Architectures For dense prediction tasks, deep multi-task architectures usually consist of an encoder that extracts a feature representation from the input frame, followed by task-specific decoders that generate predictions for each of the downstream tasks [163, 114, 152]. Researchers categorize multi-task architectures based on the location of task interactions into encoder- and decoder-focused architectures [153].

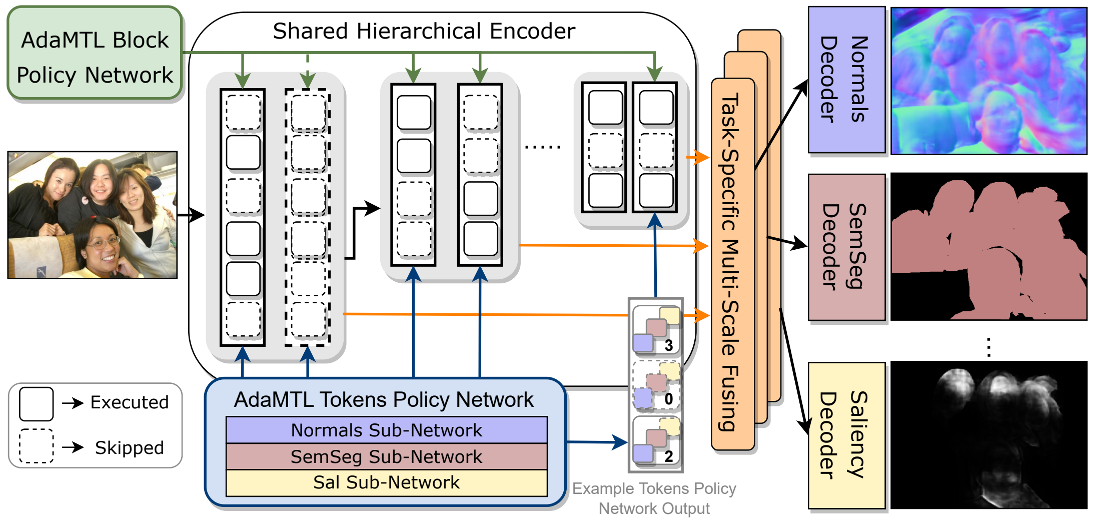
Encoder-focused architectures share the information between tasks in the encoder stage [114, 99], while the decoder-focused architectures exchange information between tasks in the decoding stage [152, 163]. Moreover, some approaches share the information across tasks in both encoder and decoder stages [140, 13]. In this chapter, our baseline MTL model follows an encoder-focused architecture where a shared encoder is used to extract visual features from the input frame, followed by task-specific decoders. Moreover, our MTL framework can easily adopt other decoder-based task-interaction techniques, as we will show in Subsection 7.4.3.

Vision Transformer Models Transformers [155] have achieved impressive performance improvements in various domains such as language understanding [33], speech recognition [47], and computer vision [34, 150]. The self-attention modules in transformers have proved to be capable of extracting strong feature representations from the inputs, improving the performance of the downstream tasks. To preserve local visual context, Vision Transformers (ViTs) split the input image into patches which are embedded as tokens [34]. Those tokens pass through successive ViT blocks. Each ViT block has a multi-head attention module followed by a multi-layer perceptron module to extract the global relationship among input tokens. Improvements have been made to ViTs enabling data-efficient training [150] as well as efficient inference [45]. Inspired by ResNets [58], Swin Transformers [101] use a hierarchical ViT block architecture as well as shifted window attention to serve as a general-purpose backbone for computer vision tasks. That is why, ViTs started replacing CNNs as a backbone for different computer vision tasks such as image classification [150, 45], object detection [18], and segmentation [139]. Moreover, several works proposed using them for multi-task learning [13, 166].

Sparsely-Activated Vision Models As computer vision models become increasingly complex, researchers have started sparsifying the models to prevent redundant computations. For example, DSelect-k [56] and M^{3}ViT [92] proposed Mixture-of-Experts (MoE) architectures that use trainable sparse gates to activate a subset of experts depending on the given input. Other methods employ early-exiting strategies to adaptively allocate computations depending on input complexity [179, 69]. In early exiting, an exit layer is added at every intermediate exit, and a confidence metric is used to take the decision. However, such strategies could be impractical for MTL models due to the huge computational overhead of adding MTL exit layers (i.e., a set of task-specific decoders) at each intermediate exit as well as the difficulty of confidence estimation in the encoder stage. Other dynamic models skip redundant layers [160], while others use a bottleneck layer to direct the computations in an input-dependent manner [121]. More recent approaches use lightweight policy networks to generate execution strategies based on the input complexity [172, 112]. Despite the effectiveness of these approaches for image classification problems, adaptive policies have not yet been explored for MTL scenarios. Sparsifying MTL models is more challenging; the multi-objective nature of those models complicates the overall objective of optimizing the performance of the different tasks in addition to the policy networks. Therefore, there is a need to explore and develop techniques that can adaptively sparsify MTL models based on input complexity while being aware of the multi-objective nature of those models.

7.3 Method

In this section, we propose AdaMTL – an adaptive end-to-end Multi-task Learning framework. We start by presenting our transformer-based MTL architecture in Subsection 7.3.1. In Subsection 7.3.2, we introduce our task-aware policy network that dynamically recognizes the unnecessary computations in the MTL model depending on the input complexity. Finally, we explain our proposed multi-staged task-aware training recipe in Subsection 7.3.3.



AdaMTL Block Policy Network
Shared Hierarchical Encoder
.....
Task-Specific Multi-Scale Fusing
Normals Decoder
SemSeg Decoder
Saliency Decoder
Executed
Skipped
AdaMTL Tokens Policy Network
Normals Sub-Network
SemSeg Sub-Network
Sal Sub-Network
3
0
2
Example Tokens Policy Network Output

Figure 7.1: Overview of our proposed AdaMTL framework integrating our AdaMTL Block Policy Network and our AdaMTL Tokens Policy Network. The AdaMTL Block policy network decides on which blocks to activate during runtime. If the decision is to activate a certain block, our AdaMTL Tokens policy network runs to decide which tokens to process through that block. Our policy network achieves a task-aware behavior that improves the quality of the generated policies for multi-task learning models.

7.3.1 Architecture Overview

Figure 7.1 illustrates an overview of our proposed adaptive end-to-end Multi-task Learning framework - AdaMTL. AdaMTL consists of two main components: a static MTL model and a lightweight policy network. Our MTL model has three parts: a Shared Hierarchical Encoder, a learnable task-specific multi-scale fusing layer, and a pool of task-specific decoders. We adopt an off-the-shelf hierarchical Vision Transformer *Swin* [101] as our shared encoder to extract visual features from the input frames. Vision Transformers (ViTs) usually split the input image into patches which are embedded as tokens. Those tokens pass through successive ViT blocks. Each ViT block has a multi-head attention module followed by a multi-layer perceptron module to extract the global relationship among input tokens. Our learnable multi-scale fusing layers use a residual blocks-based architecture [58]. They are added to combine the features at different scales (i.e., receptive fields) in an informative way for every downstream task. Finally, we use simple task-specific decoders consisting of two convolutional layers. Each decoder takes the output from the corresponding multi-scale fusing layer to generate the corresponding task predictions.

On top of our static MTL model, AdaMTL adds a lightweight policy network that runs alongside the original model to decide which parts of the model to activate, adapting to the complexity of the input frame. Our policy network works as a multi-grained decision maker; it consists of an *AdaMTL Block policy network* and an *AdaMTL Tokens policy network*. The *AdaMTL Block policy network* decides on which blocks to activate during runtime. If the decision is to activate a certain block, our *AdaMTL Tokens policy network* runs to decide which tokens to process through that block. Our intuition behind AdaMTL is that not all patches in the input frame are equally informative; for example, a patch of an input frame from the background is less informative than a patch with a person for a task such as human-parts detection. Moreover, the receptive field at which different patches should be processed differs depending on the scale of the objects in the input frame. For example, an image with multiple smaller objects might benefit from being processed by the first few layers of the models where the receptive field is small. However, the later layers with larger receptive fields are essential for an accurate output on a zoomed-in frame with one object. In other words, our *AdaMTL Policy Network* decides which patches are needed to accurately perform the downstream tasks and the receptive field at which those patches should be processed.

7.3.2 AdaMTL Policy Network

Our policy network works as a multi-grained decision maker; the *Block Policy Network* decides on which blocks to activate, while the *Tokens Policy Network* decides on which tokens (i.e., patches) to process through the activated blocks. Our *Block Policy Network* generates a learnable binary mask for each block in the shared encoder. We use this binary mask to recognize the necessary blocks needed by the MTL model in order to perform well on the downstream tasks. Similarly, for each block, we attach a *Tokens*

Policy Network that generates a learnable policy to determine the tokens that need to be processed through the activated block. Intuitively, the policy network should learn to recognize the most informative patches in every input frame as well as the receptive field at which it needs to be processed. Each policy network has two simple fully-connected layers, followed by a Gumbel Softmax activation to generate binary masks [75]. We devise two different settings for the policy network: a *task-agnostic* policy network and a *task-aware* policy network.

Task-agnostic Policy: In the *task-agnostic* setting, the policy network is unaware of the number of downstream tasks. We achieve this by co-training the whole policy network alongside the MTL model. We mainly experiment with this setting to show the necessity of task awareness while creating an effective policy network for multi-task scenarios.

Task-aware Policy: In the *task-aware* setting, we want our policy network to capture task-specific computational needs. We achieve this by dividing the policy network into sub-networks, where each sub-network is responsible for recognizing the necessary blocks/tokens for the corresponding task. As shown in Figure 7.1, the *AdaMTL Tokens Policy Network* consists of a sub-network for each task (i.e., Normals Controller, Semantic Segmentation Controller, Saliency Controller, etc.). Each sub-network makes a decision that is plausible to its respective task. Finally, to get a unified policy, we make a decision to activate a token if at least one of the tasks needs to process it. The intuitive way to combine the masks would be to apply the logical ORing operation on all the generated task-specific masks. However, to make it more learnable, we combine the masks using addition and clamping. As shown in the example output from the Tokens Policy Network in Figure 7.1, the policy network only activates a token if at least one task-specific policy sub-network decides to activate it.

7.3.3 AdaMTL Training Recipe

For our adaptive MTL framework to work effectively, we need to learn the MTL model weights as well as the execution policy (i.e., policy network's binary masks) that achieves the target efficiency without compromising the accuracy of the various tasks in our MTL model. Our end-to-end training recipe consists of 3 stages:

Stage 1: Static MTL Model Training

First, we train a static MTL model. We adopt a shared encoder along with task-specific decoders to perform multi-task learning as explained in Subsection 7.3.1. For our shared encoder, we use the publicly-available pre-trained Swin Transformer backbones [101]. Then, we attach the multi-scale fusing layer as well as task-specific decoders, and we fine-tune the end-to-end MTL model to get our static baseline. In this stage, our loss function is the weighted sum of the losses of the various downstream tasks as follows.

$$L_{stage1} = \sum_{i}^{m} \omega_{task\_i} \times L_{task\_i} \tag{7.1}$$

where ω_{task_i} and L_{task_i} are the weight and the loss of the various tasks in the MTL model, respectively, and m is the number of downstream tasks in the MTL model. We adopt the task weights used by Vandenhende *et. al.* [152].

Stage-2: Policy Network Initialization

We aim to co-train both the policy network as well as the MTL model. Randomly initializing the policy network while co-training can lead to degrading the model accuracy since the policy network would make random decisions in the earlier epochs. That's why we choose initialization weights for the policy network that activates all the blocks and the tokens. To achieve this, we freeze the static MTL model and pre-train our AdaMTL policy network with the following loss function:

$$L_{stage2} = \sum_{k}^{blocks}(1 - M_{b/k}) + \sum_{k}^{blocks}(1 - M_{t/k}) \tag{7.2}$$

Where $M_{b/k}$ and $M_{t/k}$ are the output masks generated by the Block Policy Network and the Tokens Policy Network attached to the Encoder block k, respectively. This results in an adaptive model that behaves exactly like the static model, where all blocks and tokens are activated. This acts as a good initialization point to start co-training the policy network and the MTL model.

Stage 3: Policy Network/MTL Model Co-training

In this stage, we co-train the policy network along with the MTL model. Our goal is to learn the binary masks that our policy network should generate in order to meet the target computational budget (i.e., the target percentage of the MTL model components to be activated) while maintaining the accuracy of the downstream tasks. Thus, we use a multi-objective loss as shown in Equation 7.3. Our loss function incorporates the various task losses L_{tasks} as well as the efficiency loss L_{eff} multiplied by a factor α. α represents the efficiency weight which controls the trade-off between accuracy and efficiency. In our experiments, we set α to unity; however, different values can be used to control the trade-off depending on the application requirements.

$$L_{stage3} = L_{tasks} + \alpha L_{eff} \tag{7.3}$$

The efficiency loss incorporates the efficiency of the decisions made by the blocks as well as the tokens controller, as shown in Equation 7.4. In order to minimize the number of activated blocks, we set L_{blocks} as mean squared error (MSE) between the actual percentage of the activated blocks and the target percentage of activated blocks as shown in Equation 7.5. Similarly, we set L_{tokens} as the MSE between the actual percentage of the activated tokens and the target percentage of activated tokens. However, it is

Algorithm 3 AdaMTL - Alternating Task Training (ATT)

```
Input: $model$: Model w/ initialized policy network $Tasks$: Task names in MTL model
       $m$: Number of tasks in MTL model
       $epochs_{att}$: Number of epochs for ATT
       $L_i$: Loss for the epoch i
for $i$ in $0, \dots, epochs_{att}$ do
    $current\_task = Tasks[i\%m]$   $L_i = L_{current\_task} + \alpha L_{eff}$   for $task$ in $Tasks$ do
        if $task = current\_task$ then
            model.unfreeze_decoder($task$)   model.enable_policy_network($task$)
        end
        else
            model.freeze_decoder($task$)   model.disable_policy_network($task$)
        end
    end
    train_one_epoch($model, L_i$)
end
```

common for hierarchical ViTs to have more tokens in the earlier layers compared to the later layers (i.e., the earlier layers have smaller receptive fields, thus more patches, while later layers have larger receptive fields, thus fewer patches). This means that the number of tokens does not linearly reflect the computational complexity since layers with more tokens have smaller embedding dimensions per token, while layers with fewer tokens have larger embedding dimensions. That's why we multiply the percentage of tokens in each layer by a weight ω_d equivalent to the embedding dimension in this layer as shown in Equation 7.6.

$$L_{eff} = L_{blocks} + L_{tokens} \tag{7.4}$$

$$L_{blocks} = MSE(\frac{B_{activ}}{B_{total}}, \frac{B_{target}}{B_{total}}) \tag{7.5}$$

$$L_{tokens} = MSE(\frac{\omega_d \times T_{activ}}{T_{total}}, \frac{\omega_d \times T_{target}}{T_{total}}) \tag{7.6}$$

To make our policy network task-aware, we co-train each task-specific policy subnetwork independently with the end-to-end model. Sequentially co-training the policy sub-networks along with the end-to-end model suffers from catastrophic forgetting (i.e., the model gets biased towards behaving well on the last task, and the performance deteriorates on the earlier tasks). That's why we propose an *Alternating Task Training* (ATT) where we shift the focus between the tasks every one epoch. Algorithm 3 shows the steps of our ATT technique. For each epoch, we choose a task to focus on, and we set the loss accordingly, as shown in lines 2 and 3, respectively. Then, we only activate the decoder and the policy sub-network corresponding to the chosen task, as shown in lines 4-13. We train the MTL model for one epoch using that setting. Then, we move on to co-training the decoders and the policy sub-networks of the other tasks. Finally, we perform end-to-end fine-tuning to improve the overall model accuracy. The training loss remains the same as in Equation 7.3, and we unfreeze all the model's components. This results in an adaptive MTL model that generates task-aware inference policies depending on the complexity of the input.

7.4 Experiments

7.4.1 Setup

Dataset: We evaluate our method on the PASCAL dataset [35]. Following other work in MTL literature [166, 152, 163], we use the PASCAL-Context split that has annotations for various dense prediction tasks such as semantic segmentation, human part detection, surface normals estimation, and saliency distillation. It has 4,998 images in the training split and 5,105 in the validation split.

Implementation and Training details: We implemented AdaMTL using PyTorch. As mentioned in Subsection 7.3.1, we adopt the publicly available pre-trained Swin Transformer backbone [101] as our shared encoder. To get our adaptive MTL model, we employ a three-stage training recipe as explained in Subsection 7.3.3. In Stage 1, we fine-tune the Swin Transformer backbone along with our task-specific decoders for 1000

Table 7.1: Quantitative analysis of AdaMTL on PASCAL dataset. The table shows the accuracy by our adaptive MTL model compared to the single-task model as well as static MTL models. *H* and *L* represent high and low computational complexity targets for AdaMTL, respectively. Δ m (ST) shows the change in the average accuracy of the tasks compared to the single-task model, respectively. ↓ means the lower the better, while ↑ means the higher the better. **Bolded** values represent the Pareto-frontier of the accuracy-efficiency trade-off.

Policy	**Backbone**	**Image Size**	**Sal**	**Human Parts**	**Sem Seg**	**Normals**	**Δ m (ST)**
			maxF ↑	mIoU ↑	mIoU ↑	mERR ↓	(%) ↑
Single-Task	Swin-T	224	71.93	48.63	60.35	18.45	0.00
Static MTL		224	74.15	47.62	59.08	19.20	**-1.20**
AdaMTL (H)	Swin-T	224	73.72	47.64	57.13	19.16	**-2.18**
AdaMTL (L)		224	73.01	46.85	55.9	19.54	**-3.86**
Static MTL		224	75.00	50.66	61.84		
AdaMTL (H)	Swin-S	224	75.23	51.22	61.88	18.79	**+2.65**
AdaMTL (L)		224	74.75	50.07	59.91	18.61	**+1.31**
Static MTL		384	73.93	56.68	67.47	17.69	**+8.81**
AdaMTL (H)	Swin-B	384	76.55	56.28	65.04	17.74	**+8.43**
AdaMTL (L)		384	76.23	55.04	62.63	17.92	**+6.46**

epochs. Then in stage 2, we freeze the MTL model and initialize the policy network by training it to activate the whole MTL model. We run this stage for another 80 epochs. Finally, in stage 3, we use our proposed *Alternating Task Training technique* to co-train the policy network along with the MTL model for another 150 epochs, followed by fine-tuning the end-to-end AdaMTL model for another 150 epochs. Our method does not only increase the model efficiency during inference, but it also reduces the carbon emission since we avoid retraining complex ViTs from scratch by reusing off-the-shelf pre-trained backbones; AdaMTL needs only 1 V100 GPU for around 24-48 hours (i.e., depending on the used backbone) in order to run our end-to-end training recipe.

7.4.2 Quantitative Analysis

Accuracy-Efficiency Trade-off: We apply AdaMTL on three SOTA ViTs from the Swin Transformer family [101]. We include *Swin-Tiny*, *Swin-Small*, and *Swin-Base*, representing three different scales of ViTs in terms of computational complexity. We include two different computational complexity targets: *H* and *L*. *H* represents a higher

Table 7.2: Quantitative analysis of AdaMTL on PASCAL dataset. The table shows the efficiency metrics by our adaptive MTL model compared to the single-task model as well as static MTL models. *H* and *L* represent high and low computational complexity targets for AdaMTL, respectively. Δ FLOPS (ST) show the change in percentage of FLOPS compared to the single-task model, respectively. ↓ means the lower the better, while ↑ means the higher the better. **Bolded** values represent the Pareto-frontier of the accuracy-efficiency trade-off.

Policy	**Backbone**	**Image Size**	**GFLOPS** ↓	**Δ FLOPS (ST)** (%) ↓	**Params** (M) ↓
Single-Task	Swin-T	224	18.33	1 ×	111.42
Static MTL	Swin-T	224	5.79	**0.32** ×	34.77
AdaMTL (H)	Swin-T	224	5.34	**0.29** ×	34.87
AdaMTL (L)	Swin-T	224	4.82	**0.26** ×	34.87
Static MTL	Swin-S	224	12.5	0.68 ×	67.02
AdaMTL (H)	Swin-S	224	12.51	**0.66** ×	67.12
AdaMTL (L)	Swin-S	224	10.35	**0.57** ×	67.12
Static MTL	Swin-B	384	59.39	**3.24**×	108.66
AdaMTL (H)	Swin-B	384	51.29	**2.80**×	108.88
AdaMTL (L)	Swin-B	384	41.229	**2.25**×	108.88

computational budget where the target percentage of activated tokens and blocks are 60% and 90%, respectively. *L* represents a lower computational budget where both the target percentage of activated tokens and blocks are set to 50%. In both settings, the accuracy-efficiency trade-off weight (i.e., α in Equation 7.3) is set to unity. To evaluate the accuracy-efficiency trade-off by AdaMTL, we compare it to two baselines: (1) Single-Task models and (2) Static MTL models (i.e., our base MTL model before attaching our task-aware policy network). Table 7.1 shows the accuracy, the computational complexity (i.e., FLOPS) as well as the model size (i.e., Params) of AdaMTL compared to the single-task model and the static MTL models. Δ m (ST) and Δ FLOPS (ST) show the change in the average accuracy of the tasks and percentage of FLOPS compared to the single-task model, respectively. Results show how our method enhances the accuracy-efficiency trade-off for MTL models. For example, by applying AdaMTL to *Swin-S* backbone, we can get an MTL model with 43% less FLOPS and 1.31% more accuracy compared to the single-task model. Similarly, by applying AdaMTL to *Swin-T* backbone, we can get an MTL model with 71% less FLOPS and only 2.18% drop in accuracy compared to the single-task

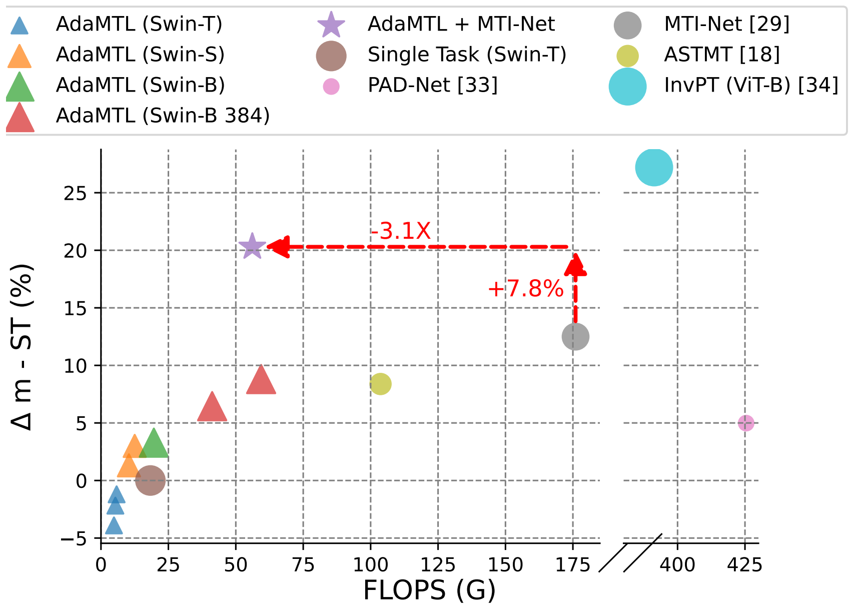


Figure 7.2: Accuracy-Efficiency trade-off by AdaMTL compared to SOTA MTL techniques. The x-axis shows the FLOPS, while the y-axis represents the average accuracy of the tasks compared to the single-task model.

Table 7.3: Comparison with SOTA MTL models.

Method	**Backbone**	Δ **m** (ST) $\uparrow$ (%)	Δ **FLOPS (ST)** $\downarrow$ (%)
PAD-Net [163]	HRNet-18	+4.98	23.21×
AdaMTL (Ours)	Swin-B	**+6.46**	**2.25** ×
ASTMT [109]	R26-DLv3	+8.38	5.66 ×
AdaMTL (Ours)	Swin-B	**+8.81**	**3.24** ×
MTI-Net [152]	ResNet-50	+13.49	9.6 ×
InvPT [166]	ViT-B	+27.20	21.35 ×

model. Therefore, given any target computational complexity, AdaMTL can meet it while potentially improving the accuracy.

Comparison to SOTA MTL models: We also compare the accuracy-efficiency trade-off of AdaMTL to four SOTA MTL models that vary in computational complexity and accuracy. Figure 7.2 shows the accuracy and the computational complexity of AdaMTL compared to those of *PAD-Net* [163], *ASTMT* [109], *MTI-Net* [152], and *InvPT* [166]. We can notice that AdaMTL dominates both *PAD-Net* and *ASTMT*. Moreover, AdaMTL achieves a more efficient trade-off compared to *MTI-Net* and *InvPT*. As shown in Table

Table 7.4: Combining AdaMTL with SOTA MTL components.

Method	**Backbone**	**Δ m** %	**Δ FLOPS** ↓	**Params** (M)
Single-Task	Swin-T	+0.00	18.33 G	111.42
AdaMTL	Swin-B	+8.81	1.1×	108.66
MTI-Net [152]	ResNet-50	+13.49	9.6×	91.00
AdaMTL + MTI-Net	Swin-B	**+20.29**	**3.1**×	101.14

7.3, AdaMTL has 3× less FLOPS than MTI-Net and 7× less FLOPS than *InvPT*. It is important to note that our effort in AdaMTL is directed specifically toward enabling efficient Multi-Task learning. That is why we argue that while the performance of *MTI-Net* and *InvPT* is impressive, their demanding computational complexity might not be suitable for real-time processing on resource-constrained devices.

7.4.3 Combining with SOTA MTL components

Our adaptive MTL framework can easily adopt other SOTA MTL components to further enhance the accuracy-efficiency trade-off. In this section, we show a case study where we integrate AdaMTL with the SOTA MTL concepts in *MTI-Net* [152] in order to improve both its efficiency and accuracy. *MTI-Net* has two main modules: (1) A multi-scale multi-modal distillation unit to model task interactions at different scales and (2) A feature propagation module that propagates distilled task information from lower to higher scales. In this experiment, we attach those two modules between the shared hierarchical encoder and the task-specific decoders in our AdaMTL framework. Following the exact same training recipe introduced in Subsection 7.3.3, the results in Table 7.4 show that AdaMTL can be integrated with other MTL modules from *MTI-Net* to boost *MTI-Net*'s accuracy by 7.8% while improving its efficiency by 3.1×.

7.4.4 Qualitative Analysis

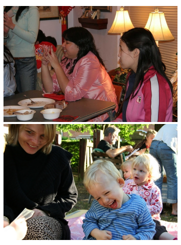
Figure 7.3 shows some insights about the allocated amount of computations by AdaMTL for input frames of different visual complexity. Figures 7.3a, 7.3b, 7.3c, and 7.3d shows examples where AdaMTL allocated 55%, 65%, 75%, and 85% of the static model FLOPS respectively. We can see that AdaMTL allocates fewer computations to simple frames with fewer objects as compared to complex scenes with multiple objects and cluttered backgrounds. Figure 7.3e shows a histogram of AdaMTL's computational complexity per example for all the images in PASCAL validation set. The red line represents the average number of FLOPS needed to process all the images. We can notice that AdaMTL adapts to the large variation in the computational complexity requirement by various images in the dataset.

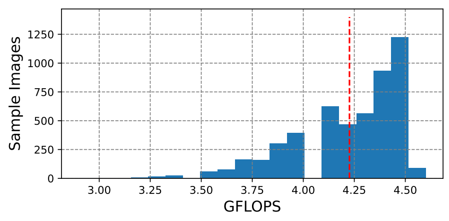
To gain more insights into the decisions made by our policy network, we visualize a sample of the generated tokens masks in Figure 7.4. Column 7.4a represents the input frames, while columns 7.4b, 7.4c, and 7.4d represent the generated tokens masks by our policy network at different layers, such that the white areas represent activated tokens. We can notice that the policy network tends to activate more tokens in the earlier layers to understand the global features of the input frame. Then, it narrows down its scope in the later layers, focusing on the most informative patches (i.e., patches with the main objects) in the input frame. We can also notice that more tokens are activated in the sample in the bottom row which is expected since it is more complex (i.e., multiple objects and cluttered background).

(a) 55% FLOPS

(b) 65% FLOPS

(c) 75% FLOPS


(d) 85% FLOPS



(e) GFLOPs Histogram of AdaMTL on PASCAL dataset

Figure 7.3: Qualitative insights on the computational budget allocated by AdaMTL to process input frames of different complexity. We can notice that AdaMTL assigns fewer computations for simpler scenes, as in (a) and (b), while it justly assigns more computations to process more complex and cluttered scenes as in (c) and (d).

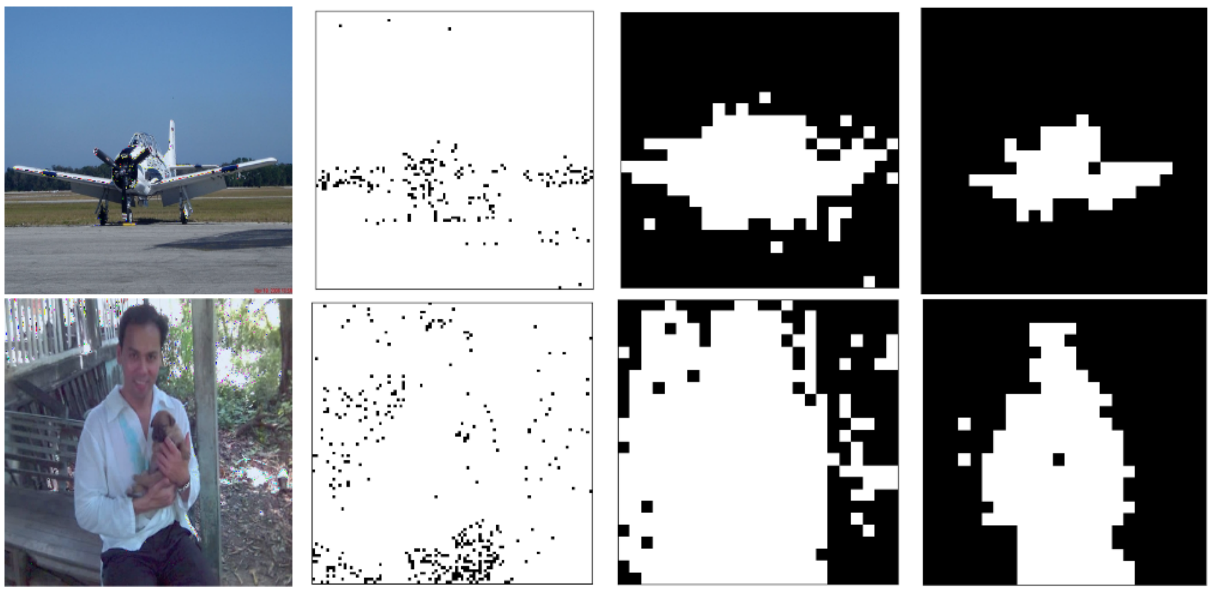


Figure 7.4: Sample of the generated masks by AdaMTL tokens policy network. The white areas in (b)-(d) represent activated tokens.

Table 7.5: Performance Analysis on Vuzix Augmented Reality Glasses [158]. We analyze the percentage of the inference latency and the energy consumption by our AdaMTL model compared to its corresponding static model.

Method	Backbone	Inference Latency (%)	Energy Consumption (%)
AdaMTL	Swin-T	-20.6%	-35.3%
AdaMTL	Swin-S	-21.8%	-37.5%

7.4.5 Deployment on Vuzix M4000 AR glasses

We compile our AdaMTL model using PyTorch for Android [129], and we deploy it on the Vuzix M4000 AR glasses [158]. The Vuzix glasses have an 8 Core 2.52Ghz Qualcomm XR1 board with 6GB RAM. It operates using Android 11.0. Using the *Battery Historian* tool [63] to profile the energy consumption on the device, we record the average latency and energy consumption across random samples from the PASCAL validation dataset. Table 7.5 shows that AdaMTL reduces the inference latency and the energy consumption by up to 21.8% and 37.5%, respectively, compared to its corresponding static MTL model.

Table 7.6: Comparison between the quality of our task-aware policy, the task-agnostic, and the random execution policy.

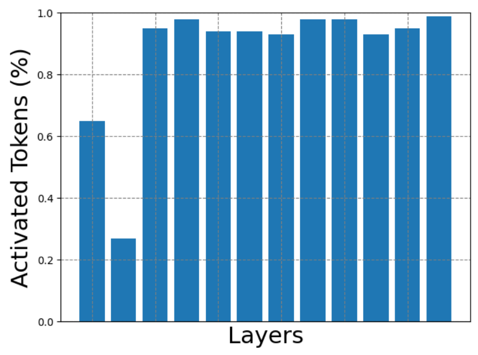
Policy	**Backbone**	Δ **m** (ST) (%) ↑	Δ **FLOPS** (ST) ↓
Random	Swin-T	-34.53	0.24×
Random+		-8.64	0.24×
Task-Agnostic		-5.68	0.24×
Task-Aware		**-3.86**	**0.26**×
Random	Swin-B	-34.86	0.82×
Random+		-8.57	0.82×
Task-Agnostic		+0.77	0.84×
Task-Aware		**+1.12**	**0.83**×

Table 7.7: The contribution of different adaptive dimensions to AdaMTL. Δ m is measured relative to the static MTL model.

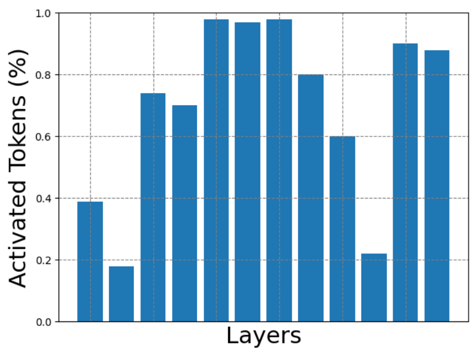
Adaptive Blocks	Adaptive Tokens	Δ m %	FLOPS (G)
×	×	-0.00	5.8
✓	×	-1.66	5.23
×	✓	-1.46	5.08
✓	✓	**-0.85**	5.37

7.4.6 Ablation Study

Quality of the learnt inference policies: To analyze the quality of the learned inference policies by our task-aware policy network. Table 7.6 compares the accuracy-efficiency trade-off achieved by our task-aware policy network (i.e., referred to as Task-Aware) to three other baselines: (1) Random where we activate random blocks and tokens from the static MTL model, (2) Random+ where we again activate random blocks and tokens from the static MTL model but after performing our adaptive training pipeline, and (3) Task-Agnostic policy network explained in Subsection 7.3.2. The results in Table 7.6 show that the policies learned by our task-aware policy network outperform the other baselines. We can also notice that performing adaptive training gives the MTL model robustness towards sparsification (i.e., Using a random policy on the static model reduced the accuracy by 34%, while it only reduced the accuracy by 8% when applied to the adaptively trained MTL model).



(a) Without weighted tokens



(b) With weighted tokens

Figure 7.5: Loss function ablation: The figure illustrates the average percentage of activated tokens over the 12 blocks in Swin-T using (a) non-weighted tokens loss and (b) weighted tokens loss.

Analysis of the adaptation along each component of our policy network To understand the contribution of both components of our policy network (i.e., blocks policy network and tokens policy network) to the accuracy-efficiency trade-off of AdaMTL, we compare the results from four different settings: (1) the static model where neither component of the policy network is activated, (2) AdaMTL while activating the block policy network only, (3) AdaMTL while activating the tokens policy network only, and (4) AdaMTL where both policy networks are activated. Table 7.7 shows that enabling both components enhances the accuracy-efficiency trade-off of AdaMTL.

Analysis of the behavior of our loss function To analyze the importance of the weight factor ω_d in Equation 7.6 of our loss function, we visualize the average percentage of activated tokens across the blocks of Swin-T encoder with and without ω_d in Figure 7.5. We can notice that adding the ω_d factor to the loss function leads to a more distributed FLOPS reduction across different blocks, which is essential for the effectiveness of our adaptive MTL framework.

7.5 Conclusion

In this chapter, we propose AdaMTL - an adaptive framework that learns task-aware inference policies for the MTL models in an input-dependent manner. We achieve this by co-training a lightweight policy network along with our MTL model. During runtime, our policy network recognizes the unnecessary computations and dynamically chooses an execution strategy depending on the input complexity and the target computational budget. Our experiments on PASCAL dataset demonstrate that AdaMTL reduces the computational complexity by 43% while improving the accuracy by 1.32% compared to the single task models. Combined with SOTA MTL components, AdaMTL boosts the accuracy by 7.8% while improving the SOTA MTL model efficiency by $3.1\times$. Finally, we deployed AdaMTL on Vuzix M4000 AR glasses showing up to 21.8% and 37.5% reduction in inference latency and energy consumption, respectively, compared to the static MTL model.

CHAPTER 8

Summary and Possible Extensions

8.1 Summary of the Dissertation

In Chapter 3, we focused on exploring compression techniques such as quantization, particularly low-precision quantization, known for enhancing neural network efficiency. Our analysis identified that non-quantized elementwise operations, prevalent in certain layers like parameterized activation functions, batch normalization, and quantization scaling, significantly contribute to the inference cost in low-precision models, an aspect often missed by current state-of-the-art (SOTA) efficiency metrics like Arithmetic Computation Effort (ACE). To address this discrepancy, the chapter introduced an improved metric, ACE_{v2}, designed to more accurately reflect the inference costs and energy consumption of quantized models on machine learning hardware. Additionally, the chapter presented *PikeLPN*, a model innovating in efficiency by applying quantization to both elementwise and multiply-accumulate operations. *PikeLPN* was shown to achieve Pareto-optimality in the efficiency-accuracy trade-off, marking up to a 3.5× improvement in efficiency over existing SOTA low-precision models.

In Chapter 4, we delved into the practicality of training deep learning models on edge devices, highlighting the opportunity for neural networks to adaptively learn from new data post-deployment, despite the memory constraints typical of such devices. Our investigation pinpointed the memory footprint, particularly from activations, as the principal challenge for on-edge training. Traditional incremental training strategies, which typically fine-tune only the latter layers, compromise on the potential accuracy benefits of full-model retraining. To address this, we introduced *BitTrain*, a methodology that leverages activation sparsity through a novel bitmap compression technique, significantly reducing memory requirements during training. This approach involved saving activations in a compressed format during the forward pass and reconstructing them for use in the backward pass, ensuring seamless integration with existing deep learning frameworks without compromising training accuracy. Our experimental findings demonstrated that *BitTrain* can achieve up to a 34% reduction in memory usage with 50% sparsity, and further pruning can lead to over 70% sparsity, resulting in up to a 56% decrease in memory footprint. This innovation represented a significant step towards enhancing machine learning capabilities on edge devices.

In Chapter 5, we investigated how extracting context from historical data patterns can significantly improve a machine learning (ML) framework, particularly focusing on optimizing compute resource allocation during runtime. This concept was applied to a human activity recognition framework for wearable technology, addressing the crucial balance between minimizing power consumption and maintaining accuracy due to the stringent power and memory constraints of wearable devices. We introduced *AdaSense*, a co-optimized framework for sensing, feature extraction, and classification tailored to Human Activity Recognition. *AdaSense* achieved energy efficiency by dynamically adapting sensor configurations based on user activity patterns over time, ensuring the selection of options that best balance accuracy and energy consumption. Utilizing low-overhead methodologies for processing and classification, the approach resulted in a significant 69% reduction in sensor power consumption with a minimal impact on activity recognition accuracy of less than 1.5%.

In Chapter 6, we delved into the enhancement of model efficiency through the integration of spatial context awareness into the architectural design of models. The approach employed a hierarchical decision-making process that starts with a preliminary, efficient evaluation of the input's spatial characteristics to find the optimal specialized processing pathway. This utilization of spatial context aimed to dynamically improve the model's efficiency. To assess the effectiveness of this approach, we applied it within an object detection framework, widely used in surveillance and augmented reality applications, where the computational and energy demands often pose challenges for deployment on resource-constrained edge devices. Object detection models process images to identify and locate various object classes present within. In this chapter, we introduced *AdaCon*, a method that enhances the efficiency of object detection models by exploiting the likelihood of different object categories occurring together within the same spatial context. Specifically, *AdaCon* categorizes objects based on the probability of their spatial co-occurrence and designs an adaptive network around these clusters. A branch controller dynamically selects network segments to activate during runtime based on the spatial context of the incoming frame. Our evaluation on the COCO dataset demonstrated that this adaptive object detection approach significantly reduces energy consumption by up to 45% and latency by up to 27%, with a marginal decrease in average precision (AP) for object detection.

In Chapter 7, we delved into the development of a complexity-aware machine learning (ML) model, specifically designed to autonomously learn and adapt its inference process based on complexity of the input, eliminating the need for manually defining the adaptivity criteria. Our exploration was applied to a vision transformer model capable of executing multiple tasks concurrently, which is particularly beneficial for applications such as augmented reality that require extensive information extraction from input. We highlighted the challenges associated with multi-task learning (MTL) models, notably the requirement for a shared encoder with substantial representational capacity to generalize across tasks and inputs, which adversely impacts inference latency. To address these challenges, we proposed *AdaMTL*, an adaptive framework designed to optimize task-specific inference within MTL models based on the input. *AdaMTL* incorporated a lightweight policy network, co-trained with the MTL model, to identify and eliminate unnecessary computations for each task based on the current input frame. This input-dependent, task-aware policy enabled selective activation of model components, significantly enhancing computational efficiency. Our experimental results on the PASCAL dataset showed that *AdaMTL* not only reduced computational complexity by 43% but also increased accuracy by 1.32% over single-task models. Furthermore, when integrated with state-of-the-art MTL strategies, *AdaMTL* achieved a 7.8% accuracy improvement and a 3.1$\times$ efficiency gain. Deployment on Vuzix M4000 smart glasses demonstrated up to a 21.8% reduction in inference latency and a 37.5% decrease in energy consumption, showcasing *AdaMTL*'s potential to significantly improve both the performance and efficiency of MTL models in practical applications.

8.2 Possible Research Extensions

Expanding on the foundations laid by this dissertation, numerous promising avenues for future research emerge. These include enhancements at both the architectural level of machine learning (ML) models and the level of ML accelerators. Such research directions aim to fully leverage the benefits of these optimizations at the architectural level of model design.

8.2.1 Heterogeneous Quantization for Multi-task Models

Future research focusing on heterogeneous quantization for multi-task learning (MTL) models presents an intriguing avenue for optimizing performance across diverse tasks while maintaining computational efficiency. Heterogeneous quantization, by applying variable quantization levels to different parts of the MTL model based on the task-specific requirements and sensitivity, could significantly enhance the balance between accuracy and computational resource usage. This approach acknowledges that not all tasks within an MTL framework demand the same level of precision, allowing for strategic allocation of computational resources where they are most needed. Investigating the application of heterogeneous quantization in MTL models involves not only developing adaptive quantization techniques but also devising intelligent mechanisms for determining the optimal quantization strategy for each task dynamically. This research could lead to more efficient MTL models capable of performing a wide array of tasks simultaneously on resource-constrained devices, pushing the boundaries of what's possible on edge devices.

8.2.2 Generalizing Power-of-two Quantization for LLMs

Future research into logarithmic quantization for large language models (LLMs) is poised to significantly address the challenges of computational efficiency and model scalability. A key advantage of logarithmic quantization lies in its ability to replace multiplication operations—typically resource-intensive in hardware—with simpler shift operations, which are considerably cheaper in terms of computational cost. This substitution is especially critical in the context of LLMs, where the sheer volume of multiplications during model training and inference can be overwhelming. By adopting a power-of-two or logarithmic scale for quantization, the process aligns more naturally with the binary nature of hardware computation, facilitating a more efficient execution path. This shift not only promises reductions in the energy consumption and hardware requirements for running LLMs but also enhances the feasibility of deploying advanced AI models on a wider range of devices, including those at the edge. Investigating and optimizing logarithmic quantization methods could lead to a new paradigm in the design and operation of LLMs, where efficiency and performance are balanced more effectively, opening the door to more sustainable and accessible AI technologies.

8.2.3 Low-Level Support for Dynamic Sparsification

Investigating low-level support for dynamic sparsification at both the hardware accelerator and compilation levels constitutes a promising avenue for future research. Dynamic sparsification, which intelligently zeroes out certain data elements or computations that have minimal impact on the overall outcome, can dramatically increase computational efficiency and reduce energy consumption in machine learning models. Future work could focus on developing hardware accelerators specifically designed to recognize and leverage sparsity in data and computations in real-time. Additionally, advancements in compiler technologies that can dynamically identify opportunities for sparsification and optimize code execution paths accordingly would further amplify these benefits. This dual approach, enhancing both hardware and software capabilities for dynamic sparsification, has the potential to significantly advance the field of efficient computing, making it possible to deploy more sophisticated machine learning algorithms on power and resource-constrained devices, opening new horizons for edge computing and beyond.

8.2.4 Accelerators Support for Heterogeneous Quantization

Exploring the development of machine learning accelerators that offer native support for heterogeneous quantization presents a compelling direction for future work. Heterogeneous quantization, which entails applying varied quantization strategies and bit-widths to different segments of a neural network, optimizes the balance between computational efficiency and model accuracy tailored to the unique requirements of each part of the network. Future advancements in accelerators capable of supporting this nuanced approach could significantly enhance the deployment of complex machine learning models in resource-constrained settings, such as edge computing devices. The ability to dynamically adjust computational precision, thereby optimizing processing efficiency and reducing power consumption, without markedly affecting performance, would mark a significant leap forward. Such research endeavors could pave the way for the widespread adoption of advanced AI applications, optimizing their performance within the stringent energy and computational limits of next-generation technology environments.

Appendix A

Detailed ACEv2 Derivations

A.1 Elementwise Multiplications

For simplicity, we assume both operands have the same number of bits (i.e., $i = j$) in this derivation. Elementwise multiplications requires a multiplier as well as an adder to account for the dot pattern at the completion of the multiplication [24]. We base our derivation on the established implementation of Dadda multiplier [24] and Ripple-Carry Adder (RCA) [6] to estimate the cost of elementwise multiplications.

To multiply two i-bit numbers, the Dadda multiplier requires i^2-3i+2 adders [24], while the RCA adds another $2i-2$ adders. This leads to a total number of adders equal to i^2-i. To generalize to operands with different precisions, the cost for elementwise multiplications between an i-bit number and a j-bit number can be derived as $i \cdot j - \max(i, j)$, with $i \cdot j$ reflecting the cost of the multiplier and $\max(i, j)$ representing the final addition. Independently, we performed an empirical verification for $1 \leq i, j \leq 64$ which confirmed the correctness of this formula, showing zero error in predicted adder counts. This refinement in ACE_{v2} cost calculation enhances our understanding of multiplier complexity.

A.2 Floating Point Elementwise Additions

We improve ACE by extending it to include the cost of floating-point elementwise addition. We derive the cost of adding an i-bit and a j-bit floating point numbers, using the formula

$$ACE_{fp-add} = c_a \cdot \max(i, j) \tag{A.1}$$

For simplicity, we assume both operands have the same number of bits (i.e., $i = j$) in this derivation. c_a reflects the added complexity of floating point operations compared to fixed-point addition. To derive c_a, we look into the components of floating-point adders [134] and analyze the ACE_{v2} cost for each component. Assuming e bits for the exponent and m bits for the mantissa, the main components of the floating-point adder and their corresponding ACE_{v2} costs are as follows:

1. *Exponent Subtraction*: Involves subtracting the exponent bits resulting in an ACE_{v2} cost of $\mathbf{e}$.
2. *Operand Swapping*: Requires a single multiplexer with negligible ACE_{v2} cost.
3. *Limitation of Alignment Shift Amount*: Involves adding the mantissa bits resulting in an ACE_{v2} cost of $\mathbf{m}$.
4. *Alignment Shift*: Involves shifting by the mantissa bits adding an ACE_{v2} cost of $\mathbf{m \cdot \log_2(m)/5}$[1].
5. *Significand Negation*: Involves one bit subtraction resulting in an ACE_{v2} cost of $\mathbf{1}$.
6. *Significand Addition*: Requires mantissa bits addition resulting in an ACE_{v2} cost of $\mathbf{m}$.
7. *Significand Conversion*: Requires two additions adding an ACE_{v2} cost of $\mathbf{2m}$.
8. *Normalization*: Requires shifting e bits resulting in an ACE_{v2} cost of $\mathbf{e \cdot \log_2(e)/5}$.

[1] ACE_{v2} cost for shift operation is derived as $i \cdot \log_2(j)/5$ in Subsection 3.2

9. *Rounding and Post-normalization*: Requires adding m bits with an ACE_{v2} cost of $\mathbf{m}$.

Summing the costs for all the components, we get a total cost of $m(5+\log_2(m)/5)+e+e \cdot \log_2(e)/5+1$. Considering the dominant role of mantissa operations, we approximate the total cost to $i(5+log_2(i)/5)$ where i is the number of bits of the added floating point number. The upper bound for $\log_2(i)/5$ is 1 when m is 32. Therefore, we can derive the cost as $6\dot{i}$ resulting in $c_a = 6$ in Equation A.1. This approximation streamlines ACE_{v2} calculation for floating-point additions.

Bibliography

[1] Martın Abadi et al. “Tensorflow: A system for large-scale machine learning”. In: *12th {USENIX} symposium on operating systems design and implementation ({OSDI} 16)*. 2016, pp. 265–283.

[2] AmirAli Abdolrashidi et al. “Pareto-optimal quantized resnet is mostly 4-bit”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2021, pp. 3091–3099.

[3] Abien Fred Agarap. “Deep Learning using Rectified Linear Units (ReLU)”. In: (Mar. 2018).

[4] S. Ananthanarayan et al. “Pt Viz: towards a wearable device for visualizing knee rehabilitation exercises”. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM. 2013, pp. 1247–1250.

[5] Sajid Anwar and Wonyong Sung. “Compact deep convolutional neural networks with coarse pruning”. In: *arXiv preprint arXiv:1610.09639* (2016).

[6] S. Archana and G. Durga. “Design of low power and high speed ripple carry adder”. In: *2014 International Conference on Communication and Signal Processing*. 2014, pp. 939–943. DOI: `10.1109/ICCSP.2014.6949982`.

[7] ARM. *Cortex M4 Technical Reference Manual*. 2010. URL: `https://developer.arm.com/documentation/ddi0439/b/Floating-Point-Unit/FPU-Functional-Description/FPU-instruction-set` (visited on 05/17/2021).

[8] K. Bark et al. “A wearable skin stretch device for haptic feedback”. In: *World Haptics 2009 - Third Joint EuroHaptics conference and Symposium on Haptic*

Interfaces for Virtual Environment and Teleoperator Systems. Mar. 2009, pp. 464–469.

[9] R Bellman. “Dynamic programming princeton university press princeton”. In: *New Jersey Google Scholar* (1957).

[10] Joseph Bethge et al. “Meliusnet: Can binary neural networks achieve mobilenet-level accuracy?” In: *arXiv preprint arXiv:2001.05936* (2020).

[11] G. Bhat et al. “Online human activity recognition using low-power wearable devices”. In: *Proceedings of the International Conference on Computer-Aided Design*. ACM. 2018, p. 72.

[12] G. Bhat et al. “Reap: Runtime energy-accuracy optimization for energy harvesting iot devices”. In: *arXiv preprint arXiv:1902.02639* (2019).

[13] Deblina Bhattacharjee et al. “Mult: an end-to-end multitask learning transformer”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2022, pp. 12031–12041.

[14] Davis Blalock et al. “What is the state of neural network pruning?” In: *arXiv preprint arXiv:2003.03033* (2020).

[15] Tolga Bolukbasi et al. “Adaptive neural networks for efficient inference”. In: *arXiv preprint arXiv:1702.07811* (2017).

[16] *Bosch Sensortec BMI160*. 2019. URL: `https://www.bosch-sensortec.com/bst/products/all_products/bmi160/`.

[17] Han Cai et al. *TinyTL: Reduce Memory, Not Parameters for Efficient On-Device Learning*. 2021. arXiv: `2007.11622 [cs.CV]`.

[18] Nicolas Carion et al. “End-to-end object detection with transformers”. In: *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part I 16*. Springer. 2020, pp. 213–229.

[19] Arnav Chavan et al. “One-for-All: Generalized LoRA for Parameter-Efficient Fine-tuning”. In: *arXiv preprint arXiv:2306.07967* (2023).

[20] Jiasi Chen and Xukan Ran. “Deep Learning With Edge Computing: A Review.” In: *Proceedings of the IEEE* 107.8 (2019), pp. 1655–1674.

[21] Tianqi Chen et al. *Training Deep Nets with Sublinear Memory Cost*. 2016. arXiv: `1604.06174 [cs.LG]`.

[22] Yukang Chen et al. “Longlora: Efficient fine-tuning of long-context large language models”. In: *arXiv preprint arXiv:2309.12307* (2023).

[23] Yoni Choukroun et al. “Low-bit quantization of neural networks for efficient inference”. In: *2019 IEEE/CVF International Conference on Computer Vision Workshop (ICCVW)*. IEEE. 2019, pp. 3009–3018.

[24] Wesley Donald Chu. “Wallace and Dadda Multipliers Implemented Using Carry Lookahead Adders”. In: 2013. URL: `https://repositories.lib.utexas.edu/server/api/core/bitstreams/db07a4ef-e75d-4b69-8645-0ea6f30ccd56/content`.

[25] Claudionor N Coelho Jr et al. “Automatic heterogeneous quantization of deep neural networks for low-latency inference on the edge for particle detectors”. In: *Nature Machine Intelligence* 3.8 (2021), pp. 675–686.

[26] Matthieu Courbariaux, Yoshua Bengio, and Jean-Pierre David. “Training deep neural networks with low precision multiplications”. In: *arXiv preprint arXiv:1412.7024* (2014).

[27] Luigi Dadda. “Some schemes for fast serial input multipliers”. In: *1983 IEEE 6th Symposium on Computer Arithmetic (ARITH)*. 1983, pp. 52–59. DOI: `10.1109/ARITH.1983.6158074`.

[28] Steve Dai et al. “Vs-quant: Per-vector scaled quantization for accurate low-precision neural network inference”. In: *Proceedings of Machine Learning and Systems* 3 (2021), pp. 873–884.

[29] Dipankar Das et al. “Mixed precision training of convolutional neural networks using integer operations”. In: *arXiv preprint arXiv:1802.00930* (2018).

[30] Jia Deng et al. "ImageNet: A large-scale hierarchical image database". In: *2009 IEEE Conference on Computer Vision and Pattern Recognition*. 2009, pp. 248–255. DOI: 10.1109/CVPR.2009.5206848.

[31] Jia Deng et al. "Imagenet: A large-scale hierarchical image database". In: *2009 IEEE conference on computer vision and pattern recognition*. Ieee. 2009, pp. 248–255.

[32] Tim Dettmers et al. "Qlora: Efficient finetuning of quantized llms". In: *arXiv preprint arXiv:2305.14314* (2023).

[33] Jacob Devlin et al. "Bert: Pre-training of deep bidirectional transformers for language understanding". In: *arXiv preprint arXiv:1810.04805* (2018).

[34] Alexey Dosovitskiy et al. "An image is worth 16x16 words: Transformers for image recognition at scale". In: *arXiv preprint arXiv:2010.11929* (2020).

[35] Mark Everingham et al. "The Pascal Visual Object Classes (VOC) challenge". In: *International Journal of Computer Vision* 88 (June 2010), pp. 303–338. DOI: 10.1007/s11263-009-0275-4.

[36] Yuan Fang et al. "Object detection meets knowledge graphs". In: (2017).

[37] Igor Fedorov et al. "Sparse: Sparse architecture search for cnns on resource-constrained microcontrollers". In: *Advances in Neural Information Processing Systems*. 2019.

[38] Thomas MJ Fruchterman and Edward M Reingold. "Graph drawing by force-directed placement". In: *Software: Practice and experience* 21.11 (1991), pp. 1129–1164.

[39] I. Galiana et al. "Wearable soft robotic device for post-stroke shoulder rehabilitation: Identifying misalignments". In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*. Oct. 2012, pp. 317–322.

[40] Peng Gao et al. "Llama-adapter v2: Parameter-efficient visual instruction model". In: *arXiv preprint arXiv:2304.15010* (2023).

[41] Amir Gholami et al. “A survey of quantization methods for efficient neural network inference”. In: *Low-Power Computer Vision*. Chapman and Hall/CRC, 2022, pp. 291–326.

[42] S. Ghosh et al. “Deep Network Pruning for Object Detection”. In: *2019 IEEE International Conference on Image Processing (ICIP)*. 2019, pp. 3915–3919. DOI: `10.1109/ICIP.2019.8803505`.

[43] A. Godfrey *et al.* “Inertial wearables as pragmatic tools in dementia”. In: *Maturitas* 127 (2019), pp. 12–17.

[44] Abhinav Goel *et al.* “Low-power object counting with hierarchical neural networks”. In: *Proceedings of the ACM/IEEE International Symposium on Low Power Electronics and Design*. 2020, pp. 163–168.

[45] Benjamin Graham et al. “Levit: a vision transformer in convnet’s clothing for faster inference”. In: *Proceedings of the IEEE/CVF international conference on computer vision*. 2021, pp. 12259–12269.

[46] Audrunas Gruslys et al. “Memory-Efficient Backpropagation Through Time”. In: *CoRR* abs/1606.03401 (2016). arXiv: `1606.03401`. URL: `http://arxiv.org/abs/1606.03401`.

[47] Anmol Gulati et al. “Conformer: Convolution-augmented transformer for speech recognition”. In: *arXiv preprint arXiv:2005.08100* (2020).

[48] Yiwen Guo, Anbang Yao, and Yurong Chen. “Dynamic network surgery for efficient dnns”. In: *Advances in neural information processing systems* 29 (2016).

[49] C. Gupta *et al.* “ProtoNN: compressed and accurate kNN for resource-scarce devices”. In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org. 2017, pp. 1331–1340.

[50] Song Han, Huizi Mao, and William J Dally. “Deep compression: Compressing deep neural networks with pruning, trained quantization and huffman coding”. In: *arXiv preprint arXiv:1510.00149* (2015).

[51] Song Han, Huizi Mao, and William J. Dally. *Deep Compression: Compressing Deep Neural Networks with Pruning, Trained Quantization and Huffman Coding*. 2016. arXiv: `1510.00149 [cs.CV]`.

[52] S. Hashemi, R. Bahar, and S Reda. "DRUM: A Dynamic Range Unbiased Multiplier for Approximate Applications". In: *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design*. ICCAD '15. Austin, TX, USA: IEEE Press, pp. 418–425. ISBN: 9781467383899.

[53] Soheil Hashemi, R. Iris Bahar, and Sherief Reda. "A Low-Power Dynamic Divider for Approximate Applications". In: *Proceedings of the 53rd Annual Design Automation Conference*. DAC '16. Austin, Texas: Association for Computing Machinery. ISBN: 9781450342360.

[54] Soheil Hashemi et al. "Understanding the impact of precision quantization on the accuracy and energy of neural networks". In: *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE. 2017, pp. 1474–1479.

[55] Irina Hashmi and Hafiz Md. Hasan Babu. "An Efficient Design of a Reversible Barrel Shifter". In: *2010 23rd International Conference on VLSI Design*. 2010, pp. 93–98. DOI: `10.1109/VLSI.Design.2010.35`.

[56] Hussein Hazimeh et al. "Dselect-k: Differentiable selection in the mixture of experts with applications to multi-task learning". In: *Advances in Neural Information Processing Systems* 34 (2021), pp. 29335–29347.

[57] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[58] Kaiming He et al. "Deep residual learning for image recognition". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 770–778.

[59] Kaiming He et al. "Delving deep into rectifiers: Surpassing human-level performance on imagenet classification". In: *Proceedings of the IEEE international conference on computer vision*. 2015, pp. 1026–1034.

[60] Kaiming He et al. “Mask r-cnn”. In: *Proceedings of the IEEE international conference on computer vision*. 2017, pp. 2961–2969.

[61] Xuehai He et al. “Parameter-efficient model adaptation for vision transformers”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 37. 1. 2023, pp. 817–825.

[62] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. “Distilling the knowledge in a neural network”. In: *arXiv preprint arXiv:1503.02531* (2015).

[63] *Battery Historian*. `https://github.com/google/battery-historian`. Aug. 2016. URL: `https://github.com/google/battery-historian`.

[64] Mark Horowitz. “1.1 Computing’s energy problem (and what we can do about it)”. In: *2014 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*. 2014, pp. 10–14. DOI: `10.1109/ISSCC.2014.6757323`.

[65] Andrew G Howard et al. “Mobilenets: Efficient convolutional neural networks for mobile vision applications”. In: *arXiv preprint arXiv:1704.04861* (2017).

[66] Edward J Hu et al. “Lora: Low-rank adaptation of large language models”. In: *arXiv preprint arXiv:2106.09685* (2021).

[67] Zhiqiang Hu et al. “LLM-Adapters: An Adapter Family for Parameter-Efficient Fine-Tuning of Large Language Models”. In: *arXiv preprint arXiv:2304.01933* (2023).

[68] Gao Huang et al. “Densely connected convolutional networks”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 4700–4708.

[69] Gao Huang et al. “Multi-scale dense networks for resource efficient image classification”. In: *arXiv preprint arXiv:1703.09844* (2017).

[70] Yanping Huang et al. *GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism*. 2019. arXiv: `1811.06965 [cs.CV]`.

[71] Forrest N. Iandola et al. *SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and ¡0.5MB model size*. 2016. arXiv: `1602.07360 [cs.CV]`.

[72] Sergey Ioffe and Christian Szegedy. “Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift”. In: 2015, pp. 448–456. URL: http://jmlr.org/proceedings/papers/v37/ioffe15.pdf.

[73] Keishi Ishihara et al. “Multi-task learning with attention for end-to-end autonomous driving”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2021, pp. 2902–2911.

[74] Benoit Jacob et al. “Quantization and training of neural networks for efficient integer-arithmetic-only inference”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 2704–2713.

[75] Eric Jang, Shixiang Gu, and Ben Poole. “Categorical reparameterization with gumbel-softmax”. In: *arXiv preprint arXiv:1611.01144* (2016).

[76] Steven A Janowsky. “Pruning versus clipping in neural networks”. In: *Physical Review A* 39.12 (1989), p. 6600.

[77] *Jetson Nano Developer Kit*. Apr. 2021. URL: https://developer.nvidia.com/embedded/jetson-nano-developer-kit.

[78] Nandan Kumar Jha and Sparsh Mittal. “Modeling data reuse in deep neural networks by taking data-types into cognizance”. In: *IEEE Transactions on Computers* 70.9 (2020), pp. 1526–1538.

[79] Xianyan Jia et al. “Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes”. In: *arXiv preprint arXiv:1807.11205* (2018).

[80] Shenwang Jiang et al. “Tree-CNN: from generalization to specialization”. In: *EURASIP Journal on Wireless Communications and Networking* 2018 (Dec. 2018). DOI: 10.1186/s13638-018-1197-z.

[81] Jing Jin, Shaolong Shu, and Feng Lin. “Personalized Control of Indoor Air Temperature Based on Deep Learning”. In: *2019 Chinese Control And Decision Conference (CCDC)*. IEEE. 2019, pp. 1354–1359.

[82] Iosr Journals et al. "Power Optimized Multiplexer Based 1 Bit Full Adder Cell Using .18 µm CMOS Technology". In: 2015. URL: https://api.semanticscholar.org/CorpusID:149453517.

[83] Sangil Jung et al. "Learning to quantize deep networks by optimizing quantization intervals with task loss". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition.* 2019, pp. 4350–4359.

[84] I. Koryakovskiy et al. "One-Shot Model for Mixed-Precision Quantization". In: *2023 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR).* Los Alamitos, CA, USA: IEEE Computer Society, June 2023, pp. 7939–7949. DOI: 10.1109/CVPR52729.2023.00767. URL: https://doi.ieeecomputersociety.org/10.1109/CVPR52729.2023.00767.

[85] Raghuraman Krishnamoorthi. "Quantizing deep convolutional networks for efficient inference: A whitepaper". In: *arXiv preprint arXiv:1806.08342* (2018).

[86] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. "The cifar-10 dataset". In: *online: http://www. cs. toronto. edu/kriz/cifar. html* 55 (2014), p. 5.

[87] A. Kumar, S. Goyal, and M. Varma. "Resource-efficient machine learning in 2 KB RAM for the internet of things". In: *Proceedings of the 34th International Conference on Machine Learning-Volume 70.* JMLR. org. 2017, pp. 1935–1944.

[88] A. Kusupati *et al.* "Fastgrnn: A fast, accurate, stable and tiny kilobyte sized gated recurrent neural network". In: *Advances in Neural Information Processing Systems.* 2018, pp. 9017–9028.

[89] Seulki Lee and Shahriar Nirjon. "Learning in the Wild: When, How, and What to Learn for On-Device Dataset Adaptation". In: *Proceedings of the 2nd International Workshop on Challenges in Artificial Intelligence and Machine Learning for Internet of Things.* AIChallengeIoT '20. Virtual Event, Japan: Association for Computing Machinery, 2020, pp. 34–40. ISBN: 9781450381345. DOI: 10.1145/3417313.3429382. URL: https://doi.org/10.1145/3417313.3429382.

[90] He Li, Kaoru Ota, and Mianxiong Dong. "Learning IoT in Edge: Deep Learning for the Internet of Things with Edge Computing". In: *IEEE Network* 32.1 (2018), pp. 96–101. DOI: 10.1109/MNET.2018.1700202.

[91] Yuhang Li, Xin Dong, and Wei Wang. "Additive powers-of-two quantization: An efficient non-uniform discretization for neural networks". In: *arXiv preprint arXiv:1909.13144* (2019).

[92] Hanxue Liang et al. "M^{3}ViT: Mixture-of-Experts Vision Transformer for Efficient Multi-task Learning with Model-Accelerator Co-design". In: *arXiv preprint arXiv:2210.14793* (2022).

[93] Tailin Liang et al. "Pruning and quantization for deep neural network acceleration: A survey". In: *Neurocomputing* 461 (2021), pp. 370–403.

[94] Ching-Yi Lin and Radu Marculescu. "Model personalization for human activity recognition". In: *2020 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*. IEEE. 2020, pp. 1–7.

[95] Tsung-Yi Lin et al. *Focal Loss for Dense Object Detection*. 2018. arXiv: 1708.02002 [cs.CV].

[96] Tsung-Yi Lin *et al.* "Microsoft coco: Common objects in context". In: *European conference on computer vision*. Springer. 2014, pp. 740–755.

[97] Liu Liu et al. *Dynamic Sparse Graph for Efficient Deep Learning*. 2019. arXiv: 1810.00859 [cs.LG].

[98] Shikun Liu, Edward Johns, and Andrew J Davison. "End-to-end multi-task learning with attention". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2019, pp. 1871–1880.

[99] Shikun Liu, Edward Johns, and Andrew J Davison. "End-to-end multi-task learning with attention". In: *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 2019, pp. 1871–1880.

[100] Yen-Cheng Liu et al. “Polyhistor: Parameter-efficient multi-task adaptation for dense vision tasks”. In: *Advances in Neural Information Processing Systems* 35 (2022), pp. 36889–36901.

[101] Ze Liu et al. “Swin transformer: Hierarchical vision transformer using shifted windows”. In: *Proceedings of the IEEE/CVF international conference on computer vision*. 2021, pp. 10012–10022.

[102] Zechun Liu et al. “Bi-real net: Enhancing the performance of 1-bit cnns with improved representational capability and advanced training algorithm”. In: *Proceedings of the European conference on computer vision (ECCV)*. 2018, pp. 722–737.

[103] Zechun Liu et al. “How do adam and training strategies help bnns optimization”. In: *International conference on machine learning*. PMLR. 2021, pp. 6936–6946.

[104] Zechun Liu et al. “Reactnet: Towards precise binary neural network with generalized activation functions”. In: *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XIV 16*. Springer. 2020, pp. 143–159.

[105] Q. Liu *et al.* “Gazelle: Energy-Efficient Wearable Analysis for Running”. In: *IEEE Transactions on Mobile Computing* 16.9 (Sept. 2017), pp. 2531–2544. ISSN: 2161-9875.

[106] Mark Harris Luke Durant Olivier Giroux and Nick Stam. *NVIDIA Tesla V100 GPU architecture*. 2017. URL: `https://developer.nvidia.com/blog/inside-volta/`.

[107] Ningning Ma et al. “Shufflenet v2: Practical guidelines for efficient cnn architecture design”. In: *Proceedings of the European conference on computer vision (ECCV)*. 2018, pp. 116–131.

[108] Rabeeh Karimi Mahabadi et al. “Parameter-efficient multi-task fine-tuning for transformers via shared hypernetworks”. In: *arXiv preprint arXiv:2106.04489* (2021).

[109] Kevis-Kokitsi Maninis, Ilija Radosavovic, and Iasonas Kokkinos. "Attentive single-tasking of multiple tasks". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2019, pp. 1851–1860.

[110] Brais Martinez et al. "Training binary neural networks with real-to-binary convolutions". In: *arXiv preprint arXiv:2003.11535* (2020).

[111] R. McGinnis *et al.* "Rapid detection of internalizing diagnosis in young children enabled by wearable sensors and machine learning". In: *PloS one* 14.1 (2019), e0210267.

[112] Lingchen Meng et al. "Adavit: Adaptive vision transformers for efficient image recognition". In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2022, pp. 12309–12318.

[113] Paulius Micikevicius et al. *Mixed Precision Training*. 2018. arXiv: `1710.03740 [cs.AI]`.

[114] Ishan Misra et al. "Cross-stitch networks for multi-task learning". In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, pp. 3994–4003.

[115] Daisuke Miyashita, Edward H Lee, and Boris Murmann. "Convolutional neural networks using logarithmic data representation". In: *arXiv preprint arXiv:1603.01025* (2016).

[116] Eslam Mohamed and Ahmad El Sallab. "Spatio-temporal multi-task learning transformer for joint moving object detection and segmentation". In: *2021 IEEE International Intelligent Transportation Systems Conference (ITSC)*. IEEE. 2021, pp. 1470–1475.

[117] Pavlo Molchanov et al. "Pruning convolutional neural networks for resource efficient inference". In: *arXiv preprint arXiv:1611.06440* (2016).

[118] Hesham Mostafa and Xin Wang. *Parameter Efficient Training of Deep Convolutional Neural Networks by Dynamic Sparse Reparameterization*. 2019. arXiv: `1902.05967 [cs.LG]`.

[119] A. N.K. et al. “Sensor-Classifier Co-Optimization for Wearable Human Activity Recognition Applications”. In: *IEEE International Conference on Embedded Software and Systems (ICESS)*. June 2019, pp. 1–4.

[120] Sharan Narang et al. “Exploring sparsity in recurrent neural networks”. In: *arXiv preprint arXiv:1704.05119* (2017).

[121] Marina Neseem and Sherief Reda. “AdaCon: Adaptive Context-Aware Object Detection for Resource-Constrained Embedded Devices”. In: *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*. IEEE. 2021, pp. 1–9.

[122] Yu-Ting Pai and Yu-Kumg Chen. “The fastest carry lookahead adder”. In: *Proceedings. DELTA 2004. Second IEEE International Workshop on Electronic Design, Test and Applications*. 2004, pp. 434–436. DOI: 10.1109/DELTA.2004.10071.

[123] Sinno Jialin Pan and Qiang Yang. “A survey on transfer learning”. In: *IEEE Transactions on knowledge and data engineering* 22.10 (2009), pp. 1345–1359.

[124] Eunhyeok Park and Sungjoo Yoo. “Profit: A novel training method for sub-4-bit mobilenet models”. In: *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part VI 16*. Springer. 2020, pp. 430–446.

[125] Adam Paszke et al. “Automatic differentiation in pytorch”. In: (2017).

[126] Adam Paszke et al. “Pytorch: An imperative style, high-performance deep learning library”. In: *arXiv preprint arXiv:1912.01703* (2019).

[127] Hai Phan et al. “Binarizing MobileNet via Evolution-Based Searching”. In: *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. 2020, pp. 13417–13426. DOI: 10.1109/CVPR42600.2020.01343.

[128] Hai Phan et al. “Mobinet: A mobile binary network for image classification”. In: *Proceedings of the IEEE/CVF winter conference on applications of computer vision*. 2020, pp. 3453–3462.

[129] *Pytorch Mobile: End-to-end workflow from Training to Deployment for iOS and Android mobile devices.* `https://pytorch.org/mobile/android`. Mar. 2023. URL: `https://pytorch.org/mobile/android`.

[130] Mohammad Rastegari et al. “Xnor-net: Imagenet classification using binary convolutional neural networks”. In: *European conference on computer vision.* Springer. 2016, pp. 525–542.

[131] Joseph Redmon and Ali Farhadi. “Yolov3: An incremental improvement”. In: *arXiv preprint arXiv:1804.02767* (2018).

[132] Shaoqing Ren et al. “Faster r-cnn: Towards real-time object detection with region proposal networks”. In: *Advances in neural information processing systems.* 2015, pp. 91–99.

[133] Charbel Sakr, Yongjune Kim, and Naresh Shanbhag. “Analytical guarantees on numerical precision of deep neural networks”. In: *International Conference on Machine Learning.* PMLR. 2017, pp. 3007–3016.

[134] P.-M. Seidel and G. Even. “On the design of fast IEEE floating-point adders”. In: *Proceedings 15th IEEE Symposium on Computer Arithmetic. ARITH-15 2001.* 2001, pp. 184–194. DOI: `10.1109/ARITH.2001.930118`.

[135] Samuel L Smith et al. “Don’t decay the learning rate, increase the batch size”. In: *arXiv preprint arXiv:1711.00489* (2017).

[136] Nimit Sharad Sohoni et al. *Low-Memory Neural Network Training: A Technical Report.* 2019. arXiv: `1904.10631 [cs.LG]`.

[137] Steven W Squyres et al. “Athena Mars rover science investigation”. In: *Journal of Geophysical Research: Planets* 108.E12 (2003).

[138] Trevor Standley et al. “Which tasks should be learned together in multi-task learning?” In: *International Conference on Machine Learning.* PMLR. 2020, pp. 9120–9132.

[139] Robin Strudel et al. "Segmenter: Transformer for semantic segmentation". In: *Proceedings of the IEEE/CVF international conference on computer vision.* 2021, pp. 7262–7272.

[140] Ximeng Sun et al. "Adashare: Learning what to share for efficient deep multi-task learning". In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 8728–8740.

[141] Qiuling Suo et al. "Deep patient similarity learning for personalized healthcare". In: *IEEE transactions on nanobioscience* 17.3 (2018), pp. 219–227.

[142] Ahmet Ali Süzen, Burhan Duman, and Betül Şen. "Benchmark Analysis of Jetson TX2, Jetson Nano and Raspberry PI using Deep-CNN". In: *2020 International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA).* IEEE. 2020, pp. 1–5.

[143] Taiji Suzuki et al. "Spectral pruning: Compressing deep neural networks via spectral analysis and its generalization error". In: *arXiv preprint arXiv:1808.08558* (2018).

[144] Mingxing Tan and Quoc Le. "Efficientnet: Rethinking model scaling for convolutional neural networks". In: *International conference on machine learning.* PMLR. 2019, pp. 6105–6114.

[145] Mingxing Tan, Ruoming Pang, and Quoc V. Le. *EfficientDet: Scalable and Efficient Object Detection.* 2020. arXiv: 1911.09070 [cs.CV].

[146] H. Tann, S. Hashemi, and S. Reda. "Flexible Deep Neural Network Processing". In: *ArXiv* abs/1801.07353 (2018).

[147] Hokchhay Tann et al. "Hardware-software codesign of accurate, multiplier-free Deep Neural Networks". In: *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC).* 2017, pp. 1–6. DOI: 10.1145/3061639.3062259.

[148] Hokchhay Tann et al. "Runtime Configurable Deep Neural Networks for Energy-Accuracy Trade-Off". In: New York, NY, USA: Association for Computing Machinery, 2016. ISBN: 9781450344838. DOI: 10.1145/2968456.2968458.

[149] *TI CC2640R2F SimpleLink MCU*. 2019. URL: `https://www.ti.com/product/CC2640R2F`.

[150] Hugo Touvron et al. “Training data-efficient image transformers & distillation through attention”. In: *International conference on machine learning*. PMLR. 2021, pp. 10347–10357.

[151] Guido Van Rossum and Fred L Drake Jr. *Python reference manual*. Centrum voor Wiskunde en Informatica Amsterdam, 1995.

[152] Simon Vandenhende, Stamatios Georgoulis, and Luc Van Gool. “Mti-net: Multi-scale task interaction networks for multi-task learning”. In: *Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part IV 16*. Springer. 2020, pp. 527–543.

[153] Simon Vandenhende et al. “Multi-Task Learning for Dense Prediction Tasks: A Survey”. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* 44.7 (2022), pp. 3614–3633. DOI: `10.1109/TPAMI.2021.3054719`.

[154] Pavan Kumar Anasosalu Vasu et al. “MobileOne: An Improved One Millisecond Mobile Backbone”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 2023, pp. 7907–7917.

[155] Ashish Vaswani et al. “Attention is all you need”. In: *Advances in neural information processing systems* 30 (2017).

[156] Praneeth Vepakomma et al. “Split learning for health: Distributed deep learning without sharing raw patient data”. In: *arXiv preprint arXiv:1812.00564* (2018).

[157] Richard Volpe et al. “Rocky 7: A next generation mars rover prototype”. In: *Advanced Robotics* 11.4 (1996), pp. 341–358.

[158] *VUZIX M4000 SMART GLASSES*. `https://www.vuzix.com/products/m4000-smart-glasses`. Mar. 2023. URL: `https://www.vuzix.com/products/m4000-smart-glasses`.

[159] C. S. Wallace. “A Suggestion for a Fast Multiplier”. In: *IEEE Transactions on Electronic Computers* EC-13.1 (1964), pp. 14–17. DOI: 10.1109/PGEC.1964.263830.

[160] Xin Wang et al. “Skipnet: Learning dynamic routing in convolutional networks”. In: *Proceedings of the European Conference on Computer Vision (ECCV)*. 2018, pp. 409–424.

[161] J. Williamson *et al.* “Data sensing and analysis: Challenges for wearables”. In: *The 20th Asia and South Pacific Design Automation Conference*. Jan. 2015, pp. 136–141.

[162] Bichen Wu et al. “Squeezedet: Unified, small, low power fully convolutional neural networks for real-time object detection for autonomous driving”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*. 2017, pp. 129–137.

[163] Dan Xu et al. “Pad-net: Multi-tasks guided prediction-and-distillation network for simultaneous depth estimation and scene parsing”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 675–684.

[164] Hang Xu et al. “Spatial-aware graph relation network for large-scale object detection”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2019, pp. 9298–9307.

[165] Tien-Ju Yang, Yu-Hsin Chen, and Vivienne Sze. “Designing energy-efficient convolutional neural networks using energy-aware pruning”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2017, pp. 5687–5695.

[166] Hanrong Ye and Dan Xu. “Inverted pyramid multi-task transformer for dense scene understanding”. In: *Computer Vision–ECCV 2022: 17th European Conference, Tel Aviv, Israel, October 23–27, 2022, Proceedings, Part XXVII*. Springer. 2022, pp. 514–530.

[167] Joseph Yiu. *Beginner guide on interrupt latency and Arm Cortex-M processors*. 2016. URL: https://community.arm.com/developer/ip-products/processors/b/

`processors-ip-blog/posts/beginner-guide-on-interrupt-latency-and-interrupt-latency-of-the-arm-cortex-m-processors`.

[168] Haoran You, Huihong Shi, Yipin Guo, et al. “ShiftAddViT: Mixture of Multiplication Primitives Towards Efficient Vision Transformer”. In: *arXiv preprint arXiv:2306.06446* (2023).

[169] Haoran You et al. “Shiftaddnet: A hardware-inspired deep network”. In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 2771–2783.

[170] Jiahui Yu et al. “Slimmable neural networks”. In: *arXiv preprint arXiv:1812.08928* (2018).

[171] Ruichi Yu et al. “Nisp: Pruning networks using neuron importance score propagation”. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*. 2018, pp. 9194–9203.

[172] Zhongzhi Yu et al. “Mia-former: efficient and robust vision transformers via multi-grained input-adaptation”. In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 36. 2022, pp. 8962–8970.

[173] Amir R Zamir et al. “Taskonomy: Disentangling task transfer learning”. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2018, pp. 3712–3722.

[174] Q. Zhang et al. “Determination of Activities of Daily Living of independent living older people using environmentally placed sensors”. In: *Annual International Conference of the IEEE Engineering in Medicine and Biology Society Conference* (July 2013), pp. 7044–7047.

[175] Renrui Zhang et al. “Llama-adapter: Efficient fine-tuning of language models with zero-init attention”. In: *arXiv preprint arXiv:2303.16199* (2023).

[176] Tianyuan Zhang et al. “Domain-Aware Dynamic Networks”. In: *arXiv preprint arXiv:1911.13237* (2019).

[177] Yichi Zhang, Zhiru Zhang, and Lukasz Lew. “Pokebnn: A binary pursuit of lightweight accuracy”. In: *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition.* 2022, pp. 12475–12485.

[178] Yichi Zhang et al. “FracBNN: Accurate and FPGA-efficient binary neural networks with fractional activations”. In: *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays.* 2021, pp. 171–182.

[179] Wangchunshu Zhou et al. “Bert loses patience: Fast and robust inference with early exit”. In: *Advances in Neural Information Processing Systems* 33 (2020), pp. 18330–18341.

[180] J. Zhu, R. San-Segundo, and J.M. Pardo. “Feature extraction for robust physical activity recognition.” In: *Human-centric Computing and Information Sciences* 16.7 (2017).

[181] Reto Zimmermann. “Computer arithmetic: Principles, architectures, and VLSI design”. In: *Personal publication (Available at http://www. iis. ee. ethz. ch/ zimmi/-publications/comp arith notes. ps. gz)* (1999).