POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

Hardware-aware neural architecture search for quantized neural networks exploration on resource-constrained devices

ROOHOLLAH MOHAMMADZADEH

Département de génie électrique

Mémoire présenté en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*

Génie électrique

Mai 2023

© Roohollah Mohammadzadeh, 2023.

POLYTECHNIQUE MONTRÉAL

affiliée à l'Université de Montréal

Ce mémoire intitulé :

Hardware-aware neural architecture search for quantized neural networks exploration on resource-constrained devices

présenté par **Roohollah MOHAMMADZADEH**

en vue de l'obtention du diplôme de *Maîtrise ès sciences appliquées*

a été dûment accepté par le jury d'examen constitué de :

Yvon SAVARIA, président

Jean Pierre DAVID, membre et directeur de recherche

Pierre LANGLOIS, membre et codirecteur de recherche

Guy BOIS, membre

DEDICATION

To my mom, dad, sisters, and brothers

ACKNOWLEDGEMENTS

I would like to express my deepest gratitude to my supervisor, Prof. Jean Pierre David, for his invaluable guidance, encouragement, and support throughout my research journey. His expertise and insights have been instrumental in shaping this thesis. I will always be thankful for the opportunity he gave me to be in his research group.

I am also very grateful to my co-supervisor, Prof. Pierre Langlois, for his insightful comments and constructive feedback. His mentorship has been invaluable to me. During our research, I have learned a lot of things from him, especially in our weekly meetings. I am deeply thankful for the opportunity to work with him.

I would also like to appreciate my family's unwavering support and encouragement. Their love and belief in me have been a constant source of motivation and inspiration.

Finally, I would like to acknowledge the many others who have supported me, including my professors, colleagues, and mentors. Their advice and friendship have greatly benefited me throughout my academic journey.

RÉSUMÉ

Au cours des dernières années, les réseaux neuronaux profonds (Deep Neural Network - DNN) ont affiché des performances impressionnantes dans divers domaines, tels que la vision par ordinateur, la reconnaissance vocale et le traitement du langage naturel. Cependant, la mise en œuvre et le déploiement des réseaux neuronaux profonds sur des appareils aux ressources limitées est une tâche difficile en raison de leurs exigences de calcul élevées, ce qui entraîne de longs délais d'inférence et une consommation d'énergie accrue. Pour résoudre ce problème, les réseaux neuronaux quantifiés (Quantized Neural Network - QNN) sont apparus comme une solution prometteuse en réduisant la précision des poids et des activations pour les adapter aux dispositifs à ressources limitées. Néanmoins, le processus de quantification entraîne une perte importante d'informations, ce qui réduit la précision. Pour surmonter ce problème, la recherche d'architectures neuronales (Neural Architecture Search - NAS) a été développée pour découvrir des architectures QNN performantes qui maximisent la précision. Cependant, jusqu'à récemment, NAS n'a pas pris en compte les contraintes matérielles telles que la latence. Dans cette thèse, nous avons utilisé l'environnement DARTS, un NAS basé sur le gradient, pour rechercher des QNN à la fois précis et à faible latence, en considérant leur implémentation sur FPGA à l'aide de l'environnement FINN. Nous avons modifié DARTS pour l'adapter aux QNN et proposé un modèle de latence pour intégrer celle-ci comme critère d'optimisation dans le processus de recherche. Nos expériences montrent que l'intégration de la latence dans le processus de recherche ciblant le jeu de données CIFAR-10 peut conduire à une réduction d'environ 1.9× de la latence avec une faible baisse de précision (environ 3.3% en moyenne) par rapport aux architectures trouvées en ignorant la latence dans le processus de recherche.

ABSTRACT

Over the last few years, Deep Neural Networks (DNNs) have exhibited impressive performance in various domains, such as computer vision, speech recognition, and natural language processing. However, implementing and deploying DNNs on devices with limited resources is a challenging task due to their high computational requirements, resulting in long inference delays and increased energy consumption. To address this issue, Quantized Neural Networks (QNNs) have emerged as a promising solution by reducing the precision of weights and activations to make them suitable for resource-constrained devices. Nonetheless, the process of quantization leads to significant loss of information, thereby reducing accuracy. To overcome this challenge, Neural Architecture Search (NAS) has been developed to discover performant QNN architectures that maximize accuracy. However, NAS has not considered hardware constraints like latency until recently. In this thesis, we used the DARTS framework, a gradient-based NAS, to investigate QNNs that are both accurate and have low latency, considering their implementation on FPGAs using the FINN framework. We modified the DARTS framework to adapt it to QNNs and proposed a latency model to integrate latency as an optimization criterion in the search process. Our experiments show that incorporating latency into the search process targeting CIFAR-10 dataset can lead to around 1.9× reduction in latency with only a small drop in accuracy (about 3.3% on average) compared to the architectures found via ignoring latency through the search process.

TABLE OF CONTENTS

DEDICATION III

ACKNOWLEDGEMENTS IV

RÉSUMÉ V

ABSTRACT VI

TABLE OF CONTENTS VII

LIST OF TABLES IX

LIST OF FIGURES X

LIST OF SYMBOLS AND ABBREVIATIONS XIII

LIST OF APPENDICES XIV

CHAPTER 1 INTRODUCTION 1

CHAPTER 2 BACKGROUND 4

2.1 Artificial Neural Networks 4

2.1.1 Deep Neural Networks 4

2.1.2 Convolutional Neural Networks 4

2.2 Neural Architecture Search 5

CHAPTER 3 LITERATURE REVIEW 8

3.1 Quantized Neural Networks 8

3.1.1 Binary Neural Networks 9

3.1.2 Knowledge distillation 14

3.2 Hardware implementation of Neural Networks 14

3.3 Neural Architecture Search performance 16

3.3.1 Hardware-aware Neural Architecture Search 17

CHAPTER 4 EFFICIENT QNNS DEPLOYING DARTS AND FINN 19

4.1 DARTS framework and how it works 19
4.1.1 DARTS search space 19
4.1.2 DARTS search process 22
4.1.3 A DARTS numerical example 24
4.2 FINN compiler and how it works 27
4.3 Proposed approach to use the DARTS framework for QNN design 30
CHAPTER 5 EXPERIMENTAL SETUP, RESULTS AND DISCUSSIONS 36
5.1 Experimental Setup 36
5.1.1 DARTS Configuration 36
5.1.2 FINN Configuration 38
5.2 Results and discussion 40
5.2.1 Results for 1-cell architecture 40
5.2.2 Results for 2-cell architecture 45
5.2.3 Results for 3-cell architecture 51
CHAPTER 6 CONCLUSION 58
6.1 Summary of the work 58
6.2 Limitations 59
6.3 Future work 59
REFERENCES 60
APPENDICES 69

LIST OF TABLES

Table 3.1 BWN and XNOR-Net vs BinaryConnect and BinaryNet on ImageNet [39].....12

Table 4.1 Calculating the latency for a DARTS cell with six nodes (Figure 4.15) 33

Table 4.2 Latency table, sample 1 (Figure 4.16).....33

Table 4.3 Latency table, sample 2 (Figure 4.17).....34

Table 5.1 Loss function configurations.....37

Table 5.2 FINN parameters for FPS=20 k and MVU=100 k 39

Table 5.3 FINN parameters for FPS=1 k and MVU=1 k 39

Table 5.4 FPGA platform summary [59] 40

Table 5.5 Test set evaluation for derived 1-cell architectures.....44

Table 5.6 Test set evaluation for derived 2-cell architectures.....48

Table 5.7 Test set evaluation for derived 3-cell architectures.....54

LIST OF FIGURES

Figure 2.1 NAS general framework [32] 6

Figure 4.1 A DARTS architecture 20

Figure 4.2 A DAG sample 20

Figure 4.3 A DARTS cell's DAG sample 20

Figure 4.4 A DAG sample representing mixed operations 22

Figure 4.5 DART cell with seven nodes and fourteen edges 24

Figure 4.6 A derived cell example 24

Figure 4.7 A DARTS cell with five nodes and five edges 25

Figure 4.8 (a) Architecture parameter matrix, α and (b) Softmax(α) 25

Figure 4.9 α matrix and Softmax(α) after a search process iteration 26

Figure 4.10 The highest α values in each mixed operation 27

Figure 4.11 Top-2 strongest edges for node 3 27

Figure 4.12 A derived cell after a search iteration 27

Figure 4.13 The FINN compiler transformation steps [75] 29

Figure 4.14 FINN dataflow architecture [73] 30

Figure 4.15 A latency graph for a DARTS cell with six nodes 32

Figure 4.16 Latency graph, sample 1 33

Figure 4.17 Latency graph, sample 2 34

Figure 5.1 DARTS architectures for different experiments 36

Figure 5.2 Search plots for experiment 1 – experiment 3 (1-cell) 41

Figure 5.3 Search plots for experiment 4 – experiment 7 (1-cell) 42

Figure 5.4 Search plots for experiment 8 – experiment 11 (1-cell) 42

Figure 5.5 Search plots for three experiments 1, 2, and 11 (1-cell) 43

Figure 5.6 Normal cell obtained from experiment 1 (1-cell) 45

Figure 5.7 Normal cell obtained from experiment 2 (1-cell) 45

Figure 5.8 Normal cell obtained from experiment 11 (1-cell) 45

Figure 5.9 Search plots for experiment 1 – experiment 3 (2-cell) 46

Figure 5.10 Search plots for experiment 4 – experiment 7 (2-cell) 46

Figure 5.11 Search plots for experiment 8 – experiment 11 (2-cell) 47

Figure 5.12 Search plots for three experiments 1, 2, and 11 (2-cell) 47

Figure 5.13 Normal cell obtained from experiment 1 (2-cell) 49

Figure 5.14 Reduction cell obtained from experiment 1 (2-cell) 49

Figure 5.15 Normal cell obtained from experiment 2 (2-cell) 50

Figure 5.16 Reduction cell obtained from experiment 2 (2-cell) 50

Figure 5.17 Normal cell obtained from experiment 11 (2-cell) 50

Figure 5.18 Reduction cell obtained from experiment 11 (2-cell) 50

Figure 5.19 Search plots for experiment 1 – experiment 3 (3-cell) 51

Figure 5.20 Search plots for experiment 4 – experiment 7 (3-cell) 52

Figure 5.21 Search plots for experiment 8 – experiment 11 (3-cell) 52

Figure 5.22 Search plot for three experiments 1, 2, and 11 (3-cell) 53

Figure 5.23 Normal cell obtained from experiment 1 (3-cell) 54

Figure 5.24 Reduction cell obtained from experiment 1 (3-cell) 55

Figure 5.25 Normal cell obtained from experiment 2 (3-cell) 55

Figure 5.26 Reduction cell obtained from experiment 2 (3-cell) 55

Figure 5.27 Normal cell obtained from experiment 11 (3-cell) 55

Figure 5.28 Reduction cell obtained from experiment 11 (3-cell) 56

Figure 5.29 Accuracy comparison of experiments 1, 2, and 11 56

Figure 5.30 Latency comparison of experiments 1, 2, and 11 57

Figure B.1 Standard convolution 71

Figure B.2 Depthwise separable convolution 71

LIST OF SYMBOLS AND ABBREVIATIONS

ANN	Artificial Neural Network
BNN	Binarized Neural Network
CNN	Convolutional Neural Network
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
DARTS	Differentiable Architecture Search
DNN	Deep Neural Network
FPGA	Field Programmable Gate Array
FPS	Frames Per Second
GPU	Graphical Processing Unit
HLS	High-Level Synthesis
HW-NAS	Hardware-aware Neural Architecture Search
LUT	Lookup Table
MAC	Multiply-Accumulate
MVU	Matrix Vector Unit
NAS	Neural Architecture Search
ONNX	Open Neural Network Exchange
QNN	Quantized Neural Network
SDF	Streaming Data Flow

LIST OF APPENDICES

Appendix A Experimental details 69

Appendix B Different types of convolutions 70

CHAPTER 1 INTRODUCTION

In recent years, Deep Neural Networks (DNNs) have achieved impressive performance in a wide range of applications. DNNs are particularly effective for learning from high-dimensional, complex data, such as images, audio, and natural language, and have been successfully applied in various domains, including computer vision, speech recognition, natural language processing, and robotics [1]. While DNNs have shown remarkable success in various applications, implementing, and deploying them on resource-constrained devices is challenging. This is because their computationally intensive function requires a large amount of memory and computational resources, resulting in high energy consumption [2]. For example, the VGG-16 and ResNet-50 models comprise 138 and 25.6 million parameters, which occupy over 528 and 87 megabytes of storage space and require 15.5 and 3.8 billion floating point operations to classify one image, respectively. Moreover, the complex architecture of deep learning models and the limited computing capabilities of resource-constrained devices can result in extended inference latencies. As a result, it may fail to satisfy real-time application demands, such as in autonomous vehicles and safety-critical systems.

Quantized Neural Networks (QNNs) offer a promising approach to overcoming these challenges. QNNs use reduced bit precision for weights, activations, or both, so they require significantly less storage and computation, making them particularly well-suited for resource-limited devices [3]. For instance, according to Suyog et al. [4], the weights of deep neural networks can be expressed using 16-bit fixed-point values without significantly diminishing classification accuracy. Additionally, Gysel et al. developed an 8-bit AlexNet with only minor accuracy degradation [5]. Furthermore, binary quantization represents an extreme case of quantization as it restricts parameters to two values (e.g., 1 and -1). In theory, binary networks can achieve a compression rate of up to 32 times. Nevertheless, quantization, especially in extreme cases, causes severe information loss, which results in non-negligible accuracy degradation. Since we need these networks to operate accurately, as close as possible to their full-precision counterparts, we need to improve them as much as possible. Several research studies have addressed QNNs' limitations and achieved satisfying progress in recent years [3], [6]–[11]. However, there are still opportunities to improve QNNs to gain more reliable performance. Neural Architecture Search (NAS) is a newly

investigated area of research to find performant QNN architectures. NAS automatically explores architectures based on given search space parameters such as convolutional layer types, convolution filter size and the number of filters and has demonstrated encouraging outcomes by surpassing prior hand-designed architectures on diverse tasks such as image classification and object detection [12]–[19]. However, the search process implies exploring a vast number of possibilities, making it computationally demanding. For instance, when searching for an optimal architecture on the ImageNet dataset, it requires 3150 GPU days to complete [20]. Hence, in recent years, gradient-based NAS has been developed to alleviate this problem significantly. This type of NAS uses gradient descent to optimize the architecture search process, and it can efficiently explore a search space comprising several operations. Compared to non-gradient-based NAS methods, gradient-based ones are more efficient and effective, and they can converge to reasonable solutions in a shorter amount of time. For example, DARTS (Differentiable Architecture Search) utilizes gradient-based optimization achieving promising results on the ImageNet dataset in only four GPU days [21].

Although maximizing accuracy is a primary goal in neural architecture search (NAS), the consideration of hardware constraints such as latency in NAS has not been studied extensively until recently [22]. In this thesis, we aim to utilize NAS to discover QNNs which are accurate and have low latency in order to use them in computer vision applications in resource-constrained environments. To achieve this, we used Brevitas, a PyTorch-based framework for developing QNNs, to modify the DARTS framework and construct a quantized search space. In this project, we considere 4-bit quantization for all learnable parameters. Additionally, to measure the DARTS-generated architectures' latency, we propose a model based on the latency of individual operations to estimate the total latency for a target hardware platform. Measuring the latency enabled us to integrate this hardware criterion into the loss function during the search process. By doing this, we transform DARTS into a latency-aware NAS framework to discover performant architectures regarding accuracy and latency. For our target hardware platform, we employed the FINN architecture, which is a pipeline-based architecture for accelerating QNN inference on FPGA. It effectively distributes the appropriate amount of computing resources, such as BRAMs and MVUs (Matrix Vector Units), for each layer in a network model.

Our contributions are as follows:

- Modifying the DARTS framework, a gradient-based NAS, to adapt QNNs using Brevitas.
- Defining a quantized search space compatible with the FINN architecture.
- Proposing a latency model based on the FINN architecture to estimate the actual latency.
- Integrating the latency into the loss function to observe its impact on the search process.

The thesis is organized as follows:

- Chapter 2 provides some basic information about neural networks and NAS.
- Chapter 3 reviews the related literature and covers several works regarding QNNs and the hardware implementation of these networks.
- Chapter 4 details our approach to use NAS to discover performant QNN.
- Chapter 5 discusses experimental setups and results.
- Chapter 6 presents the conclusion and mentions some limitations of our approach and potential future works.

CHAPTER 2 BACKGROUND

This chapter presents the fundamental concepts of neural networks, from biological neurons through the training of artificial neural networks. Following that, it provides some fundamental details concerning the Neural Architecture Search idea and its algorithms.

2.1 Artificial Neural Networks

An Artificial Neural Network (ANN) is a model designed to mimic the structure and behavior of biological neurons in the human brain. ANNs consist of several interconnected nodes called artificial neurons receiving inputs and producing outputs based on their activation. The connections between the neurons are modelled as weighted connections, allowing the network to adjust the strength of these connections based on the data it is trained on. The input data is processed through layers of artificial neurons, each layer contributing to a higher level of information abstraction. The output produced by the final layer of neurons represents the network's prediction or decision [23], [24].

2.1.1 Deep Neural Networks

A Deep Neural Network (DNN) is an ANN with a deeper architecture comprising additional layers. The network can perform better on challenging tasks thanks to these extra layers, which enable the network to learn more abstract and hierarchical representations of the input data. In other words, having more layers increases learning capacity allowing extracting more information from the input [25].

2.1.2 Convolutional Neural Networks

The Convolutional Neural Network (CNN) is a type of ANN used primarily for image classification and recognition tasks. CNNs are designed to process image data, where the spatial relationship between image pixels is essential. A CNN network comprises various types of layers, such as convolutional, activation, pooling, and fully connected layers [26]. The convolutional layer in a CNN performs a mathematical operation on the input data called convolution. It effectively helps to identify the features or patterns in the data. To adds non-linearity to the convolutional layer's

output, there should be an activation layer. It enables describing complicated relationships in the data. The data is down-sampled using the pooling layer, which lowers both the network's computing needs and its spatial dimensions. Finally, the fully connected layer at the end of the network produces the final prediction or decision. The ability of CNNs to learn useful features from the input data and their suitability for image classification and recognition tasks have made them a popular choice in computer vision and image processing applications.

2.1.2.1 Training and inference phase

For a given task, there is a set of data points in the form of the input-output tuple, known as a dataset, fed to the network for the training. In the training phase, first, input data passes through network layers to generate the prediction output. Then, an optimization algorithm applies gradient descent to adjust the weight parameters of the network utilizing a loss function to measure the difference between the actual output value and the prediction. Afterwards, the parameters' gradients are calculated and backpropagated to the network from the last layer to the first layer updating the weights. The training process aims to minimize the loss function's output through several epochs by inferring each data sample.

It is important to note that training the network on the training data does not guarantee it will perform well on new and unseen data. Therefore, a separate set of data points, called the validation set, is used to monitor the network's performance to avoid overfitting the training data. Whenever the network's performance on the validation data starts to decline, the network is not generalizing well and is memorizing the training data. Hence, it is time to stop the training process. In the final step, the network is then evaluated on a test set, which is a set of unseen data points.

2.2 Neural Architecture Search

Designing a deep neural network architecture is time-consuming and requires rich expertise and skills. For example, well-known DNN architectures such as AlexNet [27], VGG [28], and ResNet [29] are manually created by skilled researchers with knowledge of neural networks and image processing. This process, while complex, often consists of systematically exploring a large number of options. Therefore, many researchers have attempted to introduce an automatic process to discover a performant architecture for a specific task [30].

Neural Architecture Search (NAS) is the process of developing neural networks' topology in an automatic way to improve the performance of a given application [30], [31]. NAS algorithms seek an optimal set of hyperparameters for a neural network to accomplish a specific task. In the past few years, NAS methods have helped designers find the best architecture for various types of applications based on desired performance metrics such as accuracy. These methods have shown promising results by outperforming previous hand-crafted architectures on a wide range of tasks such as image classification, semantic segmentation, and object detection [32].

The NAS process consists of three main parts, the search space, the search strategy, and the performance evaluation technique. Figure 2.1 shows the NAS general framework in which a search space consists of predefined operations (e.g., convolutional layers, pooling). The search strategy determines how a possible candidate architecture is created and chosen, and through an iterative process, candidate networks are evaluated until a particular condition is reached. In general, NAS algorithms are classified by their search strategy into three categories: reinforcement learning, evolutionary algorithms, and gradient-based methods [31].

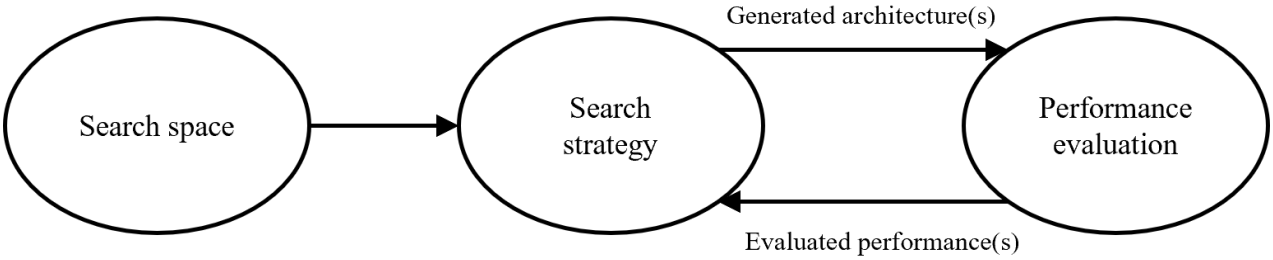

Figure 2.1 NAS general framework [32]

In the reinforcement learning strategy, we can consider the action and reward for an agent, producing an architecture and performance evaluation, respectively [30][33]. Alternatively, researchers have been utilizing evolutionary algorithms to design neural networks for decades [30]. In this strategy, an evolutionary algorithm (e.g., a genetic algorithm) is used, and a set of models is considered a population (i.e., a list of networks). Additionally, a fitness function works based on the evaluated performance of offspring generated after each mutation. The gradient-based approach employs the discrete architecture search space in a continuous relaxation manner, exploiting the gradient descent to find a high-performing solution compared to other non-gradient-based techniques. Furthermore, it can be utilized on any search space based on Directed Acyclic Graphs

(DAGs), where each edge presents various operations through the utilization of a "zero" operation to imitate the lack of an edge [21], [32].

CHAPTER 3 LITERATURE REVIEW

In Chapter 2, some fundamental principles of neural networks were introduced. Deep Neural Networks (DNNs) have significantly advanced the capabilities of Artificial Intelligence (AI) across various fields, such as image classification, object recognition, speech recognition, games like Go, and even the creation of abstract art [3]. These algorithm advancements, increasing data availability and using powerful computing devices, have delivered exceptional results, particularly in computer vision tasks where they surpassed traditional methods in image classification and even humans [34]. Hence, this has led to the widespread adoption of DNNs in various industries and applications, making them a key component of modern AI technologies [35]. However, implementing and deploying such networks with their computationally intensive functions require a large amount of memory and computational resources, resulting in high energy consumption. Therefore, utilizing these networks with time and energy constraints in some applications has become a challenging problem in recent years. Furthermore, using floating-point weights and activations in these networks has been standard practice [7], providing the necessary capacity for the intermediate layers to represent data. Moreover, neural network hardware was designed to accelerate full-precision MAC operations only. So, to address these limitations, numerous studies have been done at both the algorithmic and hardware levels to make neural networks more hardware-efficient. This chapter focuses on these works and their achievements.

3.1 Quantized Neural Networks

Quantized Neural Networks (QNNs) are a type of neural network that exploit quantization to reduce neural network memory and computation requirements. Quantization normally refers to the process of using a reduced number of bits used to represent the weights and activations of the network, typically from 32 bits to 8 bits or even fewer. This reduction in precision can result in a significant reduction in memory requirements and computation time, making it possible to deploy neural networks on resource-constrained devices, such as mobile phones and embedded systems. In these networks, the quantized values are used to update the weights and activations during training and to make predictions during inference. Despite the reduction in precision, QNNs can

still achieve competitive accuracy compared to their full-precision counterparts, especially when trained with quantization-aware training methods.

3.1.1 Binary Neural Networks

Binary Neural Networks (BNNs) represent the extreme case of quantization, using a single bit to represent weights or activations. BNNs potentially have a promising role in deploying deep learning models on devices with limited hardware and energy resources. However, they suffer from challenging problems. For example, the binarization process causes substantial information loss and complicates deep network optimization. There are several approaches to deal with these challenges, which are discussed in this chapter.

Courbariaux et al. proposed BinaryConnect [36], revolutionizing the study of binary neural networks. This work introduced a novel approach to designing efficient DNNs using binarization approximation. In this approach, weight parameters are binary values instead of full precision during forward and backward propagations. As a result, since the weights are constrained to only two values (e.g., -1 or +1), the multiply-accumulate operations would be reduced dramatically, which means the hardware footprint and power consumption decrease significantly. Furthermore, they showed that their proposed binary model could achieve competitive state-of-the-art performance on well-known datasets such as MNIST, CIFAR-10 and SVHN and play an essential role in the literature. For binarization, they utilized two methods called deterministic and stochastic binarization. The deterministic approach is very straightforward that is based on the sign function:

$$w_b = \begin{cases} +1, & w \geq 0 \\ -1, & otherwise \end{cases}, \tag{3.1}$$

where w_b is the binarized weight, and w is the real-valued weight. The other approach they used is to binarize weight parameters stochastically:

$$w_b = \begin{cases} +1, & with\ probability\ p = \sigma(w) \\ -1, & with\ probabiliy\ 1 - p \end{cases}, \tag{3.2}$$

where σ is the "hard sigmoid" function:

$$\sigma(x) = clip\left(\frac{x+1}{2}, 0, 1\right) = \max\left(0, \min\left(1, \frac{x+1}{2}\right)\right) \tag{3.3}$$

It is important to note that the real-valued parameters are crucial in the binarization process of the model, as they are stored and updated through the backpropagation process and then utilized in the binarization process.

Following this scheme, Hubara et al. introduced Binarized Neural Network (BNN) [37] in which both weights and activations are binarized. Similar to BinaryConnect, they showed that it is possible to achieve comparable state-of-the-art results on MNIST, CIFAR-10 and SVHN datasets. However, they reported unsuccessful preliminary results on the more challenging ImageNet dataset. For the inference phase, they explained that BNNs could reduce memory consumption substantially, and most arithmetic operations can be replaced by bitwise operations such as XNOR-count, potentially leading to a significant improvement in power efficiency. Their experiments on image classification show that a binarized CNN could reduce the time complexity by 60%, and compared with 32-bit DNNs, BNNs require 32× less memory storage.

Previous approaches extremely quantize the network and take advantage of saving computational resources. However, for various applications, these methods experience accuracy loss without mentioning the effect of binarization in the forward and backward path. Therefore, several solutions have been proposed to optimize binary neural networks to deal with the accuracy loss and successfully have shown remarkable improvement compared to the naive strategies [38]. Reducing the quantization error is a conventional approach to estimating full-precision network parameters to obtain comparable accuracy. This way, a typical method is to reduce the weight and activation quantization error for optimizing binary neural networks. A quantized parameter should approximate the full-precision parameters to maintain the expected model's performance as high as possible. In this way, Rastegari et al. proposed Binary Weight Networks (BWN) and XNOR-Net [39]. BWN utilizes binary weights and keeps the activations in full-precision mode, while XNOR-Net binarizes both weights and activations. Unlike previous works, this study approximates the floating-point parameters by considering a scaling factor besides the binarized parameters. This solution shows less quantization error than the previous methods, which directly used 1-bit,

improving network accuracy. The results show the higher accuracy of BWN and XNOR-Net compared to non-optimal naïve solutions (see Table 3.1) compared to the full-precision network. Their work provides smaller memory storage requirements up to 32 times and two times speedup in BWN and 58 times in the XNOR network on the ImageNet dataset.

Similar to XNOR-Net, Hu et al. proposed a model that uses a hash map with a scaling factor in the quantization process [40]. They adopted a transformation scheme to describe CNNs with binary weight as a hashing problem. The DoReFa-Net [11] continued the XNOR-Net idea and utilized 1-bit weight, 2-bit activation, and 6-bit quantized gradients to accelerate a derived AlexNet model's training process, especially in the backward pass. They concluded that gradients need to hold a larger bitwidth than activations. The authors state that the application of their method to AlexNet results in 55.9% top-1 accuracy in single-precision, while 8-bit quantization achieves 53.0% top-1 accuracy. Furthermore, binarizing the weights and quantizing the activations to 1, 2, or 4 bits results in top-1 accuracy of 40.1%, 47.7%, and 50.3%, respectively.

Regarding input tensors, Li et al. proposed a framework named HORQ [41] in which a recursive thresholding operation is used to quantize input tensors efficiently. They demonstrated that since the previous works suffered from information loss caused by inefficient input quantization, they adopted a recursive approximation of floating-point values. This approach compensates for information loss so that the final output in each stage would be a linear combination of recursive levels. Their method outperformed XNOR-Net on small-scale datasets such as MNIST and CIFAR-10 by 0.71% and 3%, respectively. Nevertheless, they have not tested their technique on ImageNet, and surprisingly their work did not perform well compared to BNN [37].

Further, ABC-Net [42] was introduced by Lin et al. as an extension to XNOR-Net. The idea adopts a linear combination of binary weights and activations to estimate the full-precision counterparts. They applied their method utilizing ResNet [29] architecture on the ImageNet dataset and achieved remarkable results close to the full-precision model with a 4.3% difference in Top-1 accuracy. Their model outperformed XNOR-Net by far due to multiple levels of approximation for each weight matrix and more accurate estimation for the activation stage inspired by DoReFa-Net [11]. However, hardware implementation and memory usage analysis is absent in this work. Increasing model size and energy consumption is expected by utilizing multiple weight matrix bases and binary activations. Improving the XNOR-Net idea, Bulat and Tzimiropoulos developed XNOR-

Net++ [43] to enhance the real-valued scaling factor leading to better accuracy of more than 6% over the ImageNet dataset performed on Resnet-18 architecture.

Table 3.1 BWN and XNOR-Net vs BinaryConnect and BinaryNet on ImageNet [39]

Method	Bitwidth activation/weight	Classification Accuracy (%): Top-1	Classification Accuracy (%): Top-5
BWN	32/1	**56.8**	**79.4**
BinaryConnect	32/1	35.4	61.0
XNOR-Net	1/1	**44.2**	**69.2**
BinaryNet	1/1	27.9	50.42
AlexNet	32/32	56.6	80.2

The authors have argued that the analytical description of the proposed scaling factor in XNOR-Net is sub-optimal. Therefore, they adopted a strategy that fuses the weight and activation binarization factors into a single one and learns it discriminatively through the backpropagation process. They also reported that their model outperforms ABC-Net by around 15% when the model size and parameters are the same as their proposed model.

Deep neural networks contain several layers. The approaches that target minimizing the quantization error try to decrease the loss of information in each layer. However, some research shows that it is necessary to consider the loss function since the binarization process in each layer can affect the global loss. Recently, many studies attempted to find the desired network loss function that can control network parameters learning with binarization limitations. This way, Hou et al. proposed Loss-Aware Binarization (LAB) [44] that utilizes the proximal Newton algorithm to obtain optimal binarized weights. The authors showed that the binarization effect could be considered in the overall loss by defining an objective function in the form of a composite optimization problem. Their method outperforms XNOR-Net and other naïve models by less than

1% in the test error rate on small-scale datasets like MNIST. Regardless of their excellent analytical work, this work is not worth implementing from the hardware implementation viewpoint because of the complexity in the training phase unless inference mode is aimed. Also, their work has not been evaluated on large-scale datasets. Ding et al. tried to add a regularization term to the loss function addressing the effect of the binarization process, including degeneration, saturation, and gradient mismatch. They showed that their method outperformed the baseline BNNs [45].

For training binary neural networks, the backpropagation algorithm is used. In the backward path, since there are non-differentiable functions (e.g., sign), the straight-through estimator (STE) [46] technique is often used to deal with gradient estimation. However, the gradient mismatch caused by utilizing STE is still considerable. This problem can potentially lead to performance degradation. One solution to reduce the gradient mismatch effect in the backward propagation is approximating the sign function. Bi-Real Net [47], proposed by Liu et al., introduces a customized function as an alternative to the sign function for the backward path. Their result shows that the gradient error can be improved, which leads to an improvement in accuracy. It significantly reduces memory usage and computation time compared to the full-precision network for both the Res-18 and the Res-34 networks. Specifically, memory usage is reduced by 11.1 times and 16.0 times, while computation is reduced by 11.1 times and 19.0 times, respectively. Additionally, the Bi-Real Net outperforms XNOR-Net in terms of FLOPS and memory usage, as it does not use real-valued weights and activations during inference and is also easier to implement.

Quantizing weights or activations to values larger than 1-bit can recover the accuracy loss through binarization to some extent, bringing performance closer to single-precision results [3], [11]. Ternary Weight Networks (TWNs) [6], introduced by Li et al., applies ternarization to the weight parameters, converting 32-bit weights into a 2-bit representation (-1, 0, and 1). TWN only affects the weights, leaving activations and gradients unchanged. When applied to LeNet-5, TWN achieved only a 0.06 % difference in accuracy on the MNIST validation set compared to single-precision. It also obtained 2.3% lower top-1 accuracy on ImageNet than its single-precision (67.6%). Similar to TWNs, Zhu et al. introduce Trained Ternary Quantization (TTQ) [48]. TTQ employs two scaling factors in each layer, one for positive weights and the other for negative weights, and these scaling factors are learned during training. With trained weights and scaling factors, TTQ has two separate gradients, one for the weights and another for the scaling factors.

When applied to ResNet-20 and ResNet-56, TTQ attains 91.13% and 93.56% accuracy on the CIFAR-10, compared to the single-precision accuracy of 91.77% and 93.2%, respectively. For AlexNet, TTQ achieves a top-1 accuracy of 57.5% on the ImageNet, compared to the single-precision accuracy of 57.2% counterpart.

3.1.2 Knowledge distillation

Knowledge distillation is another approach to improve the accuracy of low-precision deep neural networks such as BNNs. In this approach, one can train a full-precision complex network then transfer the produced information to train a BNN counterpart more accurately and decrease the discrepancies between a large model and its small counterpart. The original full-precision network and the corresponding smaller low-precision network are usually referred to as the teacher and student networks. Mishra and Marr studied knowledge distillation techniques on low-precision neural networks and proposed three schemes to improve the accuracy: jointly trained, continuously trained and a fine-tuned student network [49]. The student network has an architecture similar to the teacher network in all three schemes. They evaluated their model on ResNet architectures with a combination of different bit widths (2, 4, 8 and 32 bits) on both weights and activations; however, they did not report any results on binary quantization. Chen et al. utilized knowledge distillation techniques to train binarized DNN models in real-time speech separation applications [50]. Xu et al. proposed a filter-level pruning method utilizing a learning-based teacher/student framework to compact and accelerate BNNs on challenging datasets. However, they did not draw a comparison in accuracy between their model and the state-of-the-art binary models. The work proposed by Martinez et al. assembled several previously studied methods to propose a strong baseline that achieves state-of-the-art performance [51]. They further utilized a multi-stage teacher/student mechanism and a data-driven channel re-scaling strategy to improve the accuracy that bridged the gap between real-valued and binarized networks on CIFAR-100 and ImageNet datasets to within 3-5 % when using ResNet-18 architecture.

3.2 Hardware implementation of Neural Networks

The conventional way to implement neural network training and inference consists of utilizing CPUs and GPUs because several well-known deep learning frameworks, such as TensorFlow [52]

and PyTorch [53], specifically support these processing units. Therefore, developing deep learning algorithms can be achieved with high efficiency. On the other hand, FPGAs as reconfigurable hardware platforms have been considered for low-power configuration and cost-effective acceleration of CNN models [54]. For example, Zhang et al. [55] and Ovtcharov et al. [54] introduced an optimized single processing unit for each layer in the form of a systolic array employing an analytical roofline model tool. Other works, such as Venieris et al. [56], follow a streaming approach in which the proposed framework utilizes an analytical Synchronous Dataflow (SDF) model that considers performance-resource design space along with the platform-specific resource constraints. Usually, developing deep learning algorithms on FPGA can be cumbersome since it requires specialized hardware programming expertise. Moreover, there is still a performance gap between FPGA and GPU. However, the availability of High-Level Synthesis (HLS) tools to designers and emerging QNNs and recent improvements in their accuracy [51] facilitate the development of FPGA-friendly architectures that can achieve comparable performance to GPUs while consuming less power. In this way, Zhao et al. introduced a hardware architecture based on variable-length buffers to accelerate BNNs inference on FPGA [57]. They implemented their model using HLS design methodology on a low-cost ZedBoard FPGA development board. Similar to the model developed by Zhang et al. [55], Andri et al. proposed a low-power CNN accelerator, YodaNN [58], based on binary weights.

Umuroglu et al. from the Xilinx research group proposed FINN [59], a BNN accelerator framework based on a heterogeneous streaming architecture that enables efficient mapping of BNN models in a scalable and parameterizable design on FPGA. Their work attempts to balance accuracy, power consumption, and latency by offering adjustable classification throughput and flexible building block design. They further expanded their work to cover larger topologies for BNNs [60]. Then, FINN-R [37] was introduced to support the quantized implementation of CNNs on larger networks with more design space exploration. We cover FINN architecture in Chapter 4 in detail.

Liang et al. proposed FP-BNN [61] with a Resource-Aware Model Analysis (RAMA) method to determine FPGA resource cost for each block of computation on a BNN, then designed an efficient accelerator architecture utilizing a pop-count compressor tree and a processing element (PE) reuse procedure. Haojin Yang et al. utilized MXNet [62], a deep learning library for heterogeneous distributed systems, to develop the BMXNet library [63] for implementing BNNs and QNNs on

GPUs and CPUs. They evaluated their work on MNIST, CIFAR-10 and ImageNet datasets. Zhang et al. introduced daBNN [64], an efficient open-source framework to run BNNs in inference mode on embedded devices equipped with ARM processors by proposing an optimized bit-packing scheme and binary matrix multiplication (BGEMM) and achieved 3x and 6x speedup on Bi-Real Net-18 compared to TensorFlow lite and BMXNet [63], respectively. Kim et al. applied binarization methods on an encoder-decoder network and then proposed a binarized deconvolution engine (BDE) to accelerate it for semantic segmentation applications [65]. They used FINN architecture and implemented their model on FPGA achieving notable performance on the CamVid11 [66] dataset.

3.3 Neural Architecture Search performance

As we discussed in Chapter 2, there are different NAS search strategies to exploit search space, exploring a performant network architecture. However, despite the impressive results, current architecture search techniques are computationally intensive. For instance, the pursuit of state-of-the-art architecture for CIFAR-10 and ImageNet required a substantial amount of computational resources, including 2000 GPU days through reinforcement learning (RL) [20] or 3150 GPU days through evolutionary methods [67]. Hence, recent research focuses on developing efficient architecture search methods. For example, one well-known efficient NAS algorithm that Liu et al. proposed is Differentiable Architecture Search (DARTS) [21]. The key idea of this work is to perform architecture search in a continuously differentiable manner, allowing for gradient-based optimization of network architecture. The DARTS results showed that it outperforms hand-designed networks on various benchmark datasets in terms of accuracy. Their method found SOTA architecture for CIFAR-10 and ImageNet in only four GPU days. Their results highlight the effectiveness of the proposed method for optimizing neural network architecture. Building upon the concept of DARTS, researchers have proposed ways to improve this search strategy. One such improvement is P-DARTS [68], which was put forward by Chen et al. P-DARTS delivers superior results by gradually increasing the depth of the network during the search phase, making the overall search process faster compared to traditional DARTS.

Using gradient-based NAS, several researchers attempted to employ it to discover performant QNNs. For example, Bulat et al. introduced BATS [18], demonstrating that directly applying

DARTS on BNNs leads to poor results due to the *double approximation problem* caused by depth-wise separable convolution, which is commonly used for feature compression and consists of two successive convolutions. Additionally, they claimed that dilated convolutions experience similar issues for binarization. Therefore, the authors suggested grouped convolutions and grouped-dilated convolutions as alternative solutions to address these problems and achieve SOTA results on the ImageNet dataset with 66.1% accuracy compared to full-precision 69.7% accuracy for only 1.55 × 10^8 FLOPs compared to ~20 × 10^8 FLOPs. The authors of BATS claimed that their work outperforms [16], even though it used evolutionary NAS and achieved higher accuracy (69.65%) but with 6.6 × 10^8 FLOPs, 4.2 times higher than BATS. Other studies, including NASB [19] and BNAS v2 [17], also focus on BNNs and adopt gradient-based neural architecture search. However, unlike BATS, BNAS v2 suggested a new search objective, particularly for BNNs, to enhance diversity during the search process. Furthermore, NASB uses ResNet-18 as the backbone when constructing architecture cells and achieves 66.6% accuracy on ImageNet with 3.52 × 10^8 FLOPs.

3.3.1 Hardware-aware Neural Architecture Search

The objective of conventional NAS is to explore the search space to find an architecture that maximizes accuracy. On the other hand, Hardware-Aware Neural Architecture Search (HW-NAS) considers various hardware constraints in addition to accuracy and aims to identify an architecture that not only performs well but also meets hardware limitations, such as latency, energy consumption, and memory footprint [22]. So, in other words, we are dealing with a multi-objective optimization problem which has drawn more attention since 2017, and about 83 papers have been published that targeted HW-NAS in 2020 [22]. One example is Once-for-all (OFA) [69] which proposes a new neural architecture search (NAS) approach. OFA aims to design a super-network architecture that can be trained once and then adapted to various tasks with different hardware configurations, including different widths, depths, and resolutions. The authors claim that this approach is more efficient than traditional NAS methods, which typically require training multiple individual models for each task and hardware configuration. Furthermore, OFA uses a weight-sharing technique to efficiently search the architecture space and obtain a super-network with high performance. The results show that OFA can obtain state-of-the-art accuracy on several benchmark

datasets, such as CIFAR-10 and ImageNet while using fewer resources and time compared to other NAS methods.

A proxy task is often used in traditional NAS. A proxy task is a simpler or related task used to evaluate the performance of neural network architectures (on a smaller dataset like CIFAR-10) during the architecture search process. The objective of using it is to speed up the search process and reduce the computational cost of evaluating the performance of each candidate architecture. In fact, the goal is to find an architecture that performs well on the proxy task, which is expected to also perform well on the target task (e.g., on ImageNet). However, ProxylessNAS [70] is an example that introduces a new method for finding efficient architectures without a proxy task. In this work, the architecture parameters are binarized to consume low memory during the search process. Moreover, they use a latency model to predict the candidate architecture's latency. The results show that the architectures found by ProxylessNAS are competitive with state-of-the-art architectures and more scalable on GPU, CPU and mobile platforms, as they can be trained on larger datasets without performance degradation.

In HW-NAS, researchers sometimes directly target the hardware specification itself. For example, Jiang et al. [71] determined an approach to achieve a balance between accuracy and efficiency in FPGA applying NAS and considering different FPGA specifications like available LUTs and on-chip memory. They used an RL paradigm and proposed a tool called FNAS to estimate the latency based on their tiling strategy. For mobile devices, Wu et al. proposed Facebook-Berkeley-Nets (FBNet) [72] to search for mobile-efficient CNNs. They used gradient-based NAS and integrated latency into the loss function. As their hardware device is mobile phones, they assume that each search space operation (Conv, Pooling) will execute on the CPU consecutively. Hence, the total latency is the summation of each operation’s latency in a sample network. The results show that FBNet outperformed MobileNetV2 by being 2.4x smaller and 1.5x faster with the same accuracy.

In this thesis, the use of FINN as a framework for accelerating quantized neural networks (QNNs) on FPGA and the HW-NAS approach for balancing accuracy and hardware metrics are combined to find a high-performing, small-scale quantized neural network. Chapter 4, covers the approach and provides technical details.

CHAPTER 4 EFFICIENT QNNs DEPLOYING DARTS AND FINN

In this chapter, we first describe the DARTS and FINN frameworks and discuss how they work in detail. We then explain how we use these frameworks together and integrate latency's effect into the design process of quantized neural networks.

4.1 DARTS framework and how it works

DARTS (Differentiable Architecture Search) is a well-known NAS method proposed by Liu et al. [21]. Architecture search in DARTS is performed in a continuously differentiable manner, enabling gradient-based network architecture optimization. This is possible via a set of continuous relaxations of the discrete architecture representation, and the architecture parameters are trained along with the other network parameters. This allows the use of backpropagation to compute gradients with respect to the architecture parameters. The following sections provide details on how DARTS proposes to solve the architecture search problem.

4.1.1 DARTS search space

In the DARTS framework, a DARTS architecture is a neural network architecture constructed by stacking cells discovered by the DARTS search algorithm. A cell is a small neural network module that can be repeated and stacked to form a larger architecture. Two types of cells exist based on the input and output feature map resolution: A *Normal Cell* that preserves the resolution of the feature map and a *Reduction Cell* that reduces the input feature map's size by half. A sample DARTS architecture with two cells is depicted in Figure 4.1. As can be seen, there is a residual connection between cells. Each cell in the DARTS architecture is defined by a Directed Acyclic Graph (DAG), which forms the search space in DARTS. In a DAG, the input data and intermediate feature maps are represented by nodes, while the edges represent the operations performed on these data. The operations can be any differentiable function, such as standard, separable, or dilated convolutions. A DAG sample is depicted in Figure 4.2.

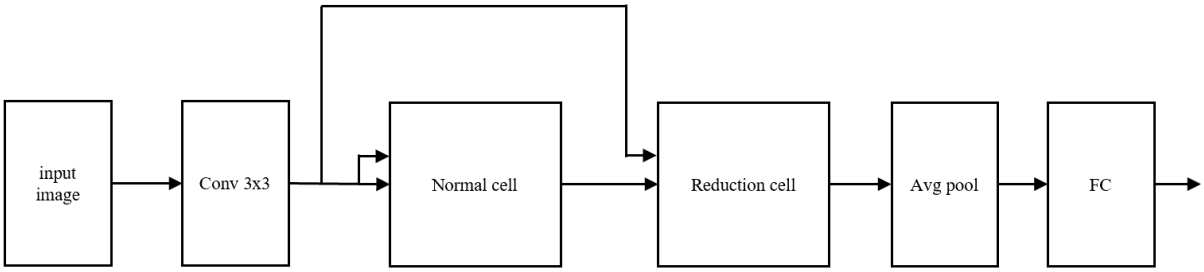

Figure 4.1 A DARTS architecture

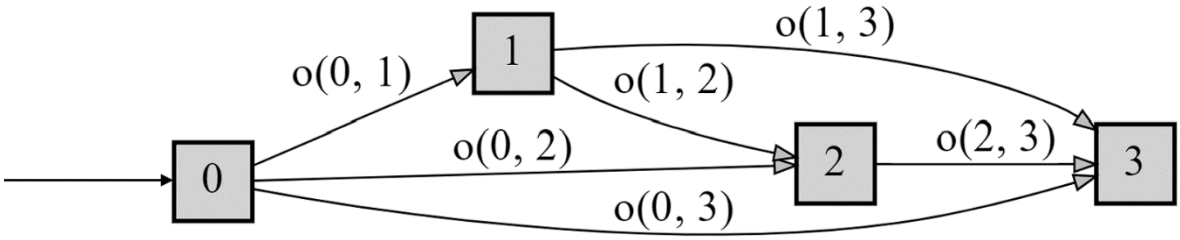

Figure 4.2 A DAG sample

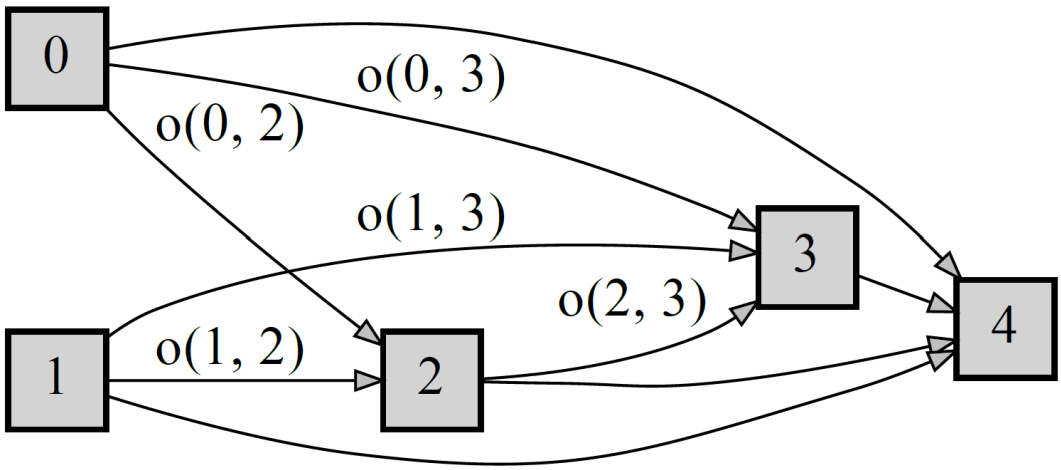

Figure 4.3 A DARTS cell's DAG sample

In a cell's DAG (Figure 4.3), each node takes in a feature map $x^{(i)}$ as input and is connected to other nodes through directed edges *(i, j)* that apply a transforming operation $o^{(i,j)}$ to $x^{(i)}$. In a cell, the input nodes (nodes 0 and 1 in Figure 4.3) are determined by the outputs from the two preceding layers (see Figure 4.1). The cell output (node 4 in Figure 4.3) combines all the intermediate nodes' outputs through concatenation. The computation of intermediate nodes is based on all their preceding nodes as shown in equation (4.1)

$$x^{(j)} = \sum_{i<j} o^{(i,j)}(x^{(i)}) \tag{4.1}$$

To make the search space continuous, DARTS includes a softmax function to replace the categorical selection of a particular operation with a continuous representation. In this regard, in equation (4.2), O is a set of available operations like convolution and max pooling in which each operation o(.) transforms $x^{(i)}$. $\bar{o}^{(i,j)}(x)$ is a weighted mixed operation of all available operations for a pair of nodes *(i, j)* which is parametrized by a vector $\alpha^{(i,j)}$ of dimension |O|, which is further referred to as the architecture parameter.

$$\bar{o}^{(i,j)}(x) = \sum_{o \in O} \frac{\exp(\alpha_o^{(i,j)})}{\sum_{o' \in O} \exp(\alpha_{o'}^{(i,j)})} o(x) \tag{4.2}$$

For example, in Figure 4.4, the left graph demonstrates a DAG with three nodes (0, 1 and 2) and three operations o0, o1 and o2, and the right graph shows the corresponding mixed operation for each pair of nodes. In this figure, a set of three operations is assumed, so there are three edges between each pair of nodes labelled with *on(i, j)* notation (n is operation number, i and j are related to node number). Each mixed operation is described in equation (4.3) in which $\bar{\alpha}_o^{(i,j)}$ is $softmax(\alpha_o^{(i,j)})$. After the search process is completed, the final architecture can be determined by selecting the most probable operation for each mixed operation $\bar{o}^{(i,j)}$, that is, $o^{(i,j)} = argmax_{o \in O}\, \alpha_o^{(i,j)}$.

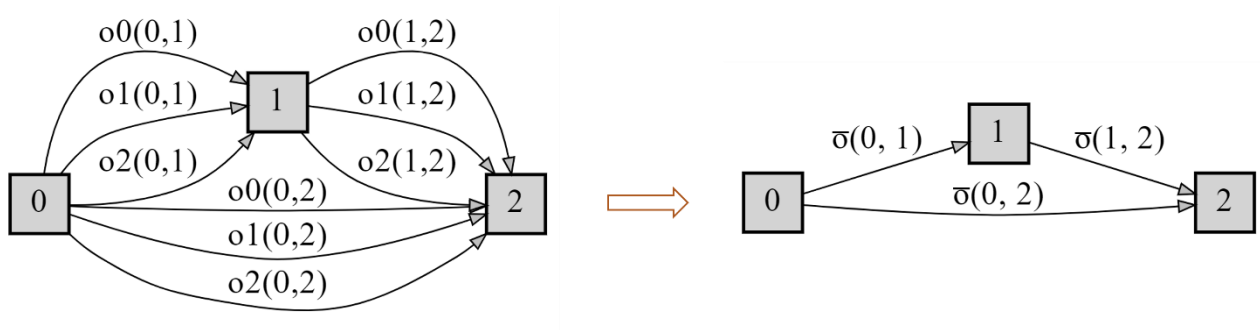

Figure 4.4 A DAG sample representing mixed operations

$$\bar{o}^{(i,j)}\,(x^{(i)}) = \bar{\alpha}_{o0}^{(i,\,j)} * o0(i,\,j)(x^{(i)}) + \bar{\alpha}_{o1}^{(i,\,j)} * o1(i,\,j)(x^{(i)}) + \bar{\alpha}_{o2}^{(i,\,j)} * o2(i,\,j)\big(x^{(i)}\big)$$

$$\bar{\alpha}_{o}^{(i,\,j)} = \frac{\exp(\alpha_{o}^{(i,\,j)})}{\sum_{o' \in O} \exp(\alpha_{o'}^{(i,\,j)})} \tag{4.3}$$

4.1.2 DARTS search process

The architecture search process consists of finding an optimal combination of operations to form a cell and further the final network architecture. Architecture search aims to learn both architecture parameters α and the weights w for the convolution operations (e.g., the convolution filters (kernels)' weights). So, in this regard, the training loss, $\mathcal{L}_{train}$, and validation loss, $\mathcal{L}_{val}$ should be determined by both the architecture parameter, α, and the weights, *w*, in the network. The objective of the architecture search is to find the optimal architecture, α^* , which minimizes the validation loss, $\mathcal{L}_{val}(w^*, \alpha^*)$, with the optimal weights, w^*, obtained by minimizing the training loss, $w^* = argmin_w\, \mathcal{L}_{train}(w\,, \alpha^*)$. This is a two-part (bilevel) optimization problem, with α being the higher-level variable and *w* being the lower-level variable described in equations (4.4)

$$\begin{aligned} &min_\alpha\, \mathcal{L}_{val}(w^*(\alpha), \alpha) \\ &s.t.\ w^*(\alpha) = argmin_w\, \mathcal{L}_{train}(w, \alpha) \end{aligned} \tag{4.4}$$

The DARTS authors adopted a simple approximation method to evaluate the architecture gradient due to the difficulty of performing an exact evaluation, which would require a costly inner optimization process which is the calculation of $w^*(\alpha)$ in (4.4). So, to obtain $w^*(\alpha)$, a single training step is performed instead of several steps to reach convergence. (4.5) shows their approximation where ξ is the learning rate in a single inner optimization iteration step outlined in Algorithm 1.

$$\nabla_\alpha \mathcal{L}_{val}(w^*(\alpha), \alpha) \approx \nabla_\alpha \mathcal{L}_{val}(w - \xi \nabla_w \mathcal{L}_{train}(w, \alpha), \alpha) \tag{4.5}$$

Algorithm 1: DARTS algorithm (adapted from [21])

Create a cell with a certain number of nodes. Then use the mixed operation formula (4.2) and initialize α for each edge (i, j) with random initialization using a normal distribution (mean 0 and variance 1). Here, ξ is the learning rate in a single inner optimization iteration step.

While *not converged* **do**

1. Update architecture α by descending $\nabla_\alpha \mathcal{L}_{val}(w - \xi \nabla_w \mathcal{L}_{train}(w, \alpha), \alpha)$
2. Update weights *w* by descending $\nabla_w \mathcal{L}_{train}(w, \alpha)$

Obtain the final architecture by applying the learned α

Figure 4.5 shows a DART cell with seven nodes: two input (green), four intermediate (blue) and one output (yellow). The fourteen edges, m0 to m13, represent mixed operations. In this cell, the architecture parameters, α, corresponding to each edge are updated after each search process step. After that, a derived cell can be obtained by keeping the two strongest edges which are the mixed operations with the highest α (except *zero* operations). A derived cell is depicted in Figure 4.6.

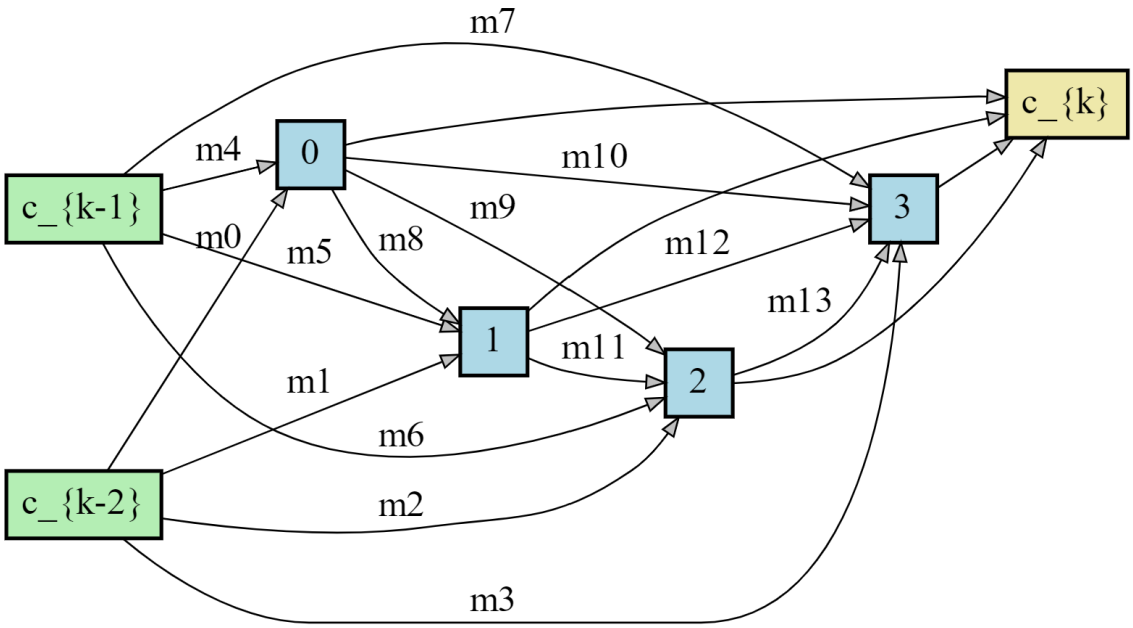

Figure 4.5 DART cell with seven nodes and fourteen edges

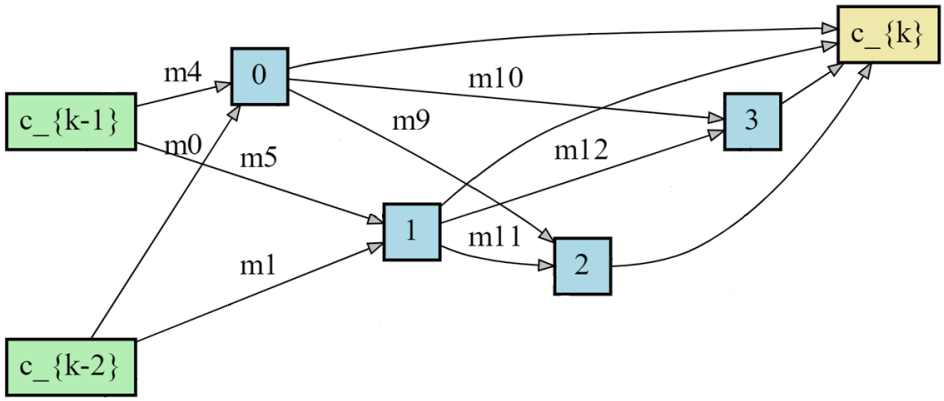

Figure 4.6 A derived cell example

4.1.3 A DARTS numerical example

In this section, we explain DARTS through a numerical example. In this example, we assume that we have a search space including the three following operations:

Search space: O = {o0:3x3_conv, o1:5x5_conv, o2:3x3_maxpool}

We created a DARTS architecture consisting of a 5-node cell (Figure 4.7) to perform the search process. Our cell has five edges, each representing a mixed operation defined according to the equation (4.3). Each mixed operation consists of three operations available in the search space and its corresponding architecture parameters, resulting in 15 values for α, as shown in Figure 4.8.

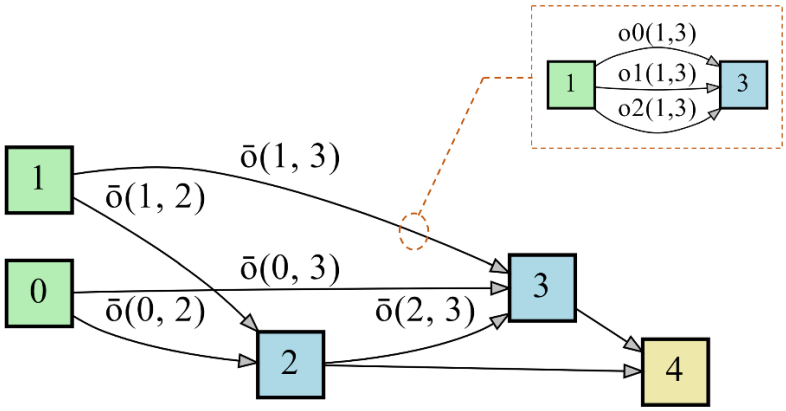

Figure 4.7 A DARTS cell with five nodes and five edges

$$\alpha = \begin{array}{c} \begin{array}{ccc} o0 & o1 & o2 \end{array} \\ \begin{pmatrix} 0.2216 & 0.2502 & 0.4172 \\ 0.9488 & 0.4231 & 0.4195 \\ 0.6891 & 0.0792 & 0.4306 \\ 0.2386 & 0.7345 & 0.1779 \\ 0.8330 & 0.3174 & 0.0807 \end{pmatrix} \end{array}$$

(a)

$$\bar{\alpha} = \text{Softmax}(\alpha) = \begin{array}{c} \\ \bar{o}(0,2) \\ \bar{o}(0,3) \\ \bar{o}(1,2) \\ \bar{o}(1,3) \\ \bar{o}(2,3) \end{array} \begin{array}{c} \begin{array}{ccc} o0 & o1 & o2 \end{array} \\ \begin{pmatrix} 0.3082 & 0.3171 & 0.3747 \\ 0.4587 & 0.2711 & 0.2702 \\ 0.4319 & 0.2347 & 0.3335 \\ 0.2791 & 0.4582 & 0.2627 \\ 0.4835 & 0.2887 & 0.2279 \end{pmatrix} \end{array}$$

(b)

Figure 4.8 (a) Architecture parameter matrix, α and (b) Softmax(α)

After creating a cell, the search process described in Algorithm 1 begins. Then, we can derive an architecture based on the learned α in each iteration. For instance, in Figure 4.7, for node 3, we have three input edges representing three mixed operations described in equations (4.6). In each search iteration, after updating α, we choose the operations with the highest corresponding α value in each mixed operation (see section 4.1.1), as described in Figure 4.9 and Figure 4.10.

$$
\begin{aligned}
\bar{o}(0,3)\,(x^{(0)}) &= \bar{\alpha}_{o0}^{(0,3)} \times o0(0,3)(x^{(0)}) + \bar{\alpha}_{o1}^{(0,3)} \times o1(0,3)(x^{(0)}) + \bar{\alpha}_{o2}^{(0,3)} \times o2(0,3)\left(x^{(0)}\right) \\
\bar{o}(1,3)\,(x^{(1)}) &= \bar{\alpha}_{o0}^{(1,3)} \times o0(1,3)(x^{(1)}) + \bar{\alpha}_{o1}^{(1,3)} \times o1(1,3)(x^{(1)}) + \bar{\alpha}_{o2}^{(1,3)} \times o2(1,3)\left(x^{(1)}\right) \qquad (4.6) \\
\bar{o}(2,3)\,(x^{(2)}) &= \bar{\alpha}_{o0}^{(2,3)} \times o0(2,3)(x^{(2)}) + \bar{\alpha}_{o1}^{(2,3)} \times o1(2,3)(x^{(2)}) + \bar{\alpha}_{o2}^{(2,3)} \times o2(2,3)\left(x^{(2)}\right)
\end{aligned}
$$

$$
\alpha = \begin{array}{c}
\begin{array}{ccc} o0 & o1 & o2 \end{array} \\
\begin{pmatrix}
0.9044 & 0.6855 & 0.2775 \\
0.7793 & 0.9904 & 0.6318 \\
0.1604 & 0.9237 & 0.4468 \\
0.1798 & 0.6322 & 0.3928 \\
0.9938 & 0.9504 & 0.9411
\end{pmatrix}
\end{array}
$$

(a)

$$
\bar{\alpha} = \text{Softmax}(\alpha) = \begin{array}{c|c}
 & \begin{array}{ccc} o0 & o1 & o2 \end{array} \\
\begin{array}{c} \bar{o}(0,2) \\ \bar{o}(0,3) \\ \bar{o}(1,2) \\ \bar{o}(1,3) \\ \bar{o}(2,3) \end{array} & \begin{pmatrix}
0.4278 & 0.3437 & 0.2285 \\
0.3228 & 0.3987 & 0.2785 \\
0.2234 & 0.4792 & 0.2974 \\
0.2625 & 0.4127 & 0.3248 \\
0.3441 & 0.3295 & 0.3264
\end{pmatrix}
\end{array}
$$

(b)

Figure 4.9 α matrix and Softmax(α) after a search process iteration

$$\bar{o}(0,3)\,(x^{(0)}) = 0.3228 \times o0(0,3)(x^{(0)}) + \boxed{0.3987 \times o1(0,3)(x^{(0)})} + 0.2785 \times o2(0,3)\left(x^{(0)}\right)$$

$$\bar{o}(1,3)\,(x^{(1)}) = 0.2625 \times o0(1,3)(x^{(1)}) + \boxed{0.4127 \times o1(1,3)(x^{(1)})} + 0.3248 \times o2(1,3)\left(x^{(1)}\right)$$

$$\bar{o}(2,3)\,(x^{(2)}) = \boxed{0.3441 \times o0(2,3)(x^{(2)})} + 0.3295 \times o1(2,3)(x^{(2)}) + 0.3264 \times o2(2,3)\left(x^{(2)}\right)$$

Figure 4.10 The highest α values in each mixed operation

Between these three mixed operations, we keep the top-2 strongest (see section 4.1.2), those with the highest α. Therefore, we consider $\bar{o}(1, 3)$ and $\bar{o}(0, 3)$ which means o1(1, 3) and o1(0, 3) will be selected as shown in Figure 4.11. We performed the same procedure for the other nodes. The derived cell is depicted in Figure 4.12.

$$\boxed{\bar{o}(0,3)\,(x^{(0)}) : 0.3987 \times o1(0,3)(x^{(0)})}$$

$$\boxed{\bar{o}(1,3)\,(x^{(1)}) : 0.4127 \times o1(1,3)(x^{(1)})}$$

$$\bar{o}(2,3)\,(x^{(2)}) : 0.3441 \times o0(2,3)(x^{(2)})$$

Figure 4.11 Top-2 strongest edges for node 3

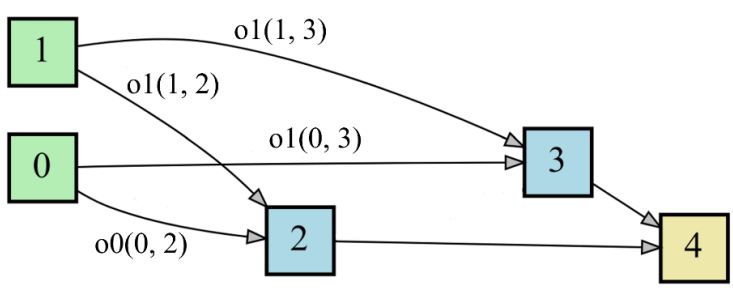

Figure 4.12 A derived cell after a search iteration

4.2 FINN compiler and how it works

Section 3.2 briefly introduced FINN [59][73], an open-source framework developed by Xilinx research labs to accelerate and deploy QNNs on FPGAs. Since QNNs and BNNs are neural networks that use, respectively, quantized and binary weights and activations, they are well-suited for deployment on resource-constrained devices such as embedded systems, FPGAs, and ASICs. The FINN framework provides various tools for designing, training, and deploying QNNs. It

implements multiple deep learning models (e.g., VGG16, Mobilenet-v1, Resnet50) optimized for deployment on FPGA devices. FINN developers also have provided a high-level API for designing and training QNNs called Brevitas, making it easy to experiment with different network architectures and sets of hyperparameters. Brevitas is a software framework for developing and deploying quantized neural networks [76]. This framework is implemented in PyTorch and offers support for a wide range of quantization techniques, which are implemented across various frameworks and compilers through a unified API. In addition, the framework enables inference acceleration for certain combinations of layers and types of quantization by exporting to FINN, *onnxruntime*, or Pytorch's quantized operators. Brevitas has proven to be effective in both research projects and large-scale commercial deployments and is compatible with CPUs, GPUs, and custom accelerators running on Xilinx FPGAs. However, according to its developers, the primary focus of the framework is on affine quantization, particularly uniform quantization, and non-uniform quantization is not currently supported [77].

Utilizing FINN begins by designing and training a QNN model with Brevitas and obtaining a Brevitas FINN-ONNX model. ONNX (Open Neural Network Exchange) is an open-source standard format for representing machine learning models. The next step is to input the ONNX model, the target FPGA device, FINN parameters such as clock frequency, FPS (Frames per Second), and the number of parallel processing elements for matrix-vector multiplication (MVU) into the FINN synthesizer. Finally, using this set of parameters [74], FINN generates the necessary Vivado HLS codes through a series of transformations. Figure 4.13 shows the FINN compiler transformation steps. The white boxes represent the state of the network at each step, while the coloured areas show the transformations applied to achieve a specific outcome. The diagram has four sections, each represented by a different colour, with each section including several flow steps. The flow begins with the Brevitas export (yellow area), followed by the preparation of the network (green area) for synthesis using Vivado HLS and stitching using Vivado IP (blue area). This section will perform a sequence of transformations on the network model to form a FINN-style dataflow architecture (Figure 4.14) in which the appropriate amount of computing resources (e.g., MVUs and BRAMs) is allocated for each layer. Further, function calls are utilized to instantiate the compute arrays, optimizing Vivado HLS building blocks from the *finn-hlslib* library.

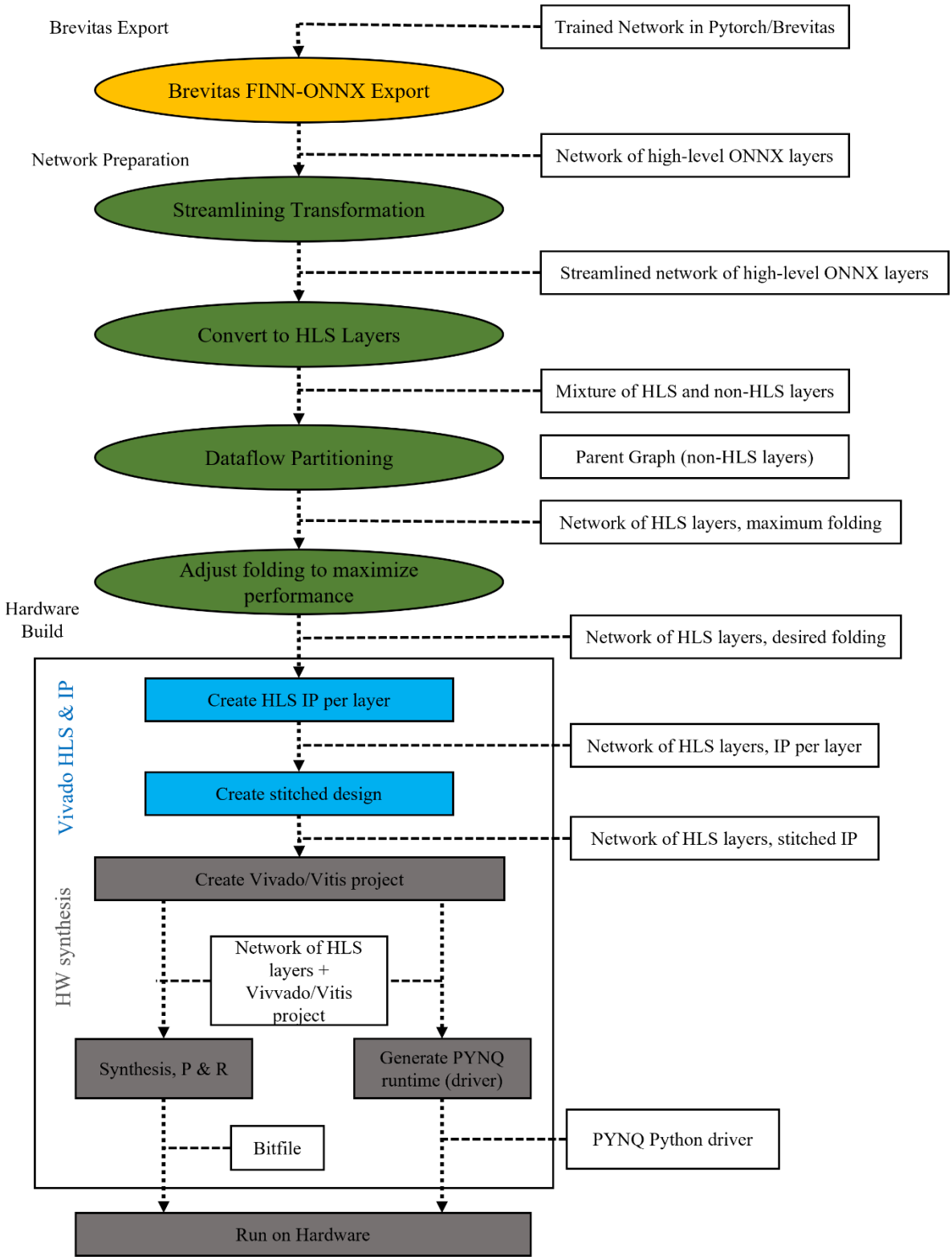

Figure 4.13 The FINN compiler transformation steps [75]

However, since these function calls can only handle particular patterns, the network must be transformed into an appropriate form so that the network layers can be replaced with these function calls. The last step is generating hardware from the network model. FINN provides two transformations, ZynqBuild and VitisBuild, for building the accelerator and integrating it into an appropriate shell, as well as generating a bitfile, depending on whether the target platform is a Zynq or Alveo. For example, in the grey area, it makes a PYNQ bitfile and runs it on a PYNQ board [75].

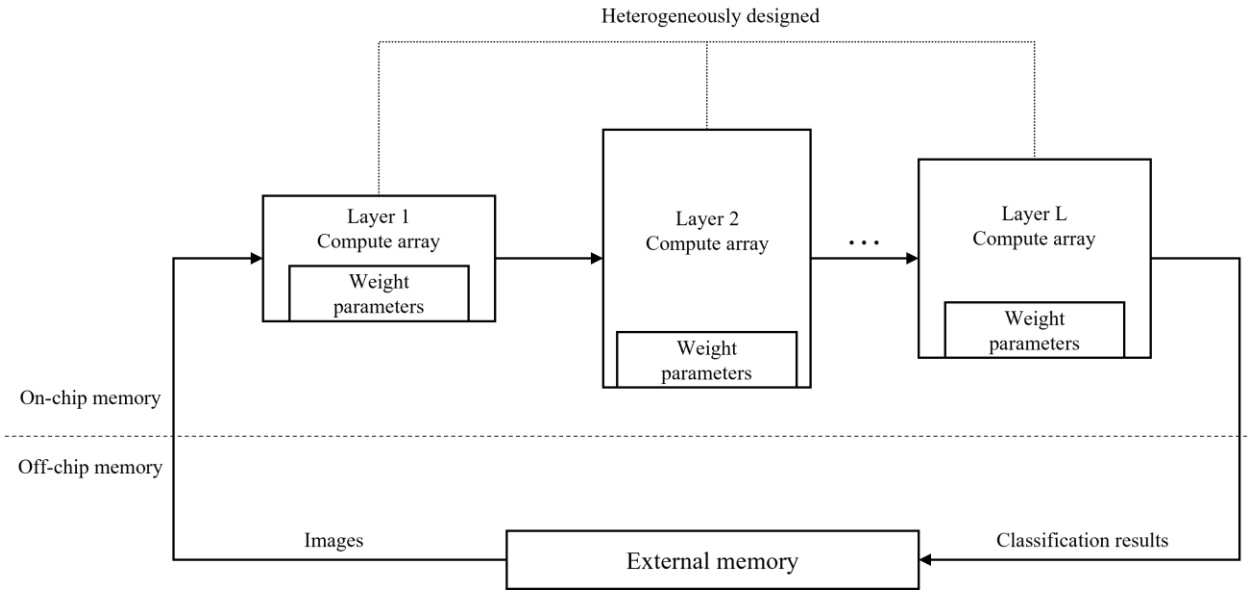

Figure 4.14 FINN dataflow architecture [73]

4.3 Proposed approach to use the DARTS framework for QNN design

This project's primary contribution is utilizing the DARTS framework to discover high-performing quantized neural networks in terms of accuracy and latency for possible implementation in FPGAs with the FINN compiler. In order to achieve this, the DARTS framework must be modified to accommodate QNNs.

In the very first step, we examined the available code for the DARTS project [78] to identify all classes and functions written in standard Pytorch form. Then, we swapped Pytorch functions and

classes regarding search space operations, such as convolutions and poolings, with their Brevitas equivalents. For example, Listing 4.1 shows a standard Pytorch convolution and its Brevitas equivalent. The original DARTS framework uses a variety of operations, such as separable and dilated convolutions, max pooling, average pooling, identity, and zero operations in its search space. However, we realized it was not feasible to use separable and dilated convolutions as the FINN compiler's automatic standard build flow does not support them and requires an advanced custom build script that must be done manually. Therefore, to comply with the FINN standard build flow, we substituted separable and dilated convolutions with standard convolutions. More information about these two types of convolutions can be found in Appendix B.

Listing 4.1 Pytorch and Brevitas convolution example

```
import torch
from torch.nn as import Conv2d
from brevitas.nn import QuantConv2d

#standard Pytorch convolution
Conv2d(kernel_size=kernel_size,
       in_channels=in_ch,
       out_channels=out_ch,
       bias=False)
#Brevitas quantized convolution
Conv2d(kernel_size=kernel_size,
       in_channels=in_ch,
       out_channels=out_ch,
       bias=False,
       weight_quant=CommonWeightQuant,
       weight_bit_width=weight_bit_width)
```

Generally, the FINN framework standard build flow can handle only standard models such as VGG with consecutive layers. It does not support more complex models with residual connections, such as Resnet50. This presents a challenge for implementing DARTS cells, which have numerous

residual connections between both nodes and cells. Having said that, the FINN developers have recently announced support for Resnet blocks [63], which is available on their GitHub repository [79]. Still, significant modifications to the build script and access to an FPGA device with increased hardware resources are necessary to fit Resnet50 into the device. Another challenge we had to deal with was measuring the latency of the generated model, since acquiring the latency report from FINN was time-consuming and not a practical method to incorporate into the DARTS searching process.

Consequently, it was decided to estimate the generated model's latency based on each individual operation implemented through FINN. As a result, a lookup table comprising operation latencies was created, and a function was written to obtain the generated model at each search step and calculate the total latency. This effectively eliminated the need to measure real-time latency through FINN reports. Table 4.1 demonstrates the model's latency measurement based on the cell structure depicted in Figure 4.15, taking into account the parallelization that can be achieved within each cell. This example is the general form of a cell in which all possible edges are available. The label *L* on edges represents the latency of the corresponding operation between nodes. Two numerical examples are provided in Figure 4.16 and Figure 4.17.

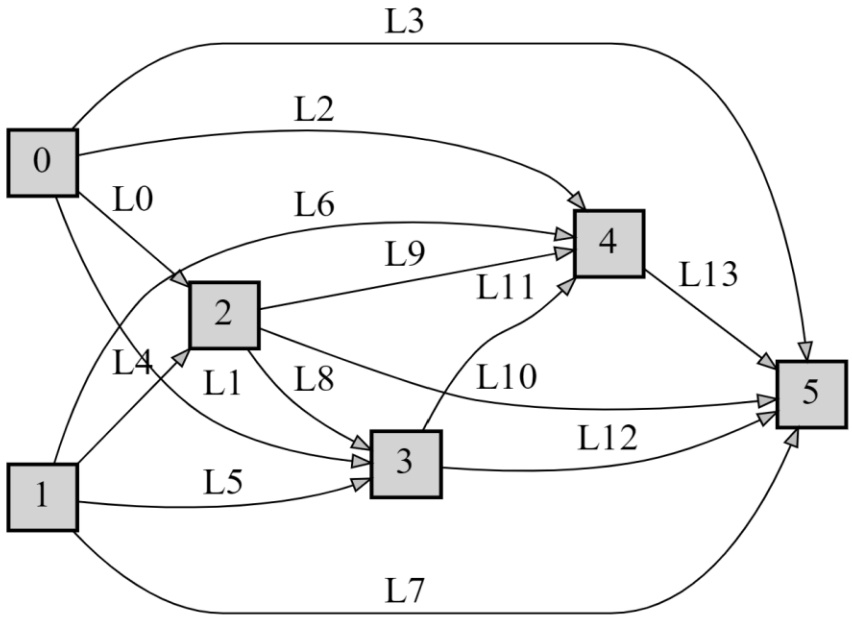

Figure 4.15 A latency graph for a DARTS cell with six nodes

Table 4.1 Calculating the latency for a DARTS cell with six nodes (Figure 4.15)

Nodes	**2**	**3**	**4**	**5**
Latency	T2 = Max(L0, L4)	T3 = Max(T2+L8, L1, L5)	T4 = Max(T2+L9, T3+L11, L2, L6)	T5 = Max(T2+L10, T3+L12, T4+L13, L3, L7)

The latency was calculated by assuming that the input edges to each node represent operations that can be executed in parallel. For each example, the latency table displays the total latency in the last column, which is calculated based on the latencies of the preceding nodes denotes as T in Table 4.1. So, for the graphs in Figure 4.16 and Figure 4.17, the total latencies are 24 and 27 (a unit of second), as shown in Table 4.2 and Table 4.3 respectively.

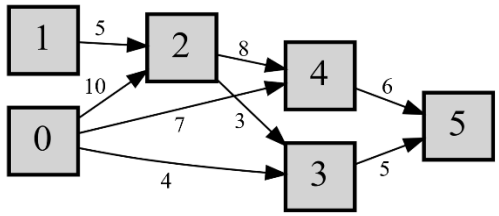

Figure 4.16 Latency graph, sample 1

Table 4.2 Latency table, sample 1 (Figure 4.16)

Nodes	**2**	**3**	**4**	**5**
Latency	T2 = Max(10, 5) = 10	T3 = Max(T2+3, 4) = 13	T4 = Max(T2+8, 7) = 18	T5 = Max(T3+5, T4+6) = 24

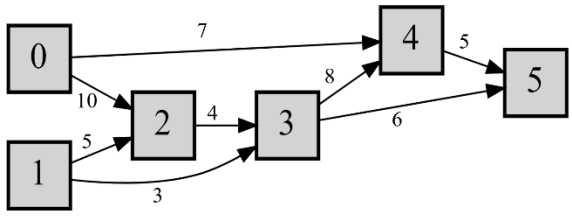

Figure 4.17 Latency graph, sample 2

Table 4.3 Latency table, sample 2 (Figure 4.17)

Nodes	**2**	**3**	**4**	**5**
Latency	T2 = Max(10, 5) = 10	T3 = Max(T2+4, 3) = 14	T4 = Max(T3+8, 7) = 22	T5 = Max(T3+6, T4+5) = 27

In choosing the quantization bitwidth, we initially decided to use 1, 2 and 3-bit quantization schemes for weights and activations. However, this decision was not successful as it did not yield acceptable results. As a result, we increased the bitwidth to 4, which led to improved results. Therefore, all experiments are performed considering a 4-bit quantization scheme in this thesis.

As we mentioned earlier in this section, we aim to consider latency as a hardware criterion in the search process to discover low-latency QNNs, so it is required to integrate this factor alongside accuracy loss into the loss function. In doing so, previous works have attempted to use NAS to explore low-latency deep neural networks, which is one of the main focuses of HW-NAS. For example, Wu et al. in FBNet [72] used the loss function described in equation (4.8), while Srinivas et al. [80] used a similar loss function as indicated in equation (4.9). In this project, we were inspired by these implementations and applied them in a similar manner in which $\boldsymbol{CE}$ is Cross-Entropy loss which represents accuracy, $\boldsymbol{\eta}$ and $\boldsymbol{\tau}$ tune the latency penalty, and $\boldsymbol{\alpha}$ represents architecture parameter described in section 4.1.2.

Our loss function: $$L(\alpha, w_\alpha) = CE(\alpha, w_\alpha) + \eta * \log\,(LAT(\alpha))^\tau \quad (4.7)$$

FBNet [72]: $$L(\alpha, w_\alpha) = CE(\alpha, w_\alpha)\,.\ \eta \log\,(LAT(\alpha))^\tau \quad (4.8)$$

Srinivas et al. [80]: $$L(\alpha, w_\alpha) = CE(\alpha, w_\alpha) + \eta * LAT(\alpha)^\tau + \gamma * ENER(\alpha)^\delta \quad (4.9)$$

In the next chapter, we present results from experiences performed to observe the effects of integrating the latency into the loss function. We also show that it is feasible to develop some performant QNNs considering the accuracy and latency using our methodology described in this section.

CHAPTER 5 EXPERIMENTAL SETUP, RESULTS AND DISCUSSIONS

In this chapter, we first explain in detail how the DARTS search process is configured. We then explain how we select the candidate networks, and we evaluate the candidate networks' performance in terms of accuracy and latency on the CIFAR10 dataset. Finally, we discuss the results and compare them with the literature.

5.1 Experimental Setup

This section explains different DARTS configurations and details the FINN compiler settings we used for our experiments.

5.1.1 DARTS Configuration

In the original DARTS paper, the authors chose an 8-cell network for the search phase [21]. Then, after finding the desired cells, they expanded the 8-cell network into a 20-cell network before obtaining and reporting their results.

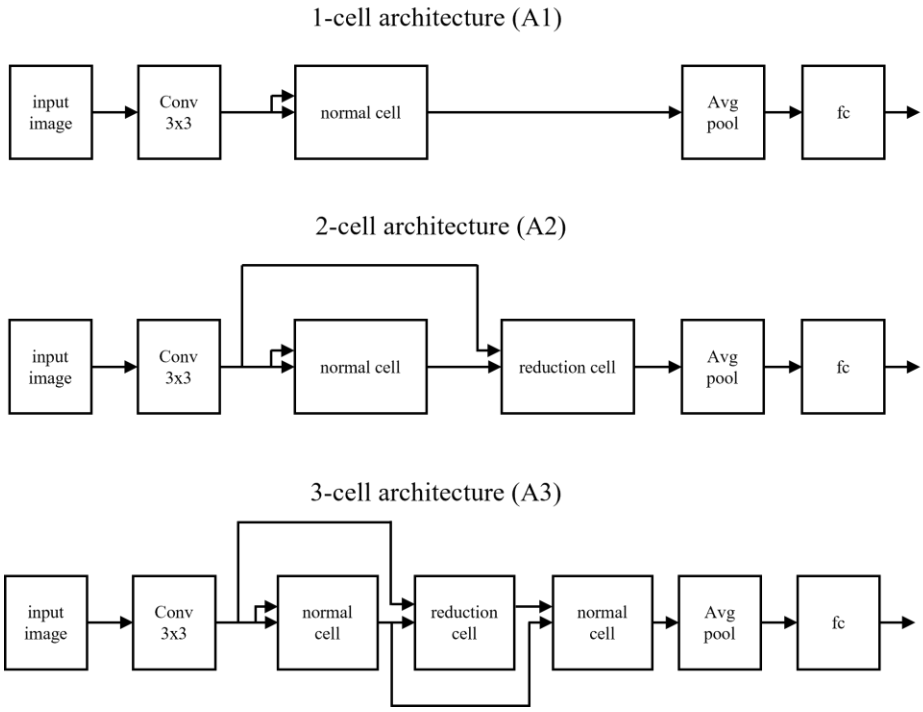

Figure 5.1 DARTS architectures for different experiments

However, for our experiences, to keep the network architecture small, we chose a smaller number of cells (1, 2 and 3) for the search phase and the evaluation phase. Figure 5.1 shows these three configurations in which reduction cells are fed by reduced input data size using stride two. Regarding the loss function, as seen in section 4.3, we use the loss function given in equation (4.7) with parameters η (eta) and τ (tau) to control the relative importance of latency. Table 5.1 shows the values of these parameters for each experiment.

Table 5.1 Loss function configurations

Experiment #	$\boldsymbol{\eta}$	$\boldsymbol{\tau}$
1	0.0	0.0
2	0.2	0.5
3	0.2	1.0
4	0.4	0.5
5	0.4	1.0
6	0.6	0.5
7	0.6	1.0
8	0.8	0.5
9	0.8	1.0
10	1.0	0.5
11	1.0	1.0

For the search process, we use the following operations: 3 × 3 and 5 × 5 quantized standard convolution, 3 × 3 max pooling, identity, and zero. For all operations [29], we use stride and

padding parameters as required to keep the convolved feature map dimensions unchanged, as proposed by Liu et al. [21]. We also use the CONV-BN-ACT (Convolution-BatchNormalization-Activation) order for a convolution block, as proposed by Umuroglu et al. [59]. For all experiments, each cell consists of seven nodes and a network is made of multiple interconnected cells. We reserve half of the training data as the validation set. In the search process, we train the network for 50 epochs, with batch size 128 for both the training and validation sets. The number of channels for the first 3 × 3 convolution block is set to 16. All the other settings are the same as the original DARTS paper, and they can be found in Appendix A.

After the search process is finished, we analyze the results for accuracy and latency. Ideally, we are looking for a network with the highest accuracy and the lowest latency, and we need to analyze the results and consider our application conditions before selecting a network. As a result, regarding given applications, we might choose networks with different combinations of accuracy and latency. Nonetheless, after choosing our desired network, we train it from scratch for 600 epochs and evaluate its accuracy with the test set [21].

5.1.2 FINN Configuration

As discussed in section 4.2, we measure the latency of each operation described in section 5.1.1 in order to create a latency lookup table which we further utilize in the latency estimation during the search process. In the FINN configuration settings, when we increase the bit width, the latency will increase unless we consume more resources (e.g., LUT, BRAMs) to increase the parallelization and keep the latency low for each operation (convolution, max pool). Therefore, one concern could be finding a bit width that best fits our project. We realized that it depends on the available hardware resources in the selected FPGA board. So, we considered different bit widths and FINN parameters, such as frames per second (FPS) and the number of parallel processing elements for matrix-vector multiplication (MVU), to find out how resource consumption changes when we choose different bit widths. Doing so, we first set the FINN compiler parameters FPS and MVU to high values [74] (Table 5.2) to get the best latency and FPS then we set them to lower values (Table 5.3).

Table 5.2 FINN parameters for FPS=20 k and MVU=100 k

Op	LUT	BRAM	Latency (us)	FPS (after compilation)
Conv_3x3_1bit	15.5 k	8	170.2	14 k
Conv_3x3_2bit	17.2 k	16	170.2	14 k
Conv_3x3_4bit	21.5 k	32	170.2	14 k
Conv_3x3_8bit	63.5 k	64	170.2	14 k

Table 5.3 FINN parameters for FPS=1 k and MVU=1 k

Op	LUT	BRAM	Latency (us)	FPS (after compilation)
Conv_3x3_1bit	2.1 k	1	1468.3	1.3 k
Conv_3x3_2bit	2.3 k	2	1468.3	1.3 k
Conv_3x3_4bit	3.1 k	3	1468.3	1.3 k
Conv_3x3_8bit	17.3 k	6	1468.3	1.3 k

As expected, configuring the FINN parameters to lower values, as shown in Table 5.3, increases latency and decreases resource consumption. As a result, we must consider an FPGA board as a baseline and examine the FINN parameters (e.g., FPS and MVU) and the bit width to have the whole generated network fit on the device. Table 5.4 lists four FPGA boards and their available resources. In our project, we set FINN parameters for the Zynq-7000 FPGA board, as done in [59].

Table 5.4 FPGA platform summary [59]

Platform	Part #	Node (nm)	DDR (GB)	kLUTs	BRAMs
AWS F1	XCVU9P-FLGB2104-2-I	16	64	1180	4320
Ultra96	XCZU3EG-SBVA484-1	16	2	71	432
PYNQ-Z1	XC7Z020-1CLG400C	28	0.5	53.2	280
Zynq-7000	XC7Z045-2FFG900C	28	2	218.6	545

5.2 Results and discussion

For this project, all experiences were done considering 4-bit quantization for weights and activations. In this section, we analyze the accuracy and latency of our work on the CIFAR-10 [81] dataset. In section 5.1.1, we mentioned that we used three types of architectures with a different number of cells to carry out the search process, and we considered different loss function configurations regarding latency. As a result, we have 11 different latency configurations for each of the three architectures, resulting in 33 different experiments. We have provided search plots for all experiments in which each epoch indicates a search iteration described in Algorithm 1. So, after the search process is finished, we can derive a network based on our latency and accuracy criteria. Afterwards, we train the chosen networks for 600 epochs and evaluate them with the test set to obtain the final accuracy.

5.2.1 Results for 1-cell architecture

In experiment 1, the values of η and τ are set to 0.0 to obtain maximum accuracy without considering latency. In this experiment, the search achieved the best accuracy but the worst latency compared to the other experiments (Figure 5.2, Figure 5.3 and Figure 5.4). During experiment 8 and 10, we observed a sudden drop in accuracy after epoch 47 and 41, respectively. This could be related to the limited learning capacity of certain operations, such as *skip_connect* and

max_pooling, and their dominance when η and τ increase. The plots in Figure 5.5 show the experiments 1, 2 and 11 which achieve the highest accuracy without considering latency, the highest accuracy considering latency and the best latency, respectively. For all experiments, to select a network from the search plot, we chose the one with the highest accuracy, indicated by coloured arrows in the corresponding figures. Table 5.5 displays the test accuracy and latency of the selected networks after 600 epochs of training.

Figure 5.2 Search plots for experiment 1 – experiment 3 (1-cell)

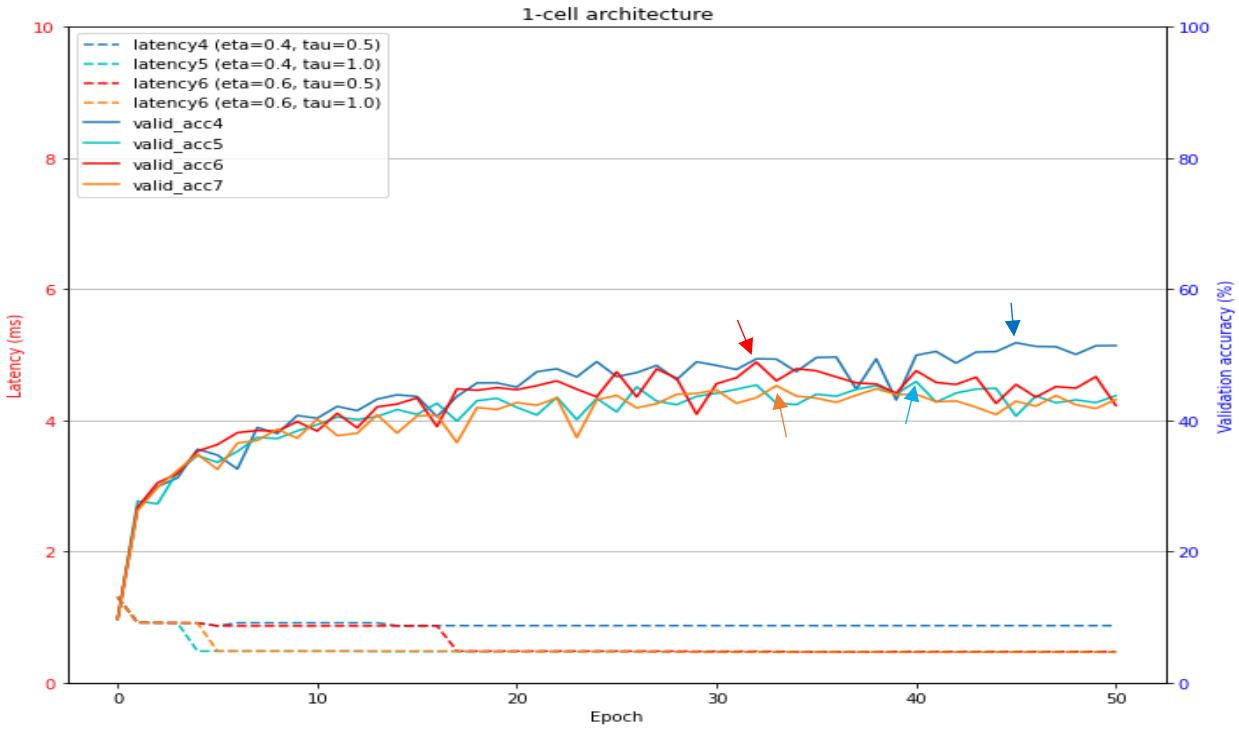

Figure 5.3 Search plots for experiment 4 – experiment 7 (1-cell)

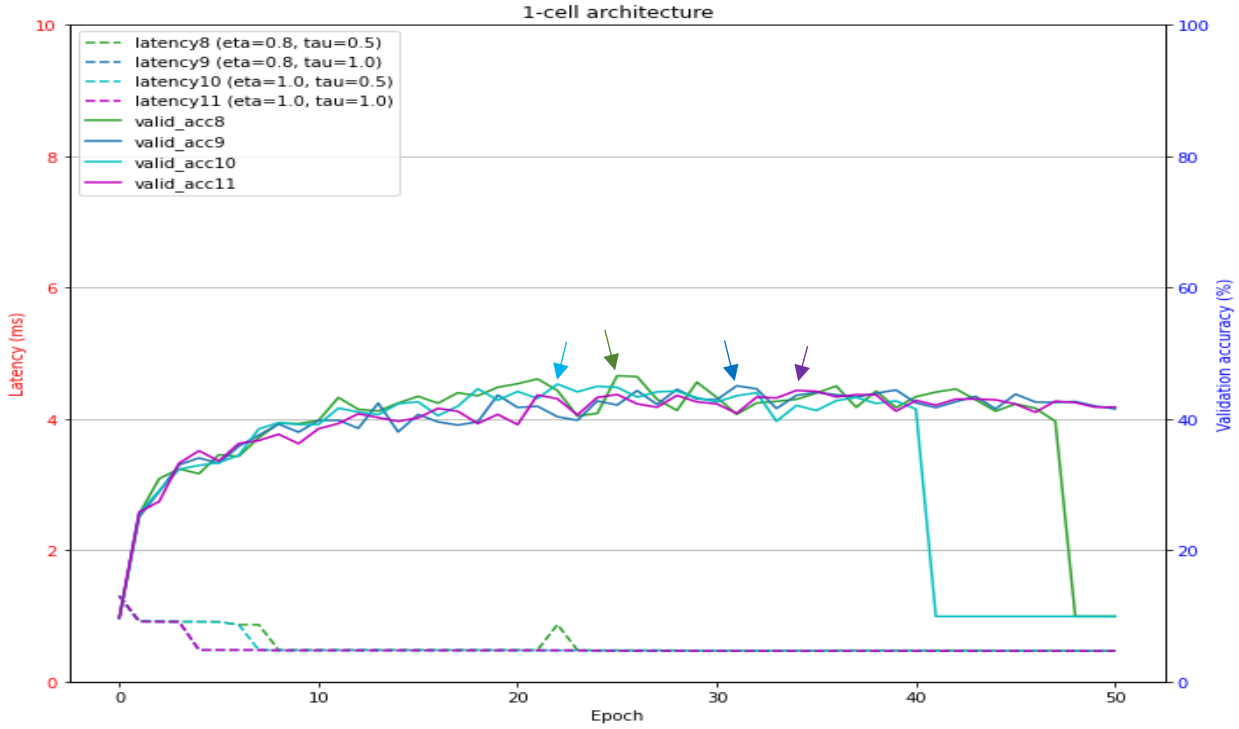

Figure 5.4 Search plots for experiment 8 – experiment 11 (1-cell)

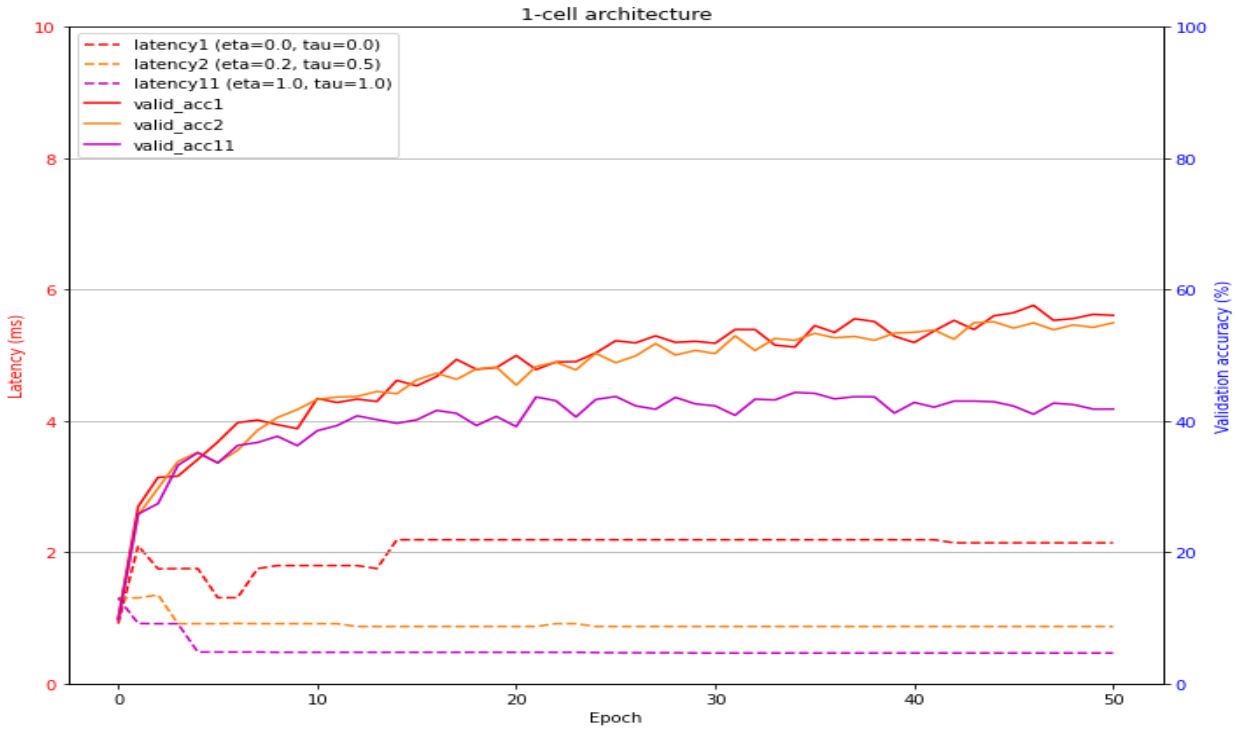

Figure 5.5 Search plots for three experiments 1, 2, and 11 (1-cell)

Table 5.5 shows that a higher value of η corresponds to lower latency. The network derived from experiment 11 has the lowest latency (0.46 ms), while the network from experiment 1 achieves the highest accuracy (63.06%). Although experiment 2 achieves an accuracy around 4.31% lower than experiment 1, it has better latency, nearly 2.46× lower than that of experiment 1. After experiment 5, we can notice that the latency value is no longer decreasing. This is likely because the search process is not able to find any combination of operations that can reduce latency more due to the dominance of low-latency operations like *skip_connect* as it happens in experiment 11 (Figure 5.8). Figure 5.6, Figure 5.7, and Figure 5.8 display the cells found during the search for experiments 1, 2 and 11. Figure 5.8 illustrates that for cells obtained from experiment 11, there are no convolution operations between nodes. This is because we increased η and τ to 1.0, which implies that the loss will be more affected by latency. Consequently, operations with lower latency (such as skip_connect and max_pool), which lack learning capacity, are favored during the search process.

Table 5.5 Test set evaluation for derived 1-cell architectures

Experiment #	$\boldsymbol{\eta}$	$\boldsymbol{\tau}$	**Test accuracy (%)**	**Latency (ms)**
1	0.0	0.0	**63.06**	2.14
2	0.2	0.5	58.75	0.87
3	0.2	1.0	54.53	0.48
4	0.4	0.5	57.95	0.87
5	0.4	1.0	47.47	0.46
6	0.6	0.5	53.08	0.48
7	0.6	1.0	48.67	0.46
8	0.8	0.5	54.46	0.48
9	0.8	1.0	48.88	0.46
10	1.0	0.5	54.25	0.48
11	1.0	1.0	50.14	0.46

In contrast, accuracy and latency increased in experiment 1, most likely because the loss function did not include latency parameters. Hence, during this scenario, the search process prioritized operations that led to higher accuracy without considering latency. As a result, the corresponding cell from this experiment, depicted in Figure 5.6, contains more convolutions than those in other experiments. Nevertheless, experiment 2 achieved a reasonable balance between accuracy and latency, as Figure 5.7 shows that the corresponding cells contained a mix of convolution and max_pooling operations.

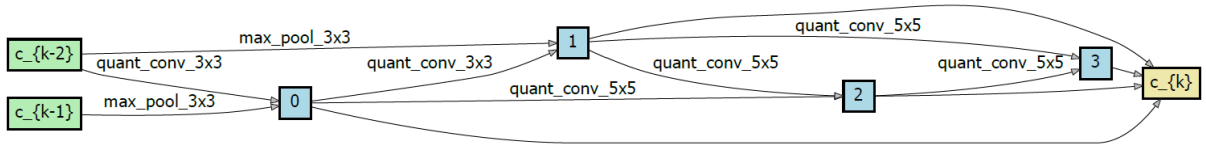

Figure 5.6 Normal cell obtained from experiment 1 (1-cell)

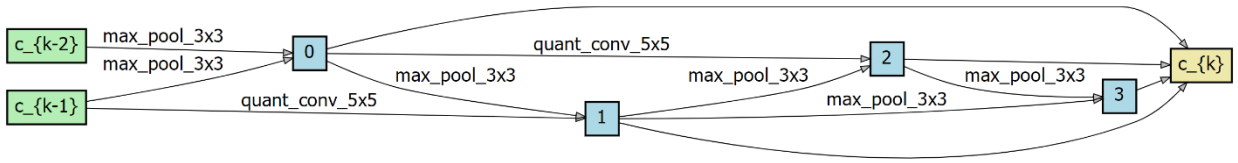

Figure 5.7 Normal cell obtained from experiment 2 (1-cell)

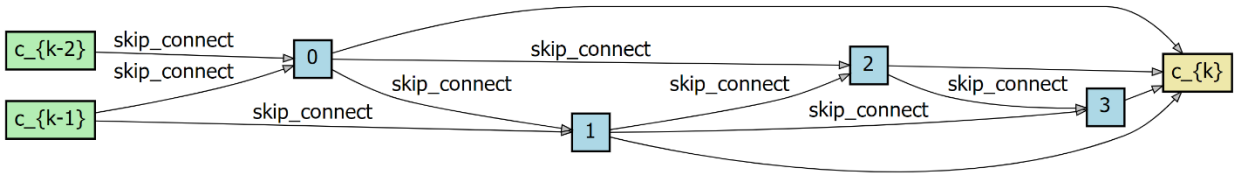

Figure 5.8 Normal cell obtained from experiment 11 (1-cell)

5.2.2 Results for 2-cell architecture

For this configuration, the search conducted in experiment 1 (Figure 5.9) achieves the best accuracy but the worst latency compared to searches conducted in other experiments, as can be seen in Figure 5.9, Figure 5.10 and Figure 5.11.This is expected because the factors η and τ are set to 0.0 in the loss function. We can see that by increasing the values of η and τ, the accuracy and latency drop gradually, however the rate of dropping in not the same when we fixed one of the latency parameters. Similar to section 5.2.1, the next step is choosing a network from the search plot based on its accuracy and latency. For all experiments, we chose the derived network indicated by coloured arrows in related figures, based on the highest accuracy. Table 5.6 shows the test accuracy and latency of the chosen networks after 600 epochs of training.

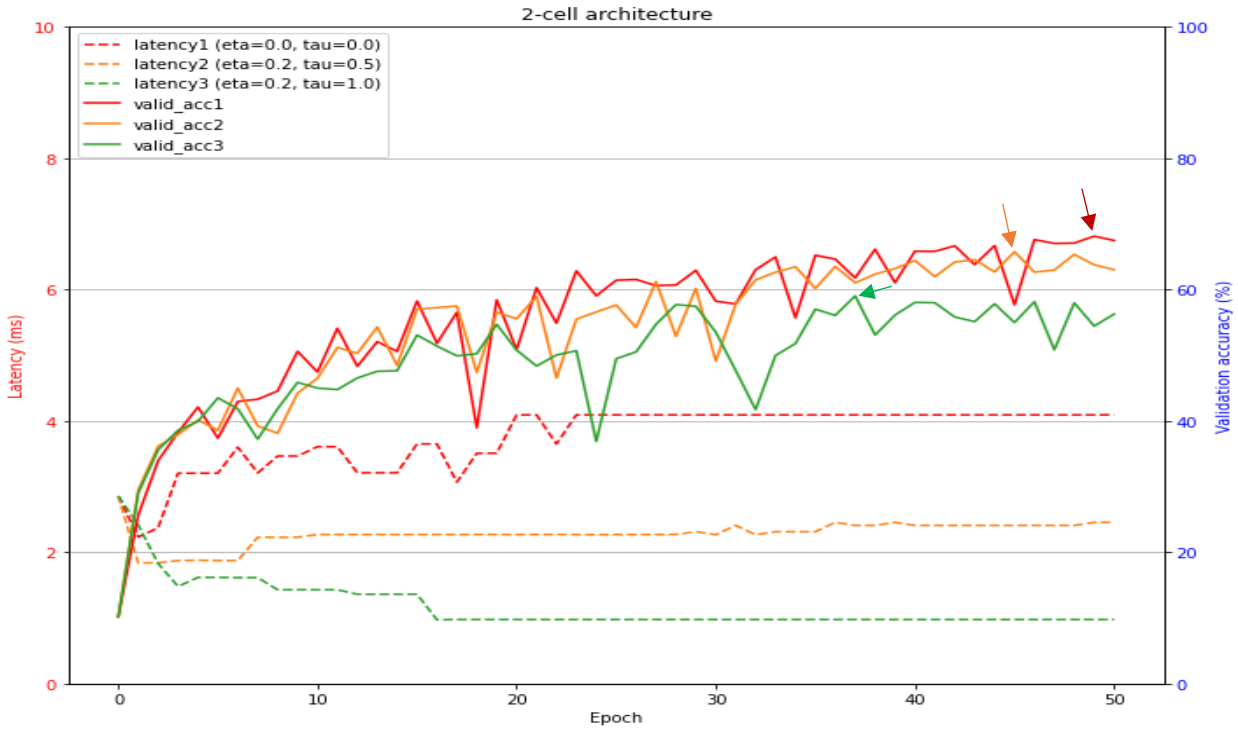

Figure 5.9 Search plots for experiment 1 – experiment 3 (2-cell)

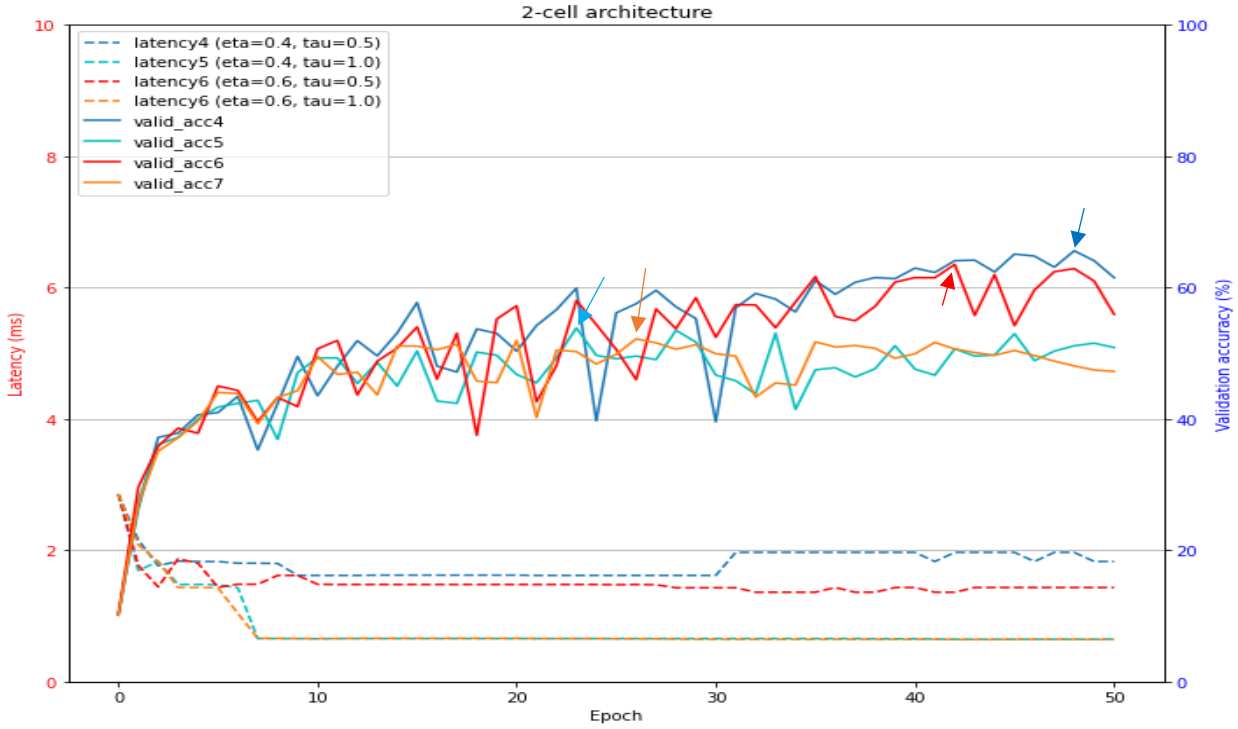

Figure 5.10 Search plots for experiment 4 – experiment 7 (2-cell)

Figure 5.11 Search plots for experiment 8 – experiment 11 (2-cell)

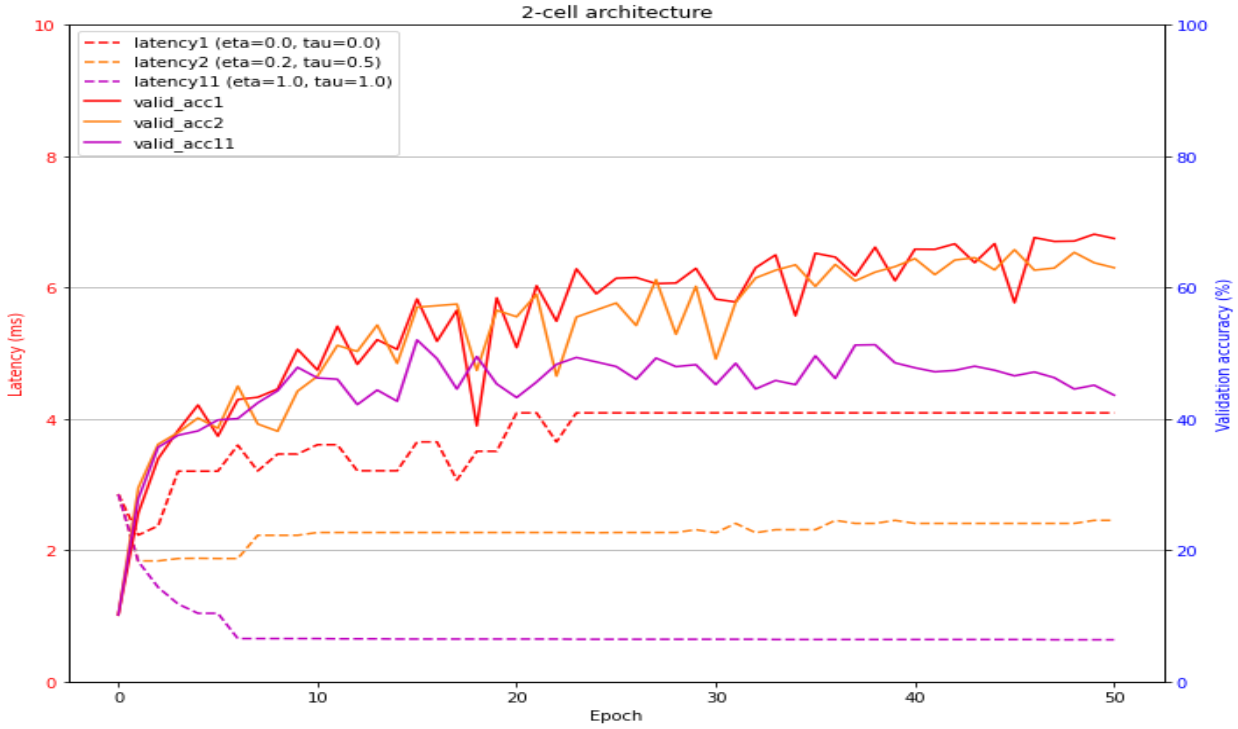

Figure 5.12 Search plots for three experiments 1, 2, and 11 (2-cell)

Table 5.6 Test set evaluation for derived 2-cell architectures

Experiment #	$\boldsymbol{\eta}$	$\boldsymbol{\tau}$	**Test accuracy (%)**	**Latency (ms)**
1	0.0	0.0	**77.93**	4.09
2	0.2	0.5	74.30	2.41
3	0.2	1.0	61.94	0.97
4	0.4	0.5	70.50	1.97
5	0.4	1.0	60.95	0.65
6	0.6	0.5	68.61	1.36
7	0.6	1.0	60.77	0.65
8	0.8	0.5	61.81	0.65
9	0.8	1.0	60.77	0.65
10	1.0	0.5	61.49	0.65
11	1.0	1.0	60.98	**0.64**

According to Table 5.6, an increase in η and τ results in a decrease in latency. The network obtained from experiment 11 exhibits the lowest latency (0.64 ms), while the network obtained from experiment 1 shows the highest accuracy (77.93%). Despite achieving an accuracy around 2.63% lower than experiment 1, experiment 2 has better latency, approximately 1.70x lower than that of experiment 1. It is worth noting that the drop in latency and accuracy is different when latency parameters increase. For example, in experiment 2 and 3, we can observe that by fixing η to 0.2 and increasing τ from 0.5 to 1.0, accuracy and latency decrease 12.36% and 1.44 ms, respectively. This is likely because of exponential effect of τ in the loss function. The following graphs, Figure

5.13-Figure 5.18, show the cells discovered from the search process for experiments 1, 2 and 11. For the cells from experiment 11, as shown in Figure 5.17 and Figure 5.18, there are no convolution operations between nodes as we similarly observed in the previous configuration in section 5.2.1, and this is because we set η and τ to 1.0, which means the loss will be more penalized by latency. Therefore, the search process tends to choose operations with lower latency (e.g., *skip_connect* and *max_pool*), which do not have learning capacity. On the other hand, in experiment 1, the accuracy and latency increased by removing latency parameters in the loss function. Therefore, similar to 1-cell experiments, in this scenario, the search process tends to choose those operations that lead to a higher accuracy without considering the latency. In this way, the related cell from this experiment, shown in Figure 5.13 and Figure 5.14, contained more convolutions than the others. However, experiment 2 achieves a balance between accuracy and latency, as can be seen in Figure 5.15 and Figure 5.16, and the corresponding cells contain a combination of *convolution*, *max_pooling* and *skip_connect* operations.

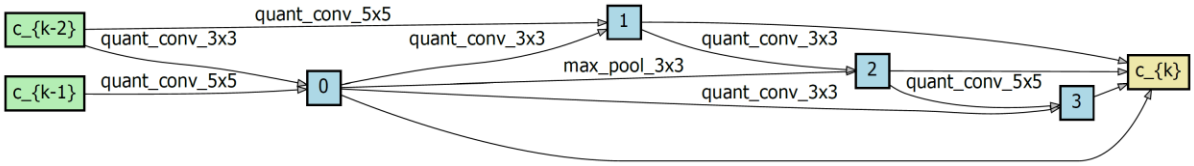

Figure 5.13 Normal cell obtained from experiment 1 (2-cell)

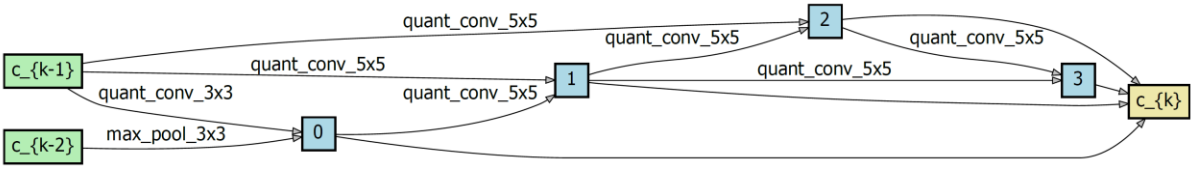

Figure 5.14 Reduction cell obtained from experiment 1 (2-cell)

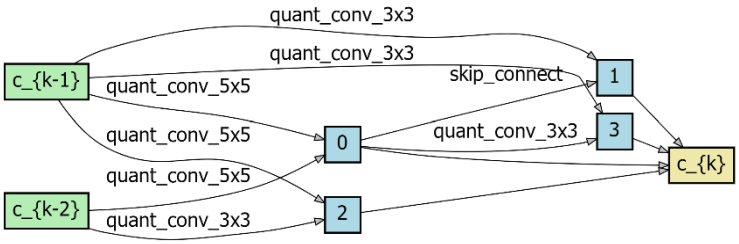

Figure 5.15 Normal cell obtained from experiment 2 (2-cell)

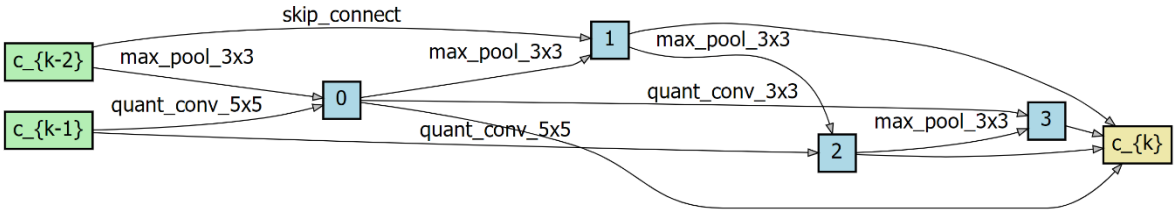

Figure 5.16 Reduction cell obtained from experiment 2 (2-cell)

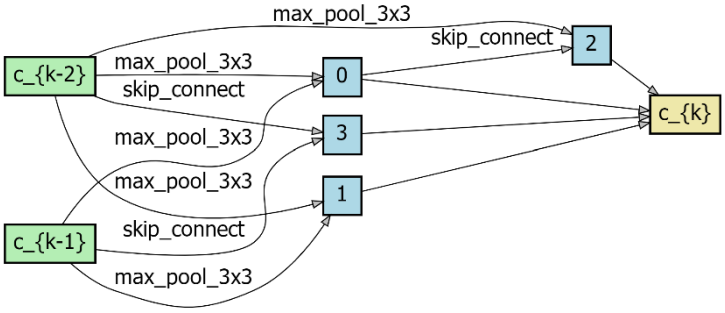

Figure 5.17 Normal cell obtained from experiment 11 (2-cell)

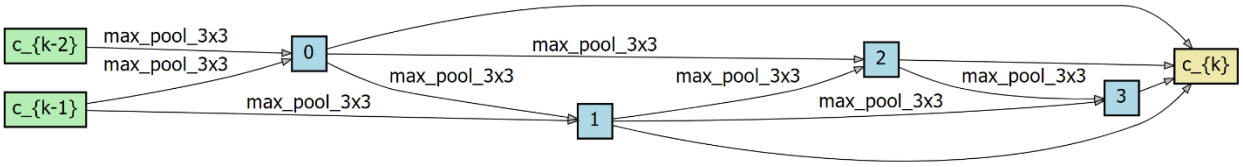

Figure 5.18 Reduction cell obtained from experiment 11 (2-cell)

5.2.3 Results for 3-cell architecture

The accuracy and latency results for a 3-cell architecture are presented in Figure 5.19, Figure 5.20 and Figure 5.21. The results indicate that experiment 1 attained the highest accuracy but a lower latency than other experiments. The observed difference is due to the factor η and τ utilized in the loss function, in which τ is varied between two values, 0.5 and 1.0, while η has five values ranging from 0.2 to 1.0, with a difference of 0.2 between each value through experiment 2 and experiment 11. In experiment 1, we set η and τ to 0.0 to obtain the maximum accuracy while ignoring latency. Figure 5.22 illustrates the comparison among experiments 1, 2 and 11.

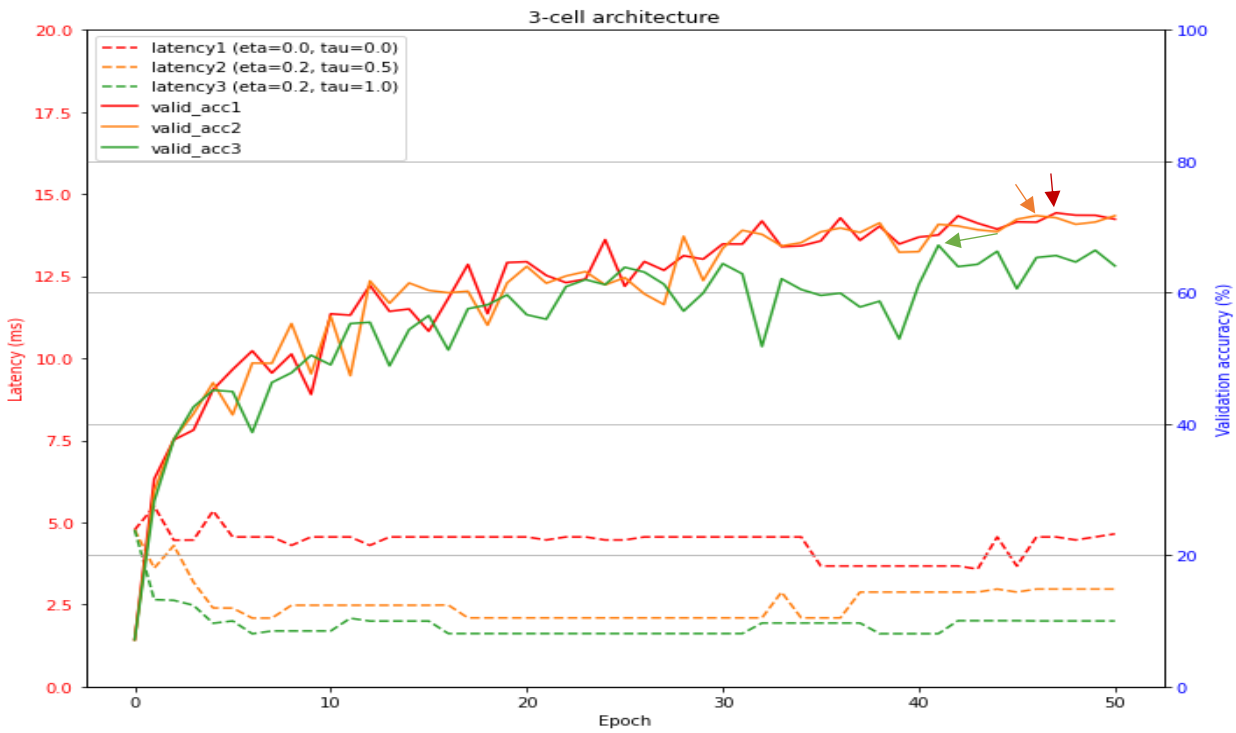

Figure 5.19 Search plots for experiment 1 – experiment 3 (3-cell)

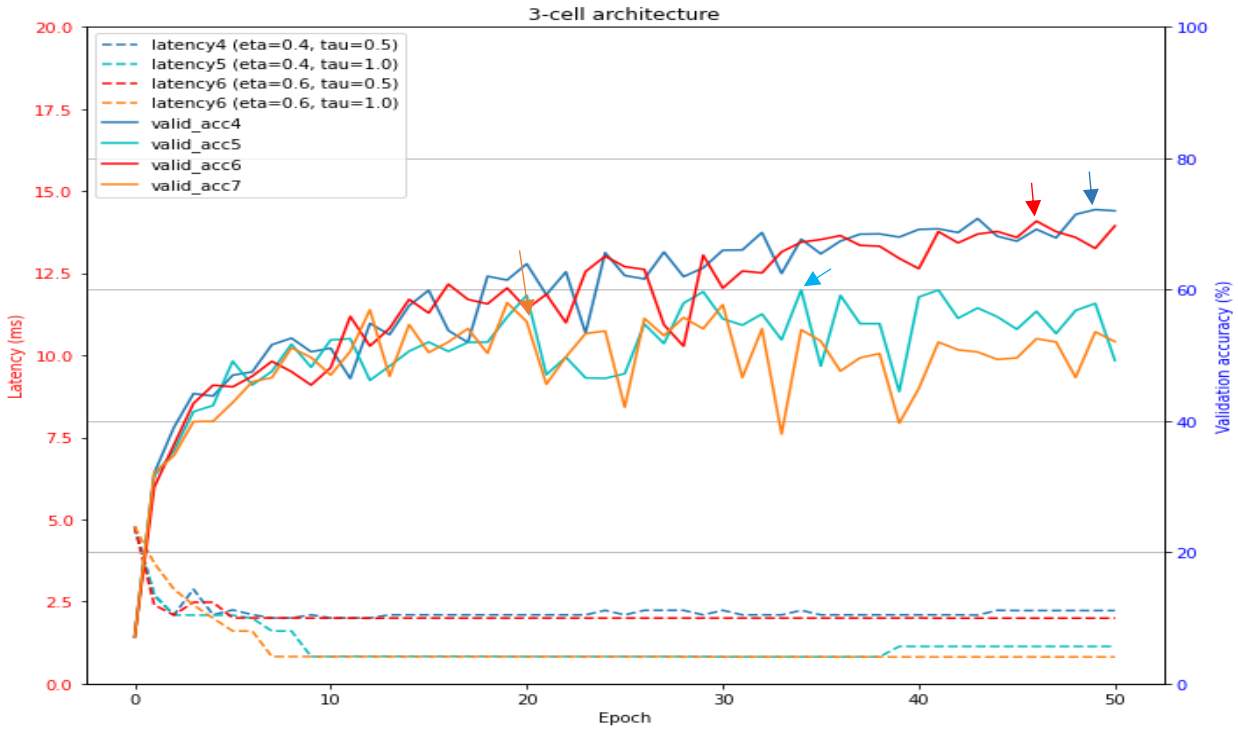

Figure 5.20 Search plots for experiment 4 – experiment 7 (3-cell)

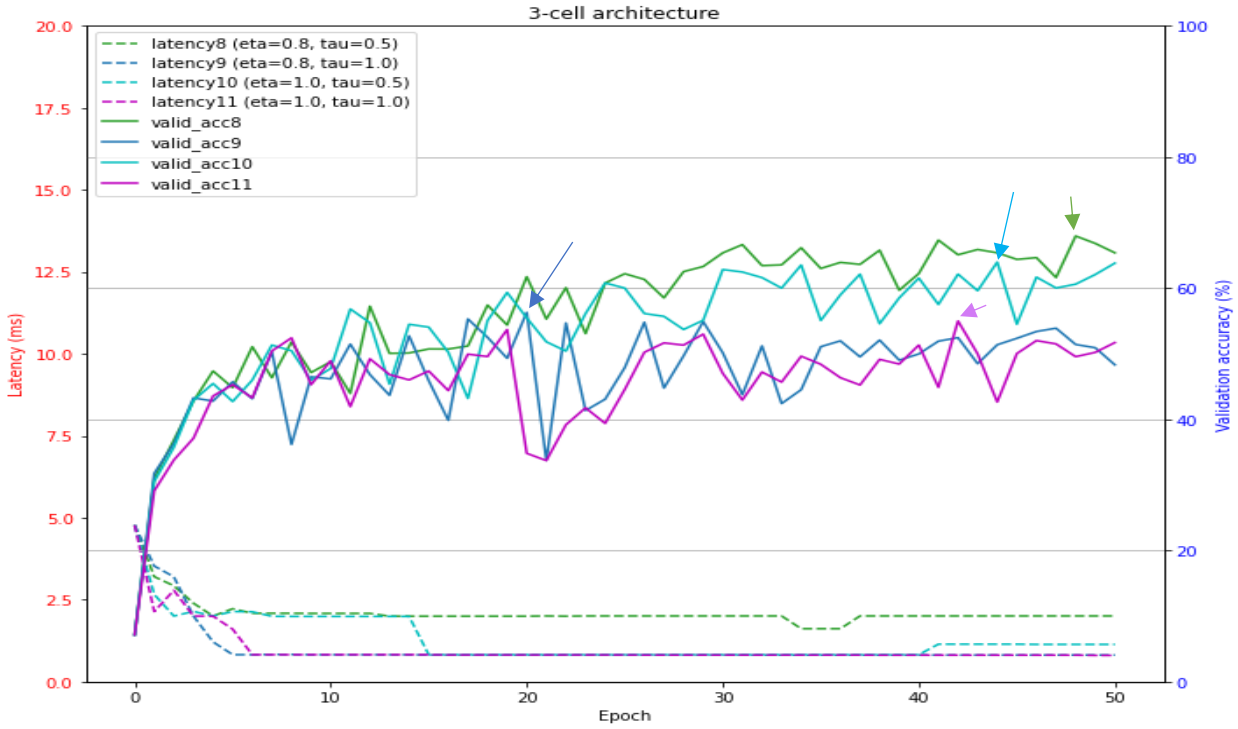

Figure 5.21 Search plots for experiment 8 – experiment 11 (3-cell)

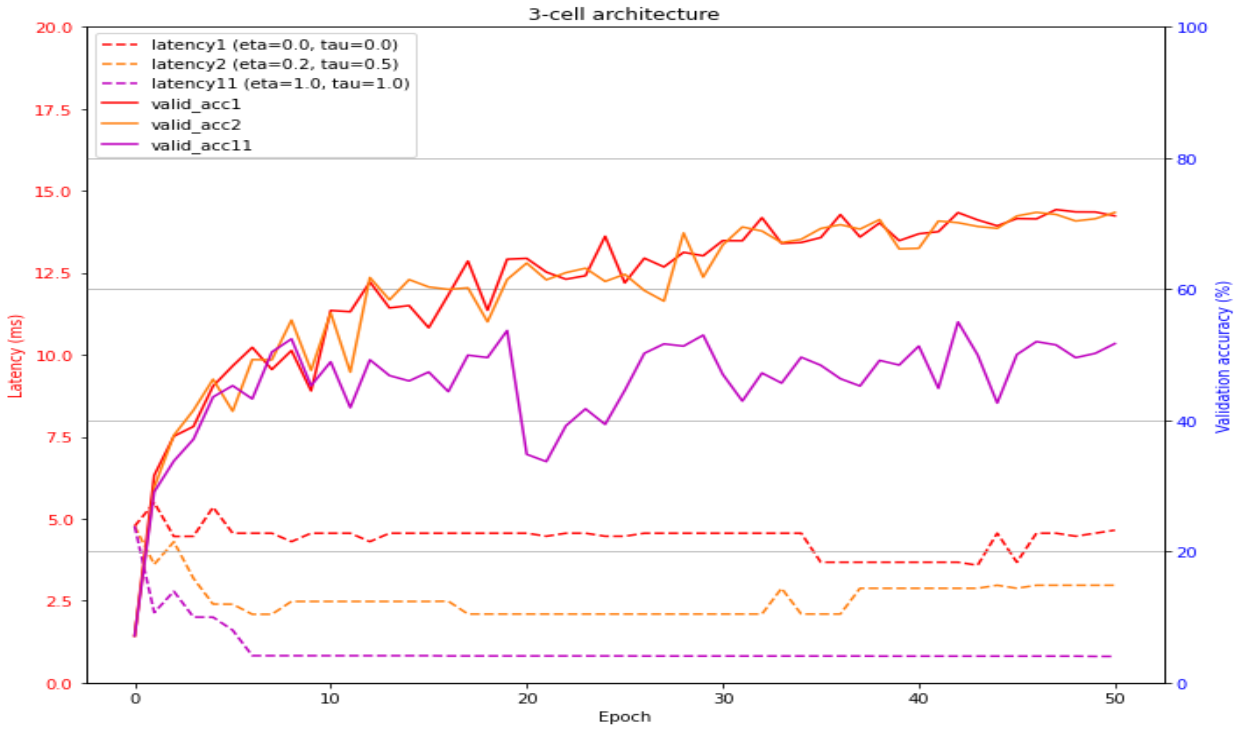

Figure 5.22 Search plot for three experiments 1, 2, and 11 (3-cell)

Table 5.7 indicates, in all experiences if τ is set to 0.5, raising η leads to a reduction in latency and accuracy. However, increasing τ to 1.0 decreases the latency even more significantly, reaching a value of ~0.81 (ms) after experiment 3. This is likely because of exponential effect of the τ parameter in the loss function (Equation (4.7)), as we have seen in other architecture configurations. The network derived from experiment 11 demonstrates the lowest latency (0.80 ms), whereas experiment 1 produces the highest accuracy (82.18%). Despite registering a slightly lower accuracy of about 1.97% than experiment 1, experiment 2 records better latency, roughly 1.54x lower than that of experiment 1. The following graphs, Figure 5.23-Figure 5.28, present the cells identified through the search process for experiments 1, 2 and 11. In comparison to the previous experiments conducted for the 1-cell and 2-cell architecture, we can draw the same conclusion: as η and τ increases, the search process favors operations with lower latency, and the accuracy can potentially decrease.

Table 5.7 Test set evaluation for derived 3-cell architectures

Experiment #	η	τ	Test accuracy (%)	Latency (ms)
1	0.0	0.0	**82.18**	4.56
2	0.2	0.5	80.21	2.97
3	0.2	1.0	70.40	1.61
4	0.4	0.5	78.54	2.22
5	0.4	1.0	64.75	0.81
6	0.6	0.5	71.39	1.99
7	0.6	1.0	62.54	0.82
8	0.8	0.5	71.09	2.00
9	0.8	1.0	63.42	0.81
10	1.0	0.5	62.25	1.14
11	1.0	1.0	60.79	**0.80**

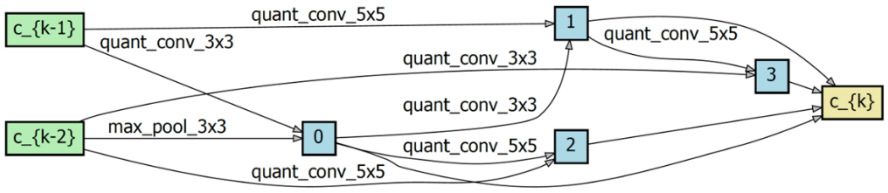

Figure 5.23 Normal cell obtained from experiment 1 (3-cell)

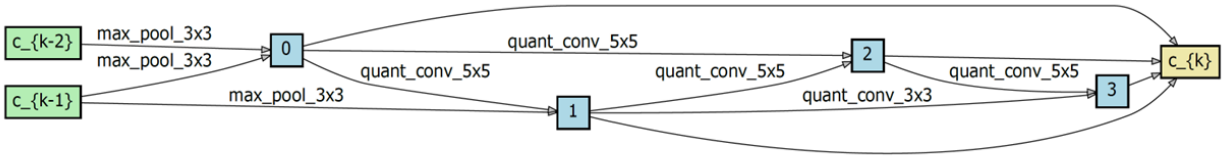

Figure 5.24 Reduction cell obtained from experiment 1 (3-cell)

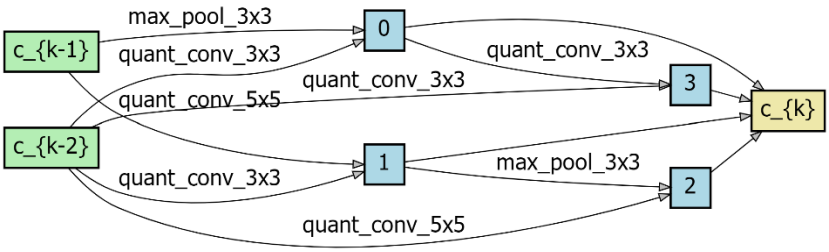

Figure 5.25 Normal cell obtained from experiment 2 (3-cell)

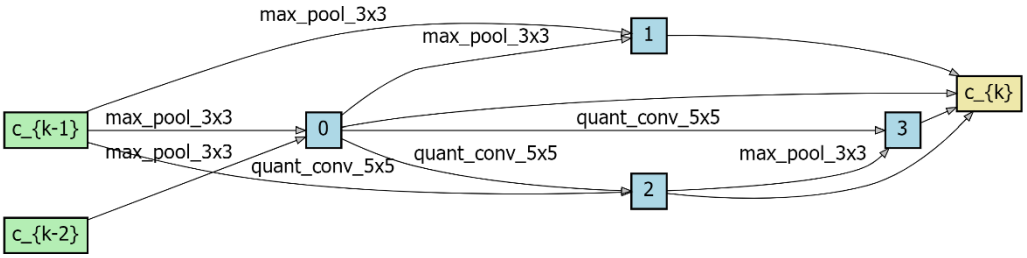

Figure 5.26 Reduction cell obtained from experiment 2 (3-cell)

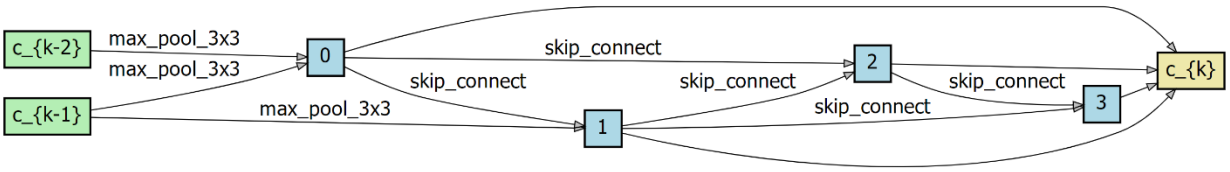

Figure 5.27 Normal cell obtained from experiment 11 (3-cell)

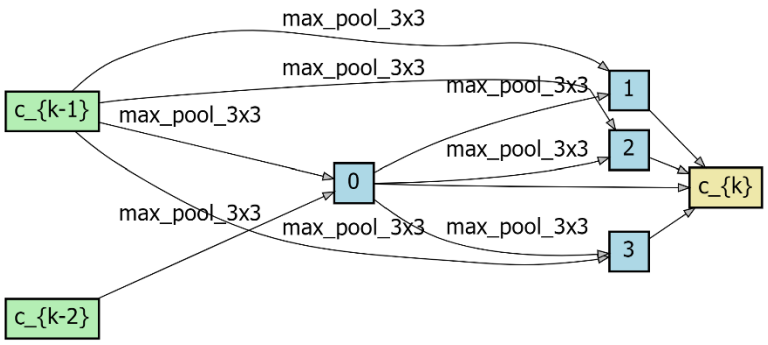

Figure 5.28 Reduction cell obtained from experiment 11 (3-cell)

Figure 5.29 and Figure 5.30 display a comparative chart of experiments 1, 2 and 11 in accuracy and latency across various architectures. We chose these experiments since they represent the best accuracy without considering latency, the best accuracy considering latency and the best latency, respectively. The chart reveals that increasing the number of cells leads to an expected growth in accuracy as well as latency. This is because adding more cells enhances the architecture's learning capacity, which results in a rise in accuracy. However, this also increases latency due to the higher number of operations involved. Overall, these conducted experiments show that it is feasible to reach a balance between accuracy and latency depending on the requirement of a given application.

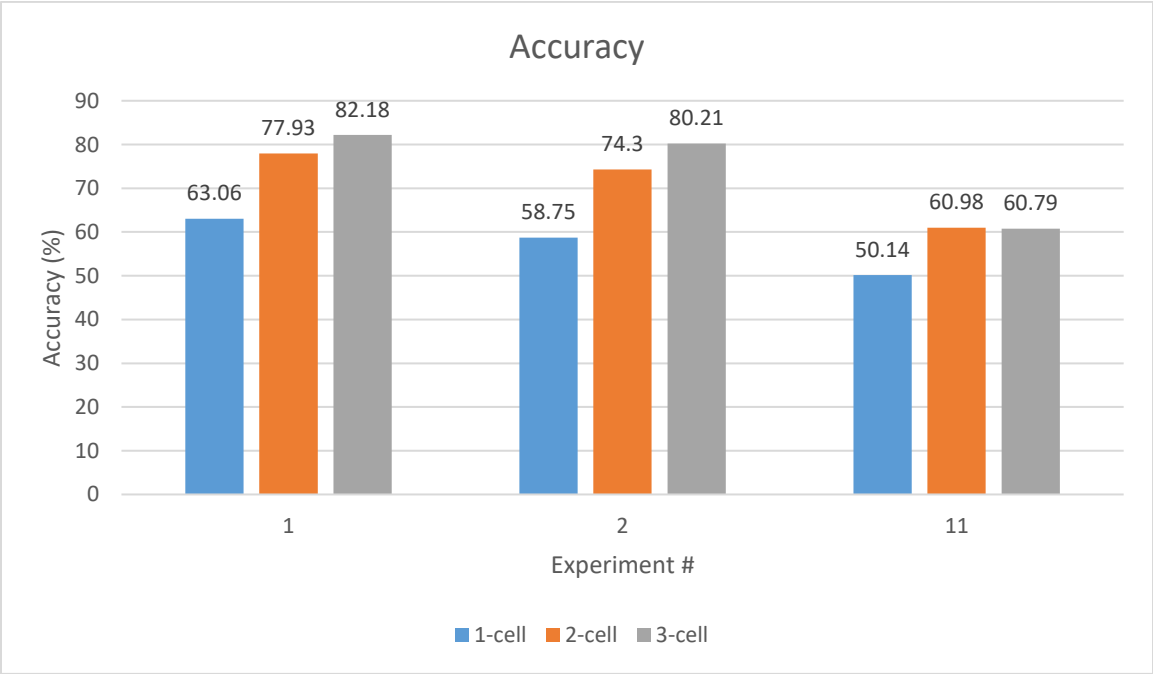

Figure 5.29 Accuracy comparison of experiments 1, 2, and 11

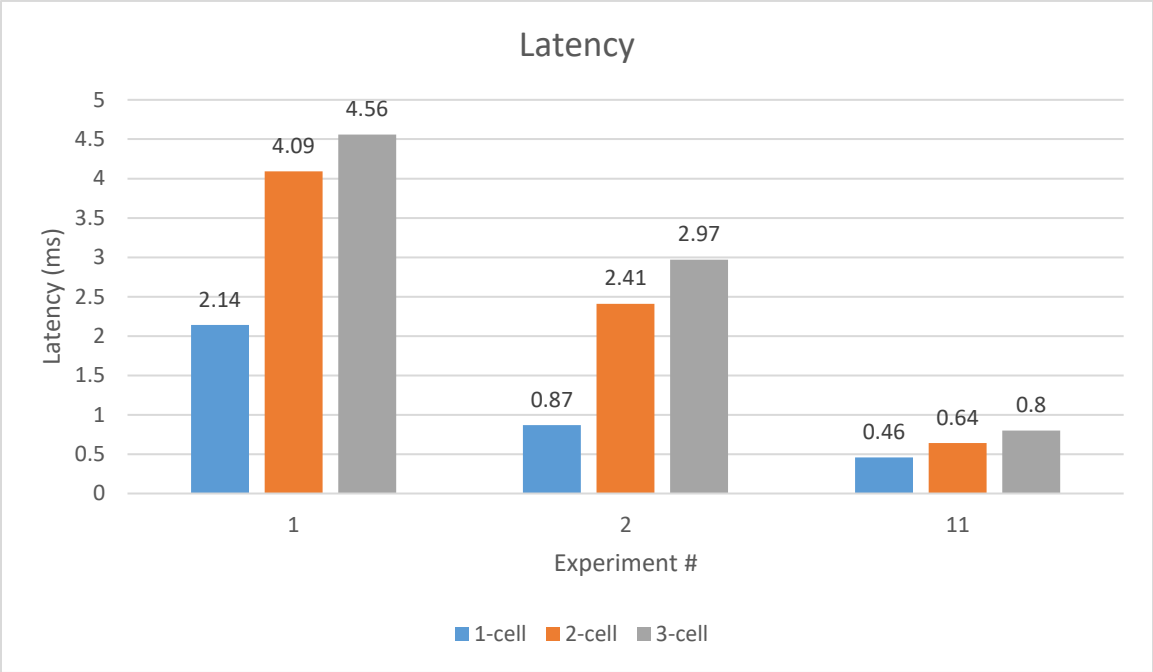

Figure 5.30 Latency comparison of experiments 1, 2, and 11

CHAPTER 6 CONCLUSION

In this thesis, we aimed to discover performant Quantized Neural Networks (QNNs) in terms of accuracy and latency utilizing Neural Architecture Search (NAS). Additionally, we introduced a method to incorporate latency as a hardware criterion into the search process. To achieve this, we investigated the feasibility of implementing a latency model based on our target hardware platform. This chapter provides an overview of the contributions made in this study and discusses its limitations and potential avenues for future works.

6.1 Summary of the work

We used a gradient-based NAS, the DARTS framework, to find Quantized Neural Networks (QNNs) that are both accurate and have low latency, aiming to implement them in FPGAs using the FINN compiler. FINN is a framework to accelerate and deploy QNNs on FPGAs. To accomplish this, we made a series of modifications to the DARTS framework to adapt it to QNNs. In this regard, we defined a search space containing operations such as quantized standard convolutions using Brevitas, a PyTorch-based software framework for developing and deploying quantized neural networks. Brevitas' unified APIs allowed us to substitute the regular Pytorch implementation of operations such as convolution and pooling with their corresponding Brevitas versions, and to export the resulting models to the FINN compiler.

Moreover, to include latency as an optimisation criterion in the search process, it was necessary to integrate it into the loss function. Therefore, we first developed a model to predict the latency of the generated network by estimating the individual operation latencies as implemented via FINN. This involved creating a lookup table of operation latencies and designing a function to calculate the total latency of the generated model at each search step. As a result, measuring the real-time latency of the whole model by implementing it on target hardware via FINN was no longer necessary. Once the latency was obtained, we combined it into the loss function along with the accuracy loss using two control parameters, *eta* and *tau*.

Furthermore, we presented the outcomes of conducted experiments on CIFAR-10 dataset to evaluate the impact of incorporating latency into the loss function. Our results show that the search process was able to adopt the latency impact and explore desired models accordingly. For example,

for all architectures (i.e., 1-cell, 2-cell and 3-cell) we achieved around 1.9x latency reduction with only a few accuracy drop about 3.3% on average compared to the architectures found via ignoring latency through the search process. Hence, we showed it is feasible to employ this methodology to develop efficient QNNs considering both accuracy and latency.

6.2 Limitations

Initially, we selected 1, 2, and 3-bit quantization schemes for both weights and activations. However, it did not yield satisfactory outcomes due to several reasons mentioned in section 3.3. Consequently, we increased the bitwidth to 4, improving performance. Therefore, all experiments conducted in this thesis utilized a 4-bit quantization scheme. In addition, the DARTS framework initially included a range of operations such as separable and dilated convolutions in its search space. Nevertheless, we learned that it was not viable to utilize separable and dilated convolutions since the FINN compiler's automatic standard build flow does not support them and necessitates a complex custom build script that must be designed manually. Hence, we opted to replace separable and dilated convolutions with standard convolutions to follow the FINN standard build flow.

Furthermore, the standard build flow in the FINN framework can handle only standard models, such as VGG16, which consist only of successive layers. It does not support more complex models, such as Resnet50, with residual connections. This is challenging for implementing DARTS cells, which have numerous residual connections. Although the FINN developers have recently added support for Resnet blocks, adapting such networks still requires significant modifications to the build script. To alleviate this limitation, we proposed an offline model to approximate the model latency.

6.3 Future work

There are several potential directions for future research. First, it would be interesting to modify the FINN compiler standard build script to cover DARTS-generated architectures. This would allow more accurate latency measurements. Secondly, considering hardware criteria other than latency, such as area and power, would be a promising way to make the search process more hardware-aware. Finally, considering quantization-aware searching methods can potentially lead to find more efficient neural networks by assigning different bit-width for different operations.

REFERENCES

[1] I. Goodfellow, Y. Bengio, and A. Courville, *Deep Learning*. MIT Press, 2016.

[2] Y. Chen, B. Zheng, Z. Zhang, Q. Wang, C. Shen, and Q. Zhang, "Deep Learning on Mobile and Embedded Devices," *ACM Computing Surveys (CSUR)*, vol. 53, no. 4, Aug. 2020

[3] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, "Quantized Neural Networks: Training Neural Networks with Low Precision Weights and Activations," *Journal of Machine Learning Research*, vol. 18, pp. 1–30, 2018

[4] S. Gupta, A. Agrawal, K. Gopalakrishnan, and P. Narayanan, "Deep Learning with Limited Numerical Precision," *32nd International Conference on Machine Learning, ICML 2015*, vol. 3, pp. 1737–1746, Feb. 2015

[5] P. Gysel, M. Motamedi, and S. Ghiasi, "Hardware-oriented Approximation of Convolutional Neural Networks," Apr. 2016

[6] F. Li, B. Liu, X. Wang, B. Zhang, and J. Yan, "Ternary Weight Networks," May 2016, [Online]. Available: http://arxiv.org/abs/1605.04711

[7] A. Andr´, A. Santos, D. Ferreira, O. Mutlu, and G. Falcao, "RedBit: An End-to-End Flexible Framework for Evaluating the Accuracy of Quantized CNNs", [Online]. Available: https://github.com/IT-Coimbra/RedBit.

[8] A. Zhou, A. Yao, Y. Guo, L. Xu, and Y. Chen, "Incremental Network Quantization: Towards Lossless CNNs with Low-Precision Weights," in *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*, Feb. 2017

[9] A. Bulat and G. Tzimiropoulos, "XNOR-Net++: Improved Binary Neural Networks," *30th British Machine Vision Conference 2019, BMVC 2019*, Sep. 2019

[10] P. Wang, Q. Hu, Y. Zhang, C. Zhang, Y. Liu, and J. Cheng, "Two-Step Quantization for Low-bit Neural Networks," *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pp. 4376–4384, Dec. 2018

[11] S. Zhou, Y. Wu, Z. Ni, X. Zhou, H. Wen, and Y. Zou, "DoReFa-Net: Training Low Bitwidth Convolutional Neural Networks with Low Bitwidth Gradients," 2016, [Online]. Available: https://arxiv.org/abs/1606.06160

[12] H. Chen, B. Zhang, X. Zheng, J. Liu, D. Doermann, and R. Ji, "Binarized Neural Architecture Search," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 34, no. 07, pp. 10526–10533, Apr. 2020

[13] H. Chen, L.A. Zhou, B. Zhang, X. Zheng, J. Liu, R. Ji, D. Doermann, and G. Guo, "Binarized Neural Architecture Search for Efficient Object Recognition," *International Journal of Computer Vision*, vol. 129, no. 2, pp. 501–516, Feb. 2021

[14] M. Loni, H. Mousavi, M. Riazati, M. Daneshtalab, and M. Sjodin, "TAS: Ternarized Neural Architecture Search for Resource-Constrained Edge Devices," In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition, DATE 2022*

[15] H. Zhu, S. Guo, W. Sheng, and L. Xiao, "SBNN: A Searched Binary Neural Network for SAR Ship Classification," *Applied Sciences 2022, Vol. 12, Page 6866*

[16] M. Shen, K. Han, C. Xu, and Y. Wang, "Searching for Accurate Binary Neural Architectures," In *Proceedings of the IEEE/CVF International Conference on Computer Vision Workshop, ICCVW 2019*, Sep. 2019, pp. 2041–2044.

[17] D. Kim, S. Korea, K. P. Singh, and J. Choi, "BNAS v2: Learning Architectures for Binary Networks with Empirical Improvements," Oct. 2021, [Online]. Available: https://arxiv.org/abs/2110.08562

[18] A. Bulat, B. Martinez, and G. Tzimiropoulos, "BATS: Binary Architecture Search," In Computer Vision–ECCV 2020: 16th European Conference, Glasgow, UK, August 23–28, 2020, Proceedings, Part XXIII 2020 Nov 3 (pp. 309-325). Cham: Springer International Publishing.

[19] B. Zhu, Z. Al-Ars, and H. P. Hofstee, "NASB: Neural Architecture Search for Binary Convolutional Neural Networks," In *Proceedings of the International Joint Conference on Neural Networks*, Jul. 2020

[20] B. Zoph, G. Brain, V. Vasudevan, J. Shlens, and Q. V. Le, “Learning transferable architectures for scalable image recognition,” In *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018

[21] H. Liu, K. Simonyan, and Y. Yang, “DARTS: Differentiable Architecture Search,” in *7th International Conference on Learning Representations, ICLR 2019*

[22] H. Benmeziane, K. el Maghraoui, H. Ouarnoughi, S. Niar, M. Wistuba, and N. Wang, “Hardware-Aware Neural Architecture Search: Survey and Taxonomy,” In *Proceedings of the Thirtieth International Joint Conference on Artificial Intelligence*, Aug. 2021

[23] S. Walczak and N. Cerpa, “Artificial Neural Networks,” *Encyclopedia of Physical Science and Technology*, pp. 631–645, 2003

[24] A. Jain, J. Mao, K. M.- Computer, and undefined 1996, “Artificial neural networks: A tutorial,” Computer 29.3, 1996, pp. 31-44

[25] W. Liu, Z. Wang, X. Liu, N. Zeng, Y. Liu, and F. E. Alsaadi, “A survey of deep neural network architectures and their applications,” *Neurocomputing*, vol. 234, pp. 11–26, Apr. 2017

[26] Y. LeCun, Y. Bengio. "Convolutional networks for images, speech, and time series." The handbook of brain theory and neural networks 3361, no. 10 (1995): 1995.

[27] A. Krizhevsky, I. Sutskever, and GE. Hinton. “Imagenet classification with deep convolutional neural networks,” *Communications of the ACM 60*, 2017

[28] K. Simonyan and A. Zisserman, “Very deep convolutional networks for large-scale image recognition,” *3rd International Conference on Learning Representations, ICLR 2015 - Conference Track Proceedings*, 2015.

[29] K. He, X. Zhang, S. Ren, and J. Sun, “Deep Residual Learning for Image Recognition,” in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, Dec. 2015, pp. 770–778.

[30] T. Elsken, J. H. Metzen, and F. Hutter, “Neural Architecture Search: A Survey,” *Journal of Machine Learning Research*, vol. 20, pp. 1–21, 2019

[31] S. Karagiannakos, “Neural Architecture Search (NAS): basic principles and different approaches,” 2021, [Online]. Available: https://theaisummer.com/neural-architecture-search/

[32] C. White, M. Safari, R. Sukthanker, B. Ru, T. Elsken, A. Zela, D. Dey, and F. Hutter, “Neural Architecture Search: Insights from 1000 Papers,” Jan. 2023, [Online]. Available: https://arxiv.org/pdf/2301.08727.pdf

[33] K. T. Chitty-Venkata and A. K. Somani, “Neural Architecture Search Survey: A Hardware Perspective,” *ACM Computing Surveys (CSUR)*, Jul. 2021

[34] A. Buetti-Dinh, V. Galli, S. Bellenberg, O. Ilie, M. Herold, S. Christel, M. Boretska, I.V. Pivkin, P. Wilmes, W. Sand, and M. Vera, “Deep neural networks outperform human expert’s capacity in characterizing bioleaching bacterial biofilm composition,” *Elsevier*, 2019

[35] A. Blanco, R. Pino-Mejías, J. Lara, and S. Rayo, “Credit scoring models for the microfinance industry using neural networks: Evidence from Peru, ” *Expert Systems with applications, Elsevier*, 2013

[36] M. Courbariaux, Y. Bengio, and J. P. David, “Binaryconnect: Training deep neural networks with binary weights during propagations,” in *Advances in Neural Information Processing Systems*, 2015

[37] I. Hubara, M. Courbariaux, D. Soudry, R. El-Yaniv, and Y. Bengio, “Binarized neural networks,” in *Advances in Neural Information Processing Systems*, 2016

[38] H. Qin, R. Gong, X. Liu, X. Bai, J. Song, and N. Sebe, “Binary neural networks: A survey,” *Pattern Recognition*, vol. 105, 2020

[39] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, “XNOR-net: Imagenet classification using binary convolutional neural networks,” *In Computer Vision–ECCV 2016: 14th European Conference, Amsterdam, The Netherlands, October 11–14, 2016, Proceedings, Part IV (pp. 525-542). Cham: Springer International Publishing*

[40] Q. Hu, P. Wang, and J. Cheng, “From hashing to CNNs: Training binary weight networks via hashing,” in *32nd AAAI Conference on Artificial Intelligence, AAAI 2018*, 2018, pp. 3247–3254.

[41] Z. Li, B. Ni, W. Zhang, X. Yang, and W. Gao, “Performance Guaranteed Network Acceleration via High-Order Residual Quantization,” in *2017 IEEE International Conference on Computer Vision (ICCV)*, Oct. 2017, pp. 2603–2611

[42] X. Lin, C. Zhao, and W. Pan, “Towards accurate binary convolutional neural network,” in *Advances in Neural Information Processing Systems*, Nov. 2017

[43] A. Bulat and G. Tzimiropoulos, “XNOR-Net++: Improved binary neural networks,” *30th British Machine Vision Conference*, 2019

[44] L. Hou, Q. Yao, and J. T. Kwok, “Loss-aware Binarization of Deep Networks,” in *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*, Nov. 2016

[45] R. Ding, T. W. Chin, Z. Liu, and Di. Marculescu, “Regularizing Activation Distribution for Training Binarized Deep Networks,” in *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, Apr. 2019

[46] Y. Bengio, N. Léonard, and A. Courville, “Estimating or Propagating Gradients Through Stochastic Neurons for Conditional Computation,” Aug. 2013, Accessed: Oct. 31, 2021. [Online]. Available: http://arxiv.org/abs/1308.3432

[47] Z. Liu, B. Wu, W. Luo, X. Yang, … W. L.-E. C. on, and undefined 2018, “Bi-Real Net: Enhancing the Performance of 1-Bit CNNs with Improved Representational Capability and Advanced Training Algorithm,” *Springer*, Accessed: Oct. 28, 2021. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-030-01267-0_44

[48] C. Zhu, H. Mao, S. Han, and W. J. Dally, “Trained ternary quantization,” *5th International Conference on Learning Representations, ICLR 2017 - Conference Track Proceedings*, 2017.

[49] A. Mishra and D. Marr, “Apprentice: Using knowledge distillation techniques to improve low-precision network accuracy,” in *6th International Conference on Learning Representations, ICLR 2018 - Conference Track Proceedings*, Nov. 2018

[50] X. Chen, G. Liu, J. Shi, J. Xu, and B. Xu, “Distilled Binary Neural Network for Monaural Speech Separation,” in *Proceedings of the International Joint Conference on Neural Networks*, Oct. 2018

[51] B. Martinez, J. Yang, A. Bulat, and G. Tzimiropoulos, "Training binary neural networks with real-to-binary convolutions," 2020, [Online]. Available: https://github.com/brais-martinez/real2binary.

[52] M. Abadi *et al.*, "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems," Mar. 2016, Accessed: Jan. 23, 2022. [Online]. Available: https://arxiv.org/abs/1603.04467v2

[53] A. Paszke *et al.*, "PyTorch: An Imperative Style, High-Performance Deep Learning Library," *Advances in neural information processing systems*, vol. 32, Dec. 2019

[54] K. Ovtcharov, O. Ruwase, J. Kim, J. Fowers, K. Strauss, and E. S. Chung, "Accelerating Deep Convolutional Neural Networks Using Specialized Hardware," *Microsoft Research Whitepaper*, pp. 3–6, 2015, Accessed: Jan. 23, 2022. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.1050.9891&rep=rep1&type=pdf

[55] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing FPGA-based accelerator design for deep convolutional neural networks," In Proceedings of the 2015 ACM/SIGDA international symposium on field-programmable gate arrays, pp. 161-170, 2015

[56] S. I. Venieris and C. S. Bouganis, "FpgaConvNet: A Framework for Mapping Convolutional Neural Networks on FPGAs," *Proceedings - 24th IEEE International Symposium on Field-Programmable Custom Computing Machines, FCCM 2016*, pp. 40–47, Aug. 2016

[57] R. Zhao, W. Song, W. Zhang, T. Xing, J.H. Lin, M. Srivastava, R. Gupta, and Z. Zhang, "Accelerating Binarized Convolutional Neural Networks with Software-Programmable FPGAs," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, pp. 15–24, Feb. 2017

[58] R. Andri, L. Cavigelli, D. Rossi, and L. Benini, "YodaNN: An ultra-low power convolutional neural network accelerator based on binary weights," *in IEEE Computer Society Annual Symposium on VLSI (ISVLSI)*, 2016

[59] Y. Umuroglu *et al.*, "FINN," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, Feb. 2017, pp. 65–74

[60] N. J. Fraser *et al.*, “Scaling Binarized Neural Networks on Reconfigurable Logic,” *ACM International Conference Proceeding Series*, pp. 25–30, Jan. 2017

[61] S. Liang, S. Yin, L. Liu, W. Luk, and S. Wei, “FP-BNN: Binarized neural network on FPGA,” *Neurocomputing*, vol. 275, pp. 1072–1086, Jan. 2018

[62] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, and Z. Zhang, “MXNet: A Flexible and Efficient Machine Learning Library for Heterogeneous Distributed Systems,” Dec. 2015, Accessed: Jan. 30, 2022. [Online]. Available: https://arxiv.org/abs/1512.01274v1

[63] H. Yang, M. Fritzsche, C. Bartz, and C. Meinel, “BMXNet: An Open-Source Binary Neural Network Implementation Based on MXNet,” *MM 2017 - Proceedings of the 2017 ACM Multimedia Conference*, pp. 1209–1212, May 2017

[64] J. Zhang, Y. Pan, T. Yao, H. Zhao, and T. Mei, “daBNN: A Super Fast Inference Framework for Binary Neural Networks on ARM devices,” in *MM 2019 - Proceedings of the 27th ACM International Conference on Multimedia*, Aug. 2019, pp. 2272–2275

[65] H. Kim, J. Kim, J. Choi, J. Lee, and Y. H. Song, “Binarized Encoder-Decoder Network and Binarized Deconvolution Engine for Semantic Segmentation,” *IEEE Access*, vol. 9, pp. 8006–8027, 2021

[66] G. Brostow, J. Shotton, J. Fauqueur, and R. Cipolla, “Segmentation and recognition using structure from motion point clouds,” *Springer*, In Computer Vision–ECCV 2008: 10th European Conference on Computer Vision, Marseille, France, October 12-18, 2008, Proceedings, Part I 10, pp. 44-57. Springer Berlin Heidelberg, 2008

[67] E. Real, A. Aggarwal, Y. Huang, and QV. Le, “Regularized evolution for image classifier architecture search,” *In Proceedings of the aaai conference on artificial intelligence*, vol. 33, no. 01, pp. 4780-4789. 2019.

[68] X. Chen, L. Xie, J. Wu, and Q. Tian, “Progressive Differentiable Architecture Search: Bridging the Depth Gap between Search and Evaluation,” *Proceedings of the IEEE International Conference on Computer Vision*, vol. 2019-October, pp. 1294–1303, Apr. 2019

[69] H. Cai, C. Gan, T. Wang, Z. Zhang, and S. Han, "Once-for-All: Train One Network and Specialize it for Efficient Deployment," Aug. 2020

[70] H. Cai, L. Zhu, and S. Han, "ProxylessNAS: Direct Neural Architecture Search on Target Task and Hardware," in *7th International Conference on Learning Representations, ICLR 2019*

[71] W. Jiang, X. Zhang, EH. Sha, L. Yang, Q. Zhuge, Y. Shi, and J. Hu, "Accuracy vs. Efficiency: Achieving Both through FPGA-Implementation Aware Neural Architecture Search," *In Proceedings of the 56th Annual Design Automation Conference 2019,* pp. 1-6, 2019

[72] B. Wu, X. Dai, P. Zhang, Y. Wang, F. Sun, Y. Wu, Y. Tian, P. Vajda, Y. Jia, and K. Keutzer, "FBNet: Hardware-Aware Efficient ConvNet Design via Differentiable Neural Architecture Search", In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 10734-10742. 2019

[73] M. Blott, T.B. Preußer, N.J. Fraser, G. Gambardella, K. O'brien, Y. Umuroglu, M. Leeser, K. Vissers, "FinN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 11, no. 3, Dec. 2018

[74] "FINN documents" https://finn-dev.readthedocs.io/en/latest/source_code/finn.builder.html

[75] "Finn/tfc_end2end_example.ipynb at main · Xilinx/finn." https://github.com/Xilinx/finn/blob/main/notebooks/end2end_example/bnn-pynq/tfc_end2end_example.ipynb (accessed Feb. 16, 2023).

[76] Volcacius *et al.*, "Xilinx/brevitas: Release version 0.8.0," Jan. 2023, doi: 10.5281/ZENODO.7520091.

[77] "Brevitas documentation." [Online]. Available : https://xilinx.github.io/brevitas/index.html

[78] "Quark0/darts: Differentiable architecture search for convolutional and recurrent networks." https://github.com/quark0/darts (accessed Feb. 16, 2023).

[79] "Finn-examples/build/resnet50 at main · Xilinx/finn-examples." [Online]. Available: https://github.com/Xilinx/finn-examples/tree/main/build/resnet50

[80] S. V. Srinivas, H. Nair, V. Vidyasagar, “Hardware aware neural network architectures (using fbnet)”, 2019, [Online]. Available: https://arxiv.org/pdf/1906.07214.pdf

[81] A. Krizhevsky, V. Nair, and G. Hinton, “CIFAR-10 (Canadian Institute for Advanced Research),” 2014, [Online]. http://www.cs.toronto.edu/~kriz/cifar.html

[82] V. Dumoulin, and F. Visin, “A guide to convolution arithmetic for deep learning,” Mar. 2016, [Online]. Available: https://arxiv.org/pdf/1603.07285.pdf

APPENDIX A EXPERIMENTAL DETAILS

For all experiments in this thesis, we optimized the weights *w* using momentum SGD with an initial learning rate of 0.025, a momentum of 0.9, and a weight decay of 0.0003. The architecture parameters (α's) in normal and reduction cells were randomly initialized using a normal distribution (mean 0 and variance 1). We used the Adam optimizer with an initial learning rate of 0.0003, the momentum of (0.5, 0.999), and weight decay of 0.001 to optimize α, and the search took 5, 9 and 20 hours for each 1-cell, 2-cell and 3-cell experiment, respectively, on a single GPU.

APPENDIX B DIFFERENT TYPES OF CONVOLUTIONS

Standard and depthwise separable convolutions

A standard convolutional layer in a neural network applies a single filter to the input data, computing a dot product between the filter and a small portion of the input and then sliding the filter over the entire input data to generate a feature map. The computational cost of this layer depends on the filter size, input size, and the number of output channels. On the other hand, a depthwise separable convolution layer splits the standard convolution into two separate steps: a depthwise convolution and a pointwise convolution. The depthwise convolution applies a filter to each input channel, generating output feature maps with the same number of channels as the input. The pointwise convolution then combines the output feature maps from the depthwise convolution using a 1x1 convolution, producing output feature maps with a new number of channels. Depthwise separable convolution lowers the computational cost by reducing the number of parameters and computations required to process the input. Figure B.1 and Figure B.2 provide examples of standard and depthwise separable convolutions, respectively. As shown, the number of learnable parameters in depthwise separable convolution is significantly less than its standard counterpart, with $3 \times (3 \times 3 \times 1) + 128 \times (1 \times 1 \times 3) = 411$ and $128 \times (3 \times 3 \times 3) = 3456$ parameters, respectively.

Dilated convolution

Dilated convolution uses a filter with spaces between values to process larger areas of an image without increasing the number of parameters. This is done by adding zeros between the filter values, allowing the network to incorporate more distant spatial information. Dilated convolution is often used in tasks requiring a large context, such as image segmentation [82].

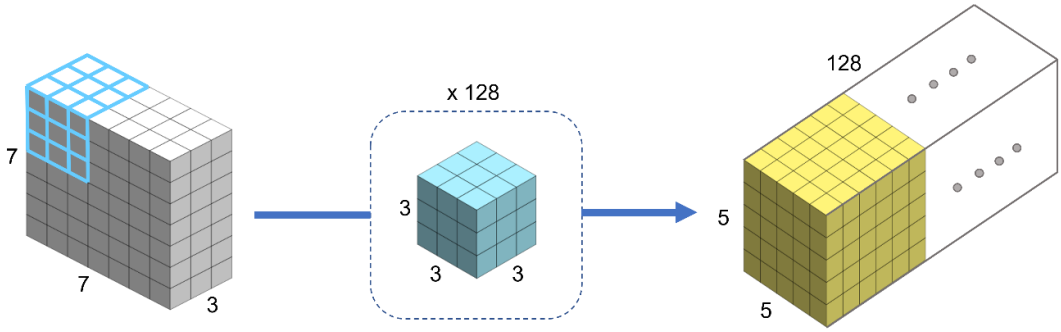

Figure B.1 Standard convolution

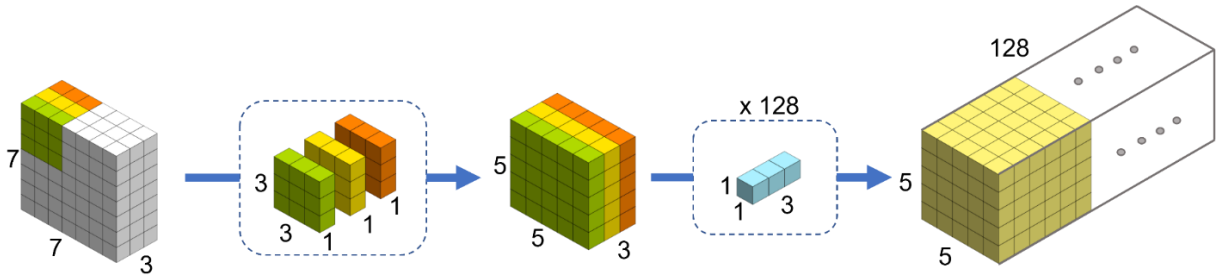

Figure B.2 Depthwise separable convolution