UNIVERSITY OF SOUTHAMPTON

Faculty of Engineering and Physical Science
School of Electronics and Computer Science
Cyber Physical Systems Research Group

Doctor of Philosophy

Supervisor: Prof Geoff Merrett and Prof Jonathon Hare

Realising the Benefits of Dynamic DNNs on Reconfigurable Hardware

by **Anastasios Dimitriou**

April 29, 2025

Declaration of Authorship

I declare that this thesis and the work presented in it is my own and has been generated by me as the result of my own original research.

I confirm that:

1. This work was done wholly or mainly while in candidature for a research degree at this University;
2. Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated;
3. Where I have consulted the published work of others, this is always clearly attributed;
4. Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work;
5. I have acknowledged all main sources of help;
6. Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself;
7. Parts of this work have been published are listed under the Research Contribution.

Signed:.. Date:..................

Acknowledgements

I would like to express my deepest gratitude to all those who have contributed to the completion of this thesis. First and foremost, I would like to thank my supervisors, Prof. Geoff Merrett and Prof. Jonathon Hare, for their invaluable guidance, constant support, and encouragement throughout my PhD journey. Your expertise and insightful feedback have been invaluable.

I extend my gratitude to my sponsor, the Engineering and Physical Sciences Research Council (EPCRC), for the financial assistance which made this research possible.

To my numerous colleagues and friends, thank you for your friendship, motivation, and collaborative spirit. Your willingness to share knowledge and experiences has enhanced my learning and made my entire PhD experience enjoyable. I am particularly thankful to Joe Dib and Evangelos Panagiotou for their continuous support during challenging times.

Finally, I wish to express my heartfelt appreciation to my parents, Christo and Doxa, and my sister, Ioanna, for their love, understanding, and encouragement. Your enduring sacrifices and unshakable confidence in me have been my greatest motivations and the root of my success.

This journey would not have been possible without each and every one of you.

University of Southampton

Abstract

Faculty of Engineering and Physical Science
School of Electronics and Computer Science

Doctor of Philosophy

Realising the Benefits of Dynamic DNNs on Reconfigurable Hardware

by Anastasios Dimitriou

Deep Neural Networks (DNNs) are increasingly capable of solving many cognitive problems, with countless everyday life applications such as autonomous driving, voice assistants, medical diagnoses, etc. Their increased computational, memory, and energy demands led to the widespread adoption of powerful GPU cloud servers for their execution. However, there is an increasing interest in moving the computation of DNNs to the edge. By processing data locally, the dependency on cloud infrastructure is reduced, thereby decreasing the data transmission load, improving privacy and security, and increasing the availability of networks in environments with limited internet connectivity. However, edge devices (IoT devices, mobile phones, etc.) are typically resource restricted in processing power, memory, and power consumption. This research aims to contribute towards enabling the deployment of complex DNNs on resource-restricted devices.

Field Programmable Gate Arrays (FPGAs) have been proven to be very effective in accelerating neural networks due to their configurability, parallelization capabilities, and low energy consumption. Nonetheless, attempting to map modern DNNs onto them without compression is not feasible. Dynamic DNNs are approaches that go beyond the limits of static model compression by tuning computational workload to the difficulty of inputs on a per-sample basis. This thesis first explores the challenges introduced by this DNN approaches when their FPGA implementation is targeted. Three limiting factors were identified: the lack of software libraries and frameworks, the lack of hardware modules and frameworks, and the dependencies on intermediate feature maps.

Having identified these challenges, the next contribution of this thesis is a first realisation of dynamic networks on FPGAs. The design followed the standard architecture and achieved a minimum of **3.2x** faster execution over a Jetson embedded device and comparable latency to a CPU/GPU system while maintaining very low energy consumption (at least **1.8x** less than the Jetson). This highlighted the feasibility and efficiency of deploying the dynamic network on FPGAs. Nonetheless, FPGAs are inherently parallel devices, and leveraging that, the third contribution of the thesis is a second design approach that explores the simultaneous execution of the two main components of dynamic networks. It addressed two main challenges concerning the dependencies on intermediate feature maps and further accelerated the execution of the dynamic network by up to **23%**.

Finally, targeting the versatility and reconfigurability of FPGAs, the last contribution of the thesis is the exploration of two confidence-controlled dynamic schemes. Utilising control values generated by dynamic networks, the first dynamically selects the location of the exit points within the network, and the second the applied quantisation level. Both approaches enhance the adaptability and performance of the early-exit dynamic DNN, achieving an **18%** reduction in computations and up to a **21.9%** reduction in latency, respectively, with minimal accuracy drops.

Contents

List of Figures

List of Tables

Abbreviations

AI	Artificial Intelligence
ASIC	Application-Specific Integrated Circuit
BNN	Binary Neural Network
BRAM	Block RAM (Random Access Memory)
CAD	Computer-Aided Design
CDFG	Control and Data-Flow Graph
CLB	Configurable Logic Block
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DDR	Double Data Rate
DNN	Deep Neural Network
DPR	Dynamic Partial Reconfiguration
DRAM	Dynamic Random Access Memory
DRP	Dynamically Reconfigurable Processor
DSP	Digital Signal Processor
FC	Fully Connected (Layer)
FLOPs	Floating Point Operations Per Second
FP32	32-bit Floating Point
FPGA	Field-Programmable Gate Array
GPU	Graphics Processing Unit
HLS	High-Level Synthesis
IEEE	Institute of Electrical and Electronics Engineers
IO	Input/Output
IoT	Internet of Things
ILA	Integrated Logic Analyzer
LSTM	Long Short-Term Memory
LTE-M	Long-Term Evolution Machine
LUT	Lookup Table
MAC	Multiply-Accumulate
ML	Machine Learning
NAS	Neural Architecture Search

NPU	Neural Processing Unit
PE	Processing Element
RAM	Random Access Memory
RIA	Regular Iterative Algorithms
RTL	Register Transfer Level
SoC	System on Chip
TPU	Tensor Processing Unit
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
VLIW	Very Long Instruction Word
VPU	Vision Processing Unit

Chapter 1

Introduction

Deep Neural Networks (DNNs) have emerged to be a transformative and very effective technology, revolutionising various cognitive applications such as computer vision [1], [2], voice recognition [3], natural language processing [4] etc. Characterized by their ability to model complex patterns, modern DNNs, such as ResNet [2], DenseNet [5], VGG [6], GoogleNET [7] and Transformers [4] have achieved unprecedented performance, while recent research on Neural Architecture Search (NAS) [8], [9] enables easier and faster design of even more powerful and complex structures. Their success derives from many reasons, including advancements in computational power, availability of larger datasets, increased model depth and innovations in training. Figure 1.1 highlights the increase in parameter numbers of deep neural networks in the decade between 2012 and 2022.

This increase in complexity and depth leads to significantly higher computational demands. Traditional CPU platforms often fail to process deep neural network models at the speeds required for real-time tasks. GPUs, on the contrary, offer a viable solution in many cases by leveraging their vast parallel computing capabilities to accelerate matrix-vector operations, which are predominant in DNN computations. Most modern DNN applications are executed on powerful cloud servers containing hundreds of GPUs. More specifically, while initially deployed locally, they retrieve the data from devices like mobile phones or embedded platforms, upload them to the cloud, where inference is executed and then receive the output. These servers have immense processing power, but their operation is very expensive for both natural resources and energy needs. For example, the execution of the ChatGPT [10] language model requires 10000 GPUs and millions of dollars for their operation and cooling [11], [12].

However, there is an increasing interest in moving the execution of inference locally [14], [15], [16], not only reduces the cost of executing DNNs [17] but also can result in lower overall latency and energy consumption (including both computing and communication) compared to using cloud servers. This reduction in latency enhances user

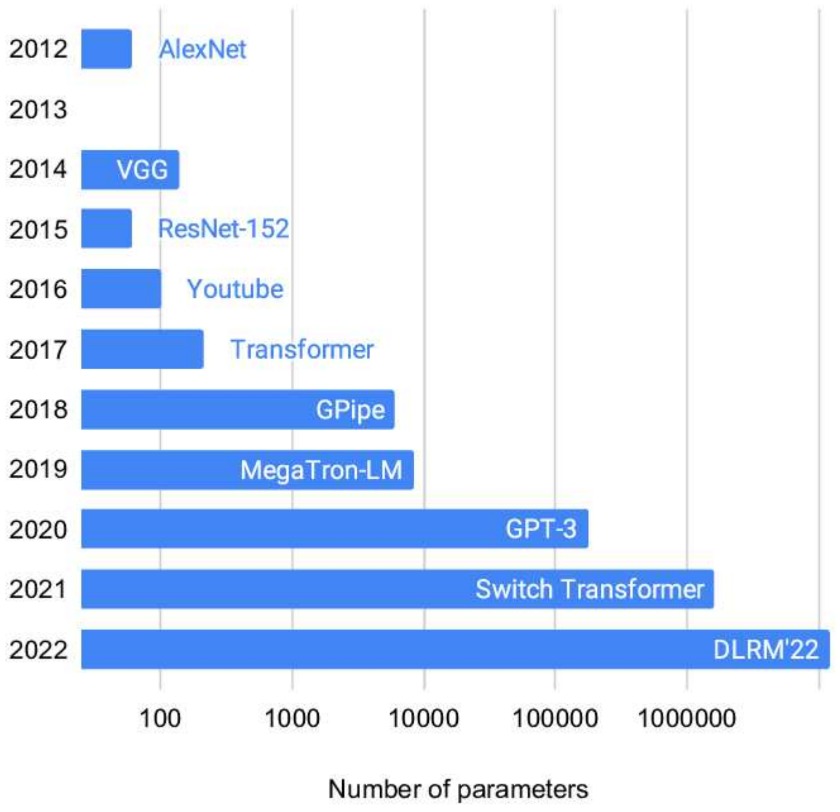


FIGURE 1.1: Number of deep neural network parameters. The x-axis is plotted on a log scale. Reprinted from B. Steiner et.al. [13].

experience [18] and allows for meeting the strict timing demands of real-time applications, such as self-driving cars [19]. Furthermore, many of the data sent to the cloud are personal, e.g. audio clips for voice assistants, and keeping them locally helps to mitigate any potential privacy concerns [20]. Finally using the cloud servers requires accessing the internet, and despite recent advancements in wireless communication (e.g. 5G, LTE-M) there are multiple areas with limited or non-existent connectivity. Performing inference locally can eliminate this issue [21].

Moving the execution of DNNs locally mandated the deployment of the neural models on devices with very restricted resources, e.g. IoT devices, mobile phones, etc. For low-resource settings, custom hardware architectures can be specifically designed for DNNs, optimizing processing elements to meet the required computational characteristics. A device that is proven a strong candidate for accelerating neural network operations is Field-Programmable Gate Arrays (FPGAs) due to their flexibility and parallelization capabilities, all while maintaining very low power consumption. They consist of a flexible collection of logic elements and IP blocks that can be configured for a specific application. When designed efficiently, they can achieve higher power efficiency than GPUs and CPUs [22].

Nevertheless, large networks can require around 40 billion operations [6] and hundreds of millions of parameters to process even a single 224 × 224 image. Despite the capabilities of FPGAs, they are devices with a restricted amount of resources, and the aforementioned DNN requirements render their deployment infeasible. To overcome that, the networks' compression is mandatory.

1.1 Research Justification

Typical approaches to reduce the inference workload and memory footprint of DNNs, which consequently translates to reduced energy consumption, primarily focus on parameter reduction. Quantization [23] entails lowering the precision of DNN weights and activations from the standard IEEE floating-point value, reducing the memory footprint and the complexity of the computations. Extreme quantization levels involve representing parameters as single bits or employing Binarised Neural Networks (BNNs) [24]. Pruning [25] is another approach through which a subset of the network's redundant or less significant parameters are removed, which greatly reduces the model's memory footprint. Nonetheless, this often entails computing large, sparse, and sometimes unstructured matrix multiplications. Also, removing certain connections or neurons may reduce the network's capacity to generalize across diverse datasets, leading to over-fitting or under-fitting in certain scenarios. Finally, Knowledge Distillation [26] has gained popularity as a technique to decrease the size of networks during training by incorporating the loss from larger capacity networks into smaller ones. All of these approaches result in more compact and efficient models, however they act in a static manner, permanently affecting the network which leads in the reduction of accuracy and representation power. To further enhance efficiency and performance, dynamic approaches are utilized.

As opposed to static methods of compression, dynamic approaches generate DNN architectures that can adapt their characteristics to various variables, including network inputs, hardware resources, etc. They are able to strategically allocate computations on demand at run-time by selectively activating model components (e.g. layers [27], channels [28] or sub-networks [29]) which consequently leads to a reduction of computations spent on easy to recognise samples or less informative parts of an input. Furthermore, data-driven network architectures enhance DNNs by enabling a significantly enlarged parameter space and improving representation power. For example, by applying feature-conditioned attention weights on a convolution layer's kernel, the model's capacity can be boosted [30], [31]. The popular soft attention mechanism could also be integrated into the framework of dynamic networks, as different channels of features are dynamically re-weighted or pruned at run-time (Figure 1.2). Finally, dynamism can also be driven by environmental hardware variables (i.e. available memory, power consumption constraints, etc.), giving them the ability to deal and adapt with different hardware platforms and varying computational budgets [32].

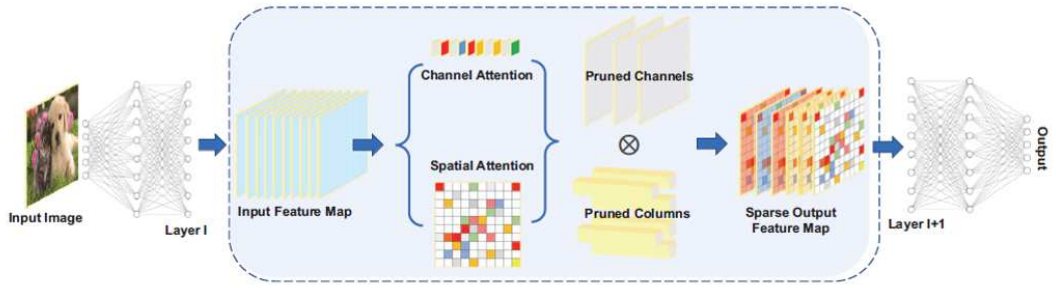


FIGURE 1.2: Dynamic channel and spatial column pruning. Reprinted from J. Shen et.al. [33].

Alongside dynamic network architectures, hardware-focused research on dynamic execution of DNNs aims to adapt the hardware platform to the model's characteristics. These approaches utilize the programmability and reconfigurability of FPGAs to adjust them to the layers' computational needs of the target DNN. While networks are getting deeper and more complex, their layers' types and characteristics are changing and have an increased variation [34]. For example, by altering the size of a systolic array of process elements, more efficient computation of depth-wise over normal convolution can be achieved [35]. This exploits even more the parallelism and acceleration potential of the FPGAs and makes them more appealing to deeper networks.

Although dynamic networks showcase great results, not only accelerating but also increasing the effectiveness, simplicity and adaptability of complex and computationally intensive networks, there is a gap between theoretical findings and real-world applications [36]. Dynamic networks are designed using libraries optimized for static models and evaluated on conventional CPU-GPU systems, resulting in inconsistency between anticipated and actual effectiveness. CPU performance still cannot meet the real-time processing requirements in embedded devices, and GPU power consumption is too high. On the contrary, FPGAs have already proven to be very effective at realising DNNs and maintaining very low energy demands. At the same time, the dynamic hardware approaches unlock further adaptability to modern networks' needs. Combining the hardware capabilities of FPGAs and the benefits of dynamic DNN architectures can be a vital contribution towards enabling the deployment of modern neural networks locally on edge devices, yet nobody has considered this to date.

1.2 Research Questions

Following the research justification above and the literature review in Chapter 2, this thesis investigates the benefits of realising a dynamic neural network in a custom hardware platform such as FPGAs. Dynamic DNNs have already been proven on a theoretical level to significantly reduce computational needs and, as a result, the energy needed to execute a neural network's inference. However, there aren't any practical results, so this thesis aims to further exploit their capabilities and accelerate and reduce their power needs with FPGAs. This platform has already been shown to offer enhanced capabilities to efficiently accommodate DNNs while achieving very low energy consumption, and its programmability and reconfigurability render it the perfect device for research. In summary, the three questions answered in this thesis are:

Q1. **What are the challenges Dynamic DNNs' architectural characteristics introduce to their implementation on FPGAs? Can Dynamic DNNs be implemented on FPGAs?**

Research question **Q1** aims to investigate the feasibility of realising dynamic DNNs on FPGAs. As their architecture differs from static networks, they introduce limitations in deploying state-of-the-art libraries, tools and frameworks, making their implementation challenging. This question is answered in Chapter 3.

Q2. **How can the performance of Dynamic networks be enhanced with the parallelization capabilities of FPGAs?**

FPGAs are very capable of parallelising computations. How can the dynamic approaches' computations be parallelised and produce more efficient hardware designs? This question is answered in Chapter 4.

Q3. **How the versatility/reconfigurability of FPGAs can be utilised to enable dynamic exit point placement and quantisation in dynamic DNNs?**

FPGAs allow the use of custom data flows, representation resolutions, and even dynamic reconfiguration of the hardware design. How can this versatility and reconfigurability enhance the dynamic network effectiveness and lead to lower latency and energy consumption by dynamically controlling the depth of exit points and the quantisation level? This question is answered in Chapter 5.

1.3 Research Contributions

To address the above research questions, the contributions that have been made during this research are summarized as follows:

C1. **Study on the challenges of realising Dynamic DNNs on FPGAs.** - Chapter 3 highlights the differences in the dynamic networks' architecture and the limitations they introduce both to software design and to state-of-the-art hardware approaches, targeting the realisation of dynamic DNNs on FPGAs. This contribution partially answers Question 1.

Anastasios Dimitriou, Benjamin Biggs, Jonathon Hare, Geoff V. Merrett. **FPGA Acceleration of Dynamic Neural Networks: Challenges and Advancements**. In *IEEE International Conference on Omni-Layer Intelligent Systems (COINS), London, UK, 2024.*

C2. **The first realisation of Dynamic DNNs on FPGAs** - Chapter 3 presents a first FPGA design that fully deploys early-exit dynamic DNNs on FPGAs. The design's experimental results showcased the capability of FPGAs to efficiently deploy and execute dynamic DNNs while achieving very low energy consumption. This contribution partially answers Question 1.

Anastasios Dimitriou, Mingyu Hu, Jonathon Hare, Geoff V. Merrett. **Exploration of Decision Sub-Network Architectures for FPGA-based Dynamic DNNs.** *In Design Automation and Test in Europe (DATE) Conference, Antwerp, Belgium, 2023.*

C3. **Exploration on how the parallelisation capabilities of FPGAs can be leveraged to increase the efficiency of early-exit dynamic DNNs implementations** - Chapter 4 proposes a second design approach that, by utilising the parallelization capabilities of FPGAs, addresses two of the early-exit networks' implementation challenges. Aditionally a first modelisation of the execution time and the energy consumption of early-exit dynamic DNNs inference on FPGAs is presented. This contribution answers Question 2.

Anastasios Dimitriou, Lei Xun, Jonathon Hare, Geoff V. Merrett. **Realisation of Early-Exit Dynamic Neural Networks on Reconfigurable Hardware**. *Submitted to IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems (TCAD), 2024.*

C4. **Two dynamic approaches that utilise the confidence values of input samples that fail to trigger an early exit.** - Chapter 5 introduces two confidence-controlled dynamic schemes concerning the placement of intermediate classifiers and quantisation. Employing the versatility and reprogrammability of FPGAs, the dynamic schemes increase the efficiency and performance of the early-exit dynamic DNN designs with minimal accuracy drop. This contribution answers Question 3.

1.4 Thesis Structure

This thesis is divided into several chapters that address different aspects of realising dynamic DNNs on FPGA platforms. Chapter 2 provides an overview of the fundamentals and current state-of-the-art in deploying efficiently DNN models, as well as techniques for both software and hardware dynamic execution approaches of neural networks.

Chapter 3 focuses on answering (**Q2**). Initially, the architecture differences between static and dynamic networks and how they introduce limitations in their implementation on FPGAs are presented. More specifically, a thorough exploration of the architecture of early-exit dynamic networks and the challenges they introduce in software and hardware design are presented, alongside some recent advancements to resolve them . A first realisation of the early-exit dynamic networks is explored, and experimental results comparing CPU, CPU-GPU, embedded and FPGA platforms over execution time and energy consumption are discussed.

Chapter 4 extends this work and introduces a second design approach, leveraging the parallelisation capabilities of FPGAs and answers to the second research question (**Q2**). Experimental results are also discussed, while a thorough comparison between the two design approaches over execution time, energy consumption, memory footprint and decision threshold levels highlights which better fits different target applications. Furthermore, two equations are presented to model the energy demands and the execution time of the early-exit dynamic networks' inference on FPGAs, targeting both architectures.

Chapter 5 explores how the information from the input sample's confidence level, obtained by the intermediate classifiers, can be combined with the capabilities of the FPGAs to enhance the performance of the early-exit networks. More specifically, this Chapter explored two different directions, answering research question 3 (**Q3**). The first focuses on a dynamic placement of the early-exit networks' intermediate classifiers scheme, taking advantage of the ***parallel*** design approach in order to further adapt the early-exiting architecture to the variety of the input samples. Following that, the second direction explores the quantisation of the early-exit dynamic DNNs initially applied only on the intermediate classifiers and then on the entirety of the dynamic network. Finally, it introduces a dynamic quantisation scheme, where different network parts have different quantisation levels, to reduce resource needs with minimal effects on early-exiting behaviour and accuracy.

Finally, Chapter 6 summarizes this thesis's key findings and discusses plans for future research in this area.

Chapter 2

Background Theory

Efficient deployment and execution of DNNs on mobile and embedded platforms are crucial for their widespread adoption [14]. However, this task presents many challenges [15], [16], as despite achieving state-of-the-art performance, deep neural networks (DNNs) are inherently both computationally and memory access intensive. This Chapter begins with an overview of modern DNNs in Section 2.1, followed by an exploration and comparison of the different hardware platforms targeting their deployment in Section 2.2, focusing on the FPGAs, which is the target device of this thesis. To achieve the deployment of DNNs in resource-restricted devices, however, their compression is necessary. Section 2.3 provides an overview of popular network compression methods and, finally, Section 2.4 and Section 2.5 presents a thorough review of current literature in dynamic deep neural networks, covering both software and hardware directions.

2.1 Deep Neural Networks

Deep Neural Networks (DNNs) are one of the foundation models of modern machine learning, inspired by the structure and function of the human brain. They consist of interconnected layers of artificial neurons that process data through a series of transformations. Unlike traditional neural networks, DNNs contain multiple hidden layers between the input and output layers (Figure 2.1), allowing them to learn and model intricate patterns and relationships within data. This depth and different model architectures enable DNNs to tackle highly complex tasks, such as convolutional neural networks [37] for image recognition and sequential models (e.g. LSTMs [38]) for natural language processing.

At the core of DNNs are neurons, which are the basic units that perform computations. Each neuron receives inputs, applies a set of weights, adds a bias, and then passes the

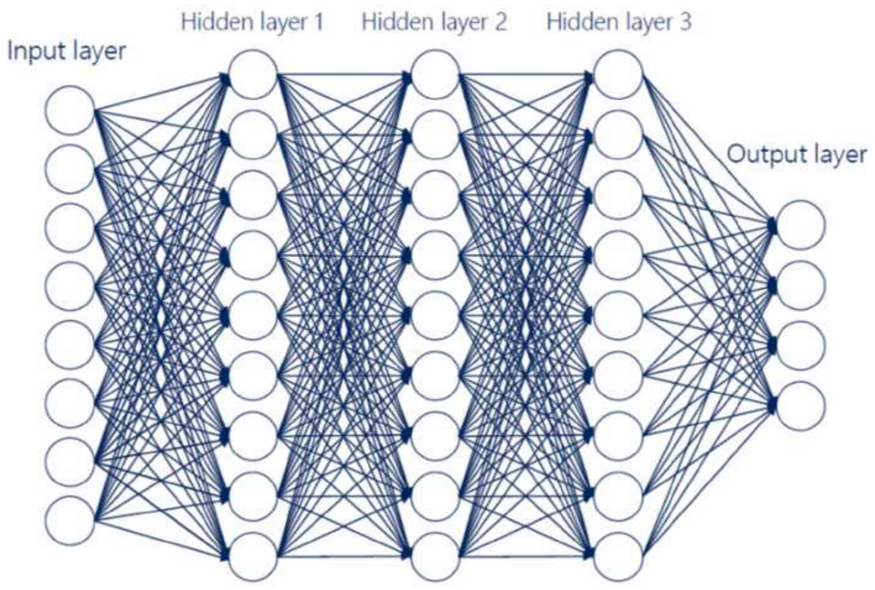


FIGURE 2.1: Deep Neural Network (DNN) example. Reprinted from M. Merenda et.al. [39].

result through an activation function. This process helps to introduce non-linearity into the model, enabling it to learn from errors and adjust the weights during training. The layers of neurons are typically organized into three main types: the input layer, which receives raw data; multiple hidden layers, where intermediate computations are carried out; and the output layer, which produces the final predictions or classifications.

However, before deploying DNNs, they have to be trained. Training a DNN involves optimizing the weights and biases using a process called backpropagation [40], which adjusts these parameters to minimize the error between the predicted output and the actual target. This is achieved through iterative algorithms such as stochastic gradient descent. The depth and complexity of DNNs allow them to capture high-level abstractions in data, making them particularly effective for tasks that require an understanding of hierarchical structures, such as recognizing objects in images or understanding the context in language.

2.1.1 Convolutional Neural Networks

Convolutional Neural Networks (CNNs or ConvNets) are a specialized class of deep neural networks designed primarily for processing structured grid-like data, such as images. Inspired by the visual cortex of the human brain, CNNs are particularly effective at capturing information in data through their unique architecture, which includes convolutional, pooling, and fully connected layers. This allows CNNs to automatically and adaptively learn spatial hierarchies of features, making them exceptionally powerful for tasks such as image classification, object detection, and segmentation.

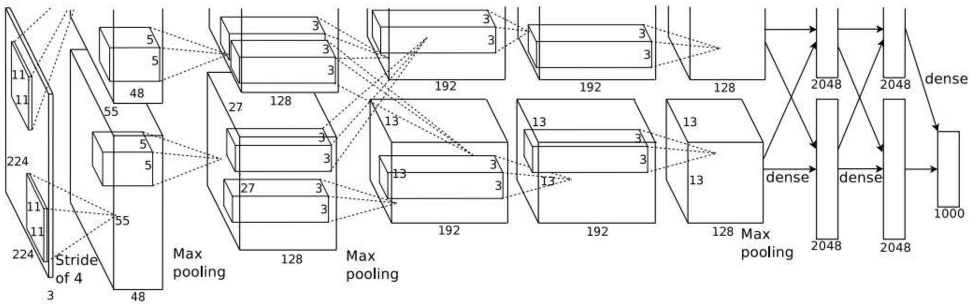


FIGURE 2.2: Architecture of AlexNet CNN model. Reprinted from A. Krizhevsky et.al. [1].

The core of CNNs are convolutional layers, which apply a series of learnable filters (or kernels) to the input data. These filters slide over the input data to produce feature maps, capturing local patterns such as edges, textures, and simple shapes in the initial layers. As the data passes through deeper layers, the network combines these simple patterns to detect more complex features and structures. This hierarchical feature extraction process enables CNNs to recognize objects in images regardless of variations in position, scale, and orientation.

Pooling layers are another crucial component of CNNs. These layers perform down-sampling operations on the feature maps, reducing their spatial dimensions and the number of parameters, which helps to control over-fitting and makes the network more computationally efficient. The most common type of pooling is max pooling, which takes the maximum value from each sub-region of the feature map. This step retains the most salient features while discarding less important information, further enhancing the network's ability to generalize from the training data.

The final layers of a CNN typically consist of fully connected layers, which integrate the high-level features extracted by the convolutional and pooling layers to perform the final classification or regression tasks. These layers are similar to those in traditional neural networks, where each neuron is connected to every neuron in the previous layer. Figure 2.2 shows the architecture of the popular CNN model AlexNet [1], which consists of an input layer, five convolution layers and three fully connected layers.

2.2 Hardware Platforms for Inference

As explained above, implementing a deep learning application is a two-step process: training and inference. Training is used to meticulously clean and set up data and weights to improve the model's generalization capabilities. Machine learning libraries such as Theano [41] and TensorFlow [42] in Python can make the implementation of neural network training relatively easy, as they include a number of useful functions for backpropagation [40], optimization algorithms (e.g stochastic gradient descent (SGD)

[43], Adam [44]). However, the actual training process can take a significant amount of resources and time to complete due to the complexity of the back-propagation algorithm and the number of iterations necessary to adjust weights and biases. This operation, though, in most occasions, happens only once before the deployment of the network, usually in very powerful GPU servers.

The second step, inference, is the phase where the trained model is deployed to make predictions on new, unknown data. Unlike training, inference is computationally less demanding because it involves only forward propagation through the network without the need for backpropagation or weight updates. The primary goal of inference is to apply the learned patterns and relationships from the training phase to real-world data. The efficiency and speed of inference are critical, especially in applications requiring real-time predictions, such as autonomous driving or online recommendation systems. While training is performed on powerful hardware like GPUs to handle the extensive computations, inference can be executed on a variety of devices, including edge devices like smartphones, thanks to optimizations like model quantization and pruning that reduce the model's size and computational requirements.

2.2.1 CPU and GPU

A first approach in executing inference of a DNN is to use software and exclusively the central processing unit (CPU). This strategy is attractive because it provides generality, as a high-level software solution can run on any platform without the need for any special hardware. It is also the simplest, both in terms of development time and design complexity. Machine learning frameworks, such as Theano [41] or TensorFlow [42] provide the programmer with a lot of predefined functions that can reduce the magnitude of the code and simplify the network's training, inference and debugging processes. Additionally, they provide tools to target a specific device's needs, e.g. TensorFlow Lite [45], which was developed specifically to easily adapt TensorFlow code for mobile phone applications. The limitations of this approach are few, and often, it should be the first choice for application designers, especially if the real-time requirements for system latency and performance are modest.

However, for applications that require low latency and high productivity, a CPU-focused implementation may be insufficient. This follows from the underlying calculations performed in the forward propagation. With N hidden units and M-dimensional input, this calculation requires $M * N$ multiplications, $M * N$ additions, and N calculations of the activation function. For a basic scalar processor, these calculations must be performed sequentially, where the $M * N$ term dominates the complexity. Modern superscalar multi-core processors, as well as low-level optimized linear algebra libraries, such as BLAS [46], introduce parallelism that helps to accelerate computations between matrices and vectors. Nonetheless, there will always be a limit to the computational performance of CPU devices.

To overcome that, graphics processing units (GPUs) were introduced, initially targeting only training [1], and then every aspect of the DNN execution [47]. Due to their thousands of processing cores, GPUs provide massive parallelism potential that can speed up matrix-vector multiplication, the predominant operation in DNNs. GPUs have become a very popular tool for deep learning, and they were quickly adopted by many machine learning frameworks, too. Most of the aforementioned tools offer smooth GPU integration through their back-end (e.g. cuDNN [48]), while many hardware manufacturers have begun designing their GPUs specifically for deep learning applications, making this transition even easier.

The deployment of GPUs, however, raises some challenges. These devices are very expensive both in financial cost (a high-performance GPU can cost thousands of pounds [49]) and in power demands. For a complex neural network-based speech recognition algorithm, a GPU application requires almost twice the amount of power required by the CPU [50]. While the total energy consumption for a particular task may be lower in a GPU than in a CPU, the high-power requirement of a GPU may be prohibitive for some platforms, such as embedded systems. So, in order to meet performance requirements while minimizing power consumption, custom hardware designs are vital.

2.2.2 Neural Processing Units

NPUs, or Neural Processing Units, are specialized hardware components designed specifically for accelerating neural network computations [51], [52]. These units are tailored to handle the intense computational requirements of artificial intelligence (AI) and machine learning (ML) algorithms, particularly deep learning models. NPUs are optimized for tasks like matrix multiplication, convolution operations, and other operations commonly found in neural networks. The primary purpose of NPUs is to enhance the performance and efficiency of AI and ML workloads, enabling faster inference and training times compared to general-purpose processors like CPUs and GPUs.

A typical NPU architecture comprises processing units optimized for matrix multiplication and convolution operations, a memory subsystem for storing parameters and data, and specialized instruction sets tailored to neural network computations. Data movement engines facilitate efficient data transfer between memory hierarchies and processing units, while power management mechanisms dynamically adjust power consumption to balance performance and energy efficiency. Integration interfaces enable connectivity with host processors and peripherals. A software stack, including drivers and runtime environments, supports developers in compiling, optimizing, and deploying DNNs. Overall, NPUs are designed to accelerate AI workloads efficiently, and they have already been deployed in devices like mobile phones and laptops [53], [54].

NPUs can be implemented as standalone chips or integrated into larger systems-on-chip (SoCs) alongside other processing units like CPUs, GPUs, and DSPs (Digital Signal Processors). Nonetheless, they have fixed architectures. While they can accelerate the execution of DNNs, custom hardware must be used for applications that require reconfigurability, adaptability, and even lower energy demands.

2.2.3 Application-Specific Integrated Circuits

CPUs, GPUs, and NPUs are platforms that allow easy development and deployment of neural network applications, while the latter two also offer high effectiveness. However, in cases where there is a restriction in power consumption, latency, resources, etc., some significant limitations arise. Custom hardware designs can be the strategy to manage demanding workloads and reduce long-term energy costs.

Generally, a custom hardware solution is called Application-Specific Integrated Circuit (ASIC). Their main difference with general-purpose processors is that while they are designed to handle a wide variety of tasks, ASICs are specialized microchips explicitly designed and contain only the resources needed to perform a single specific operation. Following that, ASICs can be highly optimized in computing and latency efficiency, maintaining very low energy demands. For example, an ASIC accelerator for a neural network targeting a computer vision application achieved 13 times faster computation performance and 3,400 times less power consumption than a GPU solution [55].

Their effectiveness can be explained by the fact that inference, as an operation, has some properties that characterise it as a very good candidate to be realized and optimized in custom hardware. First of all, for every forward propagation, the type, order, and number of operations performed are known prior to execution. This makes the decision process of what types and the number of computational units (e.g. adders, multipliers, etc.) straightforward for the designer. Furthermore, knowing the data types and their access schedule can lead to high data movement optimization. Secondly, the predominant matrix multiplications are highly parallelisable. ASICs can be designed to handle these specific operations more efficiently than general-purpose processors by incorporating specialized hardware accelerators and optimized memory hierarchies. Finally, DNNs' weights and variables are known and remain unchanged during inference. This gives an opportunity to implement various compression schemes, such as quantization, which can significantly reduce the memory footprint.

However, the development of ASICs can be expensive and time-consuming due to the need for custom design and manufacturing processes. It requires skills in electronics and complicated CAD software. In addition to that, an ASIC can only be built once, and any need for alternations to its architecture require the complete reconstruction and replacement of existing devices, which is very expensive and time consuming.

2.2.3.1 Tensor Processing Units

A special case of ASIC accelerators specifically targeting the computational needs of machine learning applications is Tensor Processing Units (TPUs) (Figure 2.3). Their architecture incorporates a vector processing unit, a systolic array-based matrix multiplication unit, a Very Long Instruction Word (VLIW) instruction set, 2D vector registers, and a transpose reduction permute unit (Figure 2.4). The core of TPUs is the matrix multiplication unit, and their philosophy is to keep it as busy as possible. This way it maximizes the exploitation of the advantages of the systolic array on calculations between tensors, which is the most common operation among machine learning applications. Finally, despite the fact that TPUs do not support out-of-order execution, they accomplish very high instruction-level parallelism. This is achieved by special compilers (e.g. Google's XLA [56]) applying techniques like critical path scheduling and register allocation.

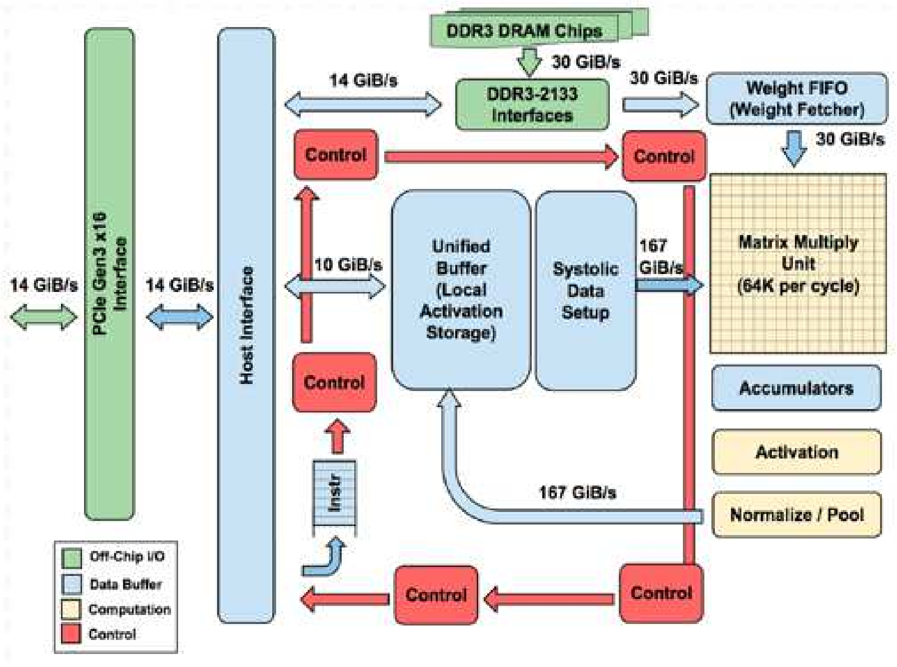


FIGURE 2.3: Google's TPU block diagram. Reprinted from N. P. Jouppi et.al. [57].

Google started using TPUs in 2015 and made them available to the public in 2018 through a cloud service for training and running machine learning algorithms [58]. Their architecture reduced the energy overhead per computation by more than 10x (Figure 2.4) and achieved 35x more trillion operations per second over a CPU solution [57].

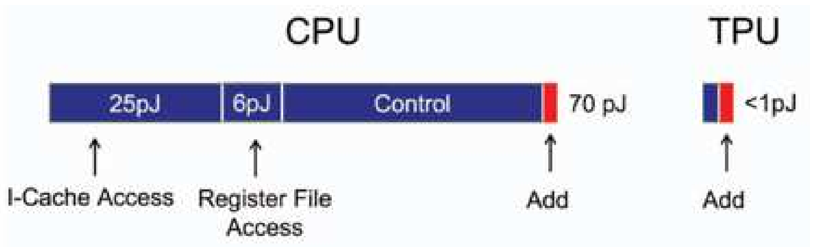


FIGURE 2.4: Instruction fetch and execute energy consumption between a traditional CPU and a TPU. Reprinted from N. Jouppi et.al.[59].

2.2.4 Field Programmable Gate Arrays

Field-programmable gate arrays (FPGAs) present a reconfigurability aspect that is missing from ASICs. FPGAs consist of a large, interconnected array of configurable logic blocks (CLBs), which can be reconfigured to implement various and complex digital circuits. They also offer on-chip block memory (BRAM) and Digital Signal Processors (DSPs) for mathematical operations, both connected to the CLBs (Figure 2.5). This architecture enables parallel processing capabilities, as multiple logic blocks can operate simultaneously to calculate different sections of an application. This parallelism makes FPGAs particularly suited for applications requiring high-speed data processing and real-time performance, such as signal processing, telecommunications, and embedded systems. Despite the fact that they lack the energy consumption levels and performance of ASICs, they are still considerably faster and more energy efficient than CPUs and GPUs. An FPGA accelerator for a speech recognition DNN application achieved 43- and 197-times better speed and energy efficiency, respectively, over a network being executed on CPU and 3 and 14 times over a network being executed on GPU [50]. Microsoft already from 2015 had deployed FPGA accelerators for its search engine (Project Brainwave [60]) and increased its ranking throughput by 50%.

One of the key advantages of FPGAs is their ability to be reprogrammed to accommodate changes in functionality or to optimize performance for different tasks. This makes them highly valuable in research and development environments, where design requirements can evolve rapidly. This is why FPGAs are the target platform in this work.

2.2.4.1 Systolic Arrays of Process Elements

One of the most prolific architectures of FPGAs and ASICs is the systolic arrays of process elements. In essence, systolic arrays are hardware structures that replace a pipeline structure with an array of homogeneous process elements (PEs) that can perform a common mathematical operation. These elements are locally interconnected, and they are able to communicate with each other in a synchronous manner. This communication pattern extends to the computations, too, as it constricts the data flow and

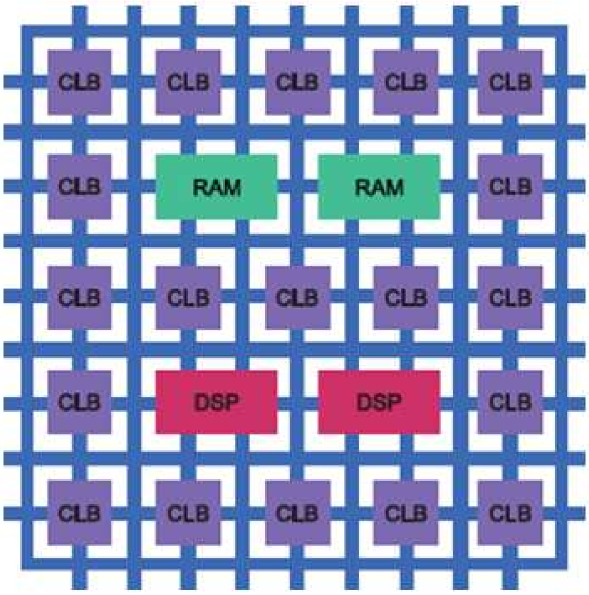


FIGURE 2.5: Typical structure of a FPGA.

consequently sets a restriction on the type of algorithms that can be mapped on systolic arrays. According to K. Rao et al. [61], these algorithms are called "Regular Iterative Algorithms" (RIAs) and must satisfy three conditions:

1. Every variable must be defined by a name and a finite set of indices
2. Variables must not be assigned multiple times but only once
3. The difference between the indices of the variables between the left-hand side and the right-hand side of an operation must be constant for every recurrence relation

Due to their structure, they can provide high-throughput capacity and great acceleration for RIAs. Some great examples of such algorithms are matrix multiplication, multi-dimensional image processing, decision-based algorithms, etc. Additionally, they add reconfigurability to the design with the ability to manipulate the active rows and columns of PEs and save fabric space, as the intercommunication and computations do not need to be driven by control units.

Therefore, as the majority of computation in DNNs are matrix-to-matrix multiplications, they became very popular with neural network accelerators. For example, Figure 2.6 shows the deployment of a systolic array for convolutional neural network's inference [62]. The weights are stationary and stored in the PEs, and the input data are continuously inputted from top to bottom. Each PE performs a multiplication and an addition with the previously calculated result from its left PE. The final result is then propagated to the nearest PE on the right.

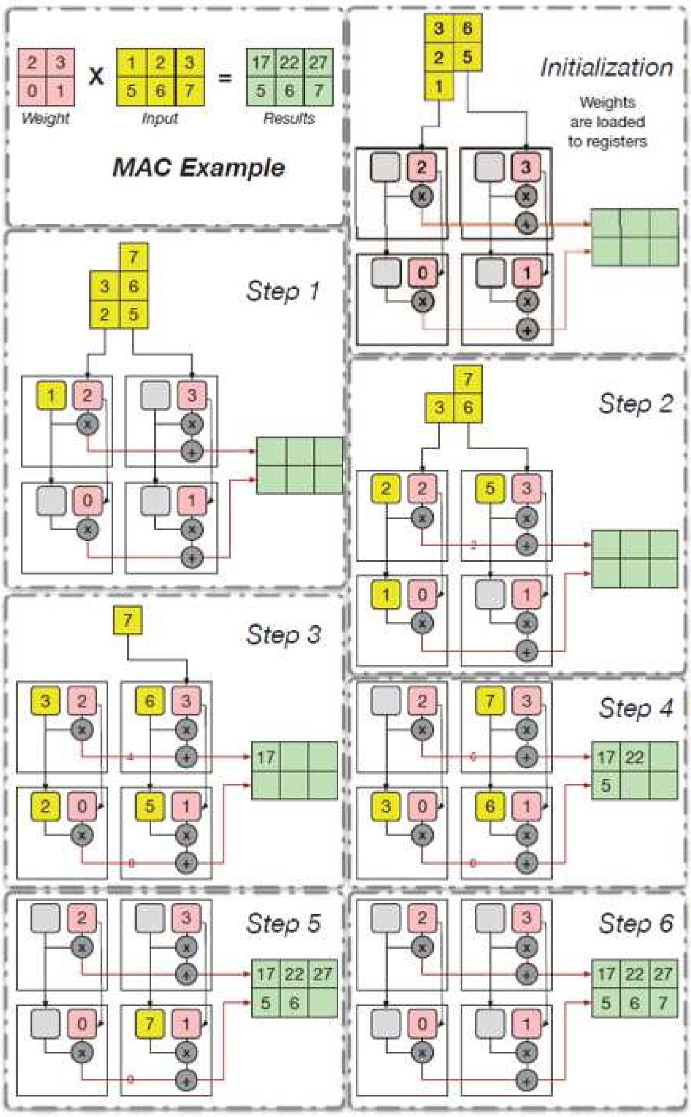


FIGURE 2.6: Matrix-Multiplication on a Systolic Array. Reprinted from J. Zhang et.al. [62].

Recent research elevates the use of systolic arrays on FPGAs. By making small changes to the structure of the PEs and by adding some controllers, researchers aim to adapt systolic arrays to some more challenging tasks arising from the evolution of acceleration techniques on neural networks. J. Zhang et al. [62] proposed an architecture to handle the very sparse weight matrices after unstructured pruning. More specifically, they considered the systolic array's size when they transformed the pruning's sparse output into a denser matrix. This way, they created blocks of data that were easier to map on the systolic array and achieved very high computation efficiency Figure 2.7, highlighting the acceleration capabilities of the systolic array.

With the same mindset, R. Xu et al. [35] achieved to create an architecture that is able to perform the computations for both normal and depth-wise convolution. The latter one transforms the matrix-matrix multiplication into matrix-vector, leading to high levels of underutilization of the systolic arrays (Figure 2.8). So by only increasing by 1% the design area, they added a number of multiplexers, a register and a control unit. They achieved a structure that could properly accelerate both techniques of convolution, illustrating the versatility of the systolic array architecture.

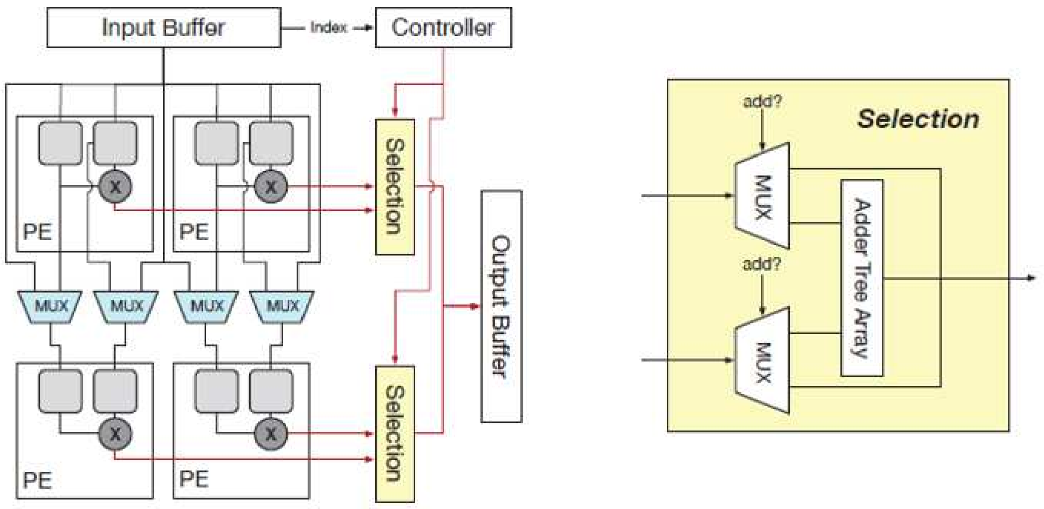


FIGURE 2.7: Example of a modified systolic array's structure. Reprinted from J. Zhang et.al. [62].

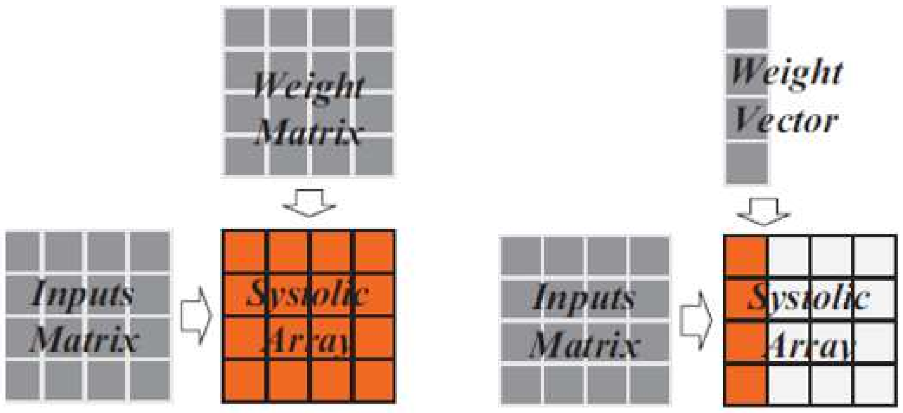


FIGURE 2.8: Normal vs Depth-wise convolution when mapped on a SA. Reprinted from R. Xu et.al. [35].

2.2.4.2 Dynamic Partial Reconfiguration

As explained above, FPGAs offer the flexibility of re-programming and redesigning without the need for re-fabrication like ASICs. Dynamic Partial Reconfiguration (DPR) enhances this flexibility even more, allowing the reconfiguration of a portion of the programmable logic while the rest of the system continues to operate normally. This capability provides significant advantages in terms of flexibility, resource optimization, and efficiency. With DPR, different tasks or functions can be loaded into specific regions of the FPGA on the fly without interrupting the operation of the entire system. This is particularly useful for applications that require real-time adaptability and responsiveness to changing conditions.

A design taking advantage of DPR is divided into two parts, a static and a dynamic. The static part (grey in Figure 2.9) is a full bit-stream of a design, and it's loaded during start-up. The dynamic part (black in Figure 2.9 labelled Reconfig Block A) is the partition of the design that can be changed by a partial configuration file in bit-form (A1.bit to A4.bit in Figure 2.9). These files can be downloaded and change the re-configurable areas of the FPGA at run time and without compromising any other non-configurable part of the initial bit-stream.

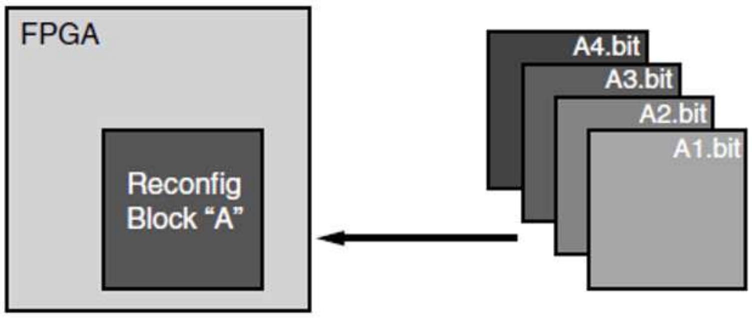


FIGURE 2.9: Basic presumption of Dynamic Partial Reconfiguration. Reprinted from *"Vivado Design Suite User Guide: Partial Reconfiguration (UG909)"* [63].

Overall, DPR expands the design space of FPGAs by allowing for greater flexibility, resource efficiency, and adaptability. The ability to multiplex hardware dynamically on a single FPGA board can generate:

- smaller architectures than a static design to implement a given function, which consequently means less production cost and power needs
- extra flexibility, increasing the amount of operations ready to run on the same piece of fabric
- extra tolerance to errors
- more choices in design security

2.2.4.3 Toolflows for mapping DNNs on FPGAs

FPGAs typically require device domain knowledge to effectively use the benefits mentioned in the above sections due to the complexities of tooling and the required fine-tuning of algorithm implementation. This makes the process of designing them an expensive and time-consuming task. However, there are many tools that can map high-level expressions of DNNs directly to the device [64]. The majority of industries related to the production of FPGA boards have developed toolkits to help hardware engineers create easier and better designs. For example, Intel has introduced Hyperflex [65], which is a Python-based command line toolkit that imports trained models from popular learning frameworks like TensorFlow [42], analyses and adjusts them to better fit a targeted device (CPU, GPU, Vision Processing Unit (VPU) or FPGA).

Similarly, Nvidia introduced NVDLA [66], which is a free and open-source architecture that provides a standard approach for DNN accelerator design. It works by grouping the DL inference operation into four categories: i) Convolutions, ii) Activations, iii) Poolling and iv) Normalisation. Then, it maps these operations using one of the pre-developed hardware components (Convolution Core, Single, Planar or Channel data processor, dedicated memory, Data Reshaping Engines). Every hardware block is then independently optimised and connected with the others to form a complete architecture.

Research towards tool-flows for mapping neural networks on FPGAs is a very popular research interest too. For example, fpgaConvNet, proposed by S. Venieris et al. [67], is an end-to-end framework that, given a trained CNN and the description of the targeted device (Xilinx's FPGA boards) as inputs, it produces an FPGA architecture file in High-Level Synthesis (HLS) code. Figure 2.10 describes the framework's processing flow.

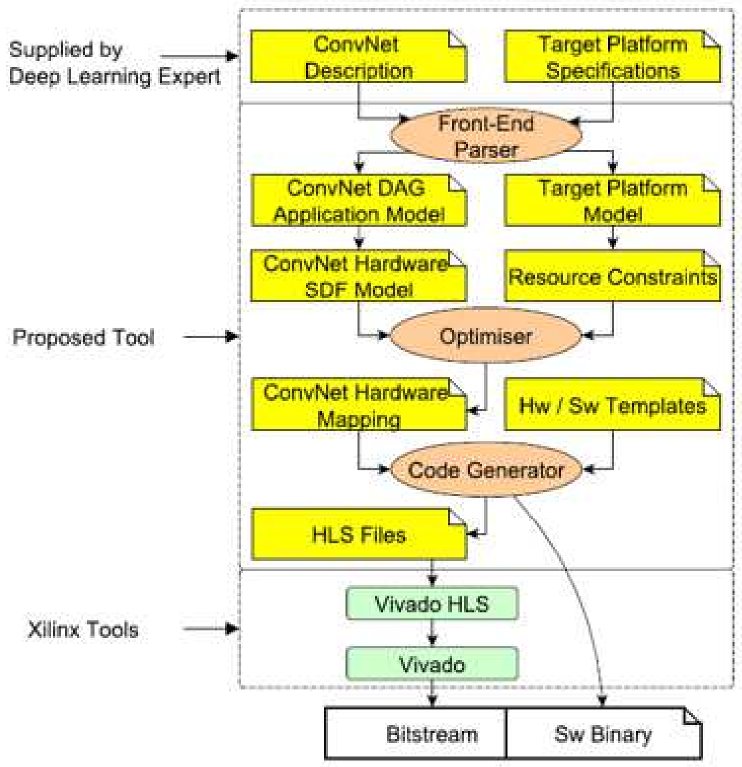


FIGURE 2.10: fpgaConvNet processing flow. Reprinted from S. I. Venieris et.al. [67].

The process begins with HLS, which translates CNN models described in high-level programming languages (Caffe [68] or Torch [69]) into hardware descriptions that can be implemented on an FPGA. This approach simplifies the development process, allowing designers to focus on the algorithmic aspects of the CNN without delving deeply into hardware design. fpgaConvNet also incorporates model compression techniques, such as quantization and pruning, which reduce the computational load and memory requirements of the CNN. These techniques help fit larger models onto the limited resources of an FPGA and improve inference speed. Moreover, the framework employs a layer-wise optimization strategy, where each layer of the CNN is mapped onto the FPGA in a manner that balances computational workload and memory access patterns. This strategy ensures efficient utilization of the FPGA's resources, such as logic gates and memory blocks, and minimizes data movement, which can be a bottleneck in CNN inference. By optimizing data flow and computation for each layer, fpgaConvNet achieves low latency and high throughput, making it suitable for real-time applications.

A similar approach is HPIPE (Heterogeneous Layer-Pipelined and Sparse-Aware CNN Inference for FPGAs) [70], an innovative convolutional neural network accelerator architecture designed specifically for FPGAs. The primary objective of HPIPE is to leverage the inherent parallelism and reconfigurability of FPGAs to accelerate CNN inference while efficiently managing computational and memory resources. It achieves this by combining two key strategies: heterogeneous layer-pipelining and sparse-aware processing.

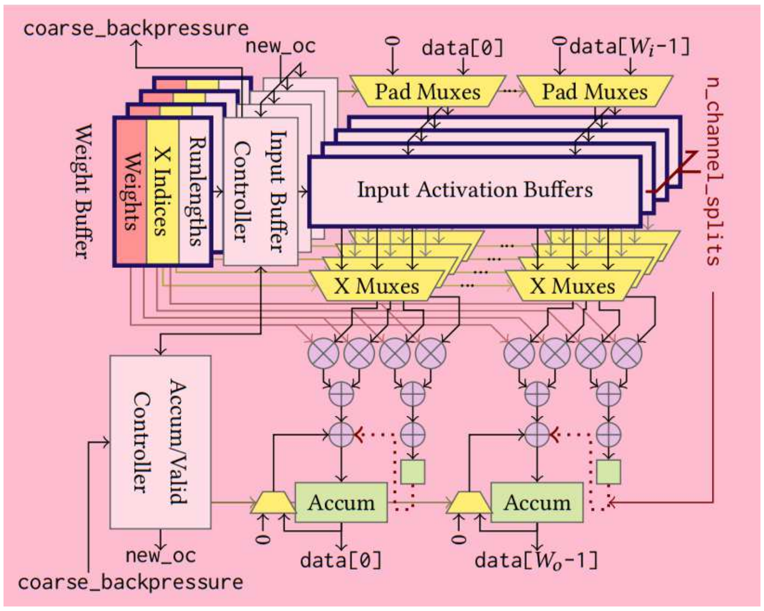


FIGURE 2.11: HPIPE architecture. Reprinted from M. Hall et.al. [70].

Heterogeneous layer-pipelining involves decomposing the CNN layers into distinct stages that can be executed in parallel across different regions of the FPGA. Each stage corresponds to a layer or a group of layers in the CNN, and these stages are configured to match the specific hardware capabilities of the FPGA. This pipelined approach ensures that data flows continuously from one layer to the next, minimizing latency and maximizing throughput. By distributing the workload across multiple FPGA regions, HPIPE can exploit the parallelism of FPGAs to process multiple layers simultaneously, thereby speeding up the inference process.

Sparse-aware processing is the second crucial component of HPIPE, which takes advantage of the sparsity in CNN weights and activations. In many CNNs, a significant portion of the weights and activations are zero, leading to unnecessary computations if not managed properly. HPIPE integrates mechanisms to identify and skip these zero values during computation, reducing the number of operations and memory accesses required. This not only accelerates the inference but also reduces power consumption. By combining these techniques, HPIPE provides a highly efficient and scalable solution for deploying CNNs on FPGAs, achieving significant improvements in speed and energy efficiency compared to traditional approaches.

2.3 Network Compression

Despite their effectiveness in accelerating DNNs and their very low power demands, FPGAs still have a limited amount of resources. On the contrary, modern neural models are growing deeper and more complex to adapt to many real-life machine learning problems (autonomous cars, language processing, healthcare applications, etc.). This translates to increased depth, variety in layer types, complexity, and computational burden. Training and inference became very time and resource-consuming operations, even for very powerful CPUs and GPUs, and made the deployment of DNNs on resource-restricted devices a very difficult or even impossible task. That led to a very broad research field that focuses on increasing model compactness. Some of the most popular techniques to reduce the size of DNNs are presented below.

2.3.1 Quantisation

Quantization is a technique used to reduce the precision of the numbers that represent a model's parameters and activations. Typically, neural networks are trained using 32-bit floating-point representations, which provide high precision but also consume significant memory and computational resources. The process of quantization involves mapping the continuous range of floating-point values to a discrete set of lower-precision values. For instance, in an 8-bit quantization, the continuous range of values is mapped to 256 discrete levels. This reduction in precision can significantly decrease the model's size, leading to lower memory bandwidth requirements and faster arithmetic operations, which are crucial for deploying models on resource-constrained hardware like FPGAs, mobile devices, and edge computing platforms. More intense quantisation can reduce the representation resolution as low as 4-bit [71], [72] or even one [24].

However, quantization can introduce some challenges, primarily related to maintaining model accuracy. Reducing the precision of weights and activations can lead to a loss of information, potentially degrading the model's performance. To address this, various techniques are employed, such as fine-tuning the quantized model to recover lost accuracy [73] and using sophisticated quantization schemes like mixed-precision quantization, where different parts of the network use different precision [74].

2.3.2 Pruning

Pruning [25] is another technique used in deep learning to reduce the size of a neural network by removing less important or redundant parameters. This process aims to simplify the network, making it more efficient in terms of memory footprint and computational burden without significantly sacrificing performance. Pruning can be particularly beneficial for deploying models on resource-constrained hardware such as FPGAs, mobile devices, and edge computing platforms.

There are two main types of pruning: structured [75], [76], and unstructured [77]. Structured pruning involves removing entire components of the network (Figure 2.12), such as filters, channels, or layers, which leads to more regular and predictable reductions in computational complexity, making it easier to implement on hardware. On the contrary, unstructured pruning involves removing individual weights that have little impact on the network's output. While this can achieve higher compression rates, it results in irregular sparsity patterns that can be harder to exploit for performance gains in standard hardware architectures.

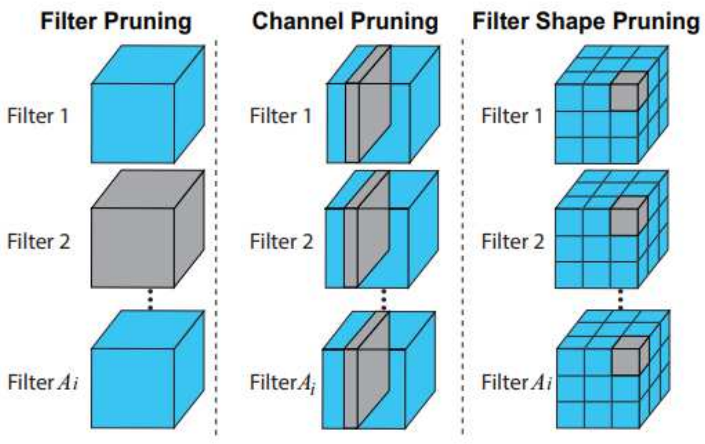


FIGURE 2.12: Different structured pruning schemes. Reprinted from N. Liu et.al. [75].

Unavoidably, the removal of these elements leads to a loss in accuracy and representation power. The pruned network is fine-tuned or retrained to help the network adjust to its new, smaller structure and regain performance [78], [79]. Although pruning can introduce some complexity in terms of implementation and fine-tuning, it is a powerful method for creating more efficient neural networks that can be deployed in a wider range of applications, balancing the trade-off between computational resources and model accuracy.

2.3.3 Knowledge Distillation

Knowledge distillation [26] is a technique where a smaller, more efficient "student" model is trained to replicate the behaviour of a larger, more complex "teacher" model. The primary goal of this approach is to transfer the knowledge learned by the large model to a smaller model that can perform similarly but with significantly reduced computational and memory requirements.

The distillation process involves training the student model to mimic the outputs of the teacher model. Instead of training the student model on the original dataset, it is trained using the soft outputs (probabilities) produced by the teacher model. These soft outputs contain more information than hard labels (the correct class) because they convey the teacher model's confidence in its predictions across all classes, not just the correct one. This additional information helps the student model learn the underlying data distribution more effectively, often resulting in improved generalization and performance compared to training directly on the original dataset with hard labels.

2.4 Dynamic DNNs

Despite their ability to compress DNNs effectively, these approaches have a limitation in common. They permanently affect the networks by using a larger network's trained weights, reducing the precision or even completely removing parameters and activations of the deployed network. This is done in a general and static way, reducing the number of parameters and the complexity of the model, which can remove useful information and limit the model's capacity to learn and generalize complex patterns. Furthermore, hardware resources are hardly ever similar or constantly available on different platforms, so their adaptability for inconsistent resources is very low.

Dynamic DNNs are architectures that adapt their structure or computation during inference based on the input data, aiming to balance computational efficiency and model performance. Their core idea is that neural models are trained to identify class-level features. Each network's input has a certain feature activation variance and will only trigger a subpart of the whole network. Unlike static DNNs, which have a fixed architecture and process every input similarly, dynamic DNNs can adjust parameters like the number of active layers, neurons, or the type of operations performed. This

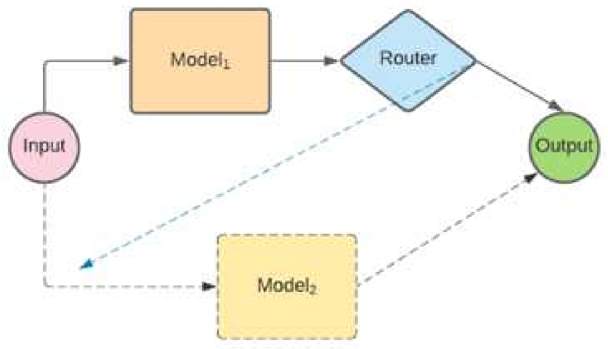


FIGURE 2.13: Cascading of models

adaptability allows dynamic DNNs to allocate computational resources more effectively, processing simpler inputs faster while dedicating more resources to complex inputs, ultimately achieving a better trade-off between speed, accuracy, and resource utilization.

Current research on Dynamic DNNs is divided into two different directions: structure-focused and parameter-focused. The first one introduces dynamic properties to the structure of the neural network, while the second dynamically adapts its parameters like weights, activations etc. Both approaches are presented below.

2.4.1 Structure-focused Dynamic DNNs

This dynamic network approach focuses on the general architecture and structural components of a model. Different input samples have different computational demands. In other words, there are "easier" (canonical) samples for the network to classify, which can be achieved at the very early stage of the inference execution. On the contrary, samples that are "harder" (non-canonical) demand the execution of deeper layers in order for the network to extract the needed amount of information to produce a correct classification. Consequently, identifying when a network is ready to produce an output is vital as a great amount of computations can be avoided.

Early-exiting is a structure-focused dynamic approach that allows inference to stop in an intermediate stage when the network has sufficient information to classify the input. If x_i is the *ith* input sample, the forward propagation of a N-layer deep network can be written:

$$y = L_{n_i} \circ L_{n_{i-1}} \circ \cdots \circ L_1(x_i), 1 \leq n_i \leq N \tag{2.1}$$

Where L_{n_i} is the *i-th* output of the N network layers, n_i the layer at which the inference stops and y the network's output.

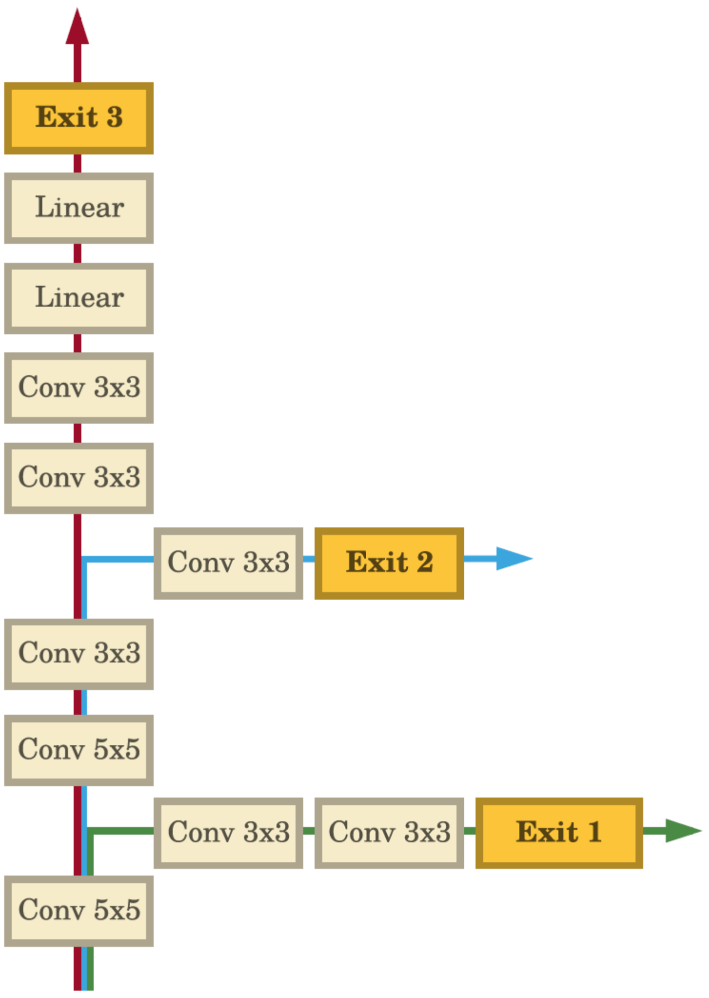


FIGURE 2.14: Network with Intermediate Classifiers. Reprinted from S. Teerapittayanon et.al. [81].

A common technique that enables early-exiting is cascading multiple models (Figure 2.13). This approach employs two models, one containing a smaller amount of layers and a second which is deeper. After the execution of the first, a decision function determines if the amount of abstracted information from the input sample is enough to classify it. If they are, then this model's output is used, and the dynamic network proceeds with the next input samples. If they are not, the second model is enabled, and the classification is determined by its output. E. Park et al. [80] propose a dynamic model with two CNN networks, where if the difference between the two largest outputs (SoftMax) of the first network exceeds a threshold, early exit occurs. Despite containing two models, they achieved over 80% reduction in energy on the MNIST dataset and over 40% on ImagNet with less than 2% accuracy loss in both cases. Nonetheless, this approach has a drawback, as all the information obtained from the first model is lost if the decision condition is not met. The two models are completely independent, and when the execution of the second is needed, the inference must start again from scratch.

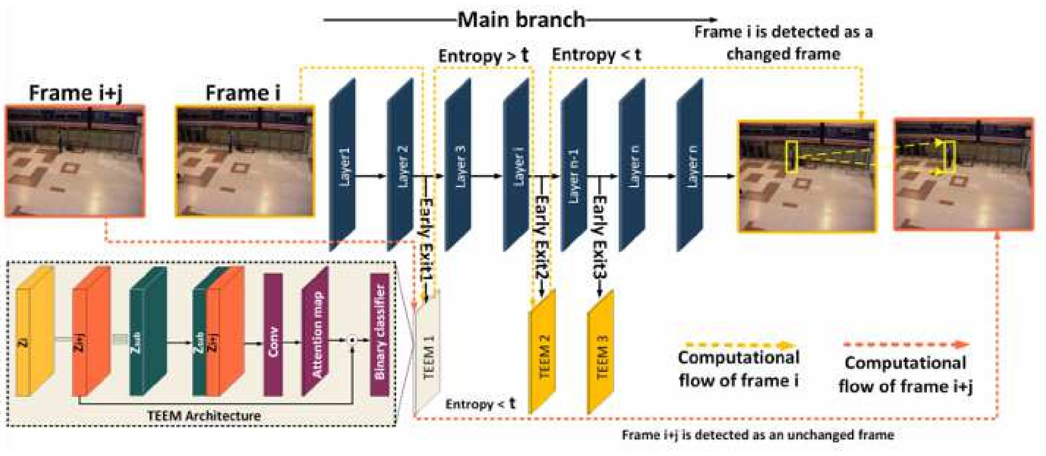


FIGURE 2.15: Dynamic model that performs early exit based on the similarity of consequent video frames. Reprinted from A. Sabet et.al. [82].

To solve this problem, instead of using multiple models, the decision for early-exiting is made between the layers of a DNN (Figure 2.14). This approach introduces intermediate classifiers, which are auxiliary output sub-networks placed at various depths within the backbone network, which from now on shall be referred to as the backbone network. They are used for the extraction of a confidence value, which is a metric of the features extracted up to that point of execution. Each of them operates independently, leveraging the partial feature representations to make its own prediction. If the confidence level of an intermediate classifier meets or exceeds a predefined threshold, the backbone network can terminate further processing and output the prediction from that classifier, effectively exiting early. S. Teerapittayanon et al. [81] using this approach, with a 2% increase in parameters amount, achieved 38% fewer computations with minimal loss in accuracy.

An extension of this approach is similarity-aware inference. This approach is applied mostly to video recognition applications. For example, A. Sabetsarvestani et al. [82] check in-between layers the similarity level of two consecutive video frames (Figure 2.15). If the abstracted features from the second frame are very similar to the previously processed one, the inference stops and restarts with the next frame.

Following the core idea of early-exiting, **layer-skipping** is a more flexible dynamic DNNs approach. Instead of completely stopping the execution of inference, it proposes skipping only the execution of specific layers. Equation 2.1 can now be written:

$$y = (S_i \circ L_{n_i}) \circ (S_{i-1} \circ L_{n_{i-1}}) \circ \cdots \circ (S_1 \circ L_1)(x_i), 1 \leq n_i \leq N \tag{2.2}$$

Where S_i denotes the decision function that decides the execution or not of the *ith* layer.

Gating functions are usually employed to deploy layer-skipping. They use the outputs of a layer and produce a binary value that denotes if the next layer will or will not be executed. SkipNet [83] is a great example. Let x_i be the input, $L_i(x_i)$ be the operation function of the *ith* layer or group of layers, then the output of the gated layer is ([83]):

$$x_{i+1} = G_i(x_i)F_i(x_i) + (1 - G_1(x_i))x_i \tag{2.3}$$

where $G_i(x_i) \in 0, 1$ is the gating function for layer i, that determines if it will be executed or not.

Z. Wu et al. [84] on the other hand, in order to implement layer-skipping, introduce an extra model (policy network) which, based on the network's inputs it is trained to produce binary gates that decide which of a pre-trained ResNet model's layers are going to be executed. Both methods add computational overhead in order to decide upon the execution of a layer, but the overall reduction in computational burden is substantially greater. On a ResNet model, SkipNet [83] achieves a 30% to 80% reduction in computations, dependent on the "harshness" of the gates, and BlockDrop [84] a 20% overall acceleration over a CPU/GPU solution.

A generalised approach combining early-exiting and layer-skipping is **dynamic routing**. In this case the network is converted into a Directed Acyclic Graph (DAG), and control modules are introduced. Every node represents either a layer or a control module and every edge a data line or a control line respectively. These control modules are similar to the intermediate classifiers presented above. Driven from input instances or user-defined global performance metrics (available resources, target latency, etc.), they either produce a gate-like output or a probability map, indicating which of the next layers or sub-graphs are going to be activated. L. Liu et al. [85] submit a novel deployment of dynamic routing, where every control node outputs a set of control scores, one for each of its outgoing control edges (Figure 2.16) achieving a 40% increased execution efficiency of ResNet-110. Z. Yan et al. [86], on the other hand, execute a small CNN network in order to broadly categorize each sample and, then, based on this categorisation, execute a specific sub-network.

It can be noticed that all of the above-mentioned approaches interfere with the depth of a network and focus on the interaction variability of the input sample with each of the backbone network's layers. Delving deeper into the layers, this translates to the number of neurons whose feature preference would align with those of the input. However, while an input sample may not stimulate the majority of a layer's neurons, there can be some that can generate useful information [87]. An alternative approach to completely skipping the execution of one or several layers is keeping all of them and selectively activating a part of their components (neurons, batches or channels). A line of work connected to this idea is **Mixture of Experts (MoE)**, where a number of

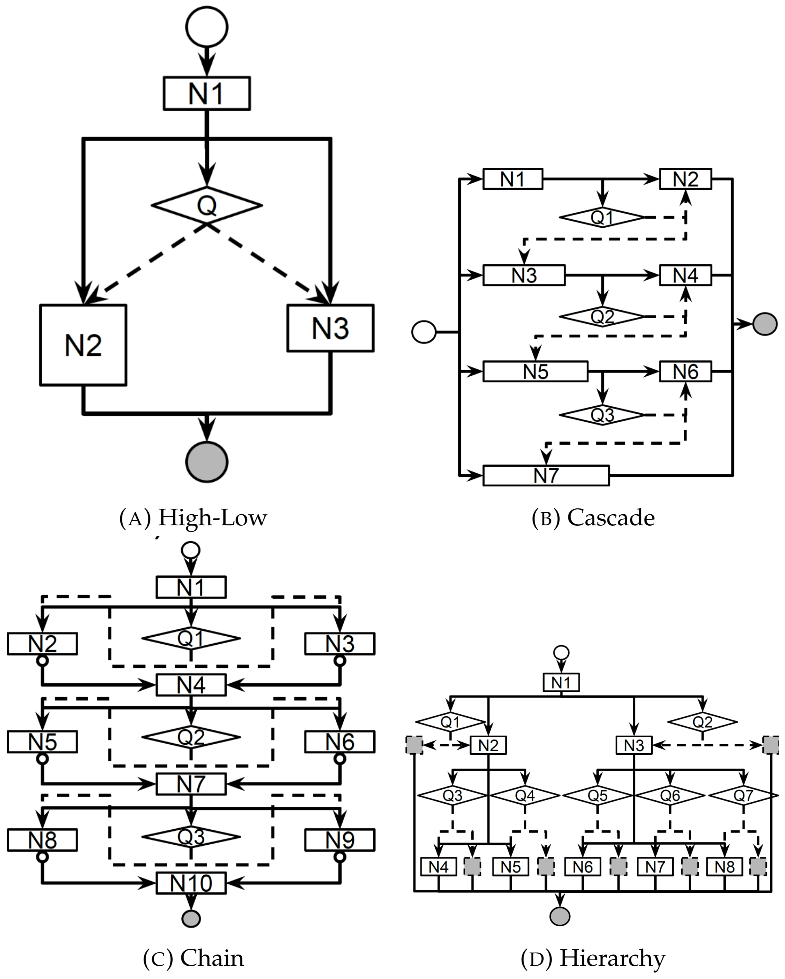


FIGURE 2.16: Four different dynamic routing networks. Reprinted from L. Liu et.al. [85].

parallel sub-networks (Experts), that each target a specific part of the problem's space, are combined, via an ensemble, using data-dependent weights. In more detail, if x is an input, G is a gating mechanism that produces an N-dimensional g vector that controls the execution of N experts ($F_1, F_2, \ldots, F_n$). The inference function can now be written as:

$$y = \sum_{K=1}^{N} [G(x)]_k F_k(x) = \sum_{k=1}^{N} g_k F_k(x) \tag{2.4}$$

If $g_n = 0$ then the n-th expert will not be executed, otherwise it will. HydraNet [88] (Figure 2.17) implements that by changing the convolutional blocks with multiple branches and selectively executing them at run-time. Doing so, N. Shazeer et al. [88] reduced the inference on a *ResNet* and a *DenseNet* model by at least 2x times.

A final approach towards enabling dynamic execution of DNNs is **dynamic pruning**. Existing static pruning methods reduce a great number of a network's parameters,

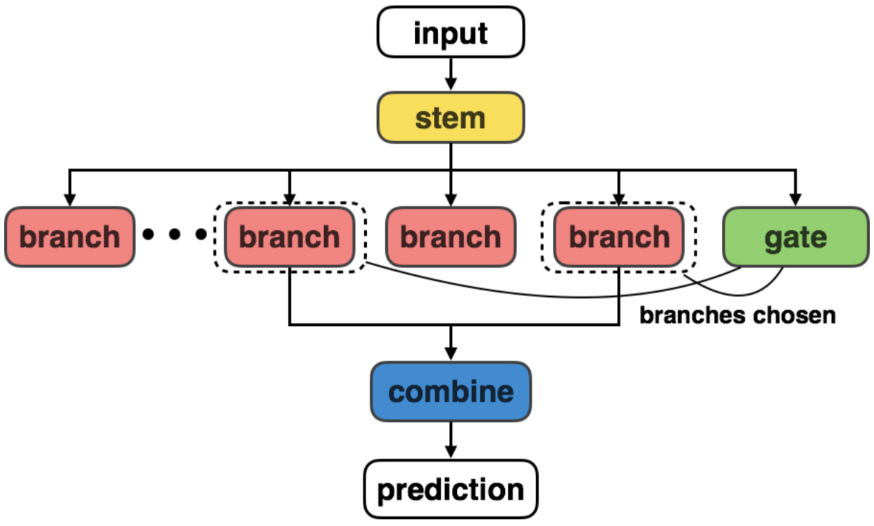


FIGURE 2.17: HydraNet's architecture: consists of multiple branches specialized for different inputs, a gate that selects which of them will be executed and a combiner that aggregates their outputs to make a final prediction. Reprinted from R. T. Mullapudi et.al. [88].

pruning permanently removes unimportant weights, neurons, or layers during or after training based on global importance metrics, resulting in a fixed, smaller network with reduced computational cost. However, since the pruned structure remains unchanged during inference, it lacks adaptability to varying input complexities and often requires fine-tuning to recover lost accuracy.

As discussed above, different inputs are better recognized by different parts of a network or layer. By permanently deleting the network's parameters, the model loses the ability to process some input samples accurately and permanently. Contrary to static, dynamic pruning suggests not deleting any weight or feature parameters but instead using a specific portion of them, decided by the current network's input. This makes the model more flexible and efficient, as it can allocate resources dynamically depending on the complexity of the input. For example, Z. Chen et al. [28] (Figure 2.18) propose GaterNet, a framework for input-dependent dynamic filter selection in CNNs. They employ similar to the target DNN but considerably smaller, which infer the input sample and generate binary gates that control the target network's filters. This way it maintains the pruning benefits while achieving less than a 2% reduction in Top-1 Error on the ResNet model using the Cifar100 test-set. J. Lin et al. [89] implement dynamic pruning by modelling it as a Markov decision process and use RNN to select specific channel groups.

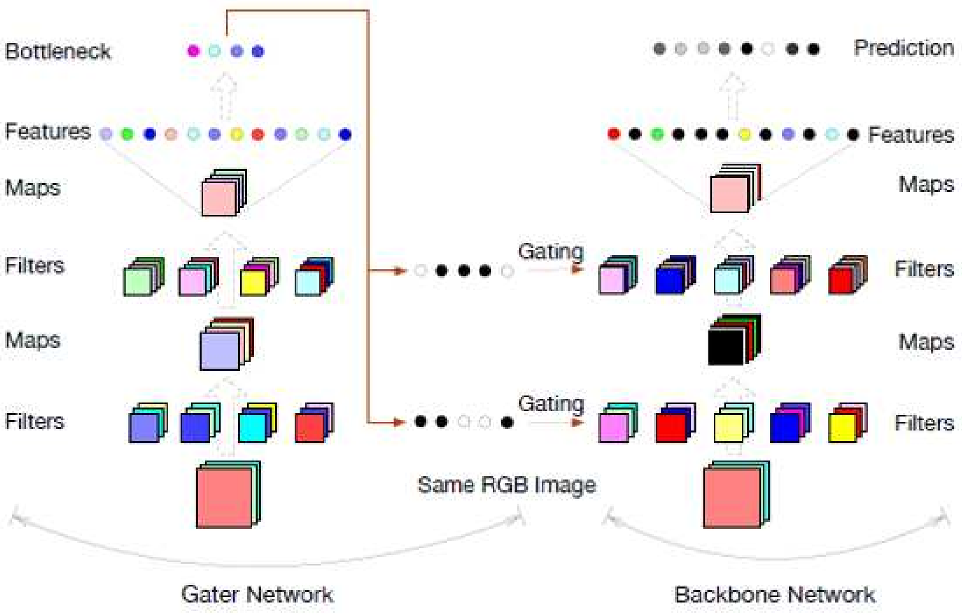


FIGURE 2.18: GaterNet's model architecture. Reprinted from Z. Chen et.al. [28].

2.4.2 Parameter-focused Dynamic DNNs

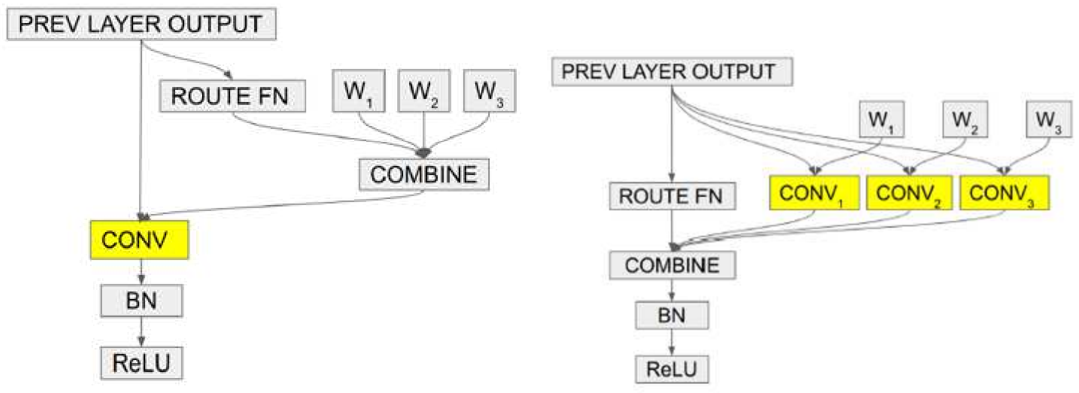
In contrast to structure-focused, parameter-focused approaches perform dynamic inference on a fixed architecture but with dynamic parameters, aiming to add more informative features and, therefore, increase the network's representation power. A fundamental assumption in the design of deep neural networks is that the same kernels are applied to each instance of a data-set. To increase a model's capacity, developers usually add more layers or increase the size of existing kernels, both of which increase the total network's computational cost.

To overcome this, published work applies soft attention mechanisms on multiple kernels in order to produce an adaptive set of parameters and increase the representation power of a model without the additional computation burden of the previous methods (**parameter adjustment**). For example, CondConv [30] computes convolutional kernels as a function of the inputs. More precisely, it parametrises the convolutional kernels as a linear combination of N experts:

$$y = \sigma((\alpha_1 W_1 + \alpha_2 W_2 + \dots \alpha_n W_n) * x) \tag{2.5}$$

where $\alpha_1, \dots, \alpha_n$ are functions of the input learned through gradient descent and $\widetilde{W} = \alpha_n W_n, n = 1, 2, \dots, M$ the weights of M kernels.

Following CondConv architecture, increasing the capacity of the network is the simple task of increasing the number of experts. This method achieves higher representation



(A) CondConv Network (B) Network following Mixture of Experts method

FIGURE 2.19: (A) CondConv architecture with three n=3 Kernels vs. (B) Mixture of Experts approach. There are two mathematically equivalent networks, but CondConv, with input-driven parameterization of convolutional kernels, requires 2 fewer convolutions. Reprinted from B. Yang et.al. [30].

power, maintaining high efficiency and minimal extra computational burden, as the result of fusing the outputs of M convolutional blocks, as in Mixture of Experts, is equal to that produced by performing one convolution with $\widetilde{W}$. This means that this approach needs only the *1/Mth* of the computation (Figure 2.19) that a Mixture of Experts architecture would require to achieve the same outcome.

Weight prediction is another technique to apply dynamic parameter adjustment. Rather than using the already calculated training weights, this approach directly generates them in the test phase. DFN [90] is a great example that leads to a 15% reduction in loss on the synthetic moving *MNIST* dataset. It consists of two parts (Figure 2.20), a filter-generating network and a dynamic filter layer. The first takes in a network's input and generates $I_A \in R^{h \times w \times c_A}$ where h,w and c_A are the width, height and number of channels of an input A. It produces filters F_θ parameterized by parameters $\theta \in R^{s \times s \times c_B \times n \times d}$ where s is the filter size, c_B the number of channels of input B, n the numbers of filters and d is a variable, equal to 1 for dynamic convolution. Finally, a dynamic filter layer takes the filters F_θ and the input I_B and generates an output $y = F_\theta(I_B)$. An LSTM network is chosen for the generation of the filters, but any other differential architecture is appropriate for use (multilayer perceptron, convolutional layer, etc.)

Finally, a soft attention mechanism can also be used to create feature kernels with adaptive size. F. Yu et al. [87] propose AntiDote, attention-based dynamic CNN, where they calculate both channel-wise and spatial-wise attention coefficients by measuring the average of the feature maps in the spatial dimensions (Height * Width) and in the dimension of depth respectively. Then, they use these coefficients to prune and reshape the feature maps. They achieved a 40% FLOPs reduction on *VGG16* running *CIFAR100*.

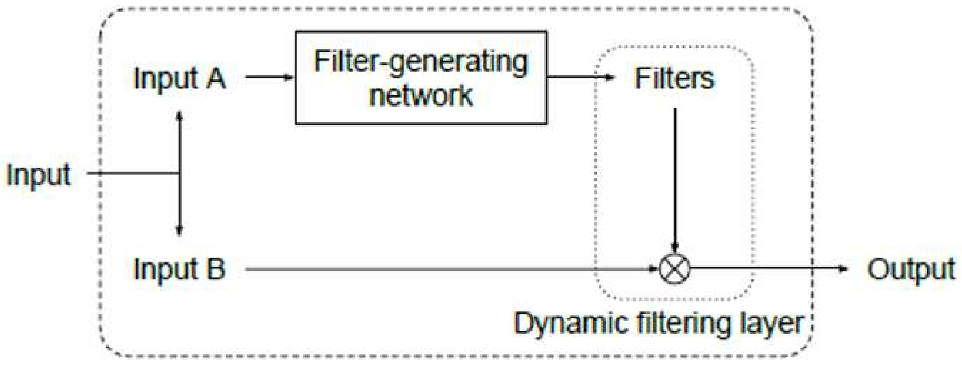


FIGURE 2.20: Dynamic Filter Network (DFN) general architecture. Reprinted from B. D. Brabandere et.al. [90].

2.4.3 Decision-Making Schemes

There are a number of slightly different decision mechanisms, but the three most commonly used are: *Confidence-based Criteria*, *Policy Networks* and *Gating Functions*

Confidence-based Criteria methods usually require the estimation of the confidence of an intermediate output, which is essentially a metric of the amount of information the backbone network had abstracted from the input sample. This value is then compared to a predefined threshold for decision-making. The confidence can be represented by the highest element of a Softmax function [27], the value of an entropy function [91] or a score margin [80]. Finally, the halting score in [92] [93] could also be viewed as the confidence of the feature to be output in the next time step.

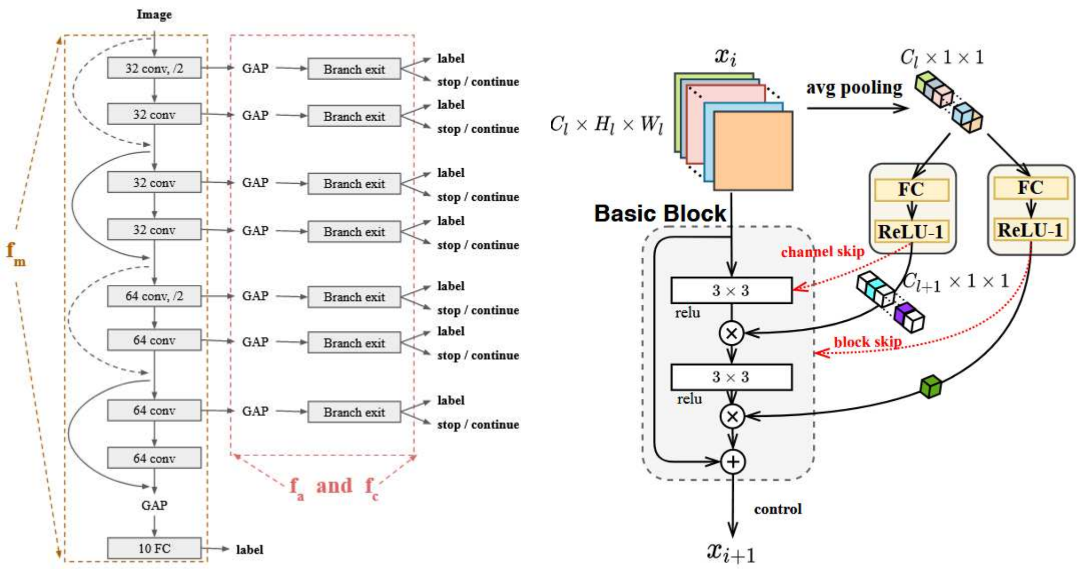
Policy Network is typically developed for a specific backbone network. Each input sample is first processed by the policy network, which is trained to learn a decision function that dictates which parts of the main network are going to be activated. BlockDrop [84] is a great example that uses this type of network to determine the depth of the backbone one.

Gating Functions can be viewed as plug-in modules that go between layers or blocks of vanilla backbone networks. The gating functions take in the input or intermediate samples and produce a binary-valued gate vector that decides which channels need to be activated [89], which layers need to be skipped [94] [95] or which path should be selected in a tree-like SuperNet [96].

2.4.4 Decision-Making Modules Architecture

Considering the architecture of the decision-making modules, there are two different directions followed in literature: the *end-to-end designed*, where these modules are integrated into the original network and the *enhanced backbone networks*, where the original network remains unchanged, and the decision-making modules are added alongside the depth. The first direction leads to an architecture that follows a progressive inference. This means that the neural model and the decision modules are co-designed and trained jointly. Such an approach gives the designer the ability to hand-write and control the entirety of the dynamic network, providing more degrees of freedom in the final architecture. However, it can lead to a considerable decline in training performance and accuracy, as it requires custom training functions that simultaneously train both the backbone and the modules. This also translates into the inability to use any of the existent coding libraries (Tensorflow [42], Pytorch [97], etc.), making the coding process harder.

On the other hand, the second direction suggests the architecture of the backbone network to be completely independent of the decision modules, giving the flexibility to the designer to lazily select their architecture. This translates to case-driven designs that can be trained separately and selected at deployment time [98]. Although it cannot achieve the overall accuracy of the *end-to-end designed*, as the backbone network and the decision-making modules are not co-designed, it is easier to train and deploy and more flexible, making it the most popular approach among conducted research. This is also the approach this research follows.



(A) EPNet Ealry-exit Architecture. Reprinted from X. Dai et.al. [99].

(B) LC-Net layer and channel skipping Architecture. Reprinted from W. Xia et.al. [100].

FIGURE 2.21: Decision-Making Mechanisms' Architecture Examples.

In most cases, decision-making modules are designed as small-sized neural networks, mirroring the structure of conventional DNNs but considerably smaller. They typically consist of convolutional layers, fully-connected (FC) layers, and pooling layers. For example, in BranchyNet [81], the early exit modules incorporate Convolutional and FC layers alongside some activation functions, while on EPNet [99] (Figure 2.21 (A)) and dynaexit [101] the convolutional layers are skipped. This decision to omit convolutional layers is often made to reduce computational complexity, as convolutional layers are among the most resource-intensive components of a neural network. Alternatively, and to mitigate the complete loss of convolution, there are works proposing depth-wise convolution [102], which significantly reduce the number of parameters and operations required. In addition to reducing the size and complexity of the decision -aking modules, some approaches opt for a more compact design by utilizing multiple smaller sub-networks to handle different types of decisions. This strategy allows for greater flexibility and efficiency in managing dynamic inference tasks. For example, LC-Net [100] employs two very small networks—one to decide whether to execute a layer and another to determine whether to use a specific channel (Figure 2.21 (B)).

Finally, the structure of the decision-making modules within dynamic networks can be either uniform or non-uniform. Decision modules organized under a uniform structure exhibit a consistent approach to decision-making. Each module within the network employs the same architecture to make decisions. This uniformity simplifies the design, improves predictability, and makes it easier to train. Networks with uniform modules often benefit from scalability and efficiency, as all components can be optimized in a similar way. However, they might lack the adaptability needed for complex environments.

A non-uniform architecture operates with different rules and processes depending on their role, position, or input data, expanding the design space. This approach allows for specialization within the network, where some modules can focus on different tasks or conditions. Non-uniform structures offer greater flexibility, enabling a network to respond more effectively to diverse or complex environments. However, this approach can introduce challenges in coordination, as varying decision processes may increase complexity in training and optimizing the network. Therefore, the design of these modules must be carefully considered, taking into account the original network architecture and the specific dynamic policy being implemented, to ensure that the benefits of dynamic inference outweigh the potential costs.

2.4.5 Dynamic Network Training

There are primarily two methods for training early-exit networks: i) **end-to-end (E2E)** and ii) **intermediate classifiers (IC) only**. Each method offers distinct trade-offs between achieved accuracy and adaptability for target-specific adjustments. In this discussion, we explore these compromises alongside additional training techniques that can enhance the overall accuracy of the model.

End-to-End training involves simultaneously training the network and its early exits. Typically, a joint loss function is formulated, combining the losses from intermediate and final outputs (denoted as L^i_{task}) in a weighted manner (Equation 2.6), and then propagating these signals back to the corresponding parts of the network. While this approach can potentially yield higher accuracy for both intermediate (y_i¡N) and final exit (y_N), it's not guaranteed due to interference between exits [103]. Specifically, the interaction of multiple backpropagation signals and the relative weighting (w_i) of the loss components [104] must be carefully designed to facilitate the extraction of reusable features across exits. Therefore, while offering greater potential, end-to-end training necessitates manual adjustment of the loss function, and collaborative design of the network architecture and its exits [105].

$$L_{e2e}(y_0, ..., y_N, y) = \sum_{i=0}^{N}(w_i * L^i_{task}(y_i, y) \tag{2.6}$$

IC-only training alternatively, can be accomplished by training the backbone of the network separately from the early exits in two distinct phases. Initially, the backbone network, whether it incorporates early exits or not, is trained or pre-trained. In the subsequent phase, the backbone network is kept frozen, and early exits are added at various points in the network and trained independently (Equation 2.7). This implies that each exit fine-tunes its own layers exclusively without influencing the convergence of the remainder of the network. Consequently, the final exit remains untouched, eliminating inter-classifier communication and the need for manual adjustment of the loss function. This approach allows for the placement of multiple exit variants at different locations within the network, enabling parallel training and scalability while deferring the selection of exit heads until deployment [106]. Consequently, a paradigm of "train-once, deploy-everywhere" is established for multi-device deployment. However, this training method imposes greater constraints on overall model modifications, potentially resulting in lower accuracy than a jointly trained and optimized variant.

$$L^i_{ic-only}(y_i, y) = L^i_{task}(y_i, y) \tag{2.7}$$

2.5 Dynamic Hardware Architectures for DNNs

Dynamic hardware approaches focus on adding a dynamic aspect to the architecture of the hardware platform in which the inference of a neural network is accelerated. Most of the published work utilises FPGAs to implement their designs and systolic arrays of process elements for the majority of the computations, mostly because of the reconfigurability potential they both offer. The core idea is to reconfigure the accelerator's different components in order to target the layers' computational needs. This leads to higher utilisation levels of the available hardware and, consequently, faster execution.

2.5.1 Diversity in Layers

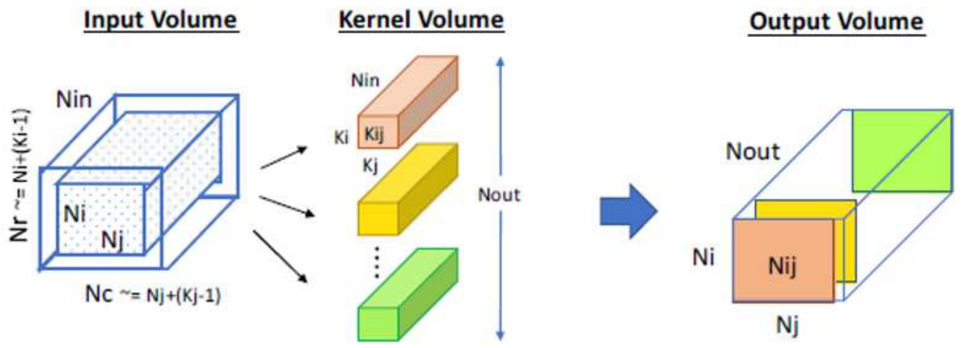


FIGURE 2.22: Parameters of DNN layers. Reprinted from M. Putic et.al. [34].

Figure 2.22 captures the main type of computations performed in the layers of a CNN. N_{in} is the number of input feature maps, N_{out} is the number of output feature maps, $N_i \times N_j(N_{ij})$ is the number of rows and columns in each feature map and $K_i \times K_j(K_{ij})$ are the dimensions of each kernel filter. In theory, two similar layers performing the same type of operation, e.g. 2D convolution, should have similar performance attributes. Putic et al. [34] (Table 2.1), however, highlights that both their data-structure shape and computational demands vary significantly. It is easily visible that, for convolutional layers, as the network gets deeper, the input and output features (N_{in} and N_{out}) increase by over two orders of magnitude while N_{ij} decreases proportionately. FC layers, on the other hand, are dominated by weights, while significant variations are noted on the kernel sizes, too, alternating between 3×3 and 1×1. This variety is depicted on the Ops/Byte requirement, too, where it differs by over an order of magnitude for Convolutional layers and two orders of magnitude overall.

	Parameters				Compute Characteristic		
Layer	N_{in}	N_{out}	N_{ij}	K_{ij}	Ops (M)	Bytes (M)	Ops/Byte
CONV$_1$	3	64	224×224	3×3	173.41	6.73	25.78
CONV$_2$	64	128	112×112	3×3	1849.69	4.96	372.59
CONV$_{3,1}$	128	256	56×56	3×3	1849.69	3.00	616.92
CONV$_{3,2}$	256	256	56×56	3×3	3699.38	4.39	842.51
CONV$_{4,1}$	256	512	28×28	3×3	1849.69	3.56	519.06
CONV$_{4,2}$	512	512	28×28	3×3	3699.38	6.32	584.95
CONV$_{5,1}$	512	512	14×14	3×3	924.84	5.12	180.63
CONV$_{5,2}$	512	512	14×14	3×3	924.84	5.12	180.63
FC$_6$	25088	4096	1×1	1×1	205.52	205.58	1.00
FC$_7$	4096	4096	1×1	1×1	33.55	33.57	1.00
FC$_8$	4096	1000	1×1	1×1	8.19	8.20	1.00

TABLE 2.1: Dimensions and compute characteristics of the layers of VGG11 image recognition DNN. Reprinted from M. Putic et.al. [34].

Creating a general static model, even though it can accelerate the execution, does not translate to the acceleration of every layer's execution. For example, when a layer has high data reuse (e.g. $Ops/ByteDNN > Ops/Byteacc$) can lead to under-provisioning the compute components, and resources should be shifted to them from, e.g. memory bandwidth. On the contrary, when a layer has high number of input and output feature maps, which means low data-reuse (e.g. $Ops/ByteDNN < Ops/Byteacc$) compute components are underutilized and resources should be shifted from them to data-movement, as data is not fed in sufficient rate. Moreover, these compute components are multi-dimensional parallel execution engines, so if the hardware dimensions are not an even multiple of the weights, inputs and outputs feature maps' dimensions, a significant utilization drop occurs. This means that by leveraging the layers' attributes, a lot more efficient resource distribution and memory scheduling can be achieved [34].

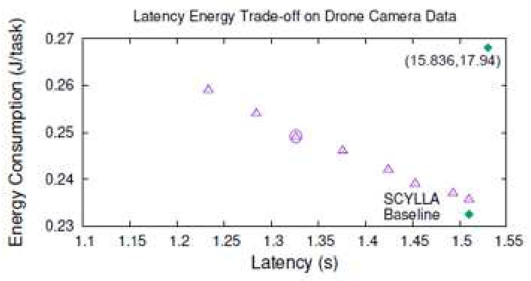
2.5.2 Hardware Design and Deployment

The information collected from analyzing the layers of a network, along with the available resources and the reconfiguration cost of the target device, can be used to explore the design space and find the most efficient architecture. Nonetheless, for this to happen, the information concerning the DNN's structure must be translated into hardware relative values such as power consumption, latency cost, etc. To evaluate performance effects, Putic et al. [34] propose a cycle-accurate hardware equivalent model of a systolic array, which is the reconfiguration target component, written in Python [107]. They simulated the MAC operations and the on- and off-chip memory data movement costs extracted from the network information in order to create different performance scenarios for each layer. Moreover, they implement and synthesize a simple design of a systolic array in RTL and use Cadence Genus [108] and ARM Memory Compiler [109] to estimate the power needs.

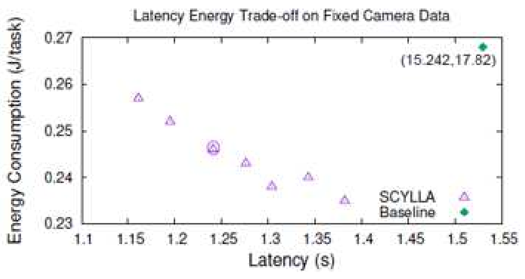
F. Kastner et al. [110] and S. Jiang et al. [111] follow a simpler approach, where taking into consideration the layer's features, they created multiple architectures with different levels of parallelization, latency and accuracy. To achieve that, they use different combinations of three major optimization parameters: the factor of unrolled loops, the pipeline interval (the number of clock cycles the systolic array needs to be available to accept a new input) and the bit-width for the quantization of the model's parameters.

However, to enable the dynamic deployment of hardware designs, dynamic partial reconfiguration is necessary. When the available resources are not static, for example, on a mobile device where a number of applications run in the background, the main architecture and the reconfigurable components must adapt to the real-time demands. S. Jiang et al. [111] formulated two functions to select the best architecture given the available resources and a preferable trade-off between latency, accuracy and energy

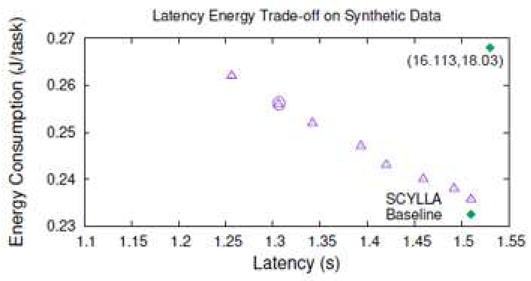
(Figure 2.23). More specifically, to calculate latency, they consider the time for the execution of the model on the FPGA, the time for its reconfiguration and the time that it must wait for available engines. Then, they pass this information to a second function, which combines the time needs with the accuracy and energy estimation of every architecture and chooses the best one for the current resource availability. In doing so, they managed to meet different real-time resource limitations while maintaining a very high quality of experience on a mobile device.



(A) Drone Data



(B) Fixed Camera Data



(C) Synthetic Data

FIGURE 2.23: Generated architectures providing different trade-offs between Energy Consumption and Latency. Reprinted from S. Jiang et.al. [111].

Contrarily, when the available resources are static, deployment methodology falls into finding the balance between reconfiguration cost and gained acceleration by the change of the design for each layer. Putic et al. [34] proposes an algorithm where the inputs, the architecture, and the reconfiguration cost are utilised to produce a configuration map (Figure 2.24). More specifically, using the performance model explained above, it explores and identifies the best design for every layer. A first network description with the best design for every layer is produced. However, reconfiguring the FPGA after the execution of each layer is very time-consuming and leads to an overall increase in latency. To resolve that, it performs between every consecutive layer to determine when the performance gain of reconfiguration is higher than its cost and only then applies it. This leads to a refined network that showed 15-65% latency and 25-90% energy improvement over static designs on 6 popular DNN models (ResNet50, GoogleNet, VGG11, etc.).

Y. Yang et al. [112] propose an approach that focuses on clustering the layers. They find the fastest one and create an empty set. Then, for every other layer, compare if the

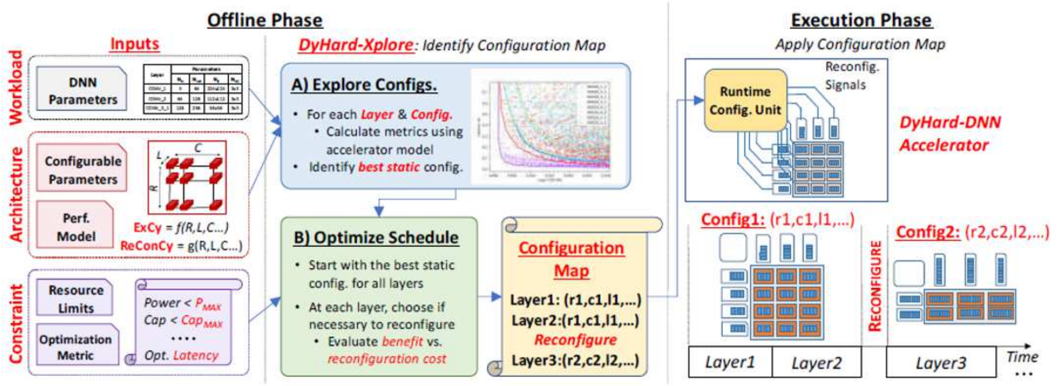


FIGURE 2.24: DyHard-DNN methodology. Reprinted from B. D. Brabandere et.al. [90].

acceleration difference between them and the fastest is greater than the reconfiguration cost. If it is then they create a new set of layers, otherwise they add it to the existing one. They repeat this process for all the layers, and they end up with a number of layer sets assigned with the design of the initial layer. Finally, in order to create the configuration map, when there are two neighbouring layers of different sets, and consequently with different assigned architectures, they recheck if the performance gain is greater than the reconfiguration cost and choose accordingly to change or not the design. Overall, they achieved a 2.11× performance improvement in the execution of AlexNet, compared with a static FPGA design.

2.6 Research Scoping and Discussion

While DNNs are getting deeper and more complex in adapting to the needs of modern AI applications, they are becoming increasingly demanding for memory and computation. CPUs are unable to accommodate large networks, especially when low latency is of high importance, leading to the widespread adoption of GPUs as the main hardware device to execute AI applications.

With the increasing interest in moving the execution of DNNs to the edge, as elaborated in Chapter 1, a lot of effort has been put into developing devices that maintain low levels of energy consumption and accelerate the execution of DNNs, like NPUs and TPUs. FPGAs are another platform that has been proven to be very effective in executing neural networks. Hardware-focused dynamic approaches further highlighted their effectiveness. Focusing on the layers' varying functions and computational needs and targeting the FPGA's ability to partially reconfigure their architecture on runtime, they propose the creation of multiple partial designs, mostly of a systolic array, to better adapt to each layer's unique characteristics. Doing so, there are approaches that manage to achieve up to 65% reduction in latency and up to 90% in energy.

However, FPGAs is a platform with a limited amount of resources, rendering the need for network compression mandatory. State-of-the-art compression techniques were presented in this Chapter, too. Static methods that permanently affect the DNNs, like quantisation or pruning, reduce the computation needs and memory footprint but result in losses in accuracy and representation power. On the other hand, dynamic approaches, following the observation that each input interacts differently with each component of a neural model, provide a dynamic aspect to neural networks by devising input-driven control mechanisms. Targeting either the network's structures or its parameters significantly increases the efficiency and adaptability of many modern deep neural models (e.g. VGG11, ResNet, GoogleNet, et al.). For example, by dynamically selecting which of the layers are going to be executed, E. Park et al. [94] reduced the needed energy by 80% on the MNIST dataset and over 40% on ImagNet compared to a static network, while F. Yu et al. [87] applying adaptive kernel sizes achieved a 40% reduction in FLOPs on a VGG16 network. These results are very promising, especially for deploying models with high computational burden on devices with limited resources.

Through the examination of the existing work, it was visible that the dynamic DNN approaches offer great benefits in reducing the computational and energy demands of modern neural networks. They also managed to achieve that with minimal to none negative effects on the accuracy of the networks. However applying these types of networks on conventional processing systems, such as computer processors and GPUs does not allow to fully highlight their capabilities, as despite their benefits, the results, especially concerning energy, does not depict a realistic application. FPGAs on the contrary, have already proven to be a very good platform to optimise and accelerate the execution of modern DNNs, while providing a low power environment. Building on the prior findings presented in this Section, a hypothesis was formulated that FPGA-specific optimizations, such as parallel execution and quantization, can further enhance the performance while improving energy efficiency of early-exit dynamic DNNs.

Chapter 3

Realising Dynamic DNNs on FPGAs

Chapter 2 showcased how the different Dynamic DNN approaches can enable a data-driven execution that adapts to the characteristics of each input sample and efficiently compresses a neural network without compromising accuracy or representation power. To achieve this, however, dynamic approaches introduce new components to the network's architecture. For example, early-exit networks deploy intermediate classifiers to enable the classification of an input at earlier stages of the inference. Layer-skipping networks incorporate gating functions to dynamically bypass certain layers of the backbone network, and weight prediction methods use filter-generating networks to dynamically predict weights adapted to each input sample's attributes. This substantially differentiates their structure from static networks, creating more adaptable but complex architectures.

Despite their very promising results, their deployment has been mostly explored in powerful GPU-CPU systems. While this is an environment very capable of highlighting their beneficial impact on the execution of DNNs, it is argued that there is a gap in knowledge between theoretical and practical results. This Chapter aims to explore the realisation of dynamic DNNs on FPGAs. FPGAs are popular for their ability to provide high-performance and low-power implementations of custom hardware designs, rendering them ideal to illustrate the capabilities of dynamic network approaches in resource-constrained environments.

The dynamic DNNs' architectural complexity, however, poses significant challenges to their deployment and hardware realization. Unlike static networks, which have a fixed structure, rendering them easier to optimize for various hardware platforms, dynamic networks introduce variability that can be difficult to manage. The new components require additional computational overhead and more sophisticated control logic. In addition to this, the reliance on software and hardware libraries designed for static networks complicates their implementation, as these tools are not optimized for dynamic operations. These challenges become even more pronounced when considering the deployment of dynamic networks on FPGAs since the dynamic reconfiguration of the network components at runtime, as required by dynamic DNNs, is not inherently supported by FPGAs.

In this Chapter, the challenges of current software and hardware infrastructure in the support of dynamic networks are outlined. Information on the limitations introduced by the characteristics of the dynamic approach is very important for the hardware designing process, and it can also highlight potential directions for future research. The state-of-art advancements towards solving some of these challenges are also illustrated and discussed. Finally, the first design approach to realising dynamic DNNs on FPGAs is proposed. The aim of the proposed architecture is not only to investigate the feasibility of the FPGA platform to deploy dynamic DNNs but also to highlight their advantages in a resource-restricted environment.

The novel contributions of this Chapter can be summarised as:

- **An exploration of the challenges the early-exit networks architecture introduces to their hardware implementation.** Identifying the limitations presented by the software and hardware infrastructure and the architectural characteristics of early-exit dynamic DNNs is of vital importance for their hardware realisation process and highlights open research problems.
- **A first consideration for realising and implementing dynamic DNNs on FPGAs (at the time this research was conducted)**. A first FPGA implementation of an early-exit network is presented.
- **An experimental evaluation of the proposed Early-Exit FPGA design**. The design is implemented on a ZCU106 FPGA, and its performance is evaluated over accuracy, execution time, power and energy consumption against a desktop CPU, CPU+GPU, and an Nvidia Jetson Xavier embedded device.

3.1 Challenges

Following Section 2.4.1, the two major components of an early-exit DNN are the backbone network and the intermediate classifiers (Figure 3.1). The backbone network (orange part of Figure 3.1) is the static DNN, unchanged and used as the main processing route for the input samples. Input samples, translated into matrices, are inference through the first layer of the network. The output feature maps are than forwarded to the intermediate classifiers (doted squares in Figure 3.1), which are the decision modules whose purpose is to enable the dynamic execution of the network and, consequently, to reduce the computational burden it introduces. The convolution and the FC

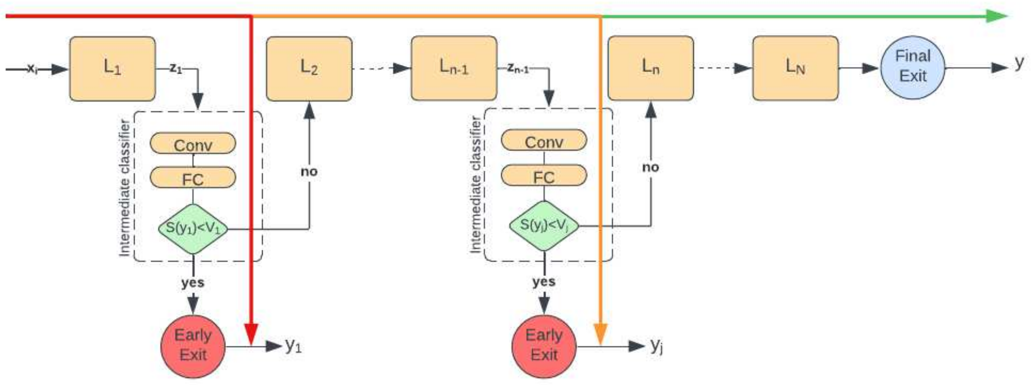


FIGURE 3.1: Early-Exit Dynamic DNN architecture.

layer are used to reduce the dimensions of the feature maps and flatten them into a vector, which size is dictated by the amount of the classification classes ($S(y_i)$). This vector is then compared with vector V_j. If the early-exit condition is met then inference stops (red line in Figure 3.1), other it continues to the execution of the backbone network till the next intermediate classifier. A check for an early exit happens again (orange line in Figure 3.1). If an early exit does not occur inference stops producing an output from the backbone network (green line in Figure 3.1).

Despite being simpler structures (see Section 2.4.4), even over small DNNs, e.g. LeNet [113], intermediate classifiers introduce significant complexity when integrated into static network architectures. They require careful training and cause alternation in the data-paths and execution of the DNNs, leading to uncharted territories both in terms of their development and their deployment.

3.1.1 Software Frameworks & Training

Training is the first step towards the deployment of a neural network. It is one of the most time-consuming and computationally heavy operations of DNNs due to the complexity of the backpropagation algorithm and the vast number of parameters modern networks require. Hence, a lot of effort is spent to simplify and accelerate this process. Frameworks like TensorFlow [42], PyTorch [97], and JAX [114] offer automatic differentiation, which automatically computes gradients needed for backpropagation. This eliminates the need for manual gradient calculations, making it easier to implement and optimize complex models. High-level APIs like Keras [115] (integrated into TensorFlow) provide intuitive interfaces for building and training models and libraries of common layers, loss functions, optimisers, etc. Thus they abstract the complexity of lower-level operations, allowing users to define and train models with minimal amount of coding. The latest advancements even support distributed training across multiple

GPUs or even clusters of machines, allowing faster training times through data parallelism (distributing different batches of data across multiple devices) and model parallelism (splitting the model across devices).

However, these frameworks have very limited, if not zero, support for dynamic DNNs. Conversely, the architectural differences of dynamic networks require more complex training methodologies in order to achieve high accuracy on a given task. As seen in Section 2.4.5, the majority of works concerning early-exiting employ either an **End-to-end** strategy, which jointly trains the backbone and intermediate classifiers with combined loss functions, or **IC-only training** in which, only the intermediate classifiers have their parameters updated during training, while the backbone parameters are frozen.

Both of these approaches require custom loss functions adapted to the structure of the early-exit network and the decision-making approach of the intermediate classifiers'. While the core of the loss function is the same as those used for static networks, the need to adapt to early-exiting necessitates creating new or altering the existing libraries. This renders the coding process considerably harder, as most of the predefined components and functions offered by DNN frameworks are infeasible to support dynamic networks. Also, their dependency on the structure of the early-exit approach makes the modelisation of training that would enable techniques like distribution training, automated tuning (e.g. Optuna [116], Ray Tune [117] etc.) very hard if not impossible.

This extends to the deployment of dynamic networks, too. A great example is Tensorflow, as it freezes the execution graph for inference, meaning that the architecture and model's parameters become constants and cannot be further trained or modified. This greatly limits the creation and deployment of models that require the run-time modification of their structure. Furthermore, dynamic architectures have a data-dependent inference procedure, which usually requires a model to handle input samples sequentially. This poses a major challenge for the parallel processing of batched input samples [118], and similarly to training, the deployment to distributed systems.

3.1.2 Hardware Design & Frameworks

Targeting custom hardware devices, especially for the deployment of complex applications like DNNs, is a difficult and time-consuming task. Likewise, to software development, numerous FPGA manufacturers and researchers have invested resources in developing mapping frameworks for DNNs. For example, Xilinx's Vitis AI [119] and Intel's OpenVINO [120] development platforms provide tools for optimizing and deploying AI models on FPGAs, including libraries for accelerating DNN inference, tools for quantization and pruning and emulators allowing the testing of DNN models before full deployment. Techniques like NAS are adapted to consider FPGA constraints during the design of network architectures, helping with design models that are not only accurate but also optimized for FPGA performance in terms of resource utilization and latency. Finally, frameworks like AngelEye and fpgaConvNet [67] automate the deployment of DNNs incorporating acceleration techniques like loop unrolling, different quantisation levels and dynamic partial reconfiguration.

However, these tools are integrated with static network libraries like Caffe, TensorFlow, Torch, etc. In other words, the limitations presented above apply to the use of FPGA-focused tool-flows, resulting in limited support in capturing and mapping the dynamic behaviour. It could be argued that, at a minimum, they could be used for the design of the backbone network. However, most of these tools generate comprehensive but often inaccessible designs. For example, Figure A.1 shows the design output of fpgaConvNet for LeNet-5. They are highly optimised architectures, but accessing and altering them in order to incorporate the intermediate classifiers. is either a very hard or impossible task.

Overall, due to the control and data-flow nature of Dynamic DNNs, mapping the architecture requires the use of custom layer types, components, and control modules, which dramatically increases the complexity of the hardware design task. Designers need to carefully balance the standard trade-offs between computational efficiency, latency, and resource utilization while ensuring proper synchronization and data flow across different inference paths.

3.1.3 Intermediate Feature Map Dependence

Focusing on early-exit dynamic networks and looking at Figure 3.1, it is notable that the intermediate classifiers use the previous layer's output to reach a decision. These intermediate feature maps are required by both the intermediate classification layers or, in cases when an early-exit does not occur, the remainder of the backbone network. Propagating the intermediate feature map through the early-exit branch destroys them, which raises the need to preserve them. In CPU/GPU systems, it is straightforward as they can be stored temporarily in the memory. However, when resource-restricted devices, like FPGAs, are targeted, preserving the intermediate feature maps can be challenging. A method to follow is buffering them on-chip, but due to the increased memory requirements in larger networks (e.g. VGG, ResNet, etc.), this is not always a viable solution. An alternative is routing them to external memory, which, although, can significantly impact the latency and energy demands of a given inference.

3.2 Advancements

One of the first attempts to overcome the challenges concerning hardware design frameworks 3.1.2 is the framework visualized in Figure 3.2. ATHEENA [121] is a framework that constructs a deeply pipelined, early-exit network accelerator to perform inference on an FPGA. It investigates the resource/throughput trade-off that can be made when factoring the probabilistic nature of the early-exit network and utilises the open-source fpgaConvNet [67] as a baseline to automate the mapping process.

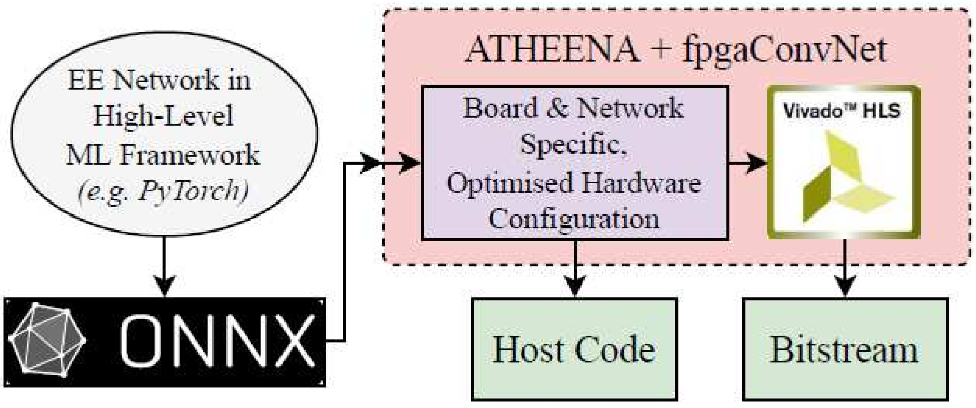

ATHEENA + fpgaConvNet
EE Network in High-Level ML Framework (e.g. PyTorch)
Board & Network Specific, Optimised Hardware Configuration
Vivado™ HLS
ONNX
Host Code
Bitstream

FIGURE 3.2: High-level overview of the components of the ATHEENA framework. Reprinted from B. Biggs et.al. [121].

Utilising fpgaConvNet, ATHEENA maps the early-exit network onto the FPGA and optimises the level of parallel computation to maximise throughput under the resource constraints of a given device. The primary distinction between the two flows is that while the original fpgaConvNet generated a data-flow graph, this approach creates a control and data-flow graph (CDFG). This CDFG not only represents the flow of data through the network layers but also incorporates the confidence decisions made at the early-exit points. To accommodate different ONNX operations and incorporate control flow into hardware translation, the parser and the optimizer are modified. Additionally, several new hardware component templates have been created to implement control flow and support confidence calculations on the FPGA. Figure A.2 is the enhanced processing flow of ATHEENA framework, corresponding to Figure 2.10 presented in Section 2.2.4.3.

AdaPEx [122] is a related FPGA work that explores dynamic network inference on FPGAs, making use of adaptive pruning and early-exit intermediate classifiers to further explore the trade-off between resources and throughput at run-time. AdaPEx is built on the FINN [71] HLS framework, which is used to construct a library of FPGA accelerators for a given traditional DNN. To achieve that, they utilised the dataflow accelerator, creating custom hardware module libraries tailored to the needs of early-exit networks. Each accelerator has a different level of channel-wise pruning, with reduced

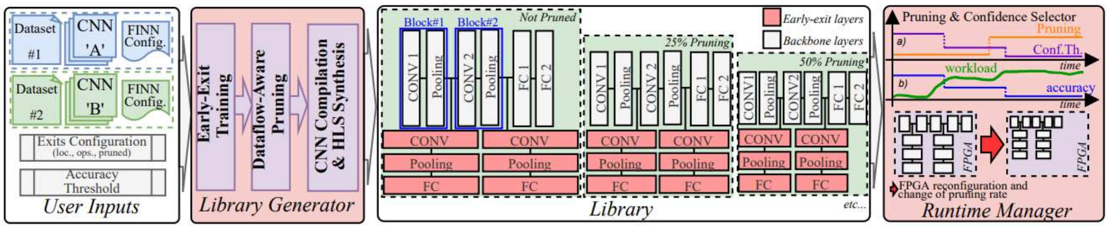


FIGURE 3.3: AdaPEx workflow. Reprinted from G. Korol et.al. [122].

accuracy corresponding to improved accelerator performance. A runtime manager is deployed to orchestrate the dynamic partial reconfiguration of the FPGA, deploying the generated accelerators based on the user requirements for accuracy and throughput. Figure 3.3

Both approaches target to overcome the aforementioned challenges and help towards making the deployment of early-exit dynamic DNNs of FPGAs easier and more efficient. Utilising and adapting frameworks like FINN or fpgaConvNet aims to make the FPGA designing process significantly easier and faster. However, as highlighted above they are designed focusing on static networks, which results in their output designs being restricted by the available libraries and frameworks limitations. For example, ATHEENA employing fgpaConvNet leads to architectures that are accelerated by function (as seen in Section 2.2.4.3) that excel at static data paths. Additionally, the limitations of the framework are highlighted by a very limited number of explored DNNs to date. Correspondingly while AdaPEx propose a dynamic hardware/software approach, combining dynamic pruning, early-exiting and FPGA reconfiguration, the developed architecture's performance is explored only on a simulation level.

3.3 Early-Exit Network FPGA Design

A different approach and the one followed in this research, is designing everything from scratch using a hardware description language (VHDL, Verilog, etc.). This technique requires considerably more time but can lead to hardware architecture with utmost efficiency and effectiveness. A first attempt to realise and fully deploy early-exit dynamic DNNs on FPGAs is presented below.

3.3.1 Early-Exiting Approach

This work employs the approach presented by Teerapittayanon et al. [81]. The network follows the early-exit architecture presented in Section 2.4.4, deploying intermediate classifiers within the backbone network in order to enable the dynamic execution of the DNN. The intermediate classifiers are trained alongside the backbone network as

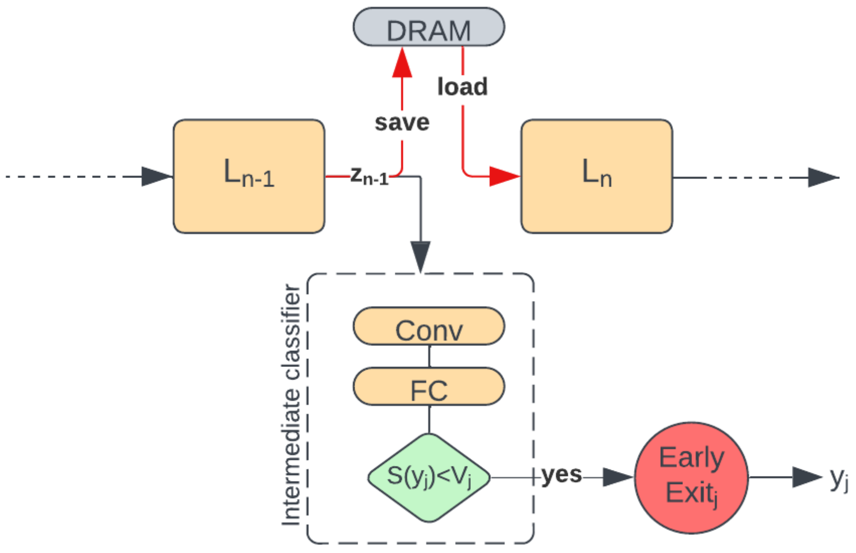


FIGURE 3.4: The Early-Exit Dynamic DNN FPGA Design Approach.

a joint optimisation problem, using the softmax cross entropy loss function. Once the network is trained, entropy is used to measure how confident the network is at an exit point. Entropy can be written as:

$$S(y) = \sum_{c \in C} y_c \cdot \log y_c \tag{3.1}$$

where y is the intermediate classifier's output that contains the probabilities for every one of the C possible labels. Entropy is then compared over a vector V, which contains the thresholds for every exit point. For example, in the *j-th* exit point after the L_n layer, entropy is compared with V_j. If it is smaller, the inference halts, and the branch's predicted class is selected. Otherwise, inference continues with the execution of the L_{n+1} layer.

3.3.2 Architecture

The proposed design approach (Figure 3.4) follows the traditional early-exit architecture. The backbone network is executed normally until the layer before the intermediate classifier L_{n-1}. The backbone network then halts, and the L_{n-1} layer's output is fed to the intermediate classifier, which starts executing. Its output is then compared with a confidence value, and the dynamic network either stops ($S(y_j) < V_j$) and requests a next input sample, or ($S(y_j) \geq V_j$) inference continues with the next layer L_n.

This architecture offers the possibility of reusing the hardware modules designed for the backbone network. Following the analysis of the structure of the intermediate classifiers in Section 2.4.4, their components (layers, activation functions, etc) are similar to the backbone network. With some careful setting of the control module on the FPGA design, reusing the same hardware modules can lead to an additional area overhead of close to 0% for the execution of the intermediate classifiers. This can cause the underutilization of the existing modules, but it can be proven to be very useful for cases with very restricted amounts of resources.

3.3.3 Hardware Implementation

To further investigate the feasibility and benefits of dynamic DNNs on FPGAs, a hardware implementation is developed. It is essential first to convert real numbers like inputs, weights, and biases into a digital format. Implementing FP32 arithmetic units in an FPGA requires significant resources, as floating-point operations are complex and consume a large number of logic elements (LUTs, flip-flops), DSP slices, and memory. In scenarios where FPGA resources are limited, fixed-point representation is utilised. An 8-bit fixed-point representation is used, as it minimizes the computational demands on hardware resources while maintaining the same levels of accuracy [123]. As illustrated in Figure 3.5, this representation comprises three components: a sign bit (S), a two-bit integer (I), and a five-bit fractional value (R) (illustrating a fixed-point representation for the number 1.375).

Sign (1 bit)	Integer (2 bits)		Real (5 bits)				
0	0	1	0	1	1	0	0
	2^1	2^0	2^{-1}	2^{-2}	2^{-3}	2^{-4}	2^{-5}
S=0	I=1		$R=(2^{-2}+2^{-3}) = 0.375$				

FIGURE 3.5: Example of fixed-point representation of the real number 1.375.

An array of processing elements is employed to accelerate matrix-vector multiplications. Systolic arrays are hardware structures that replace a pipeline structure with an array of homogeneous processing elements (PEs) that can perform a common mathematical operation. These elements are locally interconnected and able to communicate with each other in a synchronous manner. An architecture like this is very beneficial as it can naturally facilitate data reuse during matrix/tensor algebra operations. A significant portion of PEs in the array can perform their tasks without requiring communication with external memory. Consequently, a maximum of p I/O ports has the capacity to drive p^2 PE units, which substantially reduces memory bandwidth demands.

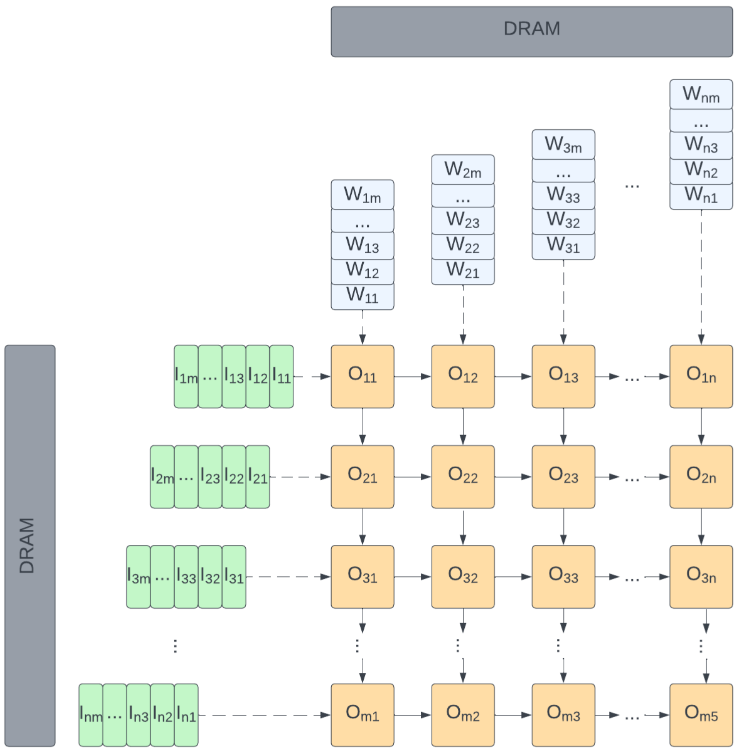

DRAM
DRAM
W_{1m} … W_{13} W_{12} W_{11}
W_{2m} … W_{23} W_{22} W_{21}
W_{3m} … W_{33} W_{32} W_{31}
W_{nm} … W_{n3} W_{n2} W_{n1}
I_{1m} … I_{13} I_{12} I_{11}
I_{2m} … I_{23} I_{22} I_{21}
I_{3m} … I_{33} I_{32} I_{31}
I_{nm} … I_{n3} I_{n2} I_{n1}
O_{11} O_{12} O_{13} … O_{1n}
O_{21} O_{22} O_{23} … O_{2n}
O_{31} O_{32} O_{33} … O_{3n}
O_{m1} O_{m2} O_{m3} … O_{m5}

FIGURE 3.6: The design of the systolic array following an Output Stationary data-flow.

Figure 3.6 shows a typical 2D systolic array configuration, containing a collection of identical and interconnected PEs, accompanied by a controller module and an on-board memory/buffer. Each PE (Figure 3.7) is constructed from fundamental arithmetic and register components, enabling it to execute multiply-accumulate operations. The data flows from the DRAM into the PEs (weights through regW and features through regI) and is multiplied and accumulated in the MAC module. Partial sums are calculated using bufc, and when all iterations are completed, the results are stored back in the memory.

Algorithm 1 3-nested loop of fully-connected (FC) layer

```
Input: $I[D, C]$ and $W[E, C]$
Output: $O[D, G]$
1: for ($d = 0; d < D; d++$) do
2:     for ($g = 0; g < G; g++$) do
3:         for ($c = 0; c < C; C++$) do
4:             $O[d, g] = O[d, g] + W[g, c] \cdot I[d, c]$
```

To improve the performance of the systolic array, it is necessary to consider how to explore the parallelism of the workload to drive multiple PEs to work simultaneously. A simple way is loop unrolling [124]. Algorithm 1, for example, describes the calculation process of an FC Layer, where d denotes the size of the mini-batch, c signifies the cth input and g represents the gth output. loop1 and loop2 can be flattened to the spatial

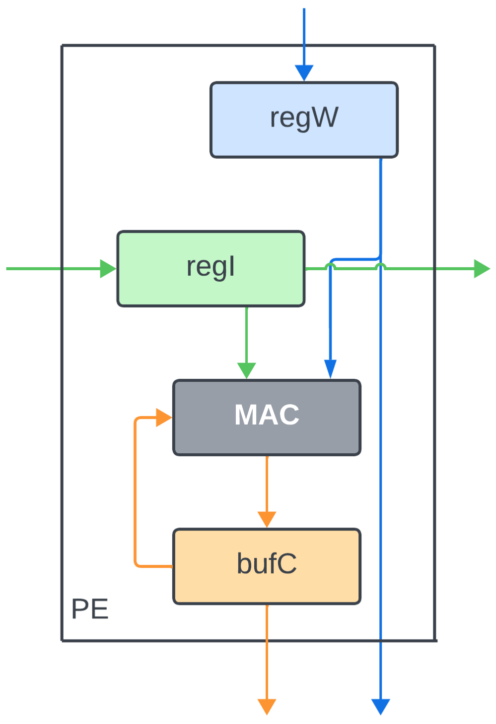


FIGURE 3.7: The design of the processing element on our FPGA architectures.

hardware and compute these loop instances in parallel. For a more detailed description, data flow can be used to represent which loops in the workload are unrolling. The Output Stationary (OS) [125] approach is followed, which refers to mapping output elements to the corresponding PEs, where each PE needs to complete all the computations required for an output element.

The target platform is the Zynq UltraScale+ MPSoC ZCU106 Evaluation Kit, including 504K logic cells, 1728 digital signal processing (DSP) slices, 38Mb of Block RAM and one 4Gb DDR4 module. Table 3.1 highlights the resource utilisation of the FPGA.

TABLE 3.1: FPGA Resource Utilisation

Data	**#PEs**	**FF**	**LUT**	**BRAM**	**DSP**	**Clock**
8-bit	300	50%	66%	96%	78%	150 Mhz

3.3.4 Layer realisation

3.3.4.1 Convolution Layer

For the implementation of convolution, input data is fed from the buffers into the Systolic Array, as explained above. The PEs calculate input-kernel multiplication, and outputs are then moved to another buffer. The activation function (ReLU), a simple

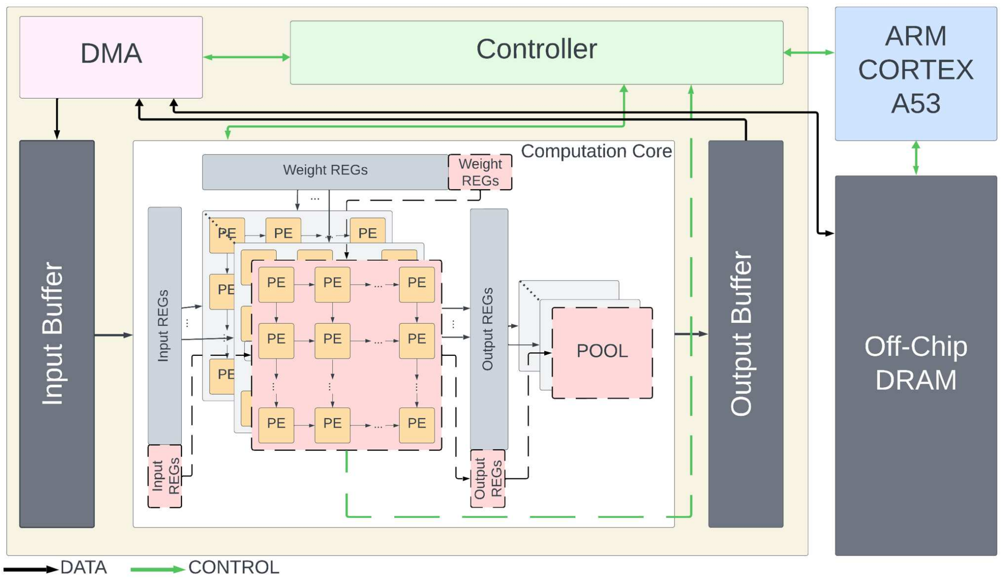


FIGURE 3.8: Design Overview.

conditional branch, is promptly applied, and its results are stored in buffers that feed the subsequent layers.

3.3.4.2 FC Layer

The architecture of fully connected (FC) layers is akin to that of convolutional layers, but instead of using convolution, they rely on vector-matrix multiplication. The input vectors consist of numerous values generated by flattening the output of the previous layer, resulting in many multiplication operations. To handle this, the inputs and weights are divided into equal segments and computed separately.

3.3.4.3 Pooling Layer

Pooling layers take the values stored in buffers from the preceding layer and apply a sliding window with a size equal to the pooling filter and a step size determined by the specified stride value. The operation performed is either max or average pooling.

3.3.5 Design Overview

Fig. 3.8 provides an overview of the design of the proposed hardware implementation. The core of the design is the 2D systolic array configuration accompanied by onboard memory/registers. The data flows from the Input Buffer into the PEs (weights through

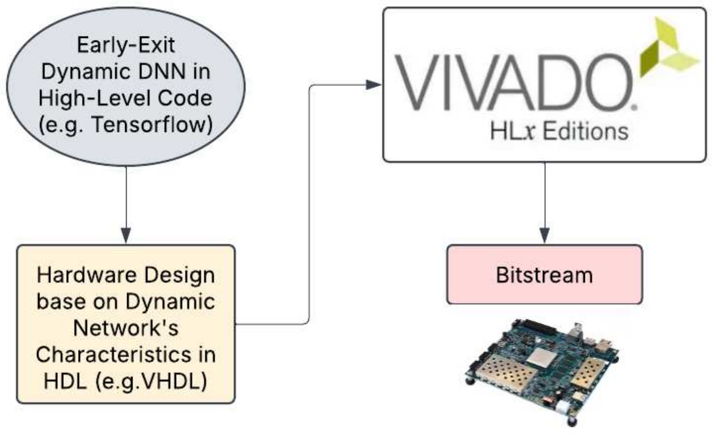

Early-Exit Dynamic DNN in High-Level Code (e.g. Tensorflow)
Hardware Design base on Dynamic Network's Characteristics in HDL (e.g.VHDL)
VIVADO HLx Editions
Bitstream

FIGURE 3.9: Tooling Flow Diagram.

regW and features through regI) and is multiplied and accumulated in the MAC module. Partial sums are calculated using bufc, and when all iterations are completed, the results are stored in the Output Buffer.

The rest of the design contains a controller module, a DMA (Direct Memory Access) module, a host processor and an Off-Chip DRAM. The controller ensures the synchronisation of the different processing units and coordinates the sequence of the dynamic DNN layer execution. It also communicates with the host processor, which, for most networks, is used to set-up the FPGA and post-process its results. In the case of ResNet-32, it is used to execute the model, too. Finally, the DMA is responsible for the efficient data movement between the accelerator buffers and the Off-Chip DRAM. As the data have an 8-bit fixed point representation, they are stored in the 32-bit per address memory in groups of 4, so the DMA is mandatory for the correct use of the external memory-fetched data. Throughout this thesis the energy consumption of the memory data move, in theoretical measurements, is the standard for LPDDR4 memory 4 pJ/bit. [126].

Fig. 3.9 finally highlights the tooling flow for the realisation of each Early-Exit network. In more detail, the network is first designed using an ML Framework, e.g. Tensorflow, PyTorch etc. Then the FPGA hardware design, described in VHDL, is adapted to the characteristics of the dynamic DNN. Vivado Design Suit is then responsible for the synthesise and implementation of the design. Finally, the generated bitstream is downloaded and applied to the FPGA, where inference is executed.

3.3.6 Experimental Setup

To verify, test and evaluate the performance of the design, the BranchyNet [81] early-exit LeNet-5 network is used. It consists of three convolutional and two FC layers, while the one deployed intermediate classifier contains one convolutional and one FC layer, and it is placed after the first convolutional layer of the backbone network. Additionally, at all exit points, the softmax function is applied to classify the input samples (y in Equation 3.1. MNIST [127] data-set is used, containing 60,000 training images and 10,000 testing images, each of which is a 28x28 pixel gray-scale image of a handwritten digit (numbers 0 to 9). The dynamic network, apart from the FPGA accelerator, is executed on a desktop computer (Intel Core i9 + Nvidia RTX 2080 TI) and on an embedded platform (Nvidia Jetson Xavier NX).

To calculate latency and power consumption of the proposed design on the FPGA, the Xilinx Vivado design suit [128] is used. In more detail, in order to acquire precise measurements of the execution time the dynamic network requires to process an input sample, an Integrated Logic Analyzer (ILA) is deployed on the design. It is a powerful debugging tool within the Design Suite that allows designers to monitor and capture real-time internal signals of an FPGA design while it is operating. By monitoring specifically selected or generated signals and with the knowledge of the design's clock frequency, the actual latency of the network can be measured with very high precision. Similarly, Vivado's Power Estimation tool is used to calculate the power needs of the design. It can estimate the power of the FPGA in various design stages, achieving the highest accuracy after implementation. It uses a Simulation Activity file (SAIF) that represents the real-world operation of the design. The file contains information with respect to the environmental conditions (device temperature, ambient temperature, airflow, etc.), the various voltages for the device's power rails, the clock constraints and the default activity rates (set and reset signal probability, IO toggle rates etc.). Utilising this information, the tool can analyse the characteristics of the design and produce a very accurate estimation of the power required by the FPGA.

This process is simpler for the Jetson Xavier NX embedded device. The execution time was measured by simply using Python's time library. For power measurements, the Tegrastat tool provided by nVidia has been adopted. It is a command-line utility used to monitor system resource usage on NVIDIA Jetson platforms, which are powered by NVIDIA's Tegra system-on-chip (SoC). This tool provides real-time information on various system resources such as CPU, GPU, memory, power consumption, et al.

3.3.7 Design Validation and Evaluation

Figure 3.10 shows the per sample execution time of the early-exit network on different platforms. The first thing to observe is that the dynamic network, when none of the early-exits is triggered (green bars), is slower than a static (normal, orange bars) network. This is explained by the additional computations introduced by the intermediate classifiers. As explained in Section 2.4.4 and in Section 3.3.2, for the followed early-exit approach, intermediate classifiers contain layers that are used to make the exit decision.

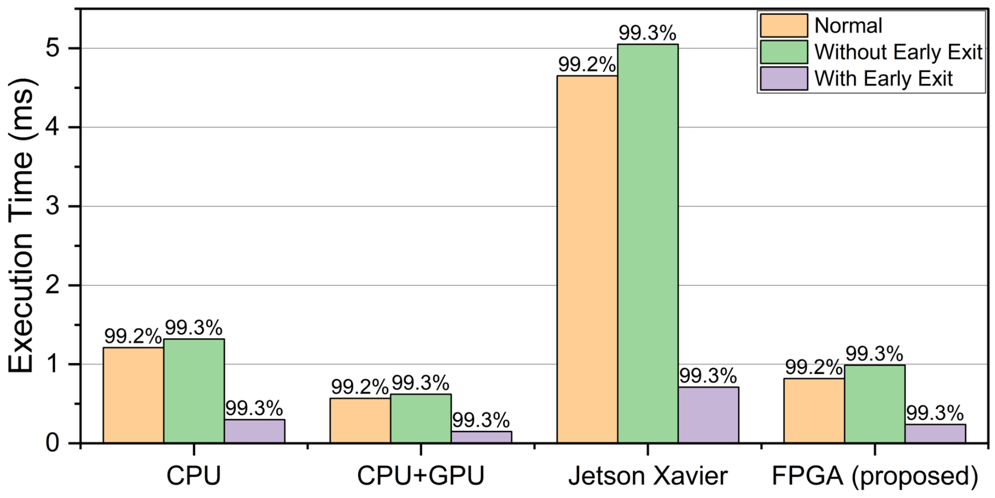


FIGURE 3.10: Experimental results comparing average Execution Time per sample. Average values are calculated based on each exit point trigger rate.

In cases where an input does not trigger an early exit, the inference contains the execution of every intermediate classifier sub-network in series with the backbone network. This delays the execution time of inference as it increases the amount of computations.

However, none of the samples triggering an early-exit is not a realistic scenario, as with the correct training, a large number of input samples will trigger an early exit. It can be seen in the results too, as the average execution time of the dynamic network executed without restricting the early exits (purple bars), is substantially faster. This occurs, as for the BranchyNet early-exit dynamic approach for LeNet-5, 94.3% of the input samples exit on the first exit point(early), 5.63% on the second (the backbone's exit). Furthermore, the accuracy of the networks is shown on top of the bars, stressing the fact that early-exiting can lead to higher accuracy values. BranchyNet jointly optimizes the weighted loss of all exit points, thus providing regularization on the exit points, preventing overfitting and improving test accuracy. Overall, the dynamic approach is at least **1.6x** faster over a static network on all platforms. At the same time, the proposed design is **1.2x** faster than a desktop CPU, comparable to a desktop CPU/GPU system and **1.7x** faster than the Jetson. These results highlight not only the effects of dynamic execution but also the capabilities of the FPGA platform. Additionally it is found that the accuracy is slightly increased in the cases where intermediate classifiers are used (99.3% over 90.2%). This can be explained by the fact that layers, especially in CNNs are attribute specific, and deeper layers target to highlight input details and characteristics harder to be found on early layers within the network. By allowing early exits, the model avoids deeper layers that might overfit to complex representations for simple inputs.

Figure 3.11 shows the average energy consumption per input sample. Similar to the execution time, it is noticeable that the dynamic network requires more energy when no early exit is triggered (green bars), logically connected with the increased amount of

TABLE 3.2: Execution Time (ms) of a static (Normal) and an early-exit dynamic LeNet-5 DNN.

	LeNet-5			
	Normal	Exit 1 (94.4%)	Exit 2 (5.6%)	Avg
CPU	1.21	0.3	1.32	0.36
CPU+GPU	0.57	0.15	0.62	0.17
Jetson	4.65	0.71	5.05	0.95
FPGA (proposed)	0.82	0.24	0.99	0.29

TABLE 3.3: Energy Consumption (mJ), with (w/ EE) and without (w/o EE) early exits of a dynamic LeNet-5 DNN.

	FPGA - Static (16.7W)		Nvidia Jetson Xavier (9.6W)			**FPGA-proposed (16.7W)**	
	w/o EE	w/ EE	Normal	w/o EE	w/ EE	w/o EE	w/ EE
LeNet-5	13.7	n/a	44.6	48.5	9.1	16.5	4.8

computations. However, as mentioned above, when it is executed without restriction on the exit triggering (purple bars), the early-exit network is significantly less energy-demanding compared to the static (orange bars). The proposed design approach requires 41% less energy over the Jetson Xavier embedded device, further illuminating the low levels of power demands FPGAs can achieve. Table 3.2 and Table 3.3 show the results in execution time and energy consumption of the early-exit LeNet-5 DNN in more detail.

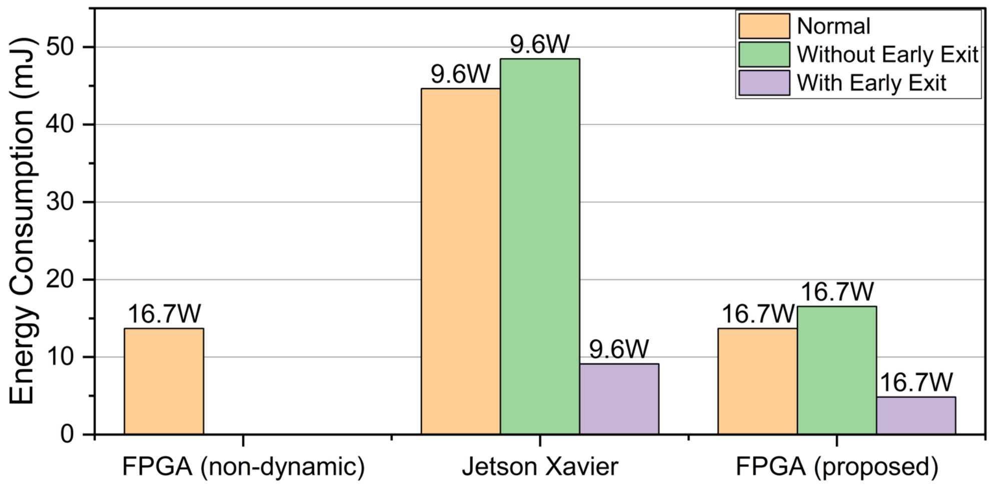


FIGURE 3.11: Experimental results comparing average Energy Consumption per sample. Average values are calculated based on each exit point trigger rate.

3.4 Discussion

Realising Dynamic DNNs on FPGAs is an effective way to reduce power consumption and flexibly adjust the computational workload of modern neural networks to meet the constraints of resource-limited environments. However, it is a very challenging task due to the changes the dynamic approaches introduce to the architecture of the neural models and the lack of libraries and frameworks, both in terms of software and hardware, that can support them. This Section presented an exploration of these limitations and identified three that present them: Software Frameworks and Training, Hardware Design Frameworks, and Early-Exit Networks' Intermediate Feature Maps Dependencies.

Following that, this Section presented a first attempt to realise and fully deploy an early-exit dynamic DNN on FPGAs. The design approach followed the traditional execution pattern of early-exit networks. It utilised systolic arrays of process elements and loop unrolling techniques, and by reusing the hardware components designed for the backbone network it achieved almost 0% of hardware overhead. The design was experimentally evaluated, and its performance was assessed over a CPU system, a CPU/GPU system, and Jetson Xavier embedded device. It achieved comparable latency results with the CPU/GPU system, a 1.7x faster execution over the embedded device, and very low energy consumption (4.8 mJ/s). This highlighted the capabilities of FPGAs to accommodate dynamic DNNs, and more specifically, early-exit networks, not only efficiently but also achieving very low power consumption. Yet training and deployment methodologies for dynamic networks are an ongoing area of research, and future directions should involve:

Libraries. The continued development of libraries targeting dynamic networks focusing on optimization and integration with mainstream software (Pytorch, Tensorflow etc.). While training and profiling are possible with existing libraries, optimized execution that can leverage batched computation and high-power computing is not well supported in mainstream tools.

Hardware-Software co-design. Existing FPGA hardware tools (hls4ml, finn, etc.) have limited support in this area and even less for Dynamic DNNs. While there is an intuitive explanation for the performance improvements of Dynamic DNNs, an exploration of why they are effective in their current state will help to derive more performant architectures and adoption in different ML tasks.

Neural Models. The majority of research on Dynamic DNNs has focused on tasks of image classification utilising CNNs. However, a diverse number of models (RNNs, GANs etc.) are already efficiently deployed on FPGAs, tackling various tasks like object detection, regression and image captioning, for which dynamism could be very beneficial.

Training Strategies. Training DNNs is inherently challenging especially, when FPGAs are targeted. Developing hardware-friendly strategies could enable on-device training leveraging the platform's low power and acceleration potential.

Distributed Execution. Beyond the resource-constrained setting, the flexibility of both Dynamic DNNs and FPGAs can be leveraged in multi-FPGA systems. Using runtime managers, reconfiguration, and high bandwidth interconnects, this system could tackle much larger-scale networks while continually adapting the level of computation on a per-sample basis.

Chapter 4

Leveraging the parallelisation capabilities of FPGAs to accelerate Early-Exit Dynamic DNNs

The design approach presented in Chapter 3 deployed an early exit dynamic DNN on an FPGA platform and achieved comparable execution time with a CPU/GPU system while maintaining very low energy demands. This highlighted the compression capabilities of dynamic DNN approaches and the feasibility and benefits of using FPGA devices. However, as elaborated in Section 3.1, the architectural interventions of the dynamic approaches to neural models raise challenges in their deployment.

Despite its promising results, the design approach still struggles with two execution limitations introduced by the early-exit networks' architecture. The first is the necessity to halt the execution of the backbone network when intermediate classifiers execute. This, especially for cases when an early exit does not occur, leads to an increase in the network's latency. Focusing on Table 3.2 and comparing the "Normal" and "Exit 2" columns, it can be seen that this leads to a latency increase by at least 8% over a static network.

The second refers to the intermediate feature map dependencies. Intermediate classifiers require the previous layer's output to execute. However, as presented in Section 3.1.3, these intermediate feature maps are needed by the backbone too, and to avoid their permanent destruction by the intermediate classifiers' execution, they must be stored. This, especially in deeper networks where the size of layers is larger (4.7Mb on VGG19 [6], 1.6Mb on ResNet-32 [2]), has a significant detrimental effect on the latency and memory footprint of inference.

However, FPGAs are inherently parallel devices, which means they can perform multiple operations simultaneously. They allow for a high degree of fine-grained parallelism, where multiple operations can be executed simultaneously at the gate level. This section presents a second design approach, aiming to exploit these parallelization capabilities and overcome the abovementioned limitations by executing both the backbone network and the intermediate classifiers simultaneously. The two designs are compared over latency, energy and memory demands and characterised over different networks and decision thresholds. Finally, a first modelisation of their time and energy demands on the FPGA is presented.

The novel contributions of this Chapter can be summarised as:

- **A design approach that leverages the parallelization capabilities of the FPGA platform.** This design approach, by executing the intermediate classifiers in parallel with the backbone network, achieves to eliminate the dependencies on the intermediate outputs and the latency cost of halting the backbone network's execution.
- **A characterisation of the two approaches over different early exit networks and different decision thresholds.** The effects on latency, energy demands, and accuracy for both designs are highlighted to achieve a thorough analysis of their benefits and disadvantages.
- **A first modelisation of the energy and time demands of an early exit network executed on an FPGA**, enabling the calculation of the trade-offs between hardware resources, accuracy and latency, allowing the exploration of the best design approach for a given scenario.

4.1 Design Approach

This design approach utilises the parallelisation capabilities of the FPGA platform, aiming to eliminate the intermediate feature map dependencies (presented in Section 3.1.3) and the latency cost caused by halting the execution of the backbone network. The basis of the design lies in the fact that intermediate classifiers are substantially smaller, and their execution is usually faster than the execution of a backbone network's layer. This, along with the ability of the FPGAs to deploy multiple execution paths in parallel, presents the opportunity for simultaneous execution of the backbone network and the intermediate classifiers without compromising the efficiency of the architecture by executing a substantial amount of unnecessary computations for the backbone network, if an early-exit occurs.

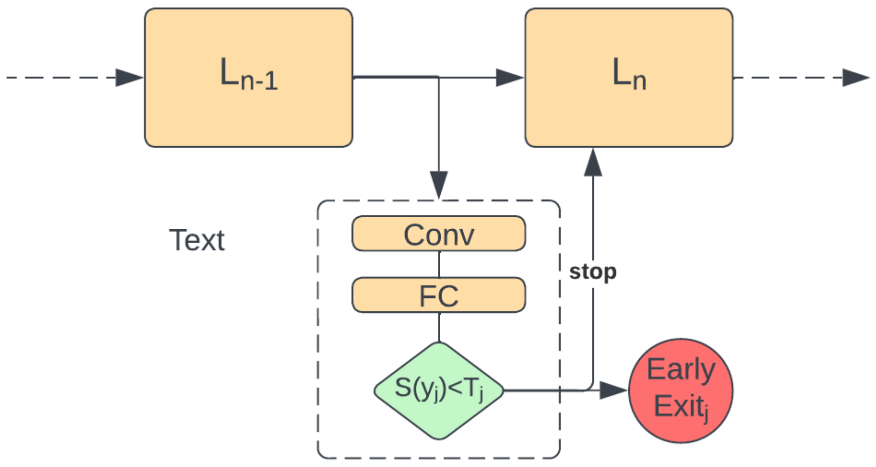


FIGURE 4.1: Parallel Approach.

Similar to the previous approach (Section 3.3), the execution of the backbone network starts normally until the layer before the intermediate classifier. In this design approach, instead of halting the backbone network, the output of the previous layer is fed to both the next layer of the backbone network and the intermediate classifier. They start executing simultaneously, but the intermediate classifier, due to its smaller size and lower complexity, always finishes executing first. Dictated by the generated decision, the execution of the backbone either continues or stops and the network resets. This eliminates the latency overhead while computing the early exit branch but also, more importantly, the need to store the intermediate layer's output. However, the simultaneous activation of both paths requires the design of separate IPs for the intermediate classifier, increasing the area and power demands of the design. For the remainder of the thesis, this approach is going to be referred to as the ***parallel*** approach, while the one explored in the previous section is the ***pipeline*** approach.

4.1.1 Designs Modelling

Before analysing the experimental results and to give a better understanding of how the designs are executed on the FPGA, a modelisation of their energy consumption and execution time is presented. To estimate the total energy consumption of the designs for the inference of I input samples, the following equation is used :

$$E_{inf} = \sum_{i=1}^{I}(P_{FPGA} \cdot T_i + E_{DRAM} \cdot M_i) \tag{4.1}$$

where P_{FPGA} and T_i are the power consumption of the FPGA design and the total execution time of inference for the ith input. M_{i} is the number of Memory Accesses to the off-chip DRAM, and can be calculated based on the systolic array size and the workload parallelisation approach presented in Section IV. Finally, E_{DRAM} is the typical per-bit access energy of LDDR4 memory (4 pJ/bit [126]).

In early-exiting approaches, however, the execution time is directly connected to every exit point's triggering rate. Essentially, this is the rate of the number of test inputs that trigger a target exit. This is dictated by one of the most influencing parameters in these dynamic approaches, which controls how harsh or lenient the decision to exit is. For example, in the target network BranchyNet, vector V is used to determine if an input i should exit at a specific exit-point. It is calculated during training, using the proposed custom optimisation function, and controls the trade-off between accuracy and latency of the early-exit scheme. After training, the rate of samples that would trigger every exit point can be experimentally calculated, creating the following average per sample execution time $\overline{T}$ equation:

$$\overline{T} = a_1 \cdot \boldsymbol{T}_1 + a_2 \cdot \boldsymbol{T}_2 + ... + a_j \cdot \boldsymbol{T}_j + ... + a_J \cdot \boldsymbol{T}_J \tag{4.2}$$

where a_{j} is the rate of samples that exits in the jth exit point, J is the total number of the dynamic network's exits, and $\mathbf{T}_{\text{j}}$ is the execution time of the backbone layers and all the intermediate classifiers, till that exit point. However, due to its different architecture, for the ***parallel*** approach, only the triggered exit point intermediate classifier execution time is added. So equation 4.2 can be correctly rewritten as:

$$\overline{T_{pipe}} = \sum_{k=1}^{J}(a_k \cdot (T_k + \sum_{i=1}^{k}\tau_i)) \tag{4.3}$$

$$\overline{T_{para}} = \sum_{k=1}^{J}(a_k \cdot (T_k + \tau_k)) \tag{4.4}$$

where now T denotes the execution time of the backbone network layers and τ the execution time of the intermediate classifier. The average execution time of the early-exit network executed on the ***pipeline*** approach is the sum of the rates a_k of every exit point multiplied by the sum of the execution time of every layer of the backbone network (T_k) and all early-exit branches up to that point (sum of τ_i). For the ***parallel*** approach, the same function applies, but instead of adding the execution time of every branch, the one that led to the stop of inference τ_k is only added. In this occasion, when no early-exit occurs, the added execution time from the intermediate classifier (τ) is 0.

In addition to the difference in execution time, the ***parallel*** approach, as seen in Section III B, does not require storing the layer's output before an exit point like the ***pipeline*** approach, decreasing the memory access needs. The following equations are proposed to calculate the energy consumption of the two designs:

$$E_{pipe} = \sum_{i=1}^{I}(P_{pipe} \cdot \overline{T_{pipe}} + E_{DRAM} \cdot (M_i + \sum_{k=1}^{i} M_{intk})) \tag{4.5}$$

$$E_{para} = \sum_{i=1}^{I}(P_{para} \cdot \overline{T_{para}} + E_{DRAM} \cdot M_i) \tag{4.6}$$

where M_{intk} is the memory access needs to store and load the intermediate layer output.

4.2 Experiment Setup

4.2.1 Hardware Implementation

The hardware architecture followed for the execution of the ***parallel*** approach is very similar to the one presented in Sections 3.3.3 and 3.3.5. The core of the hardware design is the Array of Processing Elements, which is responsible for the execution of both the Convolution, Fully-Connected and Pooling layers, while the activation functions, which are solely conditional branches, are instantly applied to the feature maps. An identical but significantly scaled-down architecture is used for the execution of the intermediate functions. The main difference between the implementation of the two design approaches is the control module. In the case of the ***parallel*** approach it is designed to follow the different data-path introduced by the simultaneous execution of the backbone network and the intermediate classifiers. Additionally, it constantly monitors an interrupt signal that indicates an early-exit and, subsequently, the rest of the network execution with the next input sample. The target platform is again the Zynq UltraScale+ MPSoC ZCU106 Evaluation Kit. Table 4.1 compares the FPGA resource utilisation for the two architectures.

The dynamic network accelerator outlined above is implemented using VHDL on a Xilinx FPGA board for four different DNNs. To verify and compare the performance of the FPGA design, the networks are also executed using PyTorch on a desktop computer with an Intel Core i9 CPU and an Nvidia RTX 2080 TI GPU and on the Nvidia Jetson

TABLE 4.1: Resource Utilisation

Design	Data	#PEs	FF	LUT	BRAM	DSP	Clock
Pipeline	8-bit	300	50%	66%	96%	78%	150 Mhz
Parallel	8-bit	300	58%	73%	89%	81%	150 Mhz

Xavier NX embedded platform. The calculation of the execution time and the power consumption is performed as explained in Section 3.3.6.

4.2.1.1 DNN Networks

The experiments were conducted with four of the most popular DNNs: LeNet-5 [37], VGG19 [6], AlexNet [1] and ResNet [2]. The networks are modified to add early exits using the structure of BranchyNet [81], an open-source and popular early exit network. Explicitly, on LeNet-5, one branch consisting of 1 convolutional layer and a fully-connected layer is added after the first convolutional layer (L_1) of the main network. On AlexNet, one branch consists of two convolutional layers and a fully-connected layer after the 2nd convolutional layer (L_2) of the main network. Finally, on VGG19 and ResNet two branches are added containing a convolutional and a fully-connected layer, after the 2nd (L_2) and 10th (L_{10}) convolutional layers on VGG19 and after the 2nd (L_2) and 14th (L_{14}) convolutional layers on ResNet32. It must be noted that an external host processor orchestrates the execution of the ResNet32 network, while the rest of the dynamic DNNs are deployed entirely on the FPGA.

4.2.1.2 Datasets

The designs are bench-marked and compared using three widely used and open-source datasets: MNIST [127], CIFAR-10 and CIFAR-100 [129]. MNIST contains 60,000 training images and 10,000 testing images, each of which is a 28x28 pixel grey-scale image of a handwritten digit. The two CIFAR datasets contain 32x32 pixel RGB natural images across 10 and 100 classes, respectively, with 50,000 training and 10,000 validation samples.

4.3 Experimental Results and Analysis

4.3.1 Design Validation and Evaluation

To validate and evaluate the FPGA architectures for the two designs, the four widely used DNNs described above are deployed and trained based on the BranchyNet [81] approach. For brevity, Figure 4.2 shows the per-sample execution time of VGG19 on

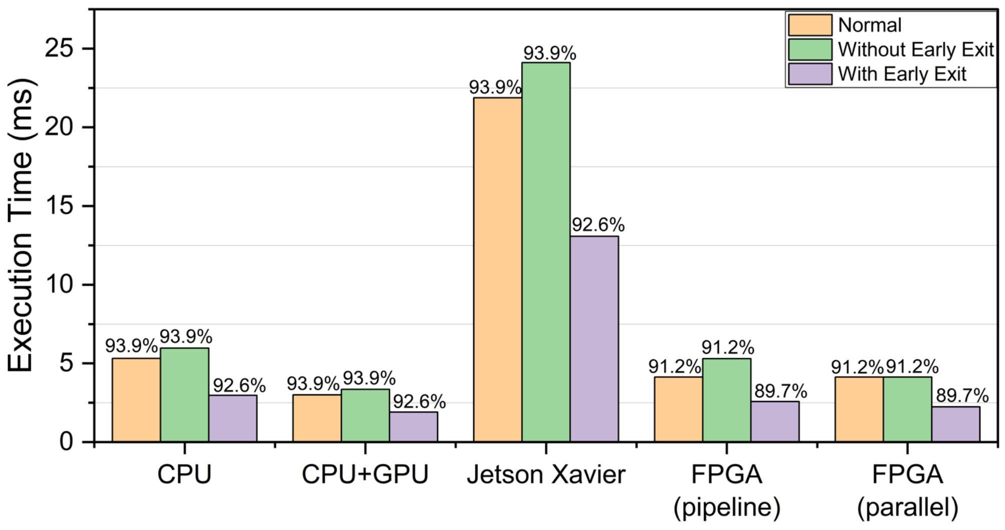


FIGURE 4.2: Experimental results comparing average Execution Time per sample. Average values are calculated based on each exit point trigger rate (a_i).

the Cifar-10 dataset, both for the original static (orange bars) and the dynamic network (green and purple bars). The green bars show the time needed to execute the early-exit network when the last exit is triggered, and comparing it with the static network (orange bars) highlights the additional latency introduced by the intermediate classifiers.

However, dynamic inference exits for 43% (a_1) of the samples at the first exit point, 45.6% (a_2) on the second and 11.4% (a_3) on the original output. The average execution time ($\overline{T}$) for the 10k test samples is portrayed on the Figure 4.2 (purple bars). As illustrated by the difference between the Normal and Early Exit bars and explained by equations 4.3 and 4.4, this leads to an execution speedup of at least 1.4x in all platforms. At the same time, for the FPGA approaches, the dynamic network is at least 1.9x faster. The benefits of executing the backbone network and the intermediate classifiers in parallel are clearly visible, especially when focusing on the results without early exiting (green bars). The parallel approach eliminates completely the execution time overhead introduced by the intermediate classifiers in an early-exit network, equalising its execution time with the static network. Also, in a normal dynamic scenario, the parallel approach is 16% faster than the pipeline approach. Finally, it is also seen that the accuracy (top of bars) of the network is barely affected, being only 1.5% less than the original static network.

The effects on the execution time of Dynamic DNNs, however, also impact their energy demands (equations 4.5 and 4.6). Figure 4.3 shows the per-sample energy consumption (E_{inf}) of VGG19 both for the original static (orange bars) and the dynamic network (green and purple bars) on the Jetson and the FPGA. Corresponding to the execution time, in cases where an early exit does not occur, the energy consumption of the dynamic network is higher than the static's. However, when the dynamic network is properly trained, the effects of early-exiting lead to a reduction in energy consumption

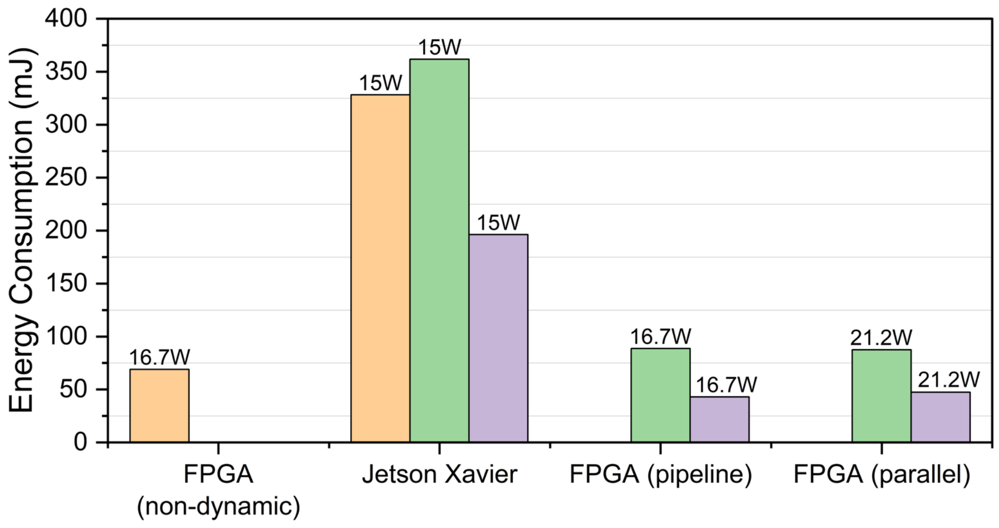


FIGURE 4.3: Experimental results comparing average Energy Consumption per sample. Average values are calculated based on each exit point trigger rate (a_i).

over a static network by at least 1.4x on the Jetson and 1.8x on the FPGA. As illustrated on Figure 4.3, the FPGA is able to achieve comparatively low energy consumption (at least 4x less than the Jetson), highlighting the capabilities and effectiveness of a custom hardware design. Focusing on the two FPGA designs, the parallel approach, despite having reduced execution time, is more energy demanding by 1.1x than the pipeline. This highlights the effects of the increased complexity of the hardware architecture and indicates the importance of the tradeoff between resource usage and performance. A more detailed exploration of the two approaches' characteristics and results is presented in Section 4.3.2.

The performance against other FPGA-based DNN accelerators is also assessed, as illustrated in Table 4.4. All of the designs employ systolic arrays of process elements and explore different techniques to accelerate DNNs on FPGAs, e.g. Winograd-GEMM [130], FFT [131] and pruning [132]. Although the targeted early-exit networks maintain the backbone network unchanged and actually burden it with the intermediate classifiers, the proposed accelerator achieves comparative power and better energy efficiency than other designs, further highlighting the benefits of dynamic approaches. In fact, there is still adequate opportunity to enhance the accelerator's performance. It is observed that DSP utilization is relatively low due to insufficient block RAM resources. By optimizing the storage structure or utilizing a platform with more resources, the processing element array can be expanded to enhance DSP utilization, thereby achieving higher throughput. Furthermore, by combining dynamic approaches with acceleration techniques, such as the ones used by the cited designs, latency and resource demands could be further reduced.

TABLE 4.2: Execution Time (ms)

	LeNet-5			AlexNet			VGG19				ResNet32			
	Exit 1 (94.4%)	Exit 2 (5.6%)	Avg	Exit 1 (75.6%)	Exit 2 (24.4%)	Avg	Exit 1 (43%)	Exit 2 (45.6%)	Exit 3 (11.4%)	Avg	Exit 1 (48.4%)	Exit 2 (18.3%)	Exit 3 (33.3%)	Avg
CPU	0.3	1.32	0.36	0.75	3.3	1.38	1.92	3.21	5.98	2.97	2.54	8.24	12.74	6.98
CPU-GPU	0.15	0.62	0.17	0.38	1.55	0.66	1.28	2.14	3.35	1.91	1.37	3.53	5.75	3.22
Jetson	0.71	5.05	0.95	1.78	12.63	4.43	8.98	14.2	24.12	13.08	9.9	15.62	26.53	16.48
Pipeline	0.24	0.99	0.29	0.6	2.48	1.06	1.49	2.99	5.31	2.58	1.84	5.1	9.4	4.95
Parallel	0.24	0.82	0.28	0.6	2.05	0.95	1.49	2.48	4.13	2.24	1.84	4.54	7.89	4.01

TABLE 4.3: Energy Consumption (mJ), with (w/ EE) and without (w/oEE) early exits

	FPGA - Static (16.7W)		Nvidia Jetson Xavier			**Pipeline (16.7W)**		**Parallel (21.2W)**	
	w/o EE	w/ EE	w/o EE	w/ EE	Power (W)	w/o EE	w/ EE	w/o EE	w/ EE
LeNet-5	16.5	n/a	48.5	9.1	9.6	13.7	4.8	17.4	5.9
AlexNet	34.2	n/a	142.7	50.1	11.3	41.4	17.7	45.5	21.1
VGG19	68.9	n/a	361.8	196.2	15	88.7	43.1	87.6	47.5
ResNet-32	99.3	n/a	398	247.2	15	143.82	75.7	141.8	85

4.3.2 Evaluating *pipeline* vs *parallel* Approaches

Table 4.2 contains the average per-sample execution time ($\overline{T}$) for each network and every exit point, alongside the percentage of samples (a_i) that would trigger them (underneath every exit). Focusing on the FPGA architectures and comparing them with the rest of the platforms, it is visible that their performance is competitive compared to the CPU/CPU-GPU and much better than that of the Jetson. The benefits of executing the intermediate classifiers in parallel with the backbone networks are also noticed, as the ***parallel*** approach is constantly faster than the ***pipeline***.

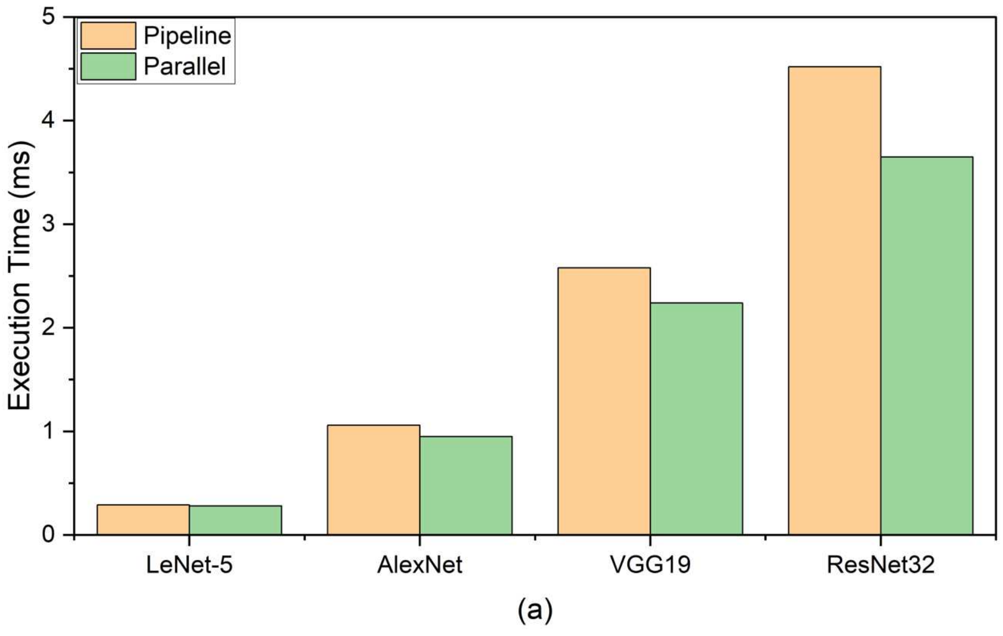


FIGURE 4.4: Comparison of the ***pipeline*** and ***parallel*** designs over average execution time across early-exit LeNet-5, AlexNet, VGG19 and ResNet32.

To further explore and characterise the two proposed designs, their latency, energy and memory demands are compared. Figure 4.4 shows the average execution time ($\overline{T}$) of the deployed early-exit networks on the two approaches. The aforementioned ***parallel*** advantages are more pronounced on deeper networks; the latency of LeNet-5 is improved by 3.7%, while VGG19 and ResNet32 are improved by 15.2% and 23.9%, respectively. This is because the design avoids halting the backbone network, which is very effective in cases where there are many samples that would require the execution of deeper layers. That occurs when more complex data sets with a higher number of non-canonical samples are targeted. Also, deeper networks usually consist of larger layers with bigger feature maps and more filters. This unavoidably affects the intermediate classifiers, too, increasing their size and, consequently, the required computation. Conversely, smaller networks targeting simpler data-sets have a high proportion of samples triggering exits in shallow layers, reducing the effectiveness of parallel execution.

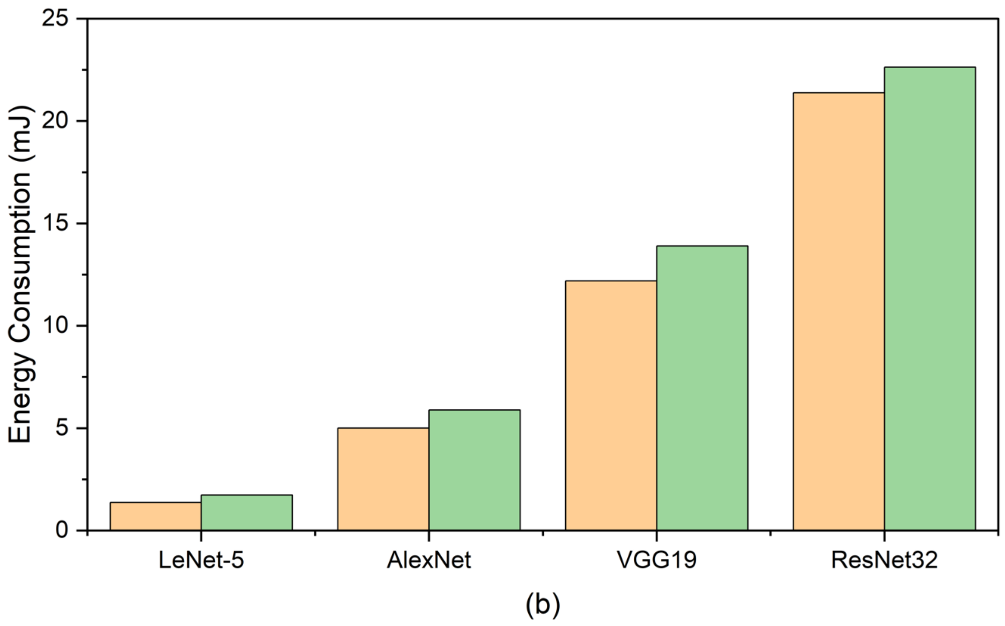


FIGURE 4.5: Comparison of the ***pipeline*** and ***parallel*** designs over average energy consumption across early-exit LeNet-5, AlexNet, VGG19 and ResNet32.

Table 4.3 shows the energy per sample for every network with and without early exiting. Based on the previously stated rates of early-exiting at each branch, it can be seen that the average energy consumption is always at least 25% lower than the static network on the embedded device and almost 50% on the FPGA (E_{inf}). It is also noticed that the ***parallel*** approach, on average, is at least 15% more energy (Figure 4.5) demanding on smaller networks than deeper networks. This is explained by the same reasons elaborated in the previous paragraph. In simpler networks, fewer samples require inference at deeper layers, while the size of the intermediate classifiers is proportional to the size of the backbone, rendering the ***parallel*** approach less effective.

Furthermore, as explained above, the ***pipeline*** approach requires the storing of the intermediate layer's output (M_{intk}); the total data for each network can be seen in Figure 4.6. It is considerably larger on deeper, more complex networks. While smaller networks like LeNet-5 and AlexNet, the data can be buffered on-chip, VGG19 and ResNet32 can reach up to 4MB, rendering the use of external memory mandatory. This significantly affects the latency and energy demands of the ***pipeline*** approach, and illustrates the effectiveness of the ***parallel*** approach on deeper networks. Overall, however, for smaller networks, the extra energy demands that are introduced in the ***parallel*** approach overshadow its benefits, making the ***pipeline*** the better design.

TABLE 4.4: Performance Comparison With Existing Implementations

	[130]	[131]	[133]	[134]	[132]	[135]	Ours (pipeline)	Ours (parallel)
DNN model	AlexNet/ VGG16	AlexNet/ VGG16	ResNet50	AlexNet/VGG16 /ResNet50	LeNet/AlexNet /VGG16	VGG16	LeNet/AlexNet/ VGG19/ResNet32	LeNet/AlexNet/ VGG19/ResNet32
Device	FPGA VX690T	FPGA ZCU102	FPGA ZC706	FPGA ZCU102	FPGA XCVU9P	Aria-10 GX1150	FPGA ZCU106	FPGA ZCU106
Frequency (MHz)	200	200	200	200	300	150	150	150
Precision	Fixed 16	Fixed 16	Fixed 16	Fixed 16	Fixed 8	Fixed 16	Fixed 8	Fixed 8
Power (W)	17.3	23.6	6.23	23.6	-	45	16.7	21.2
Performance (GOPs)	433.6/ 407.2	854.6/ 2479.6	107.9	2068.5/2641 /1240	1490.5/913.9 /862.2	232.2	583.1/911.3 /1732.5/185.1	611.9/1009 /2011.6/243.9
DSP Efficiency (GOPs/DSP)	0.30/0.28	0.34/0.98	0.24	1.81/2.31/1.08	3.88/1.79/1.68	0.153	0.78/0.96 /1.82/0.20	0.66/0.82 /1.64/0.19

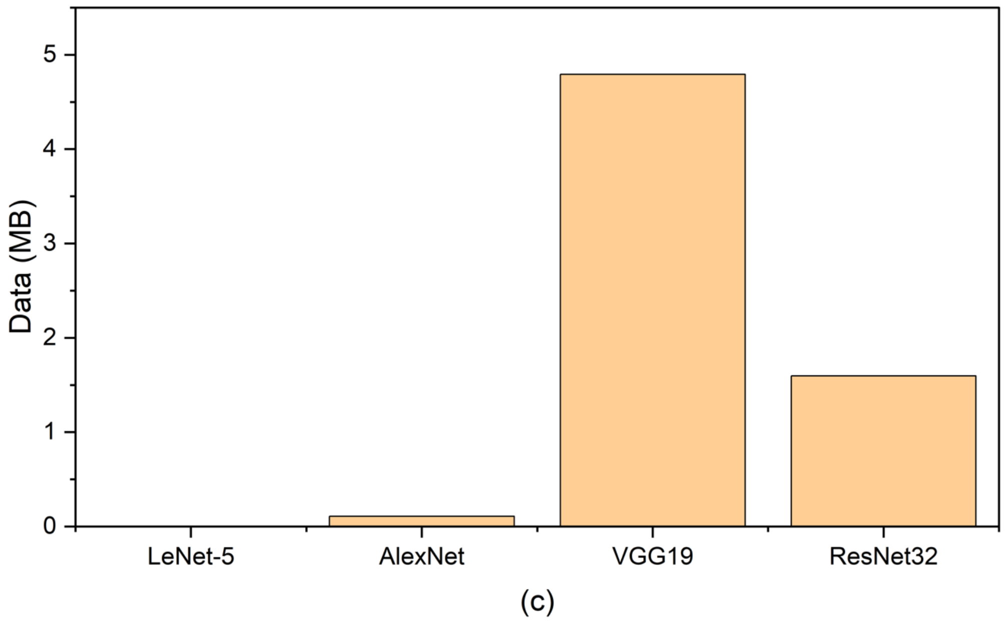


FIGURE 4.6: Comparison of the ***pipeline*** and ***parallel*** designs over average data movement across early-exit LeNet-5, AlexNet, VGG19 and ResNet32.

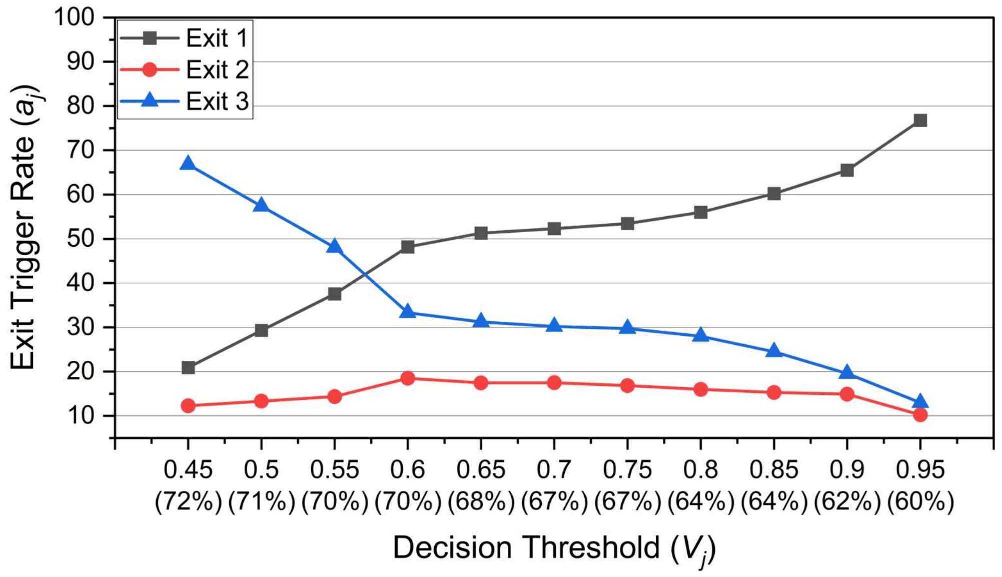


FIGURE 4.7: On a 3 point early-exit Resnet-32 a) shows each exit's trigger rate and b) the percentage difference of *parallel* over *pipeline* approaches over Energy and Time for different Decision Thresholds.

4.3.3 Effect of Decision Threshold on Performance

Section 4.1.1 presented the modelling of the two approaches regarding energy consumption and execution time. The importance of the decision threshold (V) is highlighted, as it controls the effects of early-exiting in the DNN. Figure 4.7 shows the triggering rates (a_i) of 10k samples at the three exit points of an early exit ResNet32, along with the accuracy achieved. For higher decision thresholds, most of the samples would trigger the first exit point (black line). Decreasing the decision threshold results in a significant decrease in the number of samples that would exit at the first exit point (black line), while those that would exit on the third and final exit increase (blue line). Besides affecting the exit point rates, changing the threshold essentially affects the amount of abstracted information needed to make a prediction which consequently affects the accuracy of the network, too.

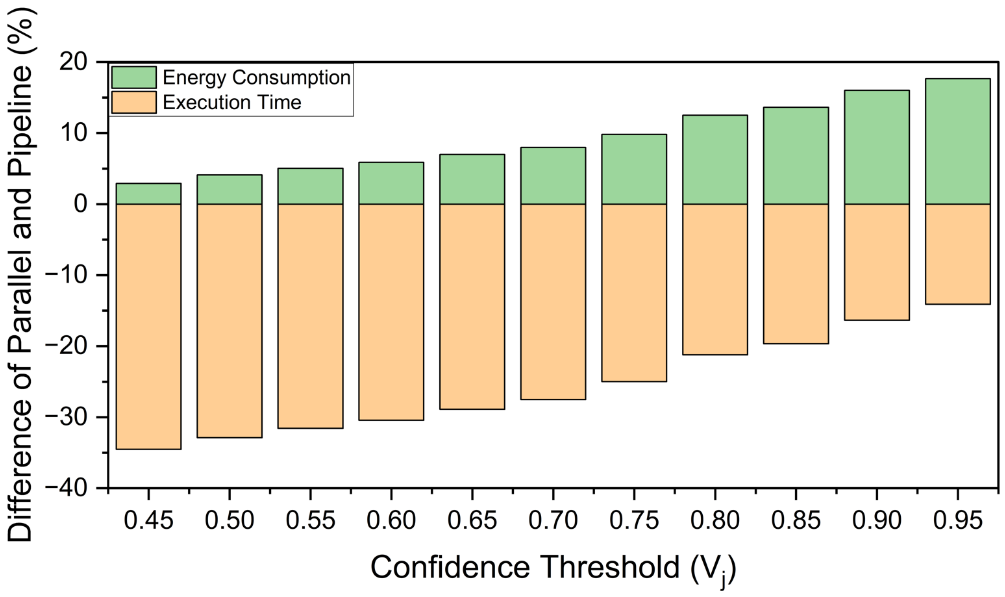


FIGURE 4.8: On an 3 point early-exit Resnet-32 a) shows each exit's trigger rate and b) the percentage difference of *parallel* over *pipeline* approaches over Energy and Time for different Decision Thresholds.

Following equations 4.3 - 4.6, it is noticeable that, by changing the thresholds, the latency and energy demands of the network can be controlled. 4.8 shows the difference between the two approaches over execution time and energy consumption. It is noted that, while using higher thresholds, the ***parallel*** approach is considerably more energy-demanding without similar gains in execution time. On the contrary, when the threshold values are lower, deeper layers are executed more often, resulting in reducing the energy consumption difference between the two approaches and increasing the time difference, making the ***parallel*** approach considerably faster. Consequently, apart from the dynamic network architecture, changing the decision threshold to exit controls the accuracy-latency trade-off and affects the effectiveness of the designs realising it.

Following the results of changing the decision threshold in execution time and energy consumption, another tradeoff is highlighted between accuracy and performance. As shown in Figure 4.7, higher threshold values lead to accuracy drops, while the opposite happens when they are lower. This is explained by the fact that more samples exit at an earlier stage without the network being able to abstract enough information to make a correct classification. On the contrary, a higher amount of samples exiting in earlier stages of the network translates to a significant drop in the necessary computations for inference and, consequently, the energy consumption. These results showcase that higher decision thresholds lead to more accurate but with increased latency and energy consumption implementations, while lower decision thresholds have the opposite effects.

4.4 Discussion

The ***pipeline*** approach demonstrated the feasibility of deploying dynamic DNNs on FPGA devices by achieving comparable execution times with CPU/GPU systems while maintaining very low energy demands. However, following the traditional execution pattern of eraly-exit networks, introduces challenges related to latency and memory footprint due to the halting of the backbone network and the need to store intermediate layer outputs. To address these limitations, this Section proposed a ***parallel*** design approach, which leverages the parallelism capabilities of FPGAs to execute the backbone and the intermediate classifiers simultaneously. The experimental results highlight that the ***parallel*** approach outperforms the pipeline approach regarding execution time, especially for deeper networks like VGG19 and ResNet32, achieving up to a 23.9% reduction in latency.

However, to achieve that, additional hardware components are introduced to incorporate the parallel execution of the backbone network and the intermediate classifiers. Through the evaluation of the two design approaches over latency, energy, memory demands and different early-exit decision thresholds, it was shown that the ***parallel*** approach is more energy-intensive for smaller networks and lower thresholds where early-exits frequently occur in shallow layers without justifying the reduction in latency. The opposite occurs when networks are deeper and thresholds are higher. Thus, the choice between ***pipeline*** and ***parallel*** approaches depends on the specific application and network characteristics, such as depth and dataset complexity.

Furthermore, a first modelisation of the early-exit networks' performance on FPGAs was presented. In more detail, four equations are proposed, two for each design approach, that take into consideration factors like exit trigger rates, memory access cost, etc., to calculate with great precision the early-exit networks' per-sample execution time and energy consumption. Overall, using these equations and controlling the thresholds of the early-exit decision, the trade-offs between accuracy, latency, and resource needs can be explored to help decide the best design approach for any application needs.

Chapter 5

Confidece Based Dynamic Control of Early-Exit Networks

The influence of the decision threshold on the performance of an early-exit dynamic DNN was illustrated in the previous Chapter, both in terms of exit trigger rates and accuracy. In essence, the confidence value, produced by the intermediate classifiers, is a metric of the amount of information the backbone network was able to abstract from the input sample up to that point of execution. Accordingly, the decision threshold sets the limit that indicates when this information is sufficient to prompt a correct classification. The ability to extract such a value is vital for the early-exit dynamic networks and is the basis for their effectiveness and compression capabilities. Despite its importance, however, the confidence level of an input sample is utilised only when it satisfies the decision threshold limit. Contradictory, when its value cannot achieve this, is completely disregarded.

In coherence with these statements, Section 4.3.3 illustrated the effectiveness of the decision threshold on the performance of the FPGA for the two proposed design approaches. Lower decision thresholds prompt a more lenient approach to exit early, increasing the number of input samples exiting at earlier stages of the network. This resulted in lower computational demands and latency, favouring the ***pipeline*** design approach. On the contrary, when a higher decision threshold is selected, the network must be very confident before an input can trigger an exit. This induced the need for more computations and higher average execution time of the network, favouring the ***parallel*** approach.

This Section explores the effects on the performance of the early exit network when the confidence level of the samples that did not succeed in triggering an early-exit is utilised. Two different directions are investigated to target the versatility and reprogrammability of the FPGA platform. The first one proposes a confidence dynamic placement of the intermediate classifiers, aiming to enhance early-exiting performance by better adapting the early-exit network architectures to the input samples' characteristics. The second introduces a confidence-controlled dynamic quantisation scheme. Quantisation is a technique widely used to reduce the resource needs of DNNs, especially when FPGAs are targeted. The proposed scheme aims to apply it in a directed way to reduce the computational and energy demands of the early-exit dynamic DNNs with minimal detrimental effects on their accuracy.

The novel contributions of this Chapter can be summarised as:

- **A first utilisation of the confidence level of input samples that did not trigger an early-exit**. The intermediate classifiers' outputs are completely disregarded when an early-exit does not occur. This Chapter highlights their importance and exploits them to increase the early-exit networks' adaptability and performance.
- **A first exploration of the effects post-training quantisation has on early-exiting behaviour.** Quantization is a widely used technique to reduce the computational demands of DNNs. However, it is shown that the unconditional application of quantisation has significant detrimental effects on the accuracy of the dynamic behaviour of the early-exit networks.
- **A first confidence-controlled dynamic placement of intermediate classifier.** This approach explores an adaptive placement of the intermediate classifiers within the backbone network governed by the input samples' confidence levels. Through this, the adaptability of the early-exit network approach is increased, and a further reduction of its computational demands is achieved.
- **A first confidence-controlled dynamic quantisation scheme.** Utilising the input sample's confidence values, this approach dynamically selects the quantisation level for different parts of the early-exit network. Doing so enables the benefits of quantisation with minimal effects on accuracy and dynamic performance.

5.1 Dynamic Early-Exit Placement

The scope of early-exiting focuses on the fact that every layer is attribute-specific, denoting that some layers are more relevant to each input sample's characteristics than others. This indicates that different inputs require the execution of different amounts of layers in order to be classified correctly. Deciding the location and the quantity of the intermediate classifiers is one of the most important tasks while designing the architecture of an early-exit dynamic network. It impacts both the granularity of the early-exit results and the overall computational overhead they introduce compared to a single-exit network. Placing an intermediate classifier after every layer of the backbone network would maximise the early-exiting efficiency, providing the capability to compute the exact number of layers needed for each input sample. However, this level of placement density can introduce an excessive computational and memory overhead, having detrimental effects on the benefits of early-exiting.

Conversely, sparse placement of intermediate classifiers may lead to longer computation of samples that would exit on earlier stages. For instance, on the ResNet-32 early-exit dynamic DNN presented in Section 4.2.1.1, an input sample that does not trigger the first exit point is bound to be executed for 12 more layers before its confidence value is assessed again. This creates a bottleneck where certain inputs are forced to go deeper into the network, even when it may not be necessary for their classification. Correspondingly, this also translates to the unnecessary computation of deeper and larger networks, reducing the overall efficiency of the network.

5.1.1 Methodology

Aiming to solve that while maintaining a balance in the number of intermediate classifiers placed in an early-exit network, a dynamic early-exit placement approach is proposed. The basis of the dynamic scheme lies in the information captured by the confidence value of each input. In more detail, the focus lies on samples that failed to trigger an early-exit. Their confidence value still represents the amount of information the backbone network achieved to abstract from them until that point of execution. It is argued that when this value is close to the decision threshold, indicates that the input sample requires the execution of only a few of the following layers in order for the network to have adequate confidence to classify it.

After the execution of an intermediate classifier and when an early-exit is not triggered, the difference (δ) between the generated confidence and the confidence threshold is calculated. Following the Equation 3.1 the calculation of δ on the jth out of J intermediate classifiers in an early-exit network can be written:

$$\delta_j = S(y_j) - V_j, j = 1, 2, \ldots, J \tag{5.1}$$

where y_j indicates the output of the jth intermediate classifier, $S(y_j)$ the confidence value of the executed input sample and V_j the decision threshold. Based on the value of δ, the location of the next exit-point is selected. For lower values, the next intermediate classifier is placed after only a few layers, while for higher values, it is placed deeper in the backbone network. The decision of the limits to determine when δ is high or low, as well as the exact location of the next intermediate classifier for both occasions, depends on the use case, the exit rate and the accuracy of each early-exit point. For example, and as shown in Section 3.3, for simpler datasets like MNIST, an early-exit

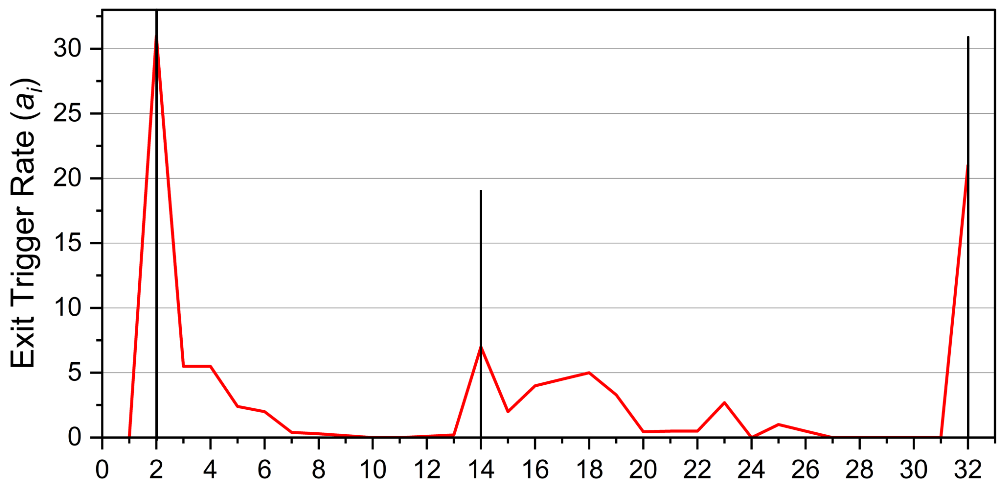


FIGURE 5.1: The exit trigger rates (a_i) of an early-exit ResNet-32 Dynamic DNN with an intermediate classifier after every layer.

point can be placed after the first layer of the backbone network and achieve a very high number of accurate classifications. For more challenging datasets, e.g. CIFAR-100, the intermediate classifiers should be placed deeper based on their classification performance.

5.1.2 Experimental Setup

To evaluate the performance of the proposed dynamic early-exit placement approach, the ResNet32 neural network was used, employing Branchynet's early-exit approach. To explore the best location to place the intermediate classifiers and to determine the high and low limits of δ, an intermediate classifier is inserted after every one of the backbone network's layers. The network is trained using the softmax cross entropy loss function, presented in Section 3.3.1, targeting the CIFAR-100 dataset.

Attempting to train the network containing all the intermediate classifiers, however, led to significant overfitting, diminishing its accuracy and generalisation capabilities. To resolve that, the training process is applied on 8 networks, each containing 4 intermediate classifiers in different equidistant points. For instance, the second network that was trained had intermediate classifiers after the *2nd*, *10th*, *18th* and *26th* layer of the backbone network. The only exception is the last network, which contains 3 intermediate classifiers, as an early exit cannot occur after layer 32. After that process, the trained parameters were combined and applied to the network containing all the intermediate classifiers.

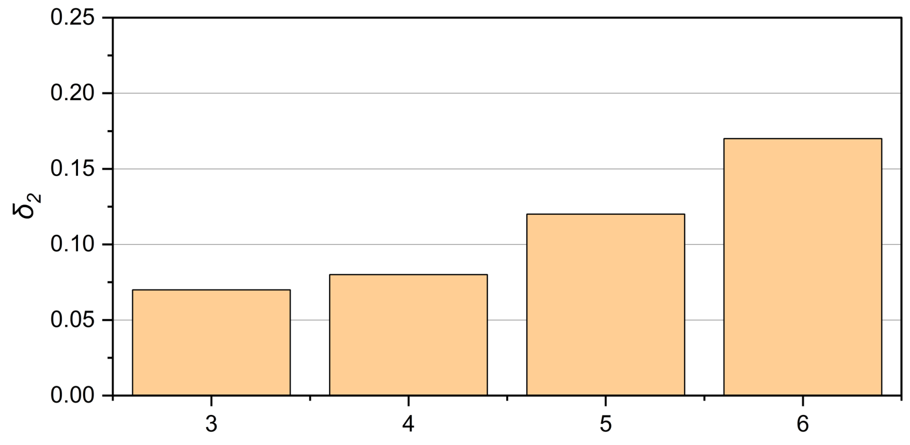
Figure 5.1 displays the percentage of input samples that would trigger an early-exit on the dynamic ResNet-32 with three exit points (black line) and when an intermediate classifier is placed after every layer (red line). It is evident that in the second case, the two exit points utilised in the original Branchynet's implementation of the early-exit ResNet-32 (at layer 2 and layer 14, as detailed in Section 4.2.1.1) maintain very high trigger rates. However, it is also noted that a considerable amount of input samples trigger an early-exit after the execution of the layers immediately following the original exit points. In more detail, 13% of the 10000 samples or 18% of the samples that didn't trigger an exit at the first intermediate classifier can be classified after processing a maximum of 4 more layers. Similarly, 16% of the total sample or 41% of the samples that did not trigger an exit after the execution of the 14th layer can be classified in the next 5 layers.

Furthermore Figure 5.2 illustrate the average δ values the input samples achieved in the intermediate classifiers after layer 2 (Figure 5.2 (A)) and layer 14 (Figure 5.2 (B)) and managed to trigger an early-exit at exit points 3 to 6 and 15 to 19 accordingly. It is noted that these values are never more than 0.21, validating the initial assumption that input samples that showcase high confidence levels but not enough to trigger an exit require the execution of only a small number of the succeeding layers to be classified. It is also important to note that the accuracy of the network was minimally affected by less than 1%.

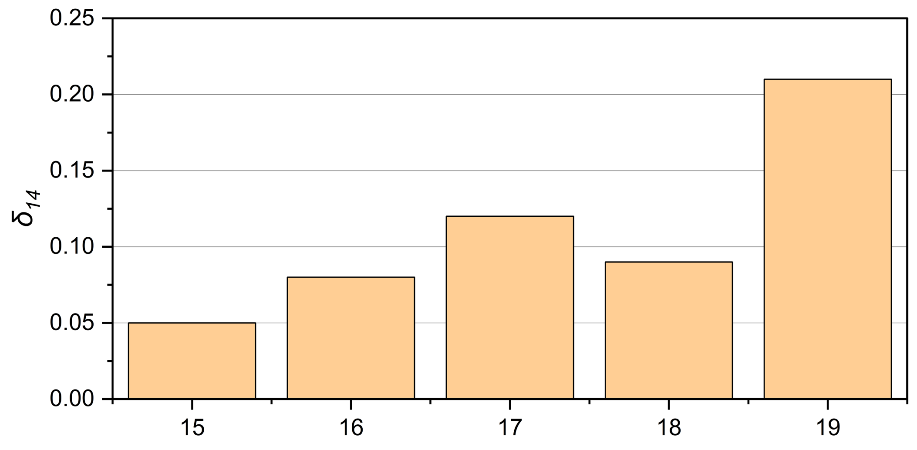
These results highlight the effectiveness of the location an intermediate classifier has. By having an intermediate classifier after every layer, the efficiency of early exiting is maximised, as the inference output for each sample requires exactly then necessary, for the network, layers to gather an adequate amount of information for a classification. This however increases a lot the computational overhead introduced by the dynamic approach, leading to a very energy and computationally intense inference, having opposite effects to what the early-exiting aims to achieve. Maintaining a balance between the amount and the location of the intermediate classifiers, as seen from the aforementioned results, is very important to enable the benefits of dynamic execution and to further enhance its performance.

Guided by the results shown in Figure 5.1 and Figure 5.2, the proposed network architecture consists of two constant intermediate classifiers and two dynamic. Likewise the Branchynet's approach, intermediate classifiers are placed after the 2nd and the 14th layers of the backbone network. However, after their execution, an assessment of the δ value is performed. If it is less or equal to 0.16, then an additional intermediate classifier is placed either after the 6th or the 19th layer, in accordance with the permanent exit points. If it is not, then the execution continues to the next permanent exit point, either 14 or 32.

Finally, to deploy the proposed network, the ***parallel*** design approach (Section 4.1) is utilised. As demonstrated in the previous Chapter, the ***parallel*** approach employs separate hardware modules to execute the intermediate classifiers in parallel with the backbone network. This implies that these dedicated modules are always available to



(A) Average δ_2 of the input samples that triggered an exit after the execution of layers 3 to 6.



(B) Average δ_2 of the input samples that triggered an exit after the execution of layers 15 to 19.

FIGURE 5.2: Average δ_j for the input samples that failed to trigger exit points 2 and 14 but achieved it at points 3 to 6 and 15 to 19 accordingly.

TABLE 5.1: Exit Trigger Rates (%)

	Exit 1	Exit 2 (Dynamic)	Exit 2	Exit 3 (Dynamic)	Exit 3
original	48.4%	-	18.3%	-	33.3%
proposed	43.9%	12.4%	7.6%	15.8%	23.3%

be used between every layer, providing the necessary versatility to accommodate the proposed approach.

5.1.3 Results and Analysis

Table 5.1 show the exit trigger rates of the three constant exit points (Exit 1 after layer 2, Exit 2 after layer 14 and Exit 3 the layer 32) and the two dynamic exit points, Exit 2 (Dynamic) and Exit 3 (Dynamic) placed after the 6th and the 19th layers of the backbone network. Focusing on the first exit point, it is seen that the proposed approach, over

TABLE 5.2: Execution Time (ms)

	Exit 1	Exit 2 (Dynamic)	Exit 2	Exit 3 (Dynamic	Exit 3	Avg
original	1.84	-	4.54	-	7.89	4.01
proposed	1.84	2.02	4.54	5.07	7.89	3.80

the original Branchynet's network, has a reduced exit trigger rate by 4.5%. This can be explained by the fact that the training process for the proposed approach included an extra intermediate classifier. This changes the balance between the exits, as they are jointly optimised. Nonetheless, it can be seen that 56.3% of the input samples trigger an early exit after executing 6 layers of the backbone network and 23.4% after executing 19 layers. In other words, compared to the original network, 7.9% of the input samples do not require the execution of at least 8 layers (between layer 6 and layer 14) and a further 10% of samples the execution of 13 layers (between layer 19 and layer 32) when dynamic early-exit placement is deployed. This translates to an overall reduction of computations by 18% over the original network while maintaining the same accuracy levels.

Following the results on exit trigger rates, Table 5.2 shows the average per sample execution time for every exit point. It can be seen that the proposed approach achieved 5.8% reduction in the latency of the early-exit dynamic DNN, which, as both designs are deployed on the same FPGA hardware design (21.2W of power consumption, see 4.3), also translates in 5.8% reduction in its energy demands. Overall, enabling the dynamic placement of the intermediate classifiers in an early-exit dynamic DNN introduces an additional level of adaptability to the characteristics of the input samples, increasing its efficiency. As a result, the proposed approach accelerates the execution of the network, reduces its energy demands, and minimises the execution of unnecessary computations for samples with high confidence levels while maintaining equivalent levels of accuracy.

5.2 Early-Exit Dynamic DNNs Quantisation

Quantization in DNNs is a technique used to reduce the computational and memory requirements of a model by representing its parameters, such as weights, biases and activations, with lower precision data types. As presented in Section 2.3.1, this process involves mapping high-precision (e.g., 32-bit floating point) numbers to lower-precision (e.g., 8-bit integer) formats, which can lead to faster computations and lower energy consumption. Due to these significant performance advantages, it became very popular for application targeting environments where computational resources and power efficiency are critical.

Such an environment is presented by FPGAs, where quantisation is of the essence to achieve efficient deployment of modern DNNs. More specifically, the most commonly used technique of quantisation in FPGAs is fixed point representation. It is a method of representing real numbers by a fixed number of bits, divided into two parts: the integer part and the fractional part. Unlike floating-point representation, where the position of the decimal point can "float," in fixed-point representation, the decimal point is fixed at a specific position (see Figure 3.5). Numerous research works, including [98], [136], [137] succeeded in reducing the representation of the DNN parameters even down to 2 bits and achieved lower computational burden and memory needs while maintaining high accuracy levels. In addition to that, FPGAs due to their customisability, offer the ability to use completely custom representations with great efficiency and without the restrictions presented by the limited support for low-bit arithmetics in general-purpose processors.

This section presents an exploration of the capability of quantisation to enhance the performance of early-exit dynamic DNNs. In more detail, two directions are investigated. The first one targets the quantisation of the intermediate classifiers. As described in Chapters 3 and 4, despite their beneficial effects, intermediate classifiers unavoidably introduce additional computational burdens to the execution of the DNN. Reducing the representation resolution can reduce their computational needs, increasing the efficiency of the early-exit networks. The second direction intends to extend this work by applying quantisation to the entirety of the network. Combining the compression capabilities of quantisation and early-exiting can further reduce the resource needs for the execution of DNNs and contribute towards enabling their deployment in resource-restricted devices.

5.2.1 Quantisation Methodology

Throughout this thesis, the post-training quantisation scheme is applied. It is a technique introduced by J. Holt and T. Hwang [138] back in 1993 and popularised by the introduction of TensorFlow Lite (TFLite) [123] in 2017. The process begins with training the network using full precision, usually a 32-bit floating point format. Then the range of values that each layer requires is determined by collecting statistics on the minimum and maximum values of the trained weights and activations. These statistics are then used to compute scaling factors that map the original high-precision values to the lower-precision representation. Once these scaling factors are established, the weights and activations are quantized by rounding the high-precision values to the nearest values in the lower-precision format.

In this work 4-bit to 8-bit fixed-point formats are used. For 4-bit and 5-bit representations, 2 bits are used for the integer part and 2 and 3 bits, respectively, for the real part. This leads to a range of -2 to 1.75 with a step of 0.25 for the 4-bit format and -2 to 1.875

with a step of 0.125 for 5-bit format. For the rest, 3 bits are used for the integer part and the rest for the real translating to a range of -4 to 3.875 with a step of 0.125 for the 6-bit format, -4 to 3.9375 with a step of 0.0625 for the 7-bit format and -4 to 3.96875 with a step of 0.03125 for 8-bit format.

5.2.1.1 Experimental Setup

To explore the effects of quantisation on early-exit dynamic DNNs, the early-exit ResNet-32 presented in Section 4.2.1.1 is deployed. It includes two intermediate classifiers containing one convolutional and one fully-connected layer, placed after the 2nd (L_2) and 15th (L_{14}) layers of the backbone network. The network requires 81.51 MB of memory to store its parameters while using 8-bit fixed-point representation, of which 227.3 kB are required by the intermediate classifiers. The network is trained using the BranchyNet approach [81] and fp32 representation. The trained weights and activations are then used to employ post-training quantisation.

When the quantisation of the intermediate classifiers is targeted, the backbone network is constantly quantised using 8-bit fixed-point representation, while for the intermediate classifiers, all 5 bit-widths presented above are explored. On the contrary, the 5 bit-widths are applied to the entirety of the early-exit network. Finally, to evaluate the effects of quantisation, the two design approaches, ***pipeline*** and ***parallel***, presented in the previous Chapters, are used.

5.2.2 Early-Exit Branch Quantisation

Despite the effectiveness of early-exiting, in order to enable the dynamic inference of a DNN, the deployment of intermediate classifiers is essential. Through correct training they can reduce the overall time consumption and energy demands of a network, however, inevitably they introduce additional computations and memory demands. Figure 4.4 and Figure 4.5 illustrated that, especially for samples that require the whole depth of the network to be classified. Comparing the *Without Early Exit* (green bar) with the *Normal* (orange bar), it is noticed that for the ***pipeline*** approach, the execution time of the early-exit VGG19 network is increased by 31%. This is resolved through the ***parallel*** approach by executing the intermediate classifiers and the backbone network in-parallel, which, however, necessitates the design of additional hardware components, leading to an increase in energy consumption by 29% over a static implementation.

In addition to that, the intermediate classifiers of the early-exit VGG19 network require an additional 125kB of memory to store their parameters. This not only contributes to the aforementioned detrimental effects in latency and energy demands but also makes the deployment of early exit networks harder. One of the primary challenges in FPGA

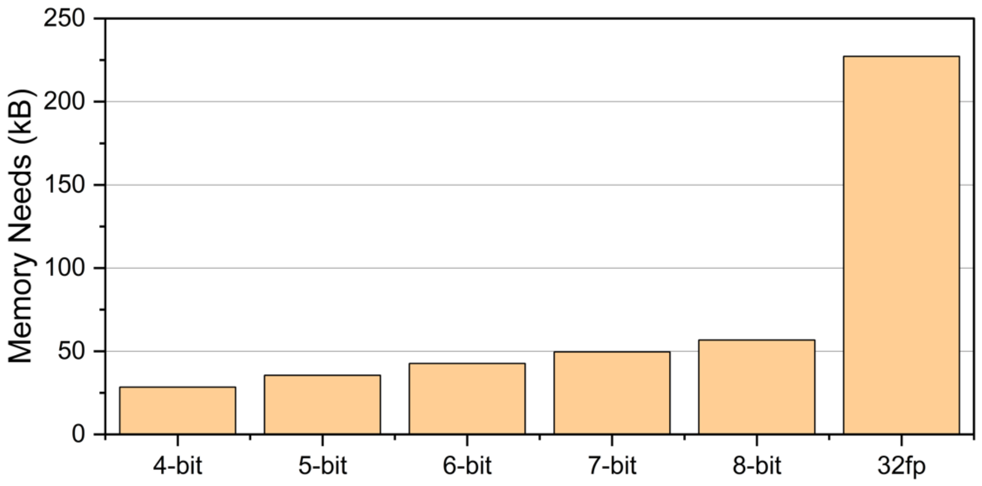


FIGURE 5.3: Early-Exit Branches Memory Needs over different quantisation levels.

design is the relatively limited on-chip memory resources compared to dedicated processors like CPUs or GPUs. FPGAs typically rely on embedded Block RAM (BRAM) or distributed RAM, which are constrained in size and can quickly become a bottleneck for data-intensive applications. Additionally, accessing off-chip memory, such as DRAM, introduces latency and bandwidth constraints and is very expensive in terms of energy consumption, further limiting performance. To resolve that, the further quantisation of intermediate classifiers, using representations lower than 8-bits, is proposed.

5.2.2.1 Experimental Results and Analysis

One of the most significant advantages of quantization is the reduction in the memory requirements of a DNN. 5.3 illustrates the memory needs for storing the parameters of the intermediate classifiers at different quantization levels. By employing 8-bit resolution, a 75% reduction in the memory requirements of the intermediate classifiers compared to fp23 can be achieved. Further reduction in representation can increase this number even more. This directly results in reduced memory accesses and simpler computations, potentially leading to lower energy consumption and faster network execution.

Before, however, examining the results of the intermediate classifiers quantisation in latency and energy consumption, it is very important to asses how it influences the early-exiting behaviour. It was presented in Chapter 4.3.3 that the decision threshold (V_i) greatly affects the performance of an early-exit network. Reducing the representation resolution of the intermediate classifiers can lead to reduced representation power that can alter the exit triggering rates. Increased numbers of input samples exiting on earlier stages could show that the network is overconfident, reducing its accuracy.

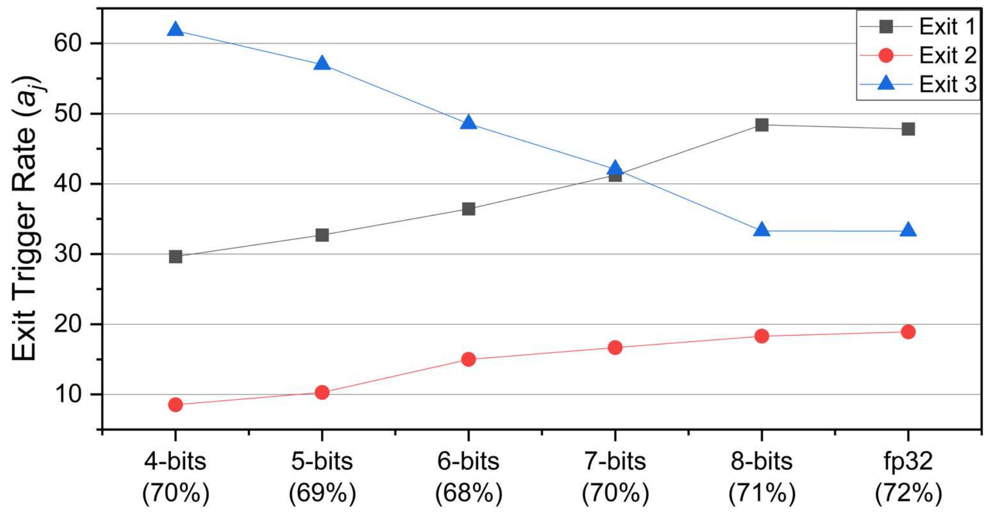


FIGURE 5.4: Percentage of samples to trigger each exit point over different quantisation levels of the exit branch.

On the contrary, increased trigger rates of deeper exits prompt that more input samples require higher amounts of computations to be classified, rendering the benefits of quantisation irrelevant.

Figure 5.4 shows the percentage of samples that would exit on each exit point. It is visible that the exit trigger rates for all three exit points (y_1, y_2 and y_3) are affected. Reducing the resolution to 4-bits and 5-bits has the biggest impact, especially on the second exit point, resulting in many input samples demanding the execution of deeper layers to be classified. This highlights the reduced representation power of lower bit-widths, which are not capable of capturing details that could potentially lead to an earlier exit. On the contrary, however, when the resolution increases, in this case, 6-bits and higher, the change in trigger rates over fp32 is less significant, always being less than 11%.

In addition to these results, bellow the bit-widths, the accuracy of the network is illustrated. In lower resolutions, where samples tend to execute till deeper stages of the DNN, the accuracy is increased. This is explained by the fact that the network, utilising more layers, can abstract more information from the samples. On the other hand, when the resolution increases, a drop in accuracy is noticed. While these representations can achieve exit trigger rates closer to the ones achieved by full resolution (fp32), the lack of representation power causes the network to commit more classification errors. Nonetheless, the drop in accuracy is not very significant, never being more than 4%.

Table 5.3 shows the per-sample execution time of the early-exit dynamic ResNet-32 network for different levels of quantisation in the intermediate classifiers. In contrast to the initial assumptions, at very low representation precision, the execution time actually increases. Specifically, employing 4-bit and 5-bit quantisation levels results in a

TABLE 5.3: Per Sample Execution Time (ms) of Early-Exit Dynamic ResNet-32 over Different Quantisation Levels of Intermediate Classifiers

		Exit Trigger Rate (a_j)	**Pipeline**	**Parallel**
4-bit	Exit 1	29.6%	1.47	1.47
	Exit 2	8.5%	4.08	3.63
	Exit 3	61.9%	7.52	6.31
	Avg	-	5.43	4.64
5-bit	Exit 1	32.7%	1.52	1.52
	Exit 2	10.3%	4.23	3.76
	Exit 3	57%	7.80	6.54
	Avg	-	5.38	4.61
6-bit	Exit 1	38.4%	1.54	1.54
	Exit 2	15%	4.28	3.81
	Exit 3	46.6%	7.89	6.62
	Avg	-	4.78	4.15
7-bit	Exit 1	43.2%	1.56	1.56
	Exit 2	16.7%	4.33	3.86
	Exit 3	40.1%	7.99	6.70
	Avg	-	4.47	3.90
8-bit	Exit 1	48.4%	1.84	1.84
	Exit 2	18.3%	5.1	4.54
	Exit 3	33.3%	9.4	7.89
	Avg	-	4.95	4.34

9% and 8% increase in execution time when using the pipeline approach, and a 6.5% and 6% increase when using the parallel approach, compared to a network using 8-bit fixed-point representation throughout its architecture. This is justified by the impact of quantisation on the exit triggering rate. As elaborated earlier, higher percentages of input samples exiting at later stages of the network lead to increased computational load, negating the performance benefits of quantisation.

However, when higher bit-widths are targeted, as seen from Figure 5.4, the exit triggering rates are a lot closer to the values achieved by a full precision network. This leads to a decrease in latency of the dynamic network. In more detail, when the 6-bit and 7-bit fixed-point representations are used, the per sample execution time is 6% and 7% less when ***pipeline*** approach is targeted and 11% and 11.5% when the ***parallel*** approach is targeted, over a network using 8-bit precision throughout his architecture.

The same outcomes can be obtained when focusing on the energy consumption of the networks. Table 5.4 shows the per sample needed energy to execute the early-exit ResNet-32 over different quantisation levels of the intermediate classifiers. It can be seen that when 4-bit and 5-bit representation is used, the energy consumption is higher than the 8-bit quantised network. This is explained by the results shown in Figure 5.4 and Table 5.3, that highlight the rates of samples that trigger each exit point. Even though the data requirements are reduced considerably, the information lost from the

TABLE 5.4: Per Sample Energy Consumption (mJ) of Early-Exit Dynamic ResNet-32 over Different Quantisation Levels of Intermediate Classifiers

	4-bit	5-bit	6-bit	7-bit	8-bit
Pipeline	78.98	78.22	67.71	64.09	72.00
Parallel	93.63	93.04	81.53	77.62	87.58

quantisation leads to a negative effect on the early-exit rates, leading to more samples executing through the entirety of the network to produce an output and overall increasing the latency and the energy demands of the network, especially over higher representations, e.g. 8-bit.

Similarly, this is caused by quantisation's effects on the early-exiting behaviour. On the contrary, it is noted that when 6-bit and 7-bit resolution is used the network is 10.5% and 15.5% respectively when the ***pipeline*** approach is used and 11.5 and 16% when the ***parallel*** approach is used. Similar to execution time, it is evident that the energy consumption is higher when using 4-bit and 5-bit representation compared to the 8-bit quantised network, attributable to the effects of quantisation on early-exit behaviour. Conversely, it is observed that when using 6-bit and 7-bit resolution, the energy consumption is 10.5% and 15.5%, respectively, when employing the ***pipeline*** approach, and 11.5% and 16% when using the ***parallel*** approach.

Overall quantising the intermediate classifiers can be beneficial towards minimising the computational overhead introduced by early-exit dynamic DNNs. As can be seen from the results, it can lead to reductions in execution time, energy consumption and, most importantly, memory demands. Nonetheless, special attention must always be paid to the effects of using lower representation precision on the early-exiting behaviour, as it can, instead of accelerating, lead to negative implications on the performance of the dynamic DNN.

5.2.3 Full-Network Quantisation

Following the results of quantising intermediate classifiers, the question is raised as to whether the same technique can be applied to the entire dynamic network. Although early-exit networks are efficient in compressing DNNs, the Achilles heel of early-exit networks is the additional computational burden of the intermediate classifiers, especially for the execution of samples that exit on the last exit point. However, quantization is a widely used technique, especially when FPGAs are the target deployment platform, to reduce the resource needs for DNN inference. Aiming to combine its benefits with those of early exiting, this section explores the effects of quantising the Early-Exit ResNet-32 Dynamic DNN to achieve further performance efficiency.

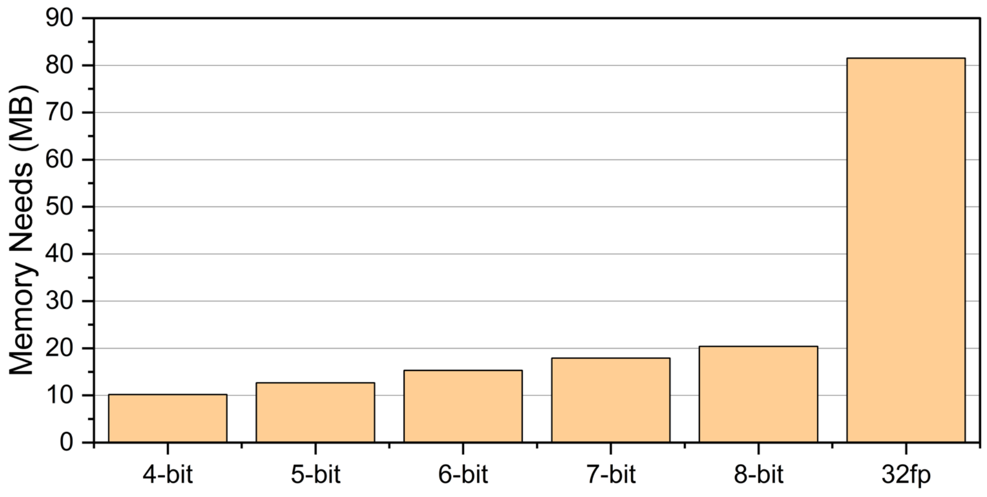


FIGURE 5.5: Early-exit ResNet-32 memory needs over different quantisation levels.

5.2.3.1 Experimental Results and Anaslysis

When quantisation is applied, the most significant result is the reduction of the memory demands of the network. The dynamic ResNet-32 requires 81.51 MB of memory (Figure 5.5) to store its parameters (weights activations etc.). Simply by reducing the representation resolution to 8-bits the data is reduced to 20.38MB (a 75% reduction). While using even lower resolutions, like 6-bit or 7-bit precision, this is further reduced by 25% and 12%, respectively.

As explained above, however, the impact of quantizing a dynamic network on its dynamic behaviour is a crucial factor to consider. Figure 5.6 shows the percentage of the input samples that would trigger each of the three exit points on the early-exit dynamic ResNet-32 DNN. In contrast to the results of the intermediate classifiers quantisation, it is apparent that the compression technique is significantly more effective in this case. The 4-bit representation resolution completely disables the first and second exit points, eliminating the dynamism from the network. Similarly, none of the input samples trigger the second exit point when the 5-bit resolution is used. This results in a substantial drop in accuracy, as both quantisation levels are at least 10% less accurate than the network using fp32 format, highlighting their lack of representation power. On the other hand, however, it is noted that 6-bit and 7-bit resolutions achieve much better results, having trigger rates substantially closer (no more than 15% different) to the ones achieved by 8-bit or fp32 representation. The effects in accuracy, although, are still more prominent in this case over the intermediate classifiers quantisation, being 9% and 4% less than what the 8-bit resolution can achieve.

Compulsorily, these results impact the energy consumption and the latency of the dynamic network too. Table 5.5 shows the per sample execution time of the early-exit dynamic ResNet-32 network for different intermediate classifiers' quantisation levels. The importance of balancing the quantisation effects on the early-exiting behaviour are

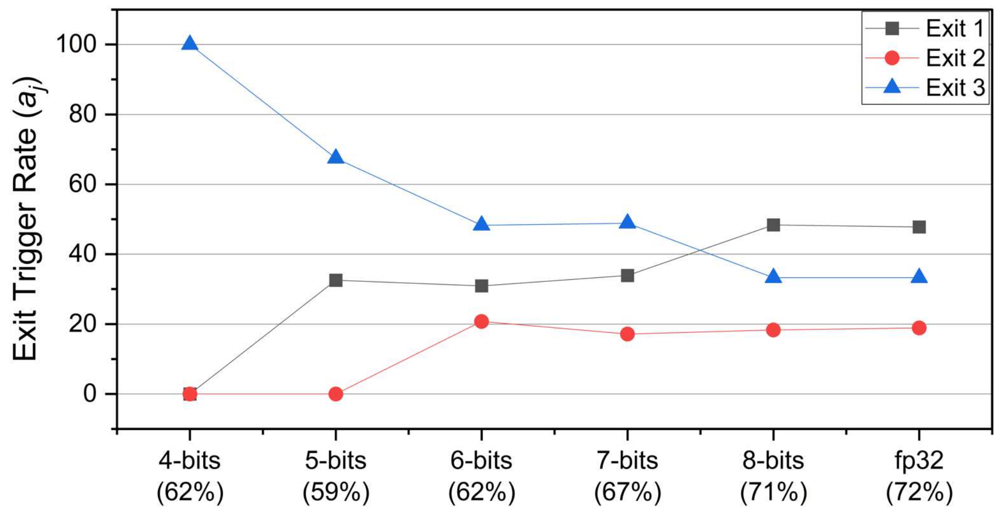


FIGURE 5.6: Percentage of samples to trigger each exit point over different quantisation levels of the early-exit network.

highlighted, as it can be seen that despite the considerable less memory needs and complexity in computations, when the 4-bit resolution is applied the overall latency of the network is actually increased, even over the 8-bit network. Similarly, while the 5-bit and 6-bit representations can achieve faster execution, the negative effects on the accuracy, being 13% and 10% less accurate, render them infeasible to use. The experimental results of the 7-bit quantized network, however, show that even by reducing the resolution by one bit, execution can be accelerated. Nevertheless, it is important to note again that early exiting plays a crucial role as the difference in triggering rates leads to only a small acceleration.

Similar conclusions can be drawn by assessing the per-sample energy consumption results (Table 5.6). Quantisation can lead to reduced energy demands. However, its effects on the exit trigger rates render its benefits futile. This is reinforced by focusing on the energy consumption of the 6-bit and 7-bit formats. While 6-bit should cause considerable reductions in energy demands, the actual decrease is less than 1% in both ***pipeline*** and ***pipeline*** approaches.

Quantization provides a great opportunity to reduce latency, energy consumption, and memory demands, particularly when targeting FPGAs for dynamic DNNs. However, applying it without considering the implications of early-exit behaviour can lead to designs that do not benefit from resolution reduction. In fact, it has been shown that it can have the opposite effects, especially when it results in more input samples exiting in the later stages of the dynamic network.

TABLE 5.5: Per Sample Execution Time (ms) of Early-Exit Dynamic ResNet-32 over Different Quantisation Levels

		Exit Trigger Rate (a_j)	**Pipeline**	**Parallel**
4-bit	Exit 1	0%	1.15	1.15
	Exit 2	0%	3.21	2.86
	Exit 3	100%	5.92	4.97
	Avg	-	5.92	4.97
5-bit	Exit 1	32.5%	1.26	1.26
	Exit 2	0%	3.51	3.13
	Exit 3	67.5%	6.48	5.44
	Avg	-	4.78	4.08
6-bit	Exit 1	30.9%	1.38	1.38
	Exit 2	20.8%	3.82	3.40
	Exit 3	48.3%	7.05	5.91
	Avg	-	4.62	3.99
7-bit	Exit 1	33.9%	1.47	1.47
	Exit 2	17.2%	4.08	3.63
	Exit 3	48.9%	7.52	6.31
	Avg	-	4.87	4.20
8-bit	Exit 1	48.4%	1.84	1.84
	Exit 2	18.3%	5.10	4.54
	Exit 3	33.3%	9.40	7.89
	Avg	-	4.95	4.34

TABLE 5.6: Per Sample Energy Consumption (mJ) of Early-Exit Dynamic ResNet-32 over Different Quantisation Levels

	4-bit	5-bit	6-bit	7-bit	8-bit
Pipeline	90.60	73.25	70.78	70.89	72.00
Parallel	105.37	86.60	84.63	84.78	87.58

5.3 Confidence Controlled Dynamic Quantisation of Early-Exit Networks

Quantisation has been acknowledged as an effective technique for reducing the inference workload of DNNs, especially when they are deployed on FPGAs, where arbitrary resolution levels can be applied. To some extent, this applies to early-exit dynamic DNNs, too; however, following the results of the previous section, it can be noted that it cannot achieve the same levels of reduction in energy demands and latency as when it is applied to static DNNs. The main reason behind that is the extent of the effect quantisation has on the early-exiting behaviour, significantly altering the exit trigger rates of the dynamic network. The limitation presented by the post-training quantisation technique is that it quantises the DNN based on the static distribution of the trained parameters. This simplifies the quantisation scheme but makes the solution fixed, disregarding the input sample characteristics and failing to dynamically capture their sensitivity and variability.

On the contrary, the main aim of the intermediate classifiers is to do exactly that. Their purpose, as explained in Chapter 2, is to extract information from the input samples and dynamically allocate computations based on their characteristics. As explained above, their output is a metric of the amount of abstracted information the networks have achieved until that execution stage. Following the results presented in Section 5.1, it is argued that input samples that displayed high confidence levels are more resilient to lower representation resolutions. This section aims to implement a dynamic quantisation scheme for early-exit Dynamic DNNs, utilising the information from the intermediate classifiers in order to efficiently reduce the computational burden, power demands, and memory needs of the network.

5.3.1 Methodology

To implement the dynamic quantisation scheme, the early-exit ResNet-32 Dynamic DNN presented in section 4.2.1.1 is deployed. The network is initially trained normally employing float-point 32-bit arithmetic representation. Then, the range of values that each layer requires is calculated by collecting statistics, as explained in Section 5.2.1, in order to apply post-training quantisation for all the presented bit-widths. Following that, the dynamic network starts executing with full precision up to the first intermediate classifier. If its output is above the decision threshold, then an early exit occurs, and inference restarts with the next input sample. If it's not, then similarly to Section 5.1, the δ is calculated, and its value is assessed. When it is low, then the succeeding part of the network until the next intermediate classifier, is quantised with lower representation resolutions. On the contrary, quantisation is either applied with higher formats or not at all (Figure 5.7). The basis of this approach lies in the fact that when the input confidence value is close to the decision threshold, the network could abstract a considerable amount of information and is less affected by the loss in representation power of lower resolution. On the contrary, when their difference is higher, the network was not able to identify the key characteristics of the input sample, and more information must be preserved for the continuation of the inference.

Similarly to Figure 5.1, Figure 5.8 shows the number of input samples that do not trigger an exit at the intermediate classifiers (Exit 1 or Exit 2), and their δ values were less or equal to 0.1, 0.2 and 0.3. A small amount of input samples satisfying these requirements would render the approach ineffective, as the application of quantisation would happen very rarely. However, the Figure 5.8 illustrates that even for samples whose confidence was at most 0.1 below the threshold, their numbers were almost 1500 for the first exit and more than 700 for the second. Considering that the exit trigger rate for the first exit point is 48.4% and for the second 18.3%, these numbers represent 27.5% and

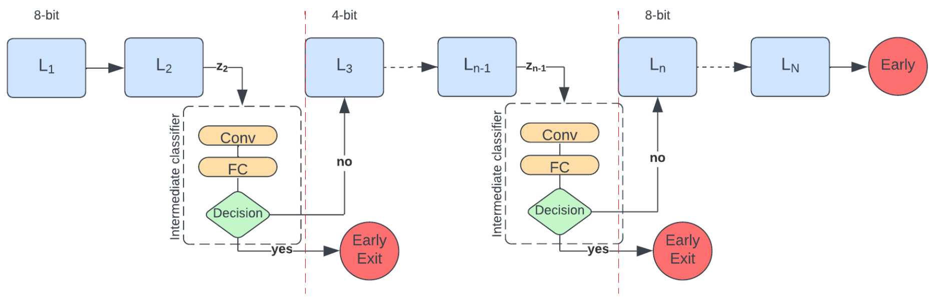


FIGURE 5.7: Dynamic Quantisation Scheme for Early-Exit Dynamic DNNs

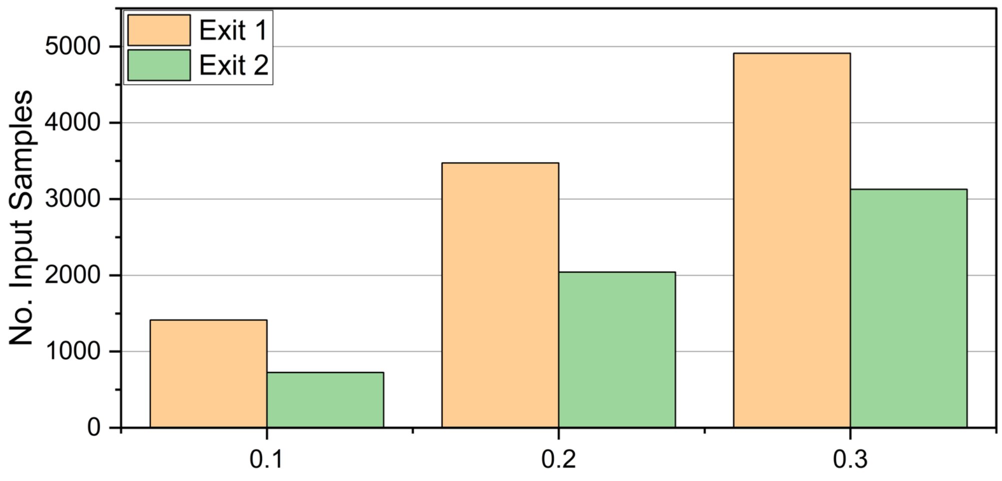


FIGURE 5.8: The number of inputs for which $\delta \leq 0.1$, $\delta \leq 0.2$ and $\delta \leq 0.3$ at Exit 1 and Exit 2.

22% of the rejected input samples, respectively. A further increase in these numbers is noticed when the δ limits are increased. These results further validate the applicability of dynamic quantisation.

Following that, Figure 5.9, Figure 5.10 and Figure 5.11 illustrate the Exit Trigger Rates of the dynamic quantised early-exit ResNet-32 when $\delta \leq 0.1$, $\delta \leq 0.2$ and $\delta \leq 0.3$ respectevely. Comparing these results with the ones presented in the previous section, it is visible that in this case, with the exception of the 4-bit representation resolution, the rates are very close to the ones achieved when the fp32 format is used. This indicates that the dynamic quantisation scheme has a considerably reduced effect on the early-exiting behaviour. The 4-bit resolution's limited representation capabilities are still insufficient to hold the necessary information for the network's accurate inference, thereby completely disabling the second exit point. Nonetheless, the rest of the representation bit-widths achieved exit rates never deviating more than 7% (5-bit when $\delta \leq 0.3$) from the results achieved with fp32.

This is depicted in the accuracy results, too (seen below the bit-widths). The reduced impact on the early-exiting behaviour leads to the minimisation of the negative effects,

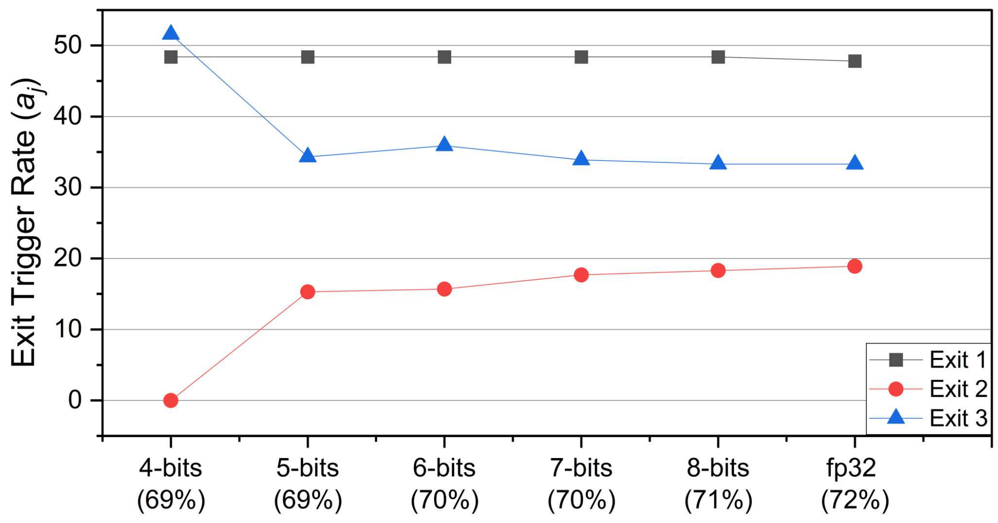


FIGURE 5.9: Percentage of samples to trigger each exit point over different dynamic quantisation levels when $\delta \leq 0.1$.

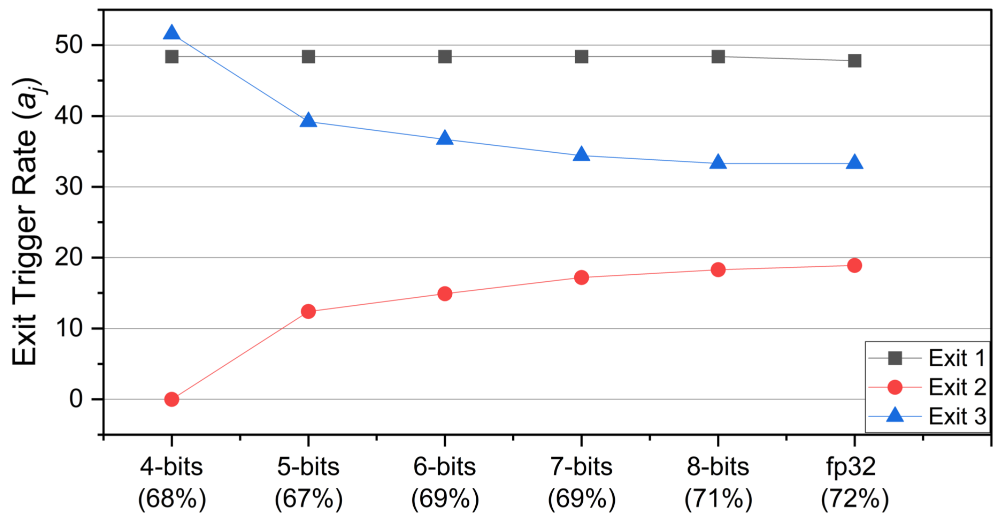


FIGURE 5.10: Percentage of samples to trigger each exit point over different dynamic quantisation levels when $\delta \leq 0.2$.

discussed in the previous Chapter, on the accuracy of the network, too. It is noticeable that the less the difference between the confidence of an input sample and the threshold, the more resilient the network to quantisation. Figure 5.9 results are both closer regarding exit trigger rates and accuracy of the quantised networks over the one using full resolution. This gives the opportunity to use different quantisation levels based on the confidence of each input sample.

To explore the proposed dynamic quantisation scheme and evaluate its effectiveness, the proposed network utilises 5-bit quantisation for samples achieving $\delta \leq 0.1$ and 6-bit for samples achieving $0.1 < \delta \leq 0.2$. Results whose δ value is higher than 0.2 are executed using full precision, which in this occasion is 8 bits. It is worth noting that the experimental results are gathered by combining the performance results of three

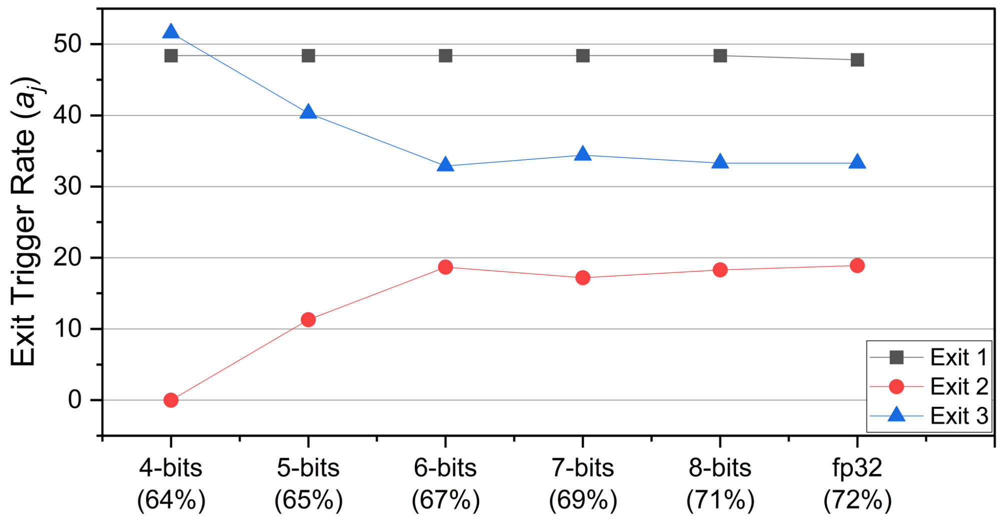


FIGURE 5.11: Percentage of samples to trigger each exit point over different dynamic quantisation levels when $\delta \leq 0.3$.

TABLE 5.7: Per Sample Execution Time (ms) of Early-Exit Dynamic ResNet-32 with and without applying dynamic quantisation

	ResNet-32				ResNet-32 (dynamic quantisation)			
	Exit 1 (48.4%)	Exit 2 (18.3%)	Exit 3 (33.3%)	Avg	Exit 1 (48.4%)	Exit 2 (16.2%)	Exit 3 (35.4%)	Avg
Pipeline	1.84	5.1	9.4	4.95	1.84	3.72	6.86	3.92
Parallel	1.84	4.54	7.89	4.01	1.84	3.22	5.60	3.39

designs with different quantisation levels. While this approach does not consider any dynamic reconfiguration costs, which is necessary to deploy the proposed scheme, it can approximate and highlight its potential benefits.

5.3.2 Experimental Results and Analysis

Table 5.7 shows the per sample execution time in ms of the early-exit ResNet-32 dynamic DNN with and without the application of the dynamic quantisation. The first thing to be noticed is that the effects of dynamic quantisation on the exit trigger rate of each exit point are affected very slightly. Obviously, the first exit point maintains the same levels of samples exiting, as the full precision is used in both cases (8 bits). The next two exits showcase some deviation, which, however, is not more than 2.1%. Having the need to use an 8-bit representation format indicates that the benefits of quantisation in memory needs cannot apply; however, focusing on the execution time results, it is visible that the quantisation scheme achieves faster execution in both architecture approaches. More specifically, the quantised network achieves a reduction in latency by 20.8% and 21.9% on the ***pipeline*** and on the ***parallel*** approach, respectively. Finally, the achieved accuracy of the network is 70%, the same as when 8-bit representation is deployed.

Overall, the dynamic quantisation scheme approach, utilising the confidence levels of input samples, managed to leverage the benefits of quantisation without compromising either the early-exiting behaviour or the accuracy of the network. Clearly, disregarding the device's reconfiguration cost cannot provide accurate energy and latency results; however, considering that the deployment of the ResNet-32 network was already orchestrated by the host processor, the latency results still provide a great approximation of the benefits of dynamic quantisation.

5.4 Discussion

In this Chapter, two dynamic approaches for the execution of the early-exit dynamic DNNs on FPGAs were proposed. The key aspect in both of them was utilising, the confidence level of input samples that did not trigger an early-exit. Utilizing a δ value, which measures the difference between their confidence level and the decision threshold, the two approaches achieved, exploiting the versatility of the FPGA platform, to further enhance the performance of the early-exit dynamic DNNs.

In more detail, the first approach presents a dynamic placing scheme for the intermediate classifiers. Noting that densely placed classifiers increase the computational overhead, while sparse placement can lead to the unnecessary computation of the backbone network's layers, a first adaptive placement scheme is presented. Employing the δ value, it dynamically places intermediate classifiers closer to the already executed ones if the rejected sample's confidence is high ($\delta \leq 0.16$) and deeper into the backbone network when it is lower. In this manner, it increased the adaptability of the network to each input sample's characteristics and achieved an 18% reduction in computation demands without sacrificing accuracy.

The second proposed approach focuses on quantisation. Quantisation is a widely used technique to reduce the computational and memory needs of a DNN, especially when deployed on FPGAs due to their ability to use custom representations. It was proven that while it can reduce the overhead introduced by the intermediate classifier when quantisation is applied undirected to the entirety of the dynamic network, it has significant detrimental effects on accuracy and early-exiting behaviour. Aiming to solve that, a first confidence-controlled quantisation scheme is proposed. Unlike conventional static quantization, which is fixed and does not consider input variability, this approach adjusts precision based on δ. For inputs with confidence close to the threshold ($\delta \leq 0.2$), lower precision is used, while on the contrary, the resolution remains unchanged. Experiments with a dynamically quantized early-exit ResNet-32 showed that while quantization slightly affects early-exit rates, it significantly reduces latency (by 21%) and computational costs without substantial accuracy loss. Both approaches demonstrated enhancements in the performance of early-exit dynamic networks. However, these results cannot be achieved without the capabilities of the FPGAs to deploy custom architectures, data-fows and representation resolutions.

Chapter 6

Conclusions & Future Work

6.1 Conclusions

Deep Neural Networks (DNNs) are the cornerstone of modern AI technologies. Their multi-layer architectures have revolutionised how machines learn and interpret complex data. Although they have showcased state-of-the-art performance/accuracy in numerous application domains, this comes at a cost due to the sheer volume of computational and memory requirements. Enabling their deployment on edge devices, like smartphones, IoT devices, etc., is vital to increasing their availability and leveraging their capabilities in everyday life and real-time applications. However, these devices have considerable resource restrictions and mandate the compression of the DNNs to accommodate their execution. Dynamic DNNs are a novel approach that aims to achieve that by strategically distributing the computational workload based on their ability to modify network structure and parameters during inference.

Conducted research on the structure and the parameters of neural networks provides a broad field of approaches to the dynamic execution of DNNs. By layer skipping, selective execution, dynamic pruning, parameter adjusting, etc., dynamic models have been proven more efficient and effective than their static equivalents. Nonetheless, their practical implementations and exploration are restricted to conventional CPU-GPU systems, which can only theoretically highlight their capabilities. This constituted the main point of interest for the conducted research presented on this thesis. In more detail, the thesis focuses on exploring and addressing the challenges of realising Dynamic DNNs on resource-restricted devices, aiming to contribute towards moving the execution of modern DNNs to the edge. FPGAs were selected as it is a platform that has already been proven very effective in implementing and accelerating DNNs. Their reprogrammability, parallelisation capabilities and low energy demands render them ideal for exploring and evaluating the capabilities of the dynamic approaches.

Despite their benefits, however, dynamic DNN approaches introduce new components (e.g. intermediate classifiers in early-exit networks) and change the architecture of the neural network and the inference data flow. These complexities require additional computational overhead and sophisticated control logic, raising many challenges in optimisation and hardware realisation. Through the exploration of these challenges (3) three main limiting factors were identified: the lack of software libraries and frameworks tailored for the needs of dynamic networks' execution and training, the lack of hardware modules and frameworks to support the architectural characteristics of dynamic DNNs and, focusing on early-exit networks, the dependencies on intermediate feature maps. These findings answered the first research question (**Q1**) and highlighted the fact that despite their positive attributes, realising dynamic DNNs on hardware is a novel and difficult task, with limitations that can affect not only the ease of design but also the performance of the architecture itself.

A first FPGA implementation of an early-exit dynamic DNN (***pipeline*** approach) achieved comparable latency to a CPU/GPU system while maintaining very low power consumption, requiring 4.8mJ per sample for the execution of a LeNet early exit network (48% less than a Jetson Xavier embedded device). The effectiveness of dynamic DNNs was also illustrated as the FPGA architecture required 70% less energy when the dynamic network was executed. Overall, the proposed design highlighted the FPGA platform's capability to efficiently deploy an early-exit dynamic network, maintaining very low power needs and minimal effects on accuracy despite the limitations presented by the dynamic architecture limitations.

Even though this approach showcased the benefits of implementing an early-exit dynamic network on FPGAs, its performance was afflicted by the need to halt the execution of the backbone network and store the intermediate feature maps every time the intermediate classifiers were executed. It is also noted that it didn't fully utilised the parallelization capabilities of the FPGA platform. These limitations were addressed by proposing a second design approach (***parallel*** approach), which explored the parallel execution of the backbone network and intermediate classifiers. Performing that eliminates the latency overhead introduced by the intermediate classifiers, especially for input samples that do not trigger an early exit, and accelerates the execution of the dynamic network by up to 23% over the ***pipeline*** design approach. These results answered the second research question (**Q2**) and illustrated how the FPGA parallelization capabilities could address some of the limitations introduced by the early-exit networks' architecture and accelerate their execution. Overall through both approaches it was proven that FPGAs are a very capable platform that can realise and enhance the performance of dynamic DNNs, further accelerating and reducing the energy demands of modern computer vision applications.

It is worth noting that throughout the exploration of early-exit network, the effects of the dynamic approach in energy and computational demands are validated and highlighted. However, it could be argued that they are very class and data dependent and could possibly not scale well under unseen data during validation or testing, especially when important layers, e.g. the last, are not executed. This could be the case when the undirected reduction of layers or filters (e.g. pruning); however, in the dynamic approaches, the intermediate classifiers that control the amount of executed layers are trained alongside the backbone network. In this way, they learn to identify the information abstracted from the backbone network and not bluntly classify the current feature map. As long as the backbone network is trained correctly to generalise on unknown information, then the early-exit approach will not affect the accuracy of the network.

An additional notable characteristic of early-exit dynamic DNNs is that to enable dynamic execution, they employ intermediate classifiers, which unavoidably introduce computational overhead. It was also noted that despite their output being a value of the amount of abstracted information the network could abstract from the input samples, it is only utilised to trigger an early exit, and otherwise, it's completely disregarded. Following these observations, two confidence-controlled dynamic schemes achieved to utilise the versatility and reprogrammability of FPGAs and enhance the performance of the early-exit dynamic DNNs, answering the last research question (**Q3**). The core of both dynamic approaches is the importance of the difference between the confidence level of input samples that didn't trigger an early exit and the decision threshold (δ). The first approach focuses on the placement of the intermediate classifiers employing the versatility of the FPGAs and the ***parallel*** design and achieved to reduce the computational burden of an early-exit network by 18% and its energy consumption and latency by 5.8%. The second proposed dynamic scheme focuses on the quantisation of the dynamic network using δ and the ability of FPGAs to accommodate various representation levels. It achieved a minimum of 20.8% reduction in the latency of the early-exit network with minimal implications on accuracy.

In conclusion, this thesis first and foremost showcased the capabilities and benefits of using FPGAs in order to realise dynamic DNNs. The versatility and reconfigurability of FPGAs where able to help create architectures that adapt to the characteristics of early-exiting and enhance their performance, both in terms of energy consumption and computational load. Nonetheless, as it was identified in Chapter 3, there is a lot room for improvement, both in terms of hardware architecture and in terms of dynamic networks design.

6.2 Future Work

This section outlines potential future research directions for extending and improving the work in this thesis.

6.2.1 Quantisation-Aware Training

Chapter 5 explored the benefits of quantisation on early-exit dynamic DNNs, employing a post-training quantisation methodology. It is a technique simple to apply, requiring little additional computational effort and has proven to be effective in reducing the memory demands and latency of a dynamic network. Nonetheless, it was also highlighted, especially in the uncontrolled full network quantisation case, that the reduced representation power has detrimental effects on the early-exiting behaviour and the accuracy of the network, rendering its application useless. To resolve that, future work can explore Quantisation-Aware Training methodologies, which consider the effects of quantisation during the training process. It is substantially more complex to apply, as it requires the fabrication and fine-tuning of complex training schemes that integrate the effects of quantisation, the sensitivity of the layers and the behavioural characteristics of the network. Achieving that, however, could minimise the negative effects quantisation has in the accuracy and early-exiting behaviour and further enhance the performance of early-exit dynamic networks by efficiently quantising them with even lower bit-widths.

6.2.2 Introduce Dynamic Partial Reconfiguration to Dynamic DNN accelerators

Section 5.3 introduced a dynamic quantisation scheme that uses the input samples' confidence to dynamically change the quantisation level of the early-exit network in between two exit points. Performing that mandates the existence of multiple bit-width designs that are available to be deployed at any time. While the followed approach (Section 5.3.1 can highlight the benefits of the proposed dynamic quantisation scheme, it disregards the costs of the dynamic reconfiguration of the FPGA. Introducing Dynamic Partial Reconfiguration to dynamically change the bit-width of the dynamic network on a single hardware design can lead to more accurate results.

In addition to that, Section 2.5.1 highlighted the diversity among the computational needs of different layers of a single DNN. This diversity can be found in the early-exit networks too. Introducing Dynamic Partial Reconfiguration to their accelerators can increase the adaptability of the hardware to their characteristics, and further increase their performance.

6.2.3 Employ NAS techniques to better explore the FPGA design space

Exploring the realization of early-exit networks in FPGAs has revealed a significant interaction between their components. Chapter 4 highlighted the effects of the decision threshold on the performance of the proposed FPGA designs, while Chapter 5 illustrated how the location of the intermediate classifiers, or the quantisation level applied to an early-exit network can alter its efficiency and accuracy. Consequently, it is beneficial to jointly optimize various design and configuration parameters such as the number of exits, their placement, architecture, and decision policies. This expands the architectural search space, making exploring and identifying optimal configurations computationally demanding. Neural Architecture Search (NAS) methods may offer an effective solution to navigate this space efficiently.

Appendix A

Additional Figures

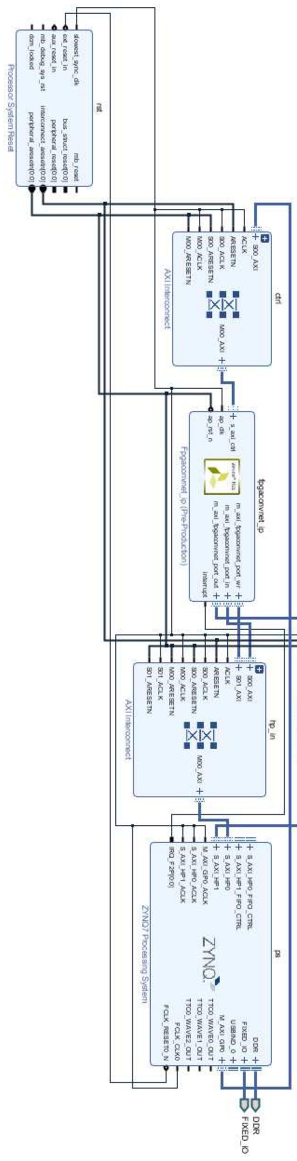


FIGURE A.1: LeNet-5 final design for Xilinx's ZC702 board

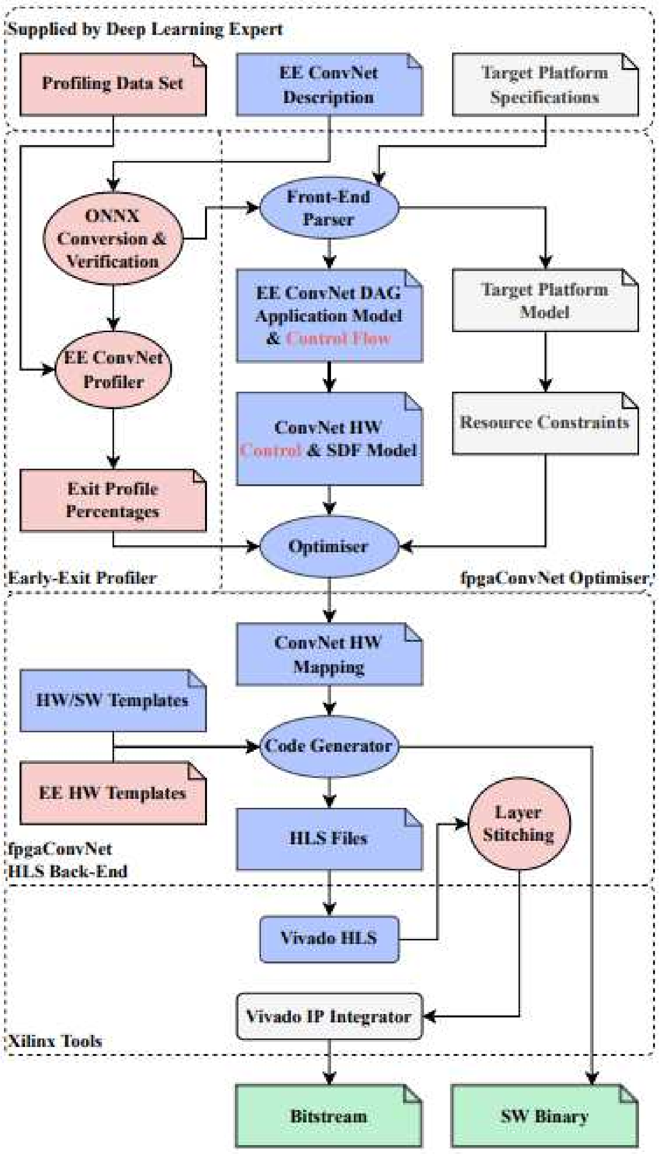


FIGURE A.2: ATHEENA processing flow. Reprinted from B. Biggs et.al. [121]

Bibliography

[1] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *Communications of the ACM*, vol. 60, pp. 84–90, 2012.

[2] K. He, X. Zhang, S. Ren, and J. Sun, "Deep residual learning for image recognition," *2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 770–778, 2015.

[3] C. Shan, J. Zhang, Y. Wang, and L. Xie, "Attention-based end-to-end speech recognition on voice search," *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 4764–4768, 2017.

[4] A. Vaswani, N. M. Shazeer, N. Parmar, J. Uszkoreit, *et al.*, "Attention is all you need," in *Neural Information Processing Systems*, 2017.

[5] G. Huang, Z. Liu, and K. Q. Weinberger, "Densely connected convolutional networks," *2017 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2261–2269, 2016.

[6] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," *CoRR*, 2014.

[7] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, *et al.*, "Going deeper with convolutions," *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 1–9, 2014.

[8] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," *CoRR*, 2016.

[9] H. Liu, K. Simonyan, and Y. Yang, "DARTS: Differentiable architecture search," in *International Conference on Learning Representations*, 2019.

[10] OpenAI, *Chatgpt: Conversational ai model*, Available at `https://chat.openai.com`, Accessed: 6 Mar. 2025, 2024.

[11] J. Koetsier, "Chatgpt burns millions every day: Can computer scientists make ai one million times more efficient?" *Forbes*, 2023.

[12] A. Mok, "How much does chatgpt cost to run? $700k/day, per analyst," *Business Insider*, 2023.

[13] B. Steiner, M. Elhoushi, J. Kahn, and J. Hegarty, *Olla: Optimizing the lifetime and location of arrays to reduce the memory usage of neural networks*, 2022. arXiv: `2210.12924 [cs.LG]`.

[14] Z. Zhou, X. Chen, E. Li, L. Zeng, *et al.*, "Edge intelligence: Paving the last mile of artificial intelligence with edge computing," *Proceedings of the IEEE*, vol. 107, pp. 1738–1762, 2019.

[15] H. Cai, J. Lin, Y. Lin, Z. Liu, *et al.*, "Enable deep learning on mobile devices: Methods, systems, and applications," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 27, pp. 1–50, 2022.

[16] T. Zhao, Y. Xie, Y. Wang, J. Cheng, *et al.*, "A survey of deep learning on mobile devices: Applications, optimizations, challenges, and research opportunities," *Proceedings of the IEEE*, vol. 110, pp. 334–354, 2022.

[17] Y. Han, X. Wang, V. C. M. Leung, D. T. Niyato, *et al.*, "Convergence of edge computing and deep learning: A comprehensive survey," *IEEE Communications Surveys & Tutorials*, vol. 22, pp. 869–904, 2019.

[18] C.-J. Wu, D. Brooks, K. Chen, D. Chen, *et al.*, "Machine learning at facebook: Understanding inference at the edge," Feb. 2019, pp. 331–344.

[19] Y. Hu, S. Liu, T. F. Abdelzaher, M. B. Wigness, *et al.*, "Real-time task scheduling with image resizing for criticality-based machine perception," *Real-Time Systems*, vol. 58, pp. 430–455, 2022.

[20] C. Shi, L. Chen, C. Shen, L. Song, *et al.*, "Privacy-aware edge computing based on adaptive dnn partitioning," *2019 IEEE Global Communications Conference (GLOBECOM)*, pp. 1–6, 2019.

[21] S. I. Venieris, M. Almeida, R. Lee, and N. D. Lane, "Nawq-sr: A hybrid-precision npu engine for efficient on-device super-resolution," *IEEE Transactions on Mobile Computing*, vol. 23, pp. 2367–2381, 2022.

[22] M. Qasaimeh, K. Denolf, J. Lo, K. Vissers, *et al.*, "Comparing energy efficiency of cpu, gpu and fpga implementations for vision kernels," in *2019 IEEE International Conference on Embedded Software and Systems (ICESS)*, 2019, pp. 1–8.

[23] S. Huang, E. Tang, S. Li, X. Ping, *et al.*, "Hardware-friendly compression and hardware acceleration for transformer: A survey," *Electronic Research Archive*, vol. 30, pp. 3755–3785, 2022.

[24] C. Yuan and S. S. Agaian, "A comprehensive review of binary neural network," *Artificial Intelligence Review*, pp. 1–65, 2021.

[25] P. Molchanov, S. Tyree, T. Karras, T. Aila, *et al.*, "Pruning convolutional neural networks for resource efficient inference," in *5th International Conference on Learning Representations, ICLR*, 2017.

[26] G. Hinton, O. Vinyals, and J. Dean, "Distilling the knowledge in a neural network," in *Proceedings of the NIPS Deep Learning and Representation Learning Workshop*, 2015.

[27] G. Huang, D. Chen, T. Li, F. Wu, *et al.*, "Multi-scale dense networks for resource efficient image classification," in *International Conference on Learning Representations*, 2017.

[28] Z. Chen, Y. Li, S. Bengio, and S. Si, "You look twice: Gaternet for dynamic filter selection in cnns," *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 9164–9172, 2018.

[29] L. Yang, Z. He, Y. Cao, and D. Fan, "Non-uniform DNN structured subnets sampling for dynamic inference," in *57th ACM/IEEE Design Automation Conference (DAC)*, San Francisco, CA, USA, Jul. 2020, pp. 1–6, 2020.

[30] B. Yang, G. Bender, Q. V. Le, and J. Ngiam, "Soft conditional computation," *CoRR*, 2019. arXiv: `1904.04971`.

[31] Y. Chen, X. Dai, M. Liu, D. Chen, *et al.*, "Dynamic convolution: Attention over convolution kernels," *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 11 027–11 036, 2019.

[32] Z. Xu, F. Yu, C. Liu, and X. Chen, "Reform: Static and dynamic resource-aware dnn reconfiguration framework for mobile device," *56th ACM/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2019.

[33] J. Shen, Y. Fu, Y. Wang, P. Xu, *et al.*, "Fractional skipping: Towards finer-grained dynamic cnn inference," in *AAAI Conference on Artificial Intelligence*, 2020.

[34] M. Putic, S. Venkataramani, S. Eldridge, A. Buyuktosunoglu, *et al.*, "Dyhard-dnn: Even more dnn acceleration with dynamic hardware reconfiguration," *2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC)*, pp. 1–6, 2018.

[35] R. Xu, S. Ma, Y. Wang, Y. Guo, *et al.*, "HeSA: Heterogeneous systolic array architecture for compact cnns hardware accelerators," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 11, pp. 2860–2871, 2022.

[36] Y. Han, G. Huang, S. Song, L. Yang, *et al.*, "Dynamic neural networks: A survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 44, pp. 7436–7456, 2021.

[37] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceeding of the IEEE*, vol. 86, pp. 2278–2324, 1998.

[38] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, pp. 1735–1780, 1997.

[39] M. Merenda, C. Porcaro, and D. Iero, "Edge machine learning for ai-enabled iot devices: A review," *Sensors (Basel, Switzerland)*, vol. 20, 2020.

[40] Y. LeCun, B. E. Boser, J. S. Denker, D. Henderson, *et al.*, "Handwritten digit recognition with a back-propagation network," in *Neural Information Processing Systems*, 1989.

[41] Theano Development Team, *Theano: A python framework for fast computation*, Available at https://theano-pymc.readthedocs.io/en/latest/, Accessed: 6 Mar. 2025, 2024.

[42] TensorFlow Developers, *Tensorflow: An open source machine learning framework*, Available at https://www.tensorflow.org/, Accessed: 6 Mar. 2025, 2024.

[43] S. Ruder, "An overview of gradient descent optimization algorithms," *ArXiv*, 2016.

[44] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," in *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, Eds., 2015.

[45] TensorFlow Developers, *Tensorflow lite: Lightweight machine learning for mobile and edge devices*, Available at https://www.tensorflow.org/lite, Accessed: 6 Mar. 2025, 2024.

[46] BLAS Developers, *Basic linear algebra subprograms (blas)*, Available at http://www.netlib.org/blas/, Accessed: 6 Mar. 2025, 2024.

[47] Y. Xiang and H. Kim, "Pipelined data-parallel cpu/gpu scheduling for multi-dnn real-time inference," *2019 IEEE Real-Time Systems Symposium (RTSS)*, pp. 392–405, 2019.

[48] NVIDIA Corporation, *Nvidia cudnn: Gpu-accelerated deep learning*, Available at https://developer.nvidia.com/cudnn, Accessed: 6 Mar. 2025, 2024.

[49] E. Eshelman, *Nvidia tesla p100 price analysis*, Available at https://www.microway.com/hpc-tech-tips/nvidia-tesla-p100-price-analysis/, Accessed: 6 Mar. 2025, 2024.

[50] S. Han, J. Kang, H. Mao, Y. Hu, *et al.*, "Ese: Efficient speech recognition engine with sparse lstm on fpga," *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017.

[51] Y. Chen, Y. Xie, L. Song, F. Chen, *et al.*, "A survey of accelerator architectures for deep neural networks," *Engineering*, vol. 6, pp. 264–274, 2020.

[52] J.-s. Seo, J. Saikia, J. Meng, W. He, *et al.*, "Digital versus analog artificial intelligence accelerators: Advances, trends, and emerging designs," *IEEE Solid-State Circuits Magazine*, vol. 14, pp. 65–79, Jul. 2022.

[53] Apple, "Deploying transformers on the apple neural engine," *Machine Learning Research*, 2022.

[54] Microsoft, "All about neural processing units (npus)," *Microsoft Support*, 2024.

[55] S. Han, X. Liu, H. Mao, J. Pu, *et al.*, "Eie: Efficient inference engine on compressed deep neural network," *ACM/IEEE 43rd Annual International Symposium on Computer Architecture (ISCA)*, pp. 243–254, 2016.

[56] A. Sabne, "Xla: Compiling machine learning for peak performance," *Google Res*, 2020.

[57] N. P. Jouppi, C. Young, N. Patil, D. Patterson, *et al.*, "In-datacenter performance analysis of a tensor processing unit," *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pp. 1–12, 2017.

[58] Google Cloud, *Cloud tpu: Scalable accelerators for machine learning*, Available at `https://cloud.google.com/tpu`, Accessed: 6 Mar. 2025, 2024.

[59] N. Jouppi, C. Young, N. Patil, and D. Patterson, "Motivation for and evaluation of the first tensor processing unit," *IEEE Micro*, vol. 38, pp. 10–19, May 2018.

[60] Microsoft Research, *Microsoft's project brainwave: Real-time ai processing*, Available at `https://www.microsoft.com/en-us/research/project/project-catapult/`, Accessed: 6 Mar. 2025, 2024.

[61] S. K. Rao and T. Kailath, "Regular iterative algorithms and their implementation on processor arrays," *Proceedings of the IEEE*, vol. 76, pp. 259–269, 1988.

[62] J. Zhang, H. Gu, G. L. Zhang, B. Li, *et al.*, "Hardware-software codesign of weight reshaping and systolic array multiplexing for efficient cnns," *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pp. 667–672, 2021.

[63] "Vivado Design Suite User Guide: Partial Reconfiguration (UG909)," en, p. 151, 2019.

[64] S. I. Venieris, A. Kouris, and C.-S. Bouganis, "Toolflows for mapping convolutional neural networks on fpgas," *ACM Computing Surveys (CSUR)*, vol. 51, pp. 1–39, 2018.

[65] "Intel® Hyperflex™ Architecture High-Performance Design Handbook," en, p. 148,

[66] *Nvidia nvdla [online].* Available: `http://nvdla.org/`.

[67] S. I. Venieris and C.-S. Bouganis, "Fpgaconvnet: Mapping regular and irregular convolutional neural networks on fpgas," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 30, pp. 326–342, 2019.

[68] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, *et al.*, "Caffe: Convolutional architecture for fast feature embedding," *Proceedings of the 22nd ACM international conference on Multimedia*, 2014.

[69] R. Collobert, K. Kavukcuoglu, and C. Farabet, "Torch7: A matlab-like environment for machine learning," in *Neural Information Processing Systems*, 2011.

[70] M. Hall and V. Betz, "Hpipe: Heterogeneous layer-pipelined and sparse-aware cnn inference for fpgas," *Proceedings of the 2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2020.

[71] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, *et al.*, "FINN: A framework for fast, scalable binarized neural network inference," *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2016.

[72] J. Ngadiuba, V. Loncar, M. Pierini, S. Summers, *et al.*, "Compressing deep neural networks on fpgas to binary and ternary precision with hls4ml," *Machine Learning: Science and Technology*, 2020.

[73] A. Vieira, F. Pratas, L. Sousa, and A. Ilic, "Accelerating cnn computation: Quantisation tuning and network resizing," in *Proceedings of the 2nd Workshop on AutotuniNg and ADaptivity AppRoaches for Energy Efficient HPC Systems*, Limassol, Cyprus: Association for Computing Machinery, 2018.

[74] Q. Jin, L. Yang, and Z. A. Liao, "Adabits: Neural network quantization with adaptive bit-widths," *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 2143–2153, 2019.

[75] N. Liu, X. Ma, Z. Xu, Y. Wang, *et al.*, "Autocompress: An automatic dnn structured pruning framework for ultra-high compression rates," in *AAAI Conference on Artificial Intelligence*, 2019.

[76] W. Niu, X. Ma, S. Lin, S. Wang, *et al.*, "Patdnn: Achieving real-time dnn execution on mobile devices with pattern-based weight pruning," *Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, 2020.

[77] X. Ma, S. Lin, S. Ye, Z. He, *et al.*, "Non-structured dnn weight pruning—is it beneficial in any platform?" *IEEE Transactions on Neural Networks and Learning Systems*, vol. 33, pp. 4930–4944, 2019.

[78] L. Pradels, S.-I. Filip, O. Sentieys, D. Chillet, *et al.*, "Fpga-based cnn acceleration using pattern-aware pruning," in *2024 IEEE 6th International Conference on AI Circuits and Systems (AICAS)*, 2024, pp. 228–232.

[79] C. Guo, B. Y. Hsueh, J. Leng, Y. Qiu, *et al.*, "Accelerating sparse dnn models without hardware-support via tile-wise sparsity," *SC20: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–15, 2020.

[80] E. Park, D. Kim, S. Kim, Y.-D. Kim, *et al.*, "Big/little deep neural network for ultra low power inference," *2015 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pp. 124–132, 2015.

[81] S. Teerapittayanon, B. McDanel, and H. T. Kung, "Branchynet: Fast inference via early exiting from deep neural networks," *2016 23rd International Conference on Pattern Recognition (ICPR)*, pp. 2464–2469, 2016.

[82] A. Sabet, J. S. Hare, B. M. Al-Hashimi, and G. V. Merrett, "Temporal early exits for efficient video object detection," *ArXiv*, 2021.

[83] X. Wang, F. Yu, Z.-Y. Dou, T. Darrell, *et al.*, "Skipnet: Learning dynamic routing in convolutional networks," in *The European Conference on Computer Vision (ECCV)*, Sep. 2018.

[84] Z. Wu, T. Nagarajan, A. Kumar, S. J. Rennie, *et al.*, "Blockdrop: Dynamic inference paths in residual networks," *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 8817–8826, 2017.

[85] L. Liu and J. Deng, "Dynamic deep neural networks: Optimizing accuracy-efficiency trade-offs by selective execution," in *Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence and Thirtieth Innovative Applications of Artificial Intelligence Conference and Eighth AAAI Symposium on Educational Advances in Artificial Intelligence*, ser. AAAI'18/IAAI'18/EAAI'18, New Orleans, Louisiana, USA: AAAI Press, 2018.

[86] Z. Yan, H. Zhang, R. Piramuthu, V. Jagadeesh, *et al.*, "Hd-cnn: Hierarchical deep convolutional neural networks for large scale visual recognition," *2015 IEEE International Conference on Computer Vision (ICCV)*, pp. 2740–2748, 2014.

[87] D. Jiang, W. Zou, S. Zhao, G. Yang, *et al.*, "An analysis of decoding for attention-based end-to-end mandarin speech recognition," in *2018 11th International Symposium on Chinese Spoken Language Processing (ISCSLP)*, 2018, pp. 384–388.

[88] R. T. Mullapudi, W. R. Mark, N. M. Shazeer, and K. Fatahalian, "Hydranets: Specialized dynamic architectures for efficient inference," *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 8080–8089, 2018.

[89] J. Lin, Y. Rao, J. Lu, and J. Zhou, "Runtime neural pruning," in *Neural Information Processing Systems*, 2017.

[90] X. Jia, B. De Brabandere, T. Tuytelaars, and L. V. Gool, "Dynamic filter networks," in *Advances in Neural Information Processing Systems*, D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, *et al.*, Eds., vol. 29, Curran Associates, Inc., 2016.

[91] J. Xin, R. Tang, J. Lee, Y. Yu, *et al.*, "Deebert: Dynamic early exiting for accelerating bert inference," in *Annual Meeting of the Association for Computational Linguistics*, 2020.

[92] A. Graves, "Adaptive computation time for recurrent neural networks," *ArXiv*, 2016.

[93] M. Elbayad, J. Gu, E. Grave, and M. Auli, "Depth-adaptive transformer," in *International Conference on Learning Representations*, 2020.

[94] A. Veit and S. J. Belongie, "Convolutional networks with adaptive inference graphs," *International Journal of Computer Vision*, vol. 128, pp. 730–741, 2017.

[95] Y. Wang, J. Shen, T.-K. Hu, P. Xu, *et al.*, "Dual dynamic inference: Enabling more efficient, adaptive, and controllable deep inference," *IEEE Journal of Selected Topics in Signal Processing*, vol. 14, pp. 623–633, 2019.

[96] Y. Li, L. Song, Y. Chen, Z. Li, *et al.*, " Learning Dynamic Routing for Semantic Segmentation," in *2020 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, Los Alamitos, CA, USA: IEEE Computer Society, Jun. 2020, pp. 8550–8559.

[97] A. Paszke, S. Gross, F. Massa, A. Lerer, *et al.*, *Pytorch: An imperative style, high-performance deep learning library*, 2019.

[98] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," in *4th International Conference on Learning Representations, ICLR*, 2016.

[99] X. Dai, X. Kong, and T. Guo, "EPNet: Learning to Exit with Flexible Multi-Branch Network," en, in *Proceedings of the 29th ACM International Conference on Information & Knowledge Management*, 2020.

[100] W. Xia, H. Yin, X. Dai, and N. K. Jha, "Fully dynamic inference with deep neural networks," *IEEE Transactions on Emerging Topics in Computing*, vol. 10, pp. 962–972, 2020.

[101] M. Wang, J. Mo, J. Lin, Z. Wang, *et al.*, "DynExit: A dynamic early-exit strategy for deep residual networks," *2019 IEEE International Workshop on Signal Processing Systems (SiPS)*, pp. 178–183, 2019.

[102] B. Fang, X. Zeng, F. Zhang, H. Xu, *et al.*, "Flexdnn: Input-adaptive on-device deep learning for efficient mobile vision," *IEEE/ACM Symposium on Edge Computing (SEC)*, pp. 84–95, 2020.

[103] H. Li, H. Zhang, X. Qi, R. Yang, *et al.*, "Improved techniques for training adaptive deep networks," *2019 IEEE/CVF International Conference on Computer Vision (ICCV)*, pp. 1891–1900, 2019.

[104] H. Hu, D. Dey, M. Hebert, and J. Bagnell, "Learning anytime predictions in neural networks via adaptive loss balancing," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, pp. 3812–3821, 2019.

[105] Y. Kaya and T. Dumitras, "How to stop off-the-shelf deep neural networks from overthinking," *ArXiv*, 2018.

[106] S. Laskaridis, S. I. Venieris, H. Kim, and N. D. Lane, "Hapi: Hardware-aware progressive inference," *2020 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1–9, 2020.

[107] P. S. Foundation, *Python: A dynamic, open source programming language*, Available at `https://www.python.org/`, Accessed: 6 Mar. 2025, 2023.

[108] Cadence Design Systems, *Cadence genus: Synthesis solution*, Available at `https://www.cadence.com`, Accessed: 6 Mar. 2025, 2023.

[109] ARM Ltd., *Arm memory compiler*, Available at https://www.arm.com/products/silicon-ip-design/memory-compilers, Accessed: 6 Mar. 2025, 2024.

[110] F. Kastner, B. Janßen, F. Kautz, M. Hübner, *et al.*, "Hardware/software codesign for convolutional neural networks exploiting dynamic partial reconfiguration on pynq," *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 154–161, 2018.

[111] S. Jiang, Z. Ma, X. Zeng, C. Xu, *et al.*, "SCYLLA: QoE-aware continuous mobile vision with FPGA-based dynamic deep neural network reconfiguration," in *IEEE Conference on Computer Communications*, Toronto, ON, Canada, Jul. 2020, pp. 1369–1378, 2020.

[112] Y. Yang, C. Wang, and X. Zhou, "Work-in-progress: Drama: A high efficient neural network accelerator on fpga using dynamic reconfiguration," in *2019 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2019, pp. 1–2.

[113] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, 1998.

[114] J. Bradbury, R. Frostig, P. Hawkins, M. J. Johnson, *et al.*, *JAX: Composable transformations of Python+NumPy programs*, 2018.

[115] F. Chollet *et al.*, *Keras*, 2015.

[116] T. Akiba, S. Sano, T. Yanase, T. Ohta, *et al.*, "Optuna: A next-generation hyperparameter optimization framework," *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*, 2019.

[117] R. Liaw, E. Liang, R. Nishihara, P. Moritz, *et al.*, "Tune: A research platform for distributed model selection and training," *ArXiv*, vol. abs/1807.05118, 2018.

[118] S. Chen, P. Fegade, T. Chen, P. B. Gibbons, *et al.*, "Ed-batch: Efficient automatic batching of dynamic neural networks via learned finite state machines," in *International Conference on Machine Learning*, 2023.

[119] Xilinx, *Vitis ai: Ai inference acceleration for xilinx fpgas and socs*, Available at https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html, Accessed: 6 Mar. 2025, 2024.

[120] I. Corporation, *Openvino: Open visual inference and neural network optimization*, 2020.

[121] B. Biggs, C.-S. Bouganis, and G. Constantinides, "Atheena: A toolflow for hardware early-exit network automation," in *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2023.

[122] G. Korol, M. G. Jordan, M. B. Rutzig, J. Castrillon, *et al.*, "Pruning and early-exit co-optimization for cnn acceleration on fpgas," in *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2023, pp. 1–6.

[123] B. Jacob, S. Kligys, B. Chen, M. Zhu, *et al.*, "Quantization and training of neural networks for efficient integer-arithmetic-only inference," *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 2704–2713, 2017.

[124] H. T. Kung, "Why systolic architectures?" *Computer*, vol. 15, pp. 37–46, 1982.

[125] A. Samajdar, J. M. Joseph, Y. Zhu, P. N. Whatmough, *et al.*, "A systematic methodology for characterizing scalability of dnn accelerators using scale-sim," *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 58–68, 2020.

[126] J. S. S. T. Association. "Jedec standard for three-dimensional integrated circuits (3dic) stacking." (2012), [Online]. Available: `https://www.jedec.org/standards-documents/docs/jesd-79-3d`.

[127] L. Deng, "The mnist database of handwritten digit images for machine learning research," *IEEE Signal Processing Magazine*, vol. 29, no. 6, pp. 141–142, 2012.

[128] X. Inc., *Vivado design suite*, version 2023.1, Accessed: 6 Mar. 2025, 2023.

[129] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," University of Toronto, Tech. Rep., 2009.

[130] S. Kala, B. R. Jose, J. Mathew, and N. Sivanandan, "High-performance cnn accelerator on fpga using unified winograd-gemm architecture," *IEEE Transactions on VLSI*, 2019.

[131] L. Lu, Y. Liang, Q. Xiao, and S. Yan, "Evaluating fast algorithms for convolutional neural networks on fpgas," *FCCM*, 2017.

[132] C. Yang, Y. Meng, K. Huo, J. Xi, *et al.*, "A sparse cnn accelerator for eliminating redundant computations in intra- and inter-convolutional/pooling layers," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 30, pp. 1902–1915, 2022.

[133] X. Li, H. Huang, T. Chen, H. Gao, *et al.*, "A hardware-efficient computing engine for fpga-based deep convolutional neural network accelerator," *Microelectron. J.*, vol. 128, 2022.

[134] Y. Liang, L. Lu, Y. Jin, J. Xie, *et al.*, "An efficient hardware design for accelerating sparse cnns with nas-based models," *Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2021.

[135] Y. Ma, Y. Cao, S. B. K. Vrudhula, and J.-s. Seo, "Optimizing loop operation and dataflow in fpga acceleration of deep convolutional neural networks," *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2017.

[136] E. Park, D. Kim, and S. Yoo, "Energy-efficient neural network accelerator based on outlier-aware low-precision computation," *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*, pp. 688–698, 2018.

[137] K. Wang, Z. Liu, Y. Lin, J. Lin, *et al.*, "Haq: Hardware-aware automated quantization with mixed precision," *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*, pp. 8604–8612, 2018.

[138] J. L. Holt and J.-N. Hwang, "Finite precision error analysis of neural network hardware implementations," *IEEE Trans. Computers*, vol. 42, pp. 281–290, 1993.