Hardware implementation of series and parallel adapter module with online training capable of multi-domain learning

NIYANA YOHANNES

Supervisor: Dr David Boland

A thesis submitted in fulfilment of
the requirements for the degree of
Bachelor of Electrical Engineering (Honours)

School of Electrical and Information Engineering
Faculty of Engineering
The University of Sydney
Australia

3 November 2023

0.1 Statement of Achievement

Throughout the course of this thesis, I have achieved:

- A hardware RTL designed adapter written in SystemVerilog in both series and parallel configuration
  - Capable of multi-domain adapting a Convolutional Neural Network with 1 convolutional layer as described in [**1**] for datasets, orignal MNIST and teduced MNIST dataset (which has the number 9 removed)
  - Able to perform inference and training online through the addition of backpropogation in hardware
- A custom built simulator written in C (CSIM)
  - Capable of testing inference and training of the unadapted CNN network on the reduced MNIST dataset
  - Capable of fine-tuning the above network using adapter modules on the original MNIST dataset
  - Capable of producing memory files that are used as inputs to test the adapter modules in hardware during simulation
- A pyTorch equivalent of the above network
  - Capable of providing a benchmark for the CSIM and hardware adapter modules

I recognise the following contributions from Nicholas Manning on the following:

- Significant contribution to the PyTorch code used as benchmark for CSIM and hardware
- Significant contribution to the SystemVerilog code of testbench, particularly the load and save logic (of input data including MNIST and Reduced MNIST dataset to testbench)
- Significant contribution to the CSIM code, particularly producing memory files used as inputs to testbench.
- Debugging final architecture of the adapters in SystemVerilog

Abstract

Historically, machine learning algorithms were primarily designed to address single, isolated tasks. Research in this field concentrated on enhancing these algorithms to optimize accuracy for specific tasks. However, as of late, there has been a shift in focus towards the development of algorithms that can effectively handle a ,ultiple problems with a single model. Transfer learning is an approach that aims to provide a solution to this recent challenge. Considering neural networks, fine-tuning stands out as a prominent and widely-used transfer learning technique. However, despite being very effective, fine-tuning require extensive retraining or modifying of the original networks' parameters for each new task. A new method of transfer learning, called multiple-domain learning aims to develop networks that can represent compactly different domain. Adapter modules present a way to perform multiple-domain learning. Adapter modules have up to date only been implemented as software solutions. This paper introduces a hardware based adapter module with online training capable of performing multi-domain learning implemented on the Zynq UltraScale+ RFSoC ZCU111 Evaluation board. Specifically, a series and parallel configuration are designed to adapt a convolutional neural network called the Super Skinny Convolutional Neural Network (SSCNN) composed of one convolutional layer, 1 max-pooling layer and 1 fully connected layer. The datasets used for evaluation of the adapter network is the original MNIST dataset, and a new reduced MNIST dataset which has the number 9 removed.

The adapter network was first designed in software and simulated for correctness. Pre-training the unadapted network on Reduced MNIST resulted in accuracy of 98.35%. Fine-tuning with the network with the series and parallel adapter on the Original MNIST resulted in an accuracy of 97.9% and 98.2%. In comparison, full vanilla fine-tuning on the original MNIST dataset resulted in an accuracy of 98.27%. Following this, the adapter modules were implemented in hardware. Floating point format was used and four different bit widths were implemented. These were 32-bit single precision , 16-bit half precision and a custom 12-bit and 10-bit precision. Floating point IP cores were used to perform all the arithmetic. These designs were synthesised and implemented on the target board at a clock cycle of 37 Mhz. The 32-bit design did not fit on the board while the other three designs fit but utilized a large number of logic elements. However, decreasing the bit-width resulted in a decrease in logic elements. The series configuration also turned out to be more hardware efficient.

This project has yielded promising results regarding hardware-based adapter modules with online training capabilities. It highlights the feasibility of utilizing adapter modules in hardware to perform multi-domain learning and effectively adapt networks for different datasets. It serves as a proof-of-concept for adapter modules in hardware, laying the foundation for future research in this area.

Acknowledgements

I would like to thank my supervisor, Dr. David Boland, for his guidance and support during the course of this thesis.

I am also thankful to my fellow thesis students who provided help and advice throughout the semseter.

Contents

List of Figures

CHAPTER 1

Introduction

1.1 Motivations and Aims

Machine learning algorithms leverage data to tackle complex problems through discovering patterns. Although discovered decades ago, the combination of an abundance of data, cheap and abundant computation, and improvements in technology, means that it is now a viable solution for plenty of problems. Originally used to play a games and plot routes [**2**], it is now used extensively in a wide variety of applications. Examples include in search engines [**3**], self-driving cars [**4**], healthcare and medicine [**5**], finance [**6**], cyber-security [**7**] and many more.

The effectiveness of machine learning algorithms in tackling complex tasks can be largely attributed to the sophisticated mathematical models they use and the parameters that makes up these models. It is this that allows them to discern patterns and relationships within data, making them capable of solving a wide array of challenging problems. However, it's worth noting that the very characteristics that provide these algorithms with their accuracy and robustness also at times pose a significant challenge in terms of computational cost.

In many real-world machine learning applications, factors such as computational power and execution time take have great importance. The growing demand for efficient machine learning solutions arise from the recognition that not all domains can afford the computational cost associated with complex models and a multitude of parameters that may come with it. Thus, balancing accuracy with computational resources is often an area of great concern

In tackling this challenge, researcher have landed at solutions like software-hardware co-design [**8**]. This is a logical step, given that the underlying hardware is the foundation machine learning algorithms are built off. Hardware accelerators have emerged as viable solutions in this area. These include a range of technologies such as Graphics Processing Units (GPUs), Application-Specific Integrated Circuits (ASICs), and Vision Processing Units (VPUs) [**9**]. They have significantly elevated the speed, performance, and energy efficiency of machine learning tasks. Yet, they can still be improved in terms of flexibility.

Field Programmable Gate-Arrays (FPGAs) are another solution. FPGAs are integrated circuits that are user-reconfigurable, providing a great level of flexibility. This adaptability is paired with power and energy efficiency, and computational capabilities, making FPGAs a compelling solution to the complexities of machine learning. Existing research has already showcased promising applications of FPGAs in machine learning contexts [**10–12**], and

Chapter 2.5.1 delves into hardware implementations of machine learning examples on FPGAs in-depth.

Based on these insights, this thesis is poised to:

- Explore the hardware implementation of a specific machine learning use case, as detailed in the subsequent paragraph in order to shed light on the potential and practicality of FPGAs in the realm of machine learning.

Historically, machine learning algorithms were primarily designed to address single, isolated tasks. Research in this field concentrated on enhancing these algorithms to optimize accuracy for specific tasks. However, as of late, there has been a shift in focus towards the development of algorithms that can solve multiple problems with 1 model. This shift is inspired by the observation that human learners possess a natural ability to transfer knowledge between tasks. Traditional machine learning algorithms, on the other hand, have been confined to addressing isolated tasks in isolation.

Transfer learning represents an approach aimed at advancing traditional machine learning paradigms [**13**]. It seeks to improve learning by using knowledge acquired from one or more source tasks and leveraging it to enhance learning in a closely related target task. It is defined as the process of improving performance in a new task by transferring knowledge obtained from a related task that has already been learned.

Within the realm of neural networks, fine-tuning stands out as a prominent and widely-used transfer learning technique. This process involves taking a model that has been pre-trained on a specific dataset and then retraining it on a new dataset, adjusting of all the network's parameters. The body of research dedicated to fine-tuning has provided promising and effective results, demonstrating its efficiency and practical relevance [**14**].

However, despite being very effective, fine-tuning require extensive retraining or modifying of the original networks' parameters for each new task. A new method of transfer learning, called multiple-domain learning introduced by Rebuffi et al. in [**15**], aims to construct networks that can represent different domains. A domain in this context refers to the type of data that a model is being trained and tested on. For example, different visual domains include animal images, vehicle images, etc... According to Rebuffi et al., multiple-domain learning looks at how deep learning techniques can be used to learn universal representations, i.e. feature extractors that can work well in several different domains [**15**].

Adapter modules present a way to perform multiple-domain learning. Adapters are modules that are used to provide a lightweight and efficient way to adapt pre-trained networks to new tasks or domains without requiring extensive retraining or modifying of the original models' parameters. They act as small learnable modules that are inserted between the layers of a pre-trained network, allowing it to learn task-specific information while preserving the majority of the original model's knowledge. Chapter 2.6 explores in depth different adapter module architectures and their use cases.

Adapter modules represent a particularly promising area of study and therefore serve as a central focus of this thesis. Building upon the foundation laid in the previous section, this thesis is also dedicated to:

- The exploration of various adapter architectures and their implementation on an FPGA. An analysis on the hardware resource usage of these adapters is also conducted and compared with existing networks in hardware. An assessment can then be made on the feasibility of these hardware adapter architectures with existing hardware networks.

1.2 Contribution

While the field of Machine Learning algorithm implementations on FPGAs continues to grow, significant research and practical applications in this area have already been conducted. In the context of Neural Networks, the prevailing hardware implementations often involve feed-forward neural networks where the network's training occurs in an offline computing environment. In this setup, the network's parameters have been previously determined for a specific task and are then configured to fit the hardware-based neural network architecture. This process is commonly referred to as "offline training," where the training phase is executed in software, and the hardware is employed for inference. Conversely, training conducted entirely in hardware is relatively rare and infrequently pursued.

In light of the various adapter architectures introduced in Chapter 2.6, it's important to note that each of these approaches relies entirely on software methods, encompassing both inference and training, which are performed online. Considering the advantages of FPGAs that have been elucidated previously, there is a compelling interest in investigating adapter architectures in a hardware context where both inference and training are executed in hardware.

As such, the primary contribution of this thesis is to:

- Explore an adapter architecture implemented entirely in hardware, encompassing both the inference and training stages.

The adapter architectures explored in Chapter 2.6 have primarily been designed for extremely large networks, such as the Wide Residual Networks (e.g., ResNeT-26 [**16**]), BERT Transformer [**17**], and large-scale Convolutional Neural Networks (ConvNets) [**18**]. These architectures were applied to exceptionally large datasets, including the Visual Decathlon benchmark [**15**], which includes ImageNet [**19**] and CIFAR100 [**20**], as well as the GLUE benchmark [**21**] and VTAB-1k [**22**]. However, the application and performance of adapter architectures for smaller networks and datasets remain largely uncharted territory.

Hence, the second significant contribution of this thesis is to:

- Delve into adapter architectures tailored for smaller networks and datasets.

The following github repository contains all the code developed throughout the course of this this. https://github.com/N-Yohannes/Adapter-Module-in-Hardware-CNN

CHAPTER 2

Literature review

2.1 Introduction

This chapter is a comprehensive literature review, delving into the essential concepts and terminology that form the foundation of this thesis paper. It commences by introducing digital arithmetic and number representations in hardware. It is the followed by a detailed explanation on machine learning, deep learning and the relevant deep learning algorithms related to this thesis. It then provides a detailed examination of Field Programmable Gate Arrays (FPGAs) and their applications within the realm of machine learning. Finally, the bulk of this chapter will be dedicated to a thorough exploration of the current body of literature on adapter architectures, which stands as the central focus of this thesis.

2.2 Digital Arithmetic

Computer arithmetic encompasses the investigation of various aspects, including number representations, algorithms for numerical operations, and the hardware circuits and designed for their execution. In this section, we introduce fixed-point and floating-point arithmetic, two fundamental arithmetic types that collectively encompass the vast majority of number systems employed in computer systems. At the core of computer data representation is the binary digit, commonly referred to as a "bit," which serves as the smallest unit of information. A bit can take one of two values, typically denoted as one or zero. By grouping these bits together, we can create numerical representations based on the specific encoding scheme in use.

2.2.1 Fixed-Point Representation

Fixed-point representation is a digital numerical format employed to represent real numbers, including those with fractional parts, using a predetermined number of digits after the radix point. In the decimal system, the radix point corresponds to the decimal point, whereas in binary systems, it's referred to as the binary point. Within a fixed-point representation, a portion of the encoding is allocated for the fractional component of real numbers, often termed "fractional bits." The part before the radix point is typically designated as the integer component.

In the base-10 decimal system, the fractional and integer segments of a fixed-point number are visibly separated by a period '.'. However, in hardware, no physical separation exists. Fixed-point fractional numbers are usually depicted as integer values that have been scaled by an appropriate factor, often referred to as the exponent. For instance, in binary scaling, when storing 'n' fractional digits, the value is an integer multiple of 2^n. This scaling factor remains consistent for all values within the fixed-point representation.

Signed fixed-point numbers are commonly expressed using two's complement notation. In this scheme, the highest place value bit serves as the sign bit, indicating whether the binary number is positive or negative. To derive the two's complement representation of a negative number, the positive of that number is calculated, and all the bits inverted and a 1 added. Figure 2.1 provides an illustration of fixed-point representation using two's complement notation.

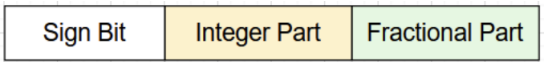


FIGURE 2.1. 2s-complement Fixed-Point Representation

In fixed-point representation, three fundamental concepts are width, range, and precision. The width **n** represents the total number of bits allocated to a fixed-point number. The range **R**, signifies the difference between the minimum and maximum values that a fixed-point number can encompass. The precision **p** defines the total number of bits allocated for the fractional part of the number.

In a two's complement fixed-point representation, the range is represented by the interval $[-2^{n-p-1}, 2^{n-p-1} - 1]$. This specifies the minimum and maximum values that can be accurately represented within the specified width and precision.

Numerical errors can emerge when there is insufficient range or precision to precisely represent a number. These errors may manifest as rounding errors attributable to limited precision or as overflow and underflow errors. Floating-point representation offers a solution to these challenges, as it allows for a more flexible and adaptable approach to representing and manipulating real numbers, reducing the limitations of fixed-point representation.

2.2.2 Floating-Point Representation

Floating point representation has three main components; the sign **S**, exponent **e** and mantissa or significand **m**.

- **Sign:** The sign bit indicates whether the number is positive or negative. Floating-point numbers are commonly represented in a sign-magnitude format, where 1 bit is allocated for the sign.
- **Significand/Mantissa:** TThe mantissa serves as an 'm' bit unsigned fixed-point number, with all bits being fractional. It is often referred to as the fraction. The position of the radix point (analogous to the decimal point in base-10) is assumed to be within the significand, typically just after or just before the most significant digit. In a normalized mantissa, the binary point is positioned immediately to the left of

the most significant non-zero digit, ensuring that the most significant digit is always '1.' However, unnormalized mantissas are also possible.

- **Exponent:** The exponent is used to shift the mantissa, enabling the dynamic range that characterizes floating-point representation. In hardware, exponents are typically encoded as unsigned integers with an added bias, allowing for the representation of a broad range of values.

The range for floating point numbers, represented as $[-min, -max]$ and $[min, max]$ can be calculated using the following formulas:

$$\text{max} = \text{largest mantissa x } 2^{\text{largest exponent}}$$
$$\text{min} = \text{smallest mantissa x } 2^{\text{smallest exponent}}$$

The IEEE Standard for Floating-Point Arithmetic, known as IEEE 754, is a technical standard for floating-point arithmetic. It was established in 1985 by the Institute of Electrical and Electronics Engineers (IEEE) [**23**]. This standard defines various aspects of floating-point arithmetic, including arithmetic formats, interchange formats, rules for rounding, operations, and exception handling. Figure 2.2 below provides a summary of IEEE-754 floating-point formats such as half, single and double precision, which are widely used in computer systems to ensure consistent and standardized representation and manipulation of real numbers.

Scheme	Format (e, m)		Bias[3]	Normal	Denormal	Range[1]	Precision[2]
	exponent	mantissa	(β)	(max^+)	(min^+)	(dB)	(ϵ)
FP16	5	10	-15	6.5e4	6.0e-8	241	2^{-11}
FP32	8	23	-127	3.4e38	1.4e-45	1668	2^{-24}
FP64	11	52	-1023	1.8e308	4.9e-324	12631	2^{-53}

[1] dynamic range in decibels $20 \log_{10}(max^+/min^+)$
[2] relative round-off error, i.e. $2^{-m} \times 2^{-1}$
[3] exponent bias $\beta = -(2^{e-1} - 1)$

FIGURE 2.2. Summary of IEEE-754 floating-point number formats

When making hardware design choices, it's important to consider the trade-offs between fixed-point and floating-point representations.

Fixed-point representation, particularly integers, can be relatively straightforward to implement in hardware. They often require fewer hardware resources and consume less power compared to floating-point representations. However, there is a trade-off in terms of accuracy, as they may not be as well-suited for applications requiring precise fractional values or a wide dynamic range.

On the other hand, floating-point representations are more complex to implement in hardware. They typically demand more hardware resources and consume additional power. However, they are much better in scenarios where high precision and a broad dynamic range are essential.

2.3 Deep Learning

Artificial Intelligence or AI refers to a computer or machines ability to mimic human intelligence and problem-solving skills. Although being a concept that has been around for centuries, AI grew in popularity around the time 1950-1956. It was at this time, John McCarthy coined the phrase 'Artificial Intelligence' [**24**].

Machine learning on the other hand is a subset of AI that leverages algorithms to automatically learn insights and patterns from data, using these findings to enhance decision-making. Going a step further, deep learning, an advancement of machine learning, employs large neural networks, networks that function like a human brain to logically analyze data, to learn complex patterns and make predictions independent of human input.

The subsequent sections delve into the fundamental component of neural networks, the perceptron, and then explore diverse types of neural networks.

2.3.1 The Perceptron

As mentioned, neural networks emulate the functioning of the human brain, which is composed of millions of interconnect neurons that are constantly transmitting data to each other. Similarly, artificial neurons in neural networks transmit information to one another. The perceptron, developed in the 1950s and 60s by scientist Frank Rosenblatt, serves as an artificial neuron within neural networks.

The perceptron computes the weighted sum of an input value with a weight added with a bias value. This operation is followed by the application of a non-linear activation function (such as sigmoid, ReLU, Leaky-ReLU, softmax [**25**]) to generate the output. This process is illustrated in Equation 2.1, where W_i, X_i, Y, and b denote the weights, input signals or input activations, output signals, and bias value, respectively. The function $f()$ represents the non-linear activation function. The functionality of a perceptron depends on the weight and bias values (parameters) which are chosen during training (see section 2.3.5).

$$Y_j = f(\sum_i W_{ji} X_i + b) \tag{2.1}$$

2.3.2 Multi-layer Perceptron

When a network comprises multiple perceptrons fully connected in a feedforward arrangement across layers, it forms a multilayer perceptron (MLP). The MLP is characterized by its interconnected layers of fully connected neurons, each equipped with a non-linear activation function. It typically includes a minimum of three layers and is able to discern data that is not linearly separable.

The input nodes, located in the input layer, receive and process input data, which then propagate through subsequent layers, often referred to as hidden layers. Ultimately, the output layer produces the neural network's predictions. This process of forward propagation is

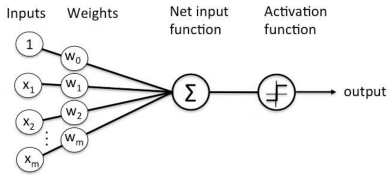


FIGURE 2.3. The mathematical model of a perceptron (or neuron)

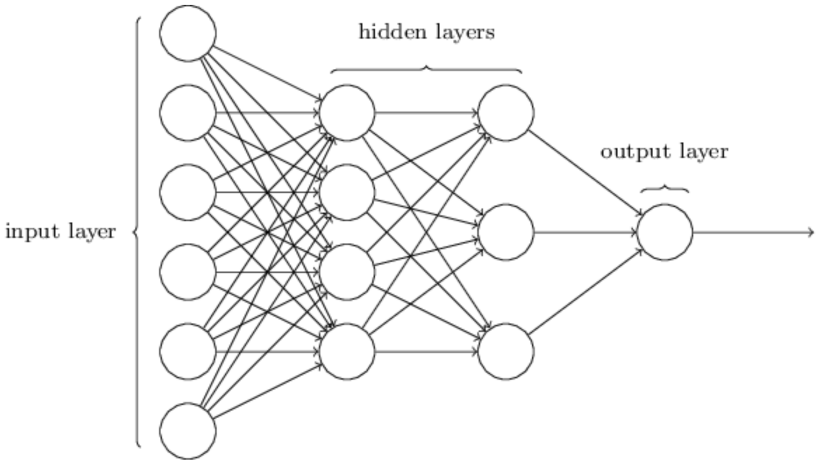


FIGURE 2.4. An example of a multilayer perceptron (or neural network)

FIGURE 2.5. The makings of a Neural Network

known as inference. Conversely, the adjustment and fine-tuning of the network's parameters constitute the training phase, a topic explored in Section 2.3.5.

The MLP neural network lays the groundwork for the subsequent neural networks discussed in the following sections.

2.3.3 Convolutional Neural Networks (CNNs)

Convolutional neural networks (CNNs) excel in processing image, speech, or audio signal inputs and are characterized by three main types of layers:

(1) Convolutional layer
(2) Pooling layer
(3) Fully-connected (FC) layer

The convolutional layer is typically the initial layer in a CNN, and is usually followed by a pooling layer. It can have multiple convolutional or pooling layers. The fully-connected layer is the final layer in the network.

The convolutional layer is a fundamental component of CNNs. It takes input data, which can have multiple channels (e.g., RGB images with 3 channels), and uses a filter to produce

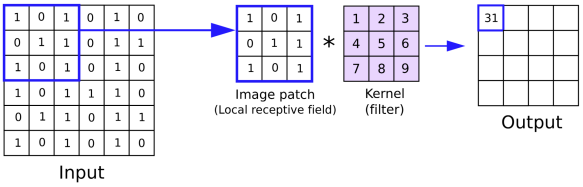


FIGURE 2.6. Convolution Operation in Convolutional Layer

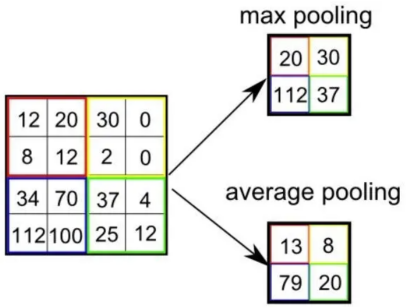


FIGURE 2.7. Pooling operation in CNN

a feature map. The filter, also called a kernel, acts as a feature detector, moving across the input data's receptive fields and mulitplying the input with the filter. This process is called convolution.

The feature detector or filter/kernel is made up of weights and biases. Filters are typically 3x3 matrices, which also determine the receptive field's size. The filter is applied to a portion of the input data, and a dot product is computed between the input and the filter. This dot product is then stored in an output array. The filter shifts with a certain stride, repeating the process until it covers the entire input. The final result, obtained from the dot products, is referred to as a feature map, activation map, or convolved feature. Following each convolution, a Rectified Linear Unit (ReLU) transformation is applied to the feature map, introducing non-linearity to the model. Unlike MLPs, in CNNs weights and biases are shared across multiple neurons.

Three main parameters influence the convolutional layer: the number of filters (determining the output feature maps), the stride length of the filter (affecting the size of the output feature map), and the padding used to align the input and filter sizes. The output size of a convolutional layer can be determined using Equation 2.2, where W is the input volume, K is the kernel size, P is the padding used, and S is the stride.

$$OutputShape = \frac{W - K + 2P}{s} + 1 \tag{2.2}$$

Pooling layers, also known as down-sampling layers, reduce the dimensionality of the input, helping to decrease the number of parameters. Unlike the convolutional layer, the pooling operation doesn't involve weights. Instead, it applies an function within a receptive field to produce the output array. There are two main types of pooling: max-pooling, which selects the maximum value within the receptive field, and average-pooling, which calculates the average within the receptive field.

After the pooling layer, the Fully Connected (FC) layer is responsible for classification. It uses the features extracted by the previous layers and their various filters. The number of input neurons in the FC layer is equal to the flattened output of the pooling layer.

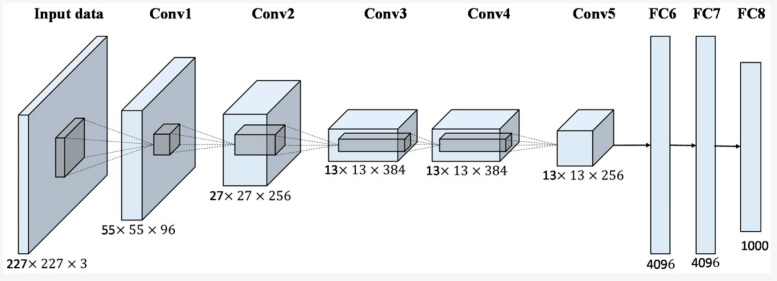


FIGURE 2.8. Example of a CNN with multiple convolutional layers (AlexNet architecture)

2.3.4 Residual Networks (ResNets)

Residual networks build of CNNs and aim to address the problem that arises when trying to train very deep CNNs. The residual block is at the heart of ResNets and can be seen in Figure 2.9 below.

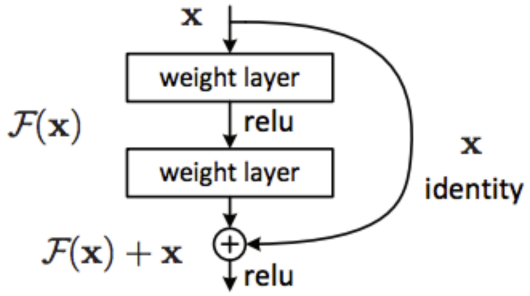


FIGURE 2.9. Residual Block: A building block of Residual Networks

Residual blocks in neural networks include a 'Skip connection,' which represents an identity mapping. This mapping has no parameters and is used to add the output from the previous layer to the subsequent layer. However, there are situations where the input x and the output $F(x)$ do not have the same dimensions. This difference may occur because a convolution operation typically reduces the spatial resolution of an image. To address this, the identity mapping is multiplied by a linear projection matrix W, which expands the channels of the shortcut to match the residual. This adjustment ensures that the input x and the output $F(x)$ can be effectively combined as input for the next layer.

Residual networks, introduced in 2015 in the paper [**16**], enable the training of exceptionally deep neural networks. The first ResNet architecture, ResNet-34, is depicted in Figure 2.10.

2.3.5 Training Neural Networks

The backpropogation algorithm is the most important algorithm when it comes to training neural networks. It is what allows a neural network to learn complex patterns from data and make predictions. It was first introduced in the 1950s and 60s but popularised in the paper

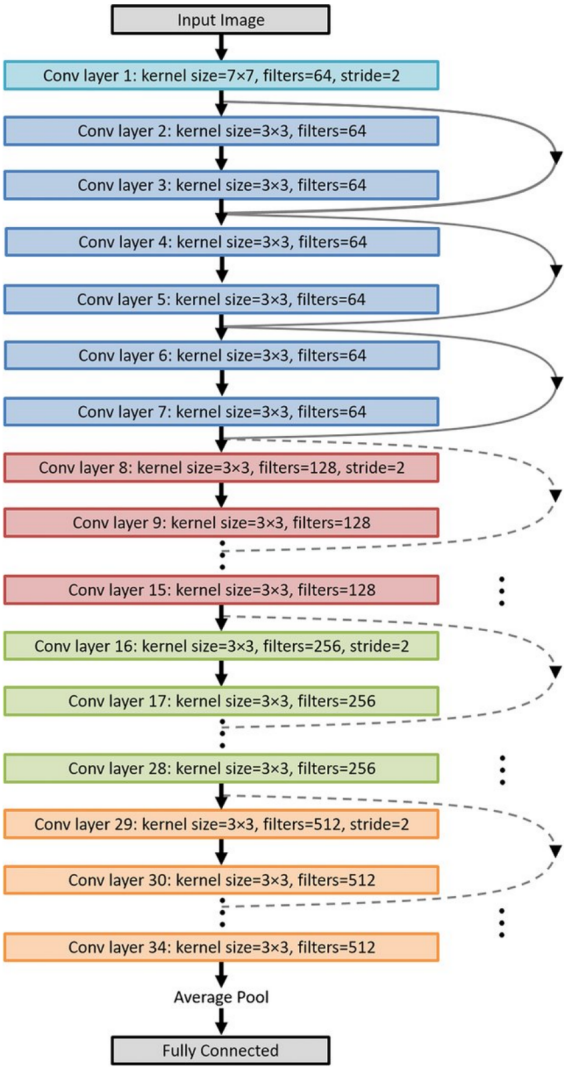


FIGURE 2.10. Architecture of ResNet-34

"Learning representations by back-propogating errors" by David E. Rumelhart et al. [**26**]. In essence, for each forward propogation of a neural network, the algorithm performs a backward pass that updates the networks parameters (weights and biases).

For each forward pass of a training instance, the last step of a neural network is to evaluate the predicted outcome vs the expected outcome. This is determined through a cost function (also known as a loss function) C. Example cost functions are mean-squared-error (MSE) and cross-entropy loss function as seen in Figures 2.11 and 2.12. Backpropagation aims to minimize the cost function by adjusting network's weights and biases. The level of adjustment is determined by the gradients (partial derivatives) of the cost function with respect to those parameters. Partial derivatives are computed because recalling from calculus, for a partial derivative $\frac{\partial f}{\partial x_i}$, the gradients shows how much a the parameter x_i needs to change (in positive

$$\text{Loss} = -\sum_{i=1}^{\text{output size}} y_i \cdot \log \hat{y}_i$$

FIGURE 2.11. Cross-Entropy (Log Loss) function for multi-class prediction

$$MSE = \frac{1}{n}\Sigma\underbrace{\left(y - \widehat{y}\right)^2}_{\text{The square of the difference between actual and predicted}}$$

FIGURE 2.12. MSE loss function for multi-class prediction

or negative direction) to minimize f. Thus, for a neural network, the partial derivatives $\frac{\partial C}{\partial w_{jk}^l}$ and $\frac{\partial C}{\partial b_j^l}$ need to be calculated where w_{jk}^l is the weight connecting neuron k at layer $l-1$ to neuron j at layer l and b_j^l is the bias at neuron j in layer l.

These partial derivatives are calculated using the chain rule. For a single weight b_j^l, the gradient is:

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial w_{jk}^l} \qquad \textit{chain rule}$$

$$z_j^l = \sum_{k=1}^{m} w_{jk}^l a_k^{l-1} + b_j^l \qquad \textit{by definition}$$

$$m - \textit{number of neurons in } l-1 \textit{ layer}$$

$$\frac{\partial z_j^l}{\partial w_{jk}^l} = a_k^{l-1} \qquad \textit{by differentiation (calculating derivative)}$$

$$\frac{\partial C}{\partial w_{jk}^l} = \frac{\partial C}{\partial z_j^l}a_k^{l-1} \qquad \textit{final value}$$

For a single bias w_{jk}^l, the gradient is:

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l}\frac{\partial z_j^l}{\partial b_j^l} \qquad \textit{chain rule}$$

$$\frac{\partial z_j^l}{\partial b_j^l} = 1 \qquad \textit{by differentiation (calculating derivative)}$$

$$\frac{\partial C}{\partial b_j^l} = \frac{\partial C}{\partial z_j^l}1 \qquad \textit{final value}$$

The common $\frac{\partial C}{\partial z_j^l}$ is known as the error of the neuron j at the layer l and is often expressed as:

$$\delta_j^l = \frac{\partial C}{\partial z_j^l}$$

At the output layer, the error δ_j^L is given by the equation:

$$\delta_j^L = \frac{\partial C}{\partial a_j^L} \times \sigma'(z_j^L)$$

At the hidden layers, the error δ_j^l is given by the equation:

$$\delta_j^L = ((w^{l+1})\delta^{l+1}) \odot \sigma'(z_j^l)$$

In summary, there are four fundamental equations for backpropogation, which can be seen in the Figure 2.13 below. For a CNN, the backpropogation equations remain largely the same but differ slightly. For a detailed explanation on that, refer [**27**].

$$\delta^L = \nabla_a C \odot \sigma'(z^L) \tag{BP1}$$

$$\delta^l = ((w^{l+1})^T \delta^{l+1}) \odot \sigma'(z^l) \tag{BP2}$$

$$\frac{\partial C}{\partial b_j^l} = \delta_j^l \tag{BP3}$$

$$\frac{\partial C}{\partial w_{j\,k}^l} = a_k^{l-1} \delta_j^l \tag{BP4}$$

FIGURE 2.13. Four fundamental backpropogation equations for a neural network

After the partial derivatives have been calculated, the network parameters can be updated according to the equation below, where ϵ is the learning rate of the network:

$$w := w - \epsilon \frac{\partial C}{\partial w}$$

$$b := b - \epsilon \frac{\partial C}{\partial b}$$

2.4 Field Programmable Gate Arrays

Field Programmable Gate Arrays (FPGAs) are integrated circuits that allow users to re-configure or reprogram them post-manufacturing. This sets them apart from Application

Specific Integrated Circuits (ASICs), which are custom-designed for specific tasks and thus lack reconfigurability. FPGAs consist of three primary components: Configurable Logic Blocks (CLBs), Input-Output Blocks (IOBs), and Programmable Interconnects. Figure 2.14 illustrates a typical FPGA structure.

The foundational building blocks within an FPGA are the Configurable Logic Blocks (CLBs). Also known as slices or logic cells, CLBs comprise two essential elements: flip-flops and lookup tables (LUTs). Flip-flops (FFs) function as registers, capable of storing data and serving as basic memory devices. Within a CLB, Lookup Tables (LUTs) are responsible for implementing much of the combinational logic. This is achieved by utilizing truth tables to define various combinatorial logic functions such as AND, OR, NAND, and XOR, which are stored within the LUT's memory. In a CLB, LUTs are interconnected with FFs to perform a wide range of combinational and sequential logic operations. Figure 2.15 offers a glimpse of a typical CLB structure where a LUT and FF can be seen.

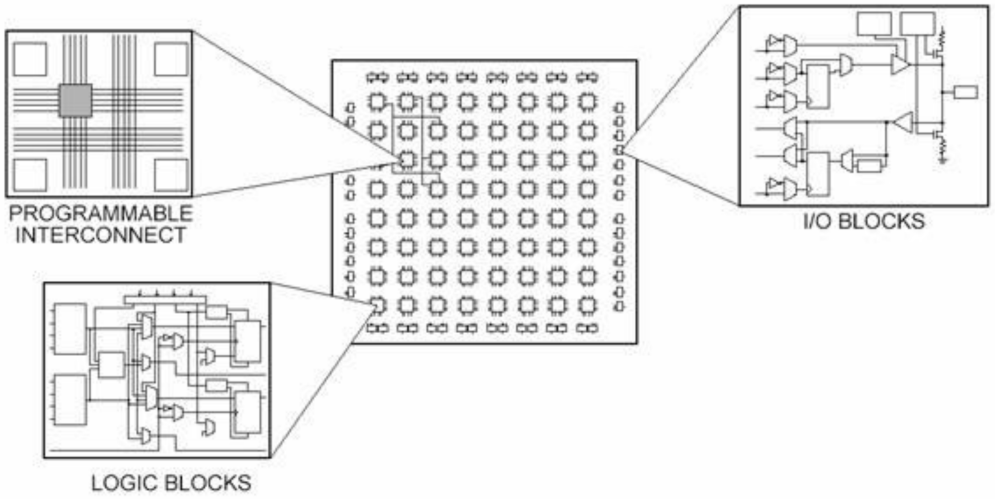


FIGURE 2.14. Typical FPGA structure

Programmable interconnects are the programmable switches and wiring resources which play a critical role in linking Configurable Logic Blocks (CLBs) and other components within the FPGA, including Input-Output Blocks (IOBs). These interconnects provide the necessary flexibility for routing and interconnecting signals between different elements on the FPGA chip. IOB blocks, on the other hand, serve as the crucial interface points between the FPGA and external devices or other components on the board. These IOBs enable the FPGA to communicate effectively with a wide range of peripheral devices. The reconfigurability of FPGAs hinges on the flexibility of their programmable CLBs and Interconnects. By allowing the reprogramming of LUTs within CLBs and the reconfigurability of the interconnects, FPGAs can be tailored and customized to perform a diverse array of tasks.

Most modern FPGAs integrate the above basic components into common complex circuits embedded onto the FPGA to reduces the area required and gives those functions increased

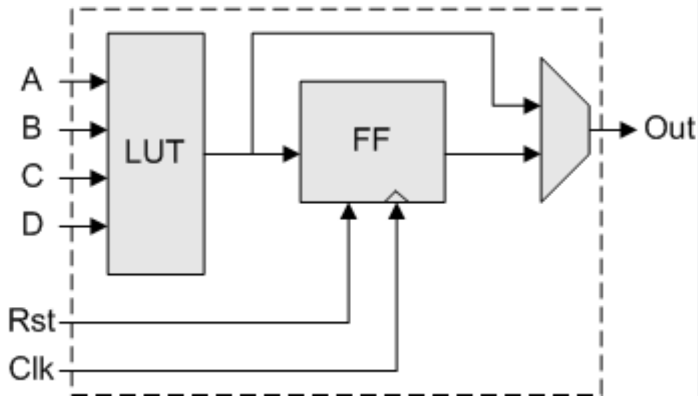


FIGURE 2.15. Typical Configurable Logic Block (CLB) structure containing a LUT and FF

performance. These include embedded memory such as block RAM (BRAM), digital signal processing (DSP) blocks, embedded processors and high speed I/O logic. Clock management resources are also a key component of FPGAs. Specialized resources like Phase-Locked Loops (PLLs) and Digital Clock Managers (DCMs) are used to handle the generation, distribution, and management of clock signals within an FPGA. These resources play a crucial role in ensuring precise timing and synchronization across the FPGA.

In 2012, a significant development emerged in the field of programmable devices when researchers successfully integrated Configurable Logic Blocks (CLBs) and interconnects from traditional FPGAs with embedded microprocessors and associated peripherals, resulting in a complete system on a programmable chip. Notable examples of such hybrid technologies can be found in products like the Xilinx Zynq-7000 All Programmable SoC. In the context of this thesis, the ZCU111 Evaluation Board will be utilized [**28**]. The ZCU111 board is equipped with the Zynq UltraScale+ XCZU28DR-2FFVG1517 RFSoC, which seamlessly combines a robust processing system (PS) with versatile programmable logic (PL) within a single integrated device.

2.4.1 FPGA Design

In traditional programming, code is typically written and then compiled into instructions that execute on a processor. However, programming with Field-Programmable Gate Arrays (FPGAs) follows a distinct approach. To define the behavior of an FPGA, users provide a design using a Hardware Description Language (HDL). Given that FPGAs are hardware devices, Hardware Description Languages (HDLs) are specify digital circuits at a high level of abstraction, which FPGAs will subsequently configure themselves to implement. The two primary HDLs commonly used in FPGA design are VHDL and Verilog. For this project, SystemVerilog, an extension of Verilog, will be utilized.

The FPGA design flow consists of several steps that go from concept ideation to a fully functional FPGA implementation. As figure 2.16 shows, these steps include (sourced from [**29**]):

(1) **Design entry:** Designers create a high-level representation of the desired digital circuit using an HDL like VHDL or Verilog.
(2) **RTL design and simulation:** The HDL code is translated into an RTL representation, which is then simulated to verify functionality and performance.
(3) **Synthesis:** The RTL design is converted into a gate-level netlist, a representation of the digital circuit using gates and flip-flops.
(4) **Implementation:** The gate-level netlist is mapped to the FPGA's resources, including CLBs, DSP slices, and programmable interconnects. This step includes place-and-route and bitstream generation.
(5) **Uploading and Testing:** FPGA devices are tested and debugged with simulation tools, testbench, or target hardware to ensure correct functionality and performance.

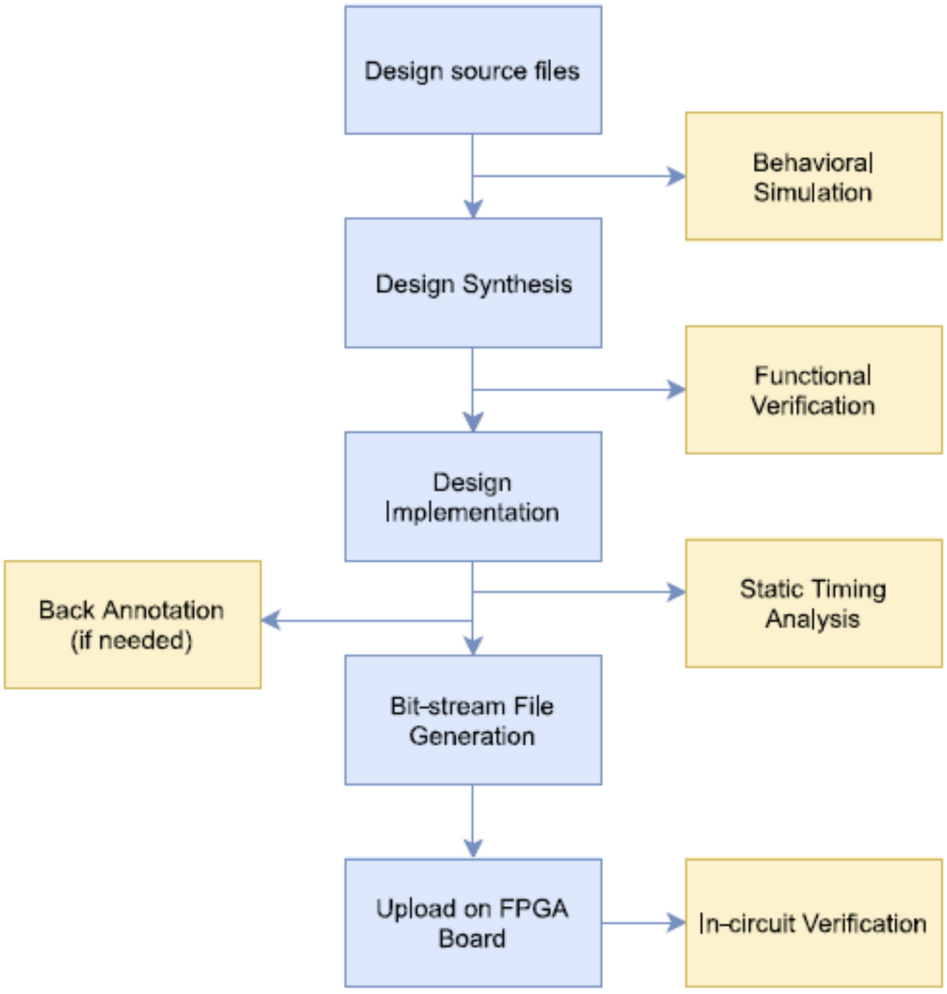


FIGURE 2.16. FPGA Design flow

Throughout the design flow, various tools and software are used to facilitate the design, simulation, synthesis, and implementation of FPGA designs. For our purposes, we will be utilizing Xilix Vivado. The Vivado simulator is also used for RTL simulation.

2.4.2 FPGA Implementations of Neural Networks

Neural network (NN) implementations on Field-Programmable Gate Arrays (FPGAs) have undergone extensive research and development. The prevalent approach involves creating a feed-forward NN, where the training of the network occurs in an offline computing environment. This involves pre-determining the weights, biases, and activations for a specific task and configuring them to fit the hardware-based NN architecture. The hardware design of NNs on FPGAs involves a series of crucial decisions. Firstly, one must determine the architecture of the NN. Subsequently, the numbering system and data precision to be employed for computations must be defined, whether it's binary, fixed-point, or floating-point representation. This decision significantly impacts the accuracy of the network, resource utilization, and power consumption. It also dictates the specific hardware modules responsible for carrying out the computational tasks.

In their paper titled 'Neural Network Implementation on an FPGA,' Chen and du Plessis [**30**] developed a feed-forward neural network on an FPGA and conducted a study focused on determining the minimum required precision to achieve a recognition rate of at least 95% for two characters in an optical character recognition application.

The neural network they employed featured 20 input nodes, 4 hidden neurons, and 2 output nodes. Throughout the network, they utilized 8-bit synaptic weight precision in integer format, including input data. Feed-forward computations were executed using multiply-accumulate (MAC) operations implemented through twos-complement serial-parallel multipliers, composed of serial adders. The activation function of choice in this neural network architecture was a piecewise linear (PWL) activation function. Network training was conducted offline.

Chen and du Plessis conducted experiments involving various activation functions, the number of hidden neurons, and data precision to determine the most effective network configuration. They found that a 3-segment PWL activation function produced the best performance, achieving a 94% accuracy rate. Furthermore, they observed that as the number of hidden neurons decreased, a significantly higher data precision was required to maintain similar performance levels. Conversely, increasing the number of hidden neurons allowed for a reduction in weight precision to as low as 2 bits. Ultimately, they concluded that a minimum of 8-bit weight precision was necessary to achieve satisfactory results in their neural network implementation.

In a related study by Gaikwad et al. titled 'Efficient FPGA Implementation of Multilayer Perceptron for Real-Time Human Activity Classification' [**31**], the researchers proposed a dedicated hardware-based system for human activity recognition (HAR) in the context of smart military wearables. Their system hinged on the utilization of a multilayer perceptron (MLP) algorithm for activity classification, and this algorithm was designed and implemented in hardware, capitalizing on the inherent parallelism of MLP computations, and executed on an FPGA.

To identify the optimal MLP configuration, Gaikwad and the team crafted and tested various MLP topologies and architectures using the UCI human activity dataset. Ultimately, they settled on a 7-6-5 MLP architecture with 16-bit fixed data precision, achieving a balance of efficient classification accuracy, resource utilization, and power consumption. In this setup, the sigmoid function was adopted as the activation function, specifically opting for the PLAN approximation due to its resource-efficient characteristics compared to other sigmoid approximations. Training for this project was conducted offline.

Regarding their findings, Gaikwad et al. observed that the classification accuracy remained relatively stable up to 16-bit precision, beyond which it began to decline. Interestingly, they noted that models with greater accuracy requirements also necessitated increased power and hardware resource usage, highlighting the trade-offs in precision, accuracy, and resource constraints in their FPGA-based MLP implementation.

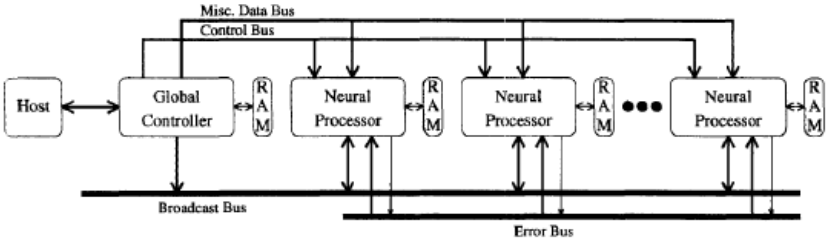


FIGURE 2.17. RRAN neural network architecture

In contrast to the preceding examples where training was conducted offline, Eldredge and Hutchings in their work titled 'A hardware implementation of the Backpropagation Algorithm using Reconfigurable FPGAs' [**32**] pursued a hardware-based training approach. Their research introduces the Run-Time Reconfiguration Artificial Neural Network (RRANN). The authors highlight that "RRANN is a hardware implementation of the backpropagation algorithm that is highly scalable and optimally utilizes FPGA resources. One notable feature of RRANN is its adeptness at harnessing parallelism across all phases of the backpropagation algorithm, including the stage involving error propagation through the network" [**32**].

The RRANN architecture can be split into three sequential stages: feed-forward, backpropagation, and update. The feed-forward stage performs the processing of input patterns and propagating them through the network, assigning activation (output) values to each neuron. The backpropagation stage is responsible for identifying output errors and subsequently propagating them backward through the network to calculate errors for neurons within hidden layers. Once error values have been assigned to all non-input neurons, the update stage comes into play. This phase utilizes activation and error values obtained from the preceding stages to calculate weight adjustments, which are then applied to update all network weights.

Figure 2.17 provides an overview of the RRANN architecture. Similar to the previous examples, RRANN leverages bit-serial hardware for its computations and employs a Look-Up Table (LUT) for implementing the activation function and its derivative.

The papers discussed above have proven to be invaluable sources of insight for this thesis, which strives to create a hardware-based adapter module with in-hardware training. These studies offer a comprehensive understanding of common hardware neural network architectures and the modular components used to construct them. They shed light on optimization strategies for hardware implementations, including decisions related to numbering systems, all of which contribute to achieving optimal results in terms of accuracy and hardware efficiency. These research findings have significantly influenced the design choices and methodologies employed in this project, as elaborated upon in Chapter 3.1.

2.5 Adapter Modules

2.5.1 Residual Adapters

In their paper [**15**], Rebuffi, Bilen, and Vedaldi introduce "residual adapter modules," which serve to parameterize a standard residual network architecture, particularly ResNet28 [**33**]. Recall that in Residual Networks, each residual block is defined by the equation $g(x) = F(x) + x$, where x represents the input to the block, and $F(x)$ constitutes a portion of a convolutional neural network composed of multiple convolutional layers.

Rather than directly altering the filter coefficients of the ResNet network, Rebuffi et al. propose the addition of parametric convolutional layers to enable domain-specific learning. Furthermore, they take an innovative step by defining these added layers as small residual modules in their own right, dubbing them "residual adapter modules."

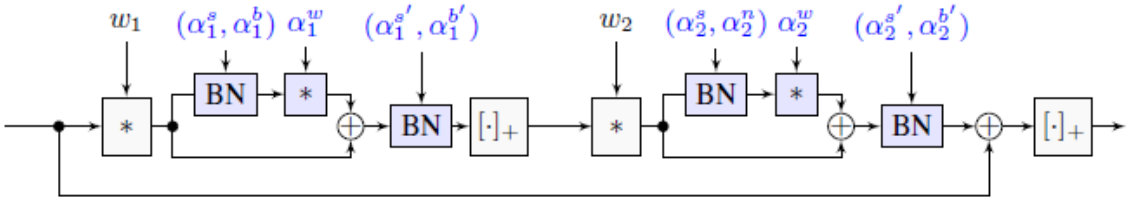


FIGURE 2.18. The figure shows a standard residual module with the inclusion of adapter modules (in blue). The filter coefficients (w_1, w_2) are domain-agnostic; (a_1, a_2) contain a small number of domain-specific parameters

These residual adapter modules have the form as follows:

$$g(x; \alpha) = x + \alpha * x \tag{2.3}$$

x here is the result of the the convolution of the input to that layer with the original filter bank at that layer. To maintain the goal of limited parameters in domain-specific learning, α is selected to be a bank of 1x1 filters. The adapter dimensionality is determined by the number of channels in the layers before and after it.

As illustrated in figure 2.18, these residual adapters are integrated within a residual block, in series with the convolutional Layer. They take the output of the convolutional layer as input

and perform their own convolution using a 1x1 kernel. The resulting output of the convolution and the adapter is then summed to create the adapter convolution layer.

Given that the adapter is embedded within a deep ResNet network, they employ Batch Normalization, followed by rescaling and shift operations that introduce additional learnable parameters. These operations are incorporated into the adapter modules and are positioned before the adapter convolution layer.

The above adapter architecture offers several distinct advantages. Firstly, it provides clear differentiation between layers that are domain-agnostic (shared across domains) and those that are domain-specific (parametric and adaptable based on the domain). Importantly, it introduces only a small fraction of model-specific parameters, typically less than 10%, promoting a high degree of parameter sharing across domains. Additionally, the architecture effectively addresses the challenge of avoiding forgetting. To achieve this, it adheres to the "tower model" [**34**], which ensures the preservation of the original model structure. Domain-agnostic parameters are initially trained on a broad data-set like ImageNet, and subsequently, domain-specific parameters are fine-tuned for each specific domain, facilitating adaptation without sacrificing previous knowledge.

Model	Airc.		C100		DPed		DTD		GTSR		Flwr		OGlt		SVHN		UCF	
Finetune	1.1	60.3	3.6	63.1	0.6	80.3	0.7	45.3	1.4	68.1	27.2	73.6	13.4	87.7	0.2	96.6	5.4	51.2
LwF [21] high lr	4.1	61.1	21.0	82.2	23.8	92.3	36.7	58.8	11.5	97.6	34.2	83.1	3.0	88.1	0.2	96.1	18.6	50.0
LwF [21] low lr	38.0	50.6	33.0	80.7	53.3	92.2	47.0	57.2	23.7	96.6	45.7	75.7	21.0	86.0	13.3	94.8	29.0	44.6
Res. adapt. finetune all	59.2	63.7	59.2	81.3	59.2	93.3	59.2	57.0	59.2	97.5	59.2	83.4	59.2	89.8	59.2	96.1	59.2	50.3

FIGURE 2.19. Residual Adapter results on Visual Decathon Challenge: Each pair of numbers report the top-1 accuracy (%) on the old task (ImageNet) and the new task after the network has undergone full fine-tuning on the latter

Rebuffi et. al. introduced a novel benchmark named "Visual Decathlon" to assess the performance of their parameterizable ResNet28 architecture. The primary objective of this benchmark is to evaluate whether a method can effectively learn to perform well in multiple diverse visual domains simultaneously. They selected ten visual domains from well-known datasets.

Two key experiments were conducted to evaluate their approach. In the first experiment, a ResNet28 model was initially trained on the ImageNet dataset. Subsequently, four different techniques were employed to adapt it to the remaining nine domains. These four baselines included:

(1) Learning an individual ResNet model from scratch for each task.
(2) Freezing all parameters of the pre-trained network and utilizing the network as a feature extractor while learning only a linear classifier.
(3) Standard fine-tuning of the entire pre-trained network.
(4) Another method, not explicitly described here.

We will focus on the fine-tuning method since it is the most relevant for comparison with the domain-adapter network.

In the second experiment, the residual adapter modules were introduced into the pre-trained ResNet28 network. The original network, having been pre-trained on ImageNet, had its domain-agnostic parameters frozen, while the domain-specific parameters were trained on the different datasets.

The results of the first experiment showed that full fine-tuning, while producing accurate results for both large and small datasets, substantially forgets the ImageNet domain knowledge. It also requires training ten complete ResNet models, effectively retraining all the network's parameters. On the other hand, the introduction and training of the residual adapter modules led to an increase of 11% in the number of parameters per domain. This approach only requires the training of one new model for the residual adapter module instead of ten new models as in full fine-tuning. Despite sharing most of the parameters, this method either matches or outperforms full fine-tuning. As observed with full fine-tuning in figure 2.19, the network exhibits significantly reduced accuracy for the original ImageNet task, indicating that it has 'forgotten' much of its original knowledge after adapting to the new dataset. In contrast, the residual adapter models retain the majority of the original model's knowledge and even result in improved accuracy after fine-tuning in some of the datasets.

The benefits of residual adapter modules for domain-specific adaptation are evidently clear. In contrast to full fine-tuning, the use of residual adapters offers several significant advantages. These advantages are readily apparent in their application on ResNet28 across ten very large datasets, but it can be confidently inferred that the same principles can be effectively extended to various other datasets and domains. The promising results obtained with this architecture on the different image domains demonstrate the considerable potential of residual adapter modules.

2.5.2 Parallel Residual Adapters

Building on their work from 2017 as published in [**15**], Rebuffi et al. made advancements in their residual adapter modules in the paper [**35**]. They introduced different residual adapter configurations, including series and parallel adaptations. Also, their research introduced designs for domain-specific parameterization, encompassing joint adapter compression and parameter allocation strategies. These developments aimed to enhance the performance and adaptability of their models.

Recall in [**15**], Rbuffi et al. introduced a residual adapter module inserted in series into a residual network architecture ResNet28. In the paper [**35**], they introduce an alternative configuration in which the adapters are connected in parallel instead of in series.

These parallel residual adapter modules have the form as follows:

$$g(x;\alpha) = f * x + \alpha * x \tag{2.4}$$

f here is the result of the the convolution of the input to that layer with the original filter bank at that layer. Like the series adapter, α is selected to be a bank of 1x1 filter. Once again, the adapter dimensionality is determined by the number of channels in the layers before and after it.

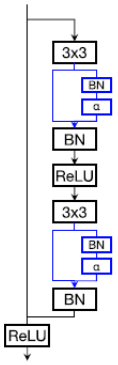


FIGURE 2.20. Series Residual adapter within a residual network inclusive of batch normalization layers

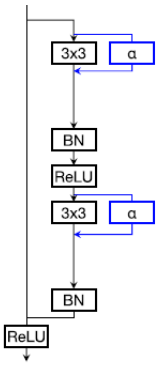


FIGURE 2.21. Parallel Residual adapter within a residual network inclusive of batch normalization layers

FIGURE 2.22. Series vs Parallel adapters with residual network

In Figure 2.22, the depiction illustrates the integration of both series and parallel adapter modules within a residual block. Notably, the parallel adapter diverges in its approach by taking the input to the residual block as its primary input and then executing its independent convolution operation, employing a 1x1 kernel. Nevertheless, similar to the series adapter, the ultimate outcome involves the summation of the original filter bank convolution and the output from the 1x1 adapter convolution, culminating in the creation of the adapter convolution layer.

The parallel adapter configuration not only preserves the same advantages as the series adapter but also introduces additional benefits. A key advantage of the parallel configuration lies in its "plug-and-play" nature. Unlike the series configuration, where adapters must be incorporated during the pretraining of the original ResNet network, parallel adapters can be seamlessly appended to any pre-trained network. This level of flexibility enables their utilization with off-the-shelf models, enhancing their versatility and ease of integration into existing architectures.

Apart from introducing a new adapter configuration, Rebuffi et al. also experimented with how best to apply the adapters to a deep neural network. Using the ResNet28 network as a base, they apply the adapter throughout the networks depth, from the shallow layers to the intermediate and deep layers. They also explore different regularization methods such as shrinkage and dropout.

These experiments are all performed on the Visual Decathlon benchmark introduced in [**15**]. Results are similar to previous chapter. The parallel configuration, like its series counter part, performs at a level equal to or sometimes greater than full fine-tuning. Notably, a new observation emerges from these experiments: the parallel configuration consistently outperforms the series counterpart. In regards to the experiments considering location of residual adapters, they are most beneficial in the last block which suggests that filters become more specialized and domain specific towards end of network. Based of these results, Rebuffi et. al. conclude parallel adapters are a simple strategy that can replace and outperform standard fine-tuning in almost every way as the parallel configuration can be applied to an off-the-shelf model a-posteriori [**35**].

Once again, this paper highlights the tremendous potential of residual adapters for multiple-domain learning. However, it goes beyond that by introducing the concept of different adapter designs, such as series and parallel residual adapters. In this specific study, the parallel adapter design outperformed the series design. This observation suggests that the exploration of various adapter architectures and designs holds great promise and could potentially lead to superior methods for multiple-domain learning.

Furthermore, this paper introduces the idea of exploring different locations within the base model to place the adapter. This innovative approach emphasizes the untapped potential in this aspect of the research. It suggests that by strategically positioning adapters within the network, there is a wealth of opportunities to enhance the adaptability and performance of models in a multi-domain context. In essence, this research opens up exciting avenues for further exploration and refinement in the field of multiple-domain learning.

2.5.3 Deep Adaptation Networks (DAN)

Building upon the foundational work of Rebuffi et al. in their 2017 and 2018 papers [**15**, **35**], researchers sought to explore alternative methods for multiple-domain learning (MDL). In this quest, Rosenfeld et al., in their paper titled 'Deep Adaptation Networks' (DAN) [**36**], introduced a model by the name of DANs designed to tackle MDL challenges. DANs shared the overarching objectives with the preceding residual adapters, which were:

- Minimize the number of additional parameters required for each domain
- Retain performance of already learned datasets (avoid forgetting)
- Utilize the same computational pipeline

DANs share a common approach with residual adapters in terms of their modification of convolutional layers within a network by introducing parameterizable filter banks. Notably, what was referred to as "adapter modules" in the previous works [**15**, **35**], are now termed "controller modules". Figure 2.23 represents a controller module designed for a single added

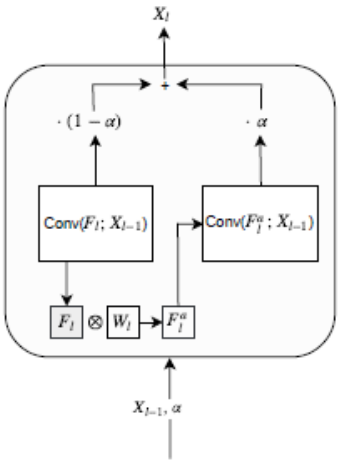


FIGURE 2.23. Deep Adaptation Network Controller Module

task. It's important to note that an arbitrary number of controller modules can be incorporated, and the α variable is extended to a vector rather than a scalar.

The core concept involves the alteration of filters within a convolutional layer of the base network. This alteration is achieved by re-combining the weights of these filters through a controller module. A switching variable denoted as α is responsible for choosing between the original filters represented by F_l and the newly created ones, denoted as F_l^a. The specific mathematical details governing the parameter re-combination process can be found in the cited source [**36**].

Like residual adapters, DANs introduce only a small number of parameters per domain, roughly 13%. However, DANs differ in key ways. They employ a switching variable α in controller modules to choose between original convolutional filters and adapter filters, allowing dynamic behavior switching for domains. Moreover, DANs allow for the encoding of multiple domains in a single network through the addition of multiple controller modules in various network layers. This flexibility sets DANs apart from residual adapters.

Method	#par	ImNet	Airc.	C100	DPed	DTD	GTSR	Flwr	Oglt	SVHN	UCF	mean	S
Scratch	10	59.87	57.1	75.73	91.2	37.77	96.55	56.3	88.74	96.63	43.27	70.32	1625
Feature	1	59.67	23.31	63.11	80.33	45.37	68.16	73.69	58.79	43.54	26.8	54.28	544
Finetune	10	59.87	60.34	82.12	92.82	55.53	97.53	81.41	87.69	96.55	51.2	76.51	2500
LWF	10	59.87	61.15	82.23	92.34	58.83	97.57	83.05	88.08	96.1	50.04	76.93	2515
Res. Adapt.	2	59.67	56.68	81.2	93.88	50.85	97.05	66.24	89.62	96.13	47.45	73.88	2118
Res. Adapt (Joint)	2	59.23	63.73	81.31	93.3	57.02	97.47	83.43	89.82	96.17	50.28	**77.17**	2643
DAN (Ours)	2.17	57.74	64.12	80.07	91.3	56.54	98.46	86.05	89.67	96.77	49.38	77.01	**2851**

FIGURE 2.24. Results on Visual Decathlon Challenge for Deep Adaptation Network

Rosenfeld et al. utilize the Visual Decathlon Benchmark introduced in [**15**] to test their network. Similar to [**15**, **35**], they use a Wide Residual Network [**33**] as the core architecture

for parameterization with their adapter modules. The initial step involved pre-training the network on the ImageNet dataset. Subsequently, they used the DANs to fine-tune and adapt the network for diverse target tasks.

In Figure 2.24, the results are presented, offering a comparison with various other approaches, including training each task from scratch, using a pre-trained network as a feature extractor, fine-tuning, and results obtained from Rebuffi et al. [**15**, **35**]. Much like their predecessors, the DAN networks demonstrate comparable or superior performance to fine-tuning, while also preserving the original network's knowledge. Notably, in terms of the Visual Decathlon S score, the DAN network surpasses the results achieved by the residual adapters.

DANs, much like residual adapters, highlight the advantages of adapter modules in the context of multi-domain learning. While they share common benefits, DANs introduce several advantages. They allow for dynamic switching between the original network and the parameterized network, improving adaptability. Also, the ability to accommodate multiple domains within a single network with the inclusion of multiple controller modules is a distinct and powerful feature. This again shows the room for improvement and untapped potential in the field of multi-domain learning, showcasing the diversity of approaches available to address this challenging problem.

2.5.4 Conv-Adapters

The paper titled 'Conv-Adapter: Exploring Parameter Efficient Transfer Learning for ConvNets' [**37**] introduces another adapter module for multi-domain learning, known as Conv-Adapter. The motivation behind creating this adapter module arose from the challenges encountered when applying previous adapter techniques developed by Rebuffi et al. [**15**, **35**] to deep convolutional neural networks (ConvNets).

Chen et al. pointed out that the use of 1x1 convolutional layers for the adapters could lead to suboptimal transfer performance. This issue stems from the loss of spatial locality, which is encoded in the structural feature maps through convolutions with kernel sizes larger than 1. This concern becomes particularly relevant when attempting to apply adapters to ConvNets. Additionally, the spatial size of feature maps in ConvNets plays a significant role in determining the effectiveness of adaptation. Previous efforts to use adapters for transferring ConvNets often involve downsampling the spatial size of features to optimize memory and parameter efficiency. However, for computer vision tasks extending beyond image classification, such as segmentation, preserving the spatial dimensions of feature maps is crucial for achieving favorable results.

To address these challenges, this paper introduces an adapter specifically designed for ConvNets, which utilizes a bottleneck structure. This structure comprises two convolutional layers with a non-linear function sandwiched in between. The first convolutional layer serves to downsample the channel dimension, employing a kernel size similar to that of the adapted blocks. Subsequently, the second convolutional layer projects the channel dimension back to its original size. The non-linear activation function employed within the bottleneck structure corresponds to the one used in the ConvNet backbone, thus maintaining consistency.

Importantly, this Conv-Adapter approach retains the spatial size of the feature maps to enhance transferability, particularly for tasks involving dense predictions. Furthermore, the paper employs depth-wise separable convolutions within the Conv-Adapter to further reduce the parameter size, optimizing its efficiency

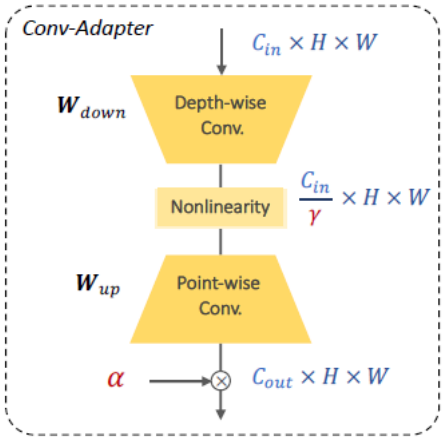


FIGURE 2.25. Architecture of ConvAdapter

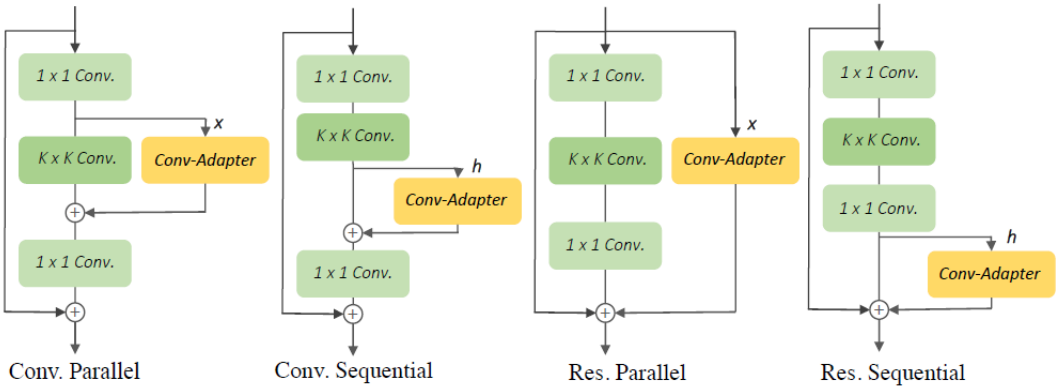


FIGURE 2.26. Four adapting schemes of Conv-Adapter to ResNet50

Figure 2.25 provides a visual representation of a Conv-Adapter module. For a comprehensive mathematical explanation, please refer to [**37**]. Figure 2.26 illustrates the integration of Conv-Adapters into ConvNets. Similar to residual adapters, Conv-Adapters offer both serial and parallel configurations, as well as the inclusion of residual connections. Just like their predecessors, Conv-Adapters introduce a relatively small number of additional parameters for each domain, approximately 3.4%. They retain the advantages associated with previous adapter architectures.

In the experimentation phase, these adapters were evaluated using two benchmark datasets, VTAB-1k and FGVC. To assess the effectiveness of the adapter method, various comparisons were conducted in conjunction with the following baseline methods:

- Full fine-tuning (FT)
- Linear probing (LP)
- Bias tuning (Bias)
- Bisual prompt tuning (VPT)

Tuning	# Param.	FGVC	VTAB-1k		
			Natural	Specialized	Structured
# Tasks	-	4	7	4	8
FT	23.89	83.46	72.19	**85.86**	**66.72**
LP	0.37	75.44 (1)	67.42 (4)	81.42 (0)	37.92 (0)
Bias	0.41	64.98 (0)	66.06 (4)	80.34 (0)	32.18 (0)
VPT	0.42	74.79 (1)	65.43 (2)	80.35 (0)	37.64 (0)
Conv. Par.	0.85	83.77 (3)	**72.60 (5)**	84.21 (1)	56.70 (1)
Conv. Seq.	0.87	79.68 (2)	72.28 (4)	83.85 (0)	58.50 (1)
Res. Par.	8.21	**84.24 (3)**	71.75 (4)	84.70 (0)	61.34 (1)
Res. Seq.	3.53	83.45 (2)	71.74 (4)	84.84 (0)	61.33 (2)

FIGURE 2.27. Results of experiments fpr ConvAdapter

Figure 2.27 presents the results of the experiments. Conv-Adapters not only show significant improvements over the baseline methods, but they also achieve performance levels that match or even surpass their fully fine-tuned counterparts on all evaluated domains. Importantly, this enhanced performance is achieved while introducing only around 3.5% of the parameters required for full fine-tuning of ResNet50. Notably, the convolution parallel configuration emerges as the best-performing option, as it has an effective balance between performance and parameter efficiency.

Once again, the advantages of adapter modules become evident through the Conv-Adapters. The architecture of Conv-Adapters is tailored to deeper networks and diverse datasets, showcasing their potential to operate effectively across a wide range of applications. This highlights the versatility and power of adapter modules in addressing various multi-domain learning scenarios and tasks.

CHAPTER 3

Methods

The primary goals of this thesis revolve around the development and integration of a hardware-based adapter architecture into a small network. The adapter will then be utilized for multi-domain learning using a small dataset. This chapter serves as an in-depth exploration of the developed adapter architecture, the network chosen for adaptation, and the dataset on which the adapter network will be evaluated.

This chapter also outlines the methodology employed throughout the thesis to achieve these objectives. The project is divided into two main components: software and hardware. In the software design phase, the adapter network is designed in software and tested using the dataset. It is initially developed and evaluated in PyTorch, an optimized Deep Learning tensor library based on Python and Torch, which serves as a benchmark. Following this, a custom C simulator is designed, involving manual coding of the adapter network, including backpropagation, and evaluation with the dataset is again performed. A comparison between the PyTorch benchmark and the custom C simulator is performed to ensure correctness before proceeding.

On the hardware side, a high-level circuit design of the adapter module is created using SystemVerilog, which is then converted into an RTL design. Unlike the software aspect, where the design invloves both the adapter and network integration, the hardware design focuses solely on the adapter module. Correctness is verified by generating inputs for the adapter module in software and providing them to the network. The adapter module is then simulated using Vivado Simulator, and its output is compared with the C simulator's output. Detailed explanations of this process can be found in Section 3.3. Once correctness is validated, the adapter modules are synthesised and inmplented onto the target board using Xilix Vivado, and resource utilization, power consumption, and latency assessment of the adapter module is carried out.

Figure 3.1 below illustrates the workflow followed throughout the thesis, as described above. For the C simulator network and the adapter modules in hardware, various numbering systems and bit-widths are utilized. For this project, floating point representation is used, specifically 32-bit IEEE single precision, 16-bit IEEE half precision, and a custom 12-bit and 10-bit format. Further details about the numbering system can be found in Section 3.2 and 3.3.

FIGURE 3.1. Workflow followed throughout thesis

Bit-width	Type	Exponent	Mantissa
32-bit	IEEE	8	23
16-bit	IEEE	5	10
12-bit	Custom	4	7
10-bit	Custom	4	5

TABLE 3.1. Various floating-point format of designed adapter modules

3.1 Adapter Architecture

In determining the adapter architecture, a comprehensive review of the adapter architectures presented in the literature, discussed in detail in Section 2.5, was performed. The network selected for implementing our adapter is a Convolutional Neural Network (CNN) known as the Super Skinny Convolutional Neural Network (SSCNN), which was developed in [**1**]. SSCNN is a relatively simple network with a single convolutional layer, pooling layer, and a fully connected layer. Detailed insights into this architecture can be found in Section 3 below.

For the development of the adapter architecture, Rebuffi et al.'s Residual Adapter [**15**, **35**] was used as the foundational framework with certain modifications. Given that the SSCNN network is not deep, residual blocks are omitted fom the adapter. Furthermore, batch normalization, and rescaling and shift operations are not utilized. However, the fundamental idea of introducing 1x1 filter banks into the convolutional layer remains consistent. Chen et al.'s adapter module [**37**], designed for deeper networks, is not employed as once again, the SSCNN network the adapters will be incorporated into do not possess significant depth. Additionally, the concept introduced by Rosenfeld's DAN adapter network [**36**], which involves using a control signal to activate or deactivate the adapter within the network, was used in the software design.

Similar to Rebuffi et al.'s residual adapters, the adapter modules developed in this paper are also in series and parallel configurations, which will be discussed in the following sections.

3.1.1 Series Adapter

The adapter architecture in series closely resembles the structure of the residual adapter series architecture. As depicted in Figure 3.2, employing the same equation as presented in Equation 2.3, this adapter is integrated within a convolutional layer, in a series configuration. It takes the output of the convolutional layer as its input and performs its own convolution using a 1x1 kernel. The resulting output from the adapter convolution and the original convolutional layer

is then combined through summation, resulting in the creation of the adapter convolution layer.

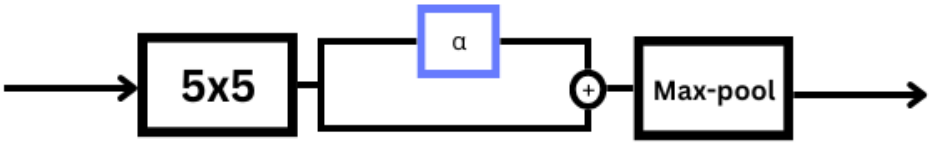


FIGURE 3.2. Architecture of Series Adapter

In the context of the SSCNN network, where there is only one convolutional layer, the adapter is positioned in series with that convolutional layer. As demonstrated in Figure 3.5, the SSCNN's convolutional layer produces six feature maps. The adapter needs to take this output as its input and perform 1x1 convolutions to generate an output of the same size, as they will be subsequently summed. This means the adapter must perform convolutions with input data that comprises multiple channels. When conducting a convolution on input data with multiple channels, the convolution kernel must have the same number of input channels as the input data to facilitate cross-correlation. In the case of an input of shape H x W x c_{in} where c_{in} > 1, a convolution kernel of shape k_h x k_w x c_{in} is required. Figure 3.3 below provides an illustration of this process.

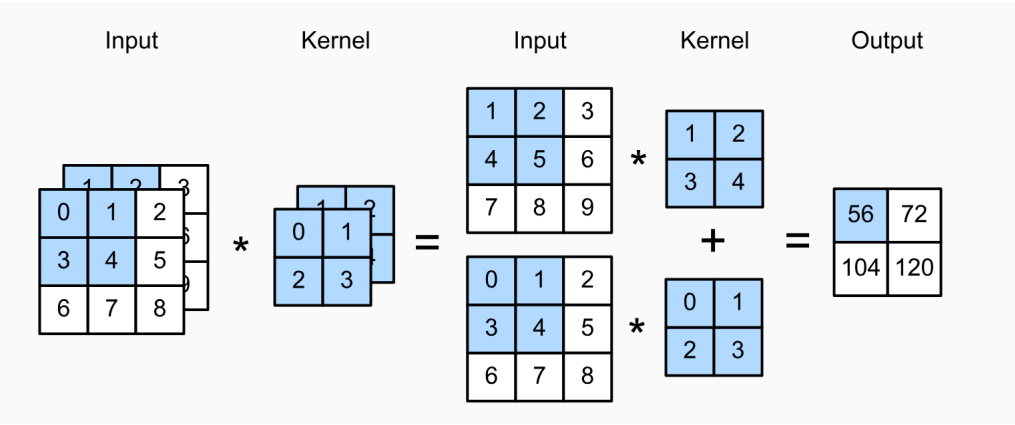


FIGURE 3.3. Cross-correlation computation with 2 input channels

This implies that 6 1x1 filter banks are needed to produce one output. However, as the output size of the adapter must match the size of the convolution output to enable summation, the above-mentioned convolution needs to be performed an additional 6 times. Thus, in total, 36 1x1 filters are required obtain an output with the same dimensions as the convolution layer output. This introduces 42 parameters (36 weights from the 1x1 filters and 6 biases for each channel output), which corresponds to only a 0.15% increase in the number of parameters. The output then passes through a ReLu activation function after the summation.

3.1.2 Parallel Adapter

The parallel adapter closely resembles the structure of the residual adapter architecture, as illustrated in Figure 3.4, utilizing the same equation as Equation 2.4. This adapter is integrated into a convolutional layer in parallel configuration. Unlike the series adapter, which takes the output of the convolutional layer as its input, the parallel adapter takes the input of that convolutional layer as its input and performs its own convolution using a 1x1 kernel. Similar to the series adapter, the output of the adapter convolution is combined with the output of the original convolutional layer, resulting in the creation of the adapter convolution layer.

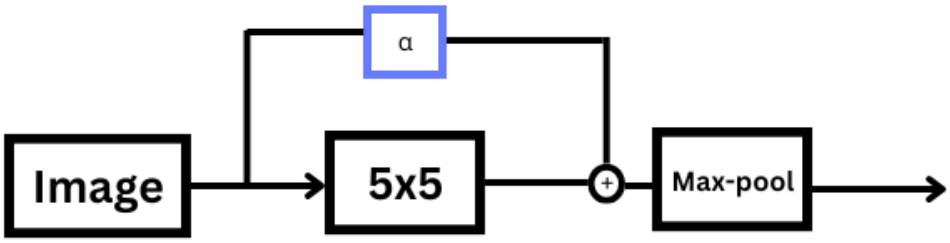


FIGURE 3.4. Architecture of Parallel Adapter

In the context of the SSCNN network, given that there is only one convolutional layer, the adapter can be placed in parallel at that position only. In this scenario, the input to the convolutional layer is the image. Therefore, the parallel adapter takes the image as its input for convolution. Since the image has only one channel, only 1 1x1 filter is needed to produce one output channel from the adapter. This is in contrast to the series adapter, which needed 6 due to the input having 6 channels. Since the convolutional layer generates six feature maps, the adapter adapter must also produce the same. Thus, the parallel adapter comprises 6 1x1 filters.

However, as previously mentioned, since the adapter's output and the convolutional layer's output are combined to create the adapter convolutional layer, their dimensions must match. With the 6 1x1 kernels, the same number of channels as the convolutional layer with the adapter can be generated. Nevertheless, as described in the following section, the input image is of dimension 28x28, which results in an adapter output size of 28x28. In contrast, the convolutional layer has an output size of 24x24. To ensure the outputs are compatible, the convolutional output is zero-padded to achieve dimensions of 28x28. This alteration also affects the size of the max-pooling layer, compared to the network with the series adapter. Hence, the network undergoes slight adjustments depending on the type of adapter used.

The parallel adapter introduces 12 additional parameters (6 weights and 6 biases), which amounts to a mere 0.03% increase in parameters.

3.1.3 Super Skinny Convolutional Neural Network (SSCNN)

The network that the adapter is being implemented into is a convolutional neural network called Super Skinny Convolutional Neural Network developed in [**1**]. As figure 3.5 shows, it has a single convolutional layer, pooling layer, and a fully connected layer. It takes as input a single channel 2D image of size $28x28$. The convolutional layer processes the image using six separate filters with a kernel size of $5x5$, resulting in the generation of six feature maps each of size $24x24$. These go through an activation function, namely ReLU. Subsequently, max-pooling is performed using $2x2$ filters, further reducing the size of the feature maps to $12x12$. The output from the max-pooling layer is then fed into a fully connected (FC) layer where a classification can be made on the input data. The hidden layer of the FC layer utilizes ReLU as its activation function while the output layer employs sigmoid. The SSCNN has in total 39542 parameters.

The choice of the Super Skinny Convolutional Neural Network (SSCNN) as the target network for implementing the adapter is based on several considerations. Firstly, one of the primary objectives of this project is to develop an adapter architecture for a small network, making the SSCNN a suitable candidate. Additionally, the project's second objective is to implement the adapter architecture in hardware. In this context, the SSCNN presents an advantageous choice because the developers of the network, as documented in [**1**], have already implemented it in hardware and provided their code and findings, including resource utilization, power consumption, and timing data. Therefore, the SSCNN offers a solid foundation for our hardware-based adapter implementation, given the available hardware-related information.

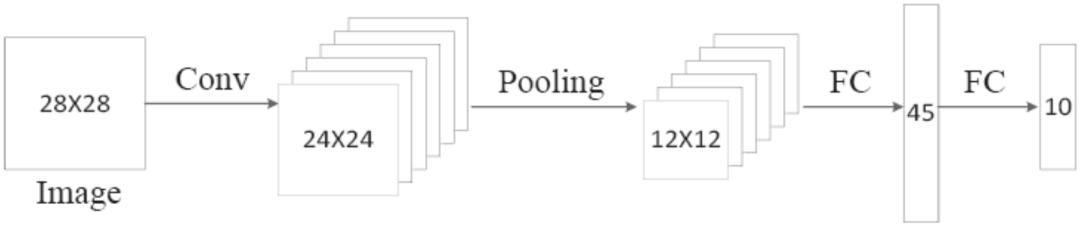


FIGURE 3.5. Architecture of Super Skinny Convolutional Neural Network

3.1.4 MNIST and Reduced MNIST Dataset

The dataset used for evaluating the adapter network is the well-known MNIST database, short for the Modified National Institute of Standards and Technology database. This dataset comprises a large collection of handwritten digits, commonly employed for training various image processing systems. It consists of 60,000 training images and 10,000 testing images. To simulate multi-domain adaptation, the original MNIST dataset is modified by removing the digit 9, resulting in the creation of a new dataset named Reduced MNIST. Initially, the original network, without the adapter, is trained using the Reduced MNIST dataset.

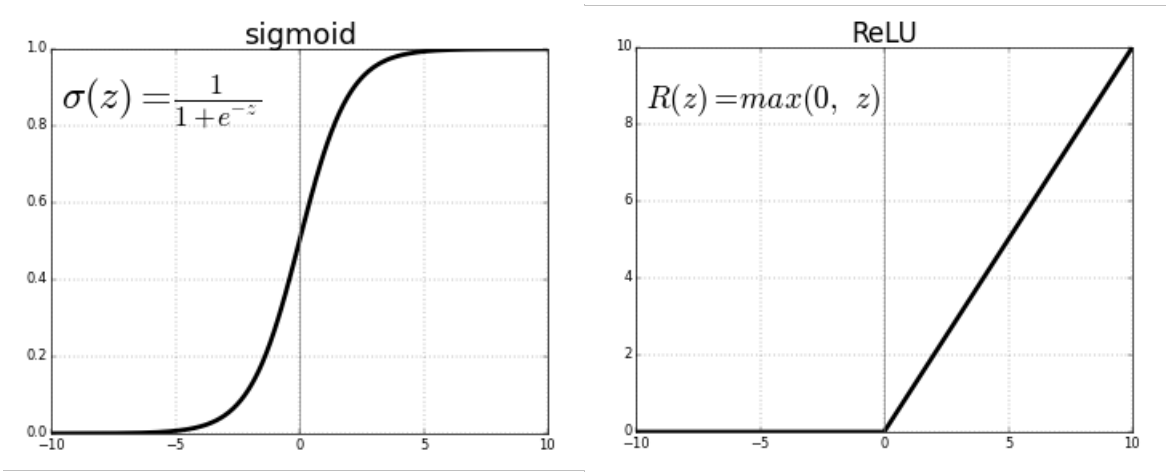


FIGURE 3.6. Activation functions used in Super Skinny Convolutional Neural Network

After the network has been trained on the Reduced MNIST, the network is fine-tuned using the adapter modules on the original MNIST dataset, which has the digit 9. This experimental setup aims to test the network's ability to adapt and learn a new digit when it has already been trained on a subset of digits, demonstrating its multi-domain learning capabilities.

FIGURE 3.7. Sample images from MNIST dataset

3.2 Software Design

The adapter network is first designed and tested in software to evaluate its performance. Initially, PyTorch is employed as a benchmark due to its capabilities in building deep learning models and its extensive features, having been developed by researchers at Facebook AI Research and other leading labs. Following this, the adapter network was designed manually, including the inference and backpropagation process, in the C programming langauge. It was also tested. The upcoming sections provide detailed insights into the methodologies adopted for both the benchmark and the C simulator

3.2.1 PyTorch Benchmark

The development of the adapter architecture and the network was carried out using Torch and it's extension Torchvision. Torch is a machine learning library in Python that is well-suited for building neural networks.

The chosen network is the SSCNN which consists of a single convolutional layer, one max-pooling layer, and a fully connected layer. The adapter modules, represented as 1x1 filter banks and placed either in series or in parallel, have also been initialized using the *nn* module. Similar to the DAN network introduced by Rosenfeld et al. [**36**], the network uses a control signal to switch between the original network configuration and the network with the adapter enabled.

```
self.conv1sa = nn.Conv2d(
        #1x1 serial
        in_channels=self.outc,
        out_channels= self.outc,
        kernel_size=1,
        stride=1,
        padding=0,
        bias=True,
    )

self.conv1pa = nn.Conv2d(
        #1x1 parallel
        in_channels=1,
        out_channels= self.outc,
        kernel_size=1,
        stride=1,
        padding=0,
        bias=True,
    )
```

FIGURE 3.8. PyTorch implementation of series and parallel adapter

Training the network is facilitated by the *nn* module, which provides convenient functions for this purpose. The *nn.train()* function prepares the network for training, while *nn(input data)* conducts inference on the provided input data. The output of the inference and the corresponding label are then used to compute the loss, with cross-entropy loss function being employed. Following the loss calculation, *loss.backwards()* is utilized to propagate the loss backward through the network and compute gradients. The network's parameters are updated with *optimizer.step()*, and a learning rate of 0.01 is used. Once the network has been trained on the training data, it is tested on the test data for evaluation.

The initial step involves pre-training the network using the Reduced MNIST dataset, which consists of the digits 0 to 8. The layers preceding the adapter are then frozen, which is the convolutional layer of the network. Subsequently, the control signal is activated for either the series or parallel adapter, and the network with the adapter is trained on the original MNIST dataset, digits from 0 to 9. This training process solely updates the parameters from the adapter layer onwards, while the convolutional layer remains frozen. This approach allows for

the evaluation of the adapter's ability to adapt the network and learn a new digit, exemplifying the concept of multi-domain learning.

3.2.2 Custom C Simulator

After implementing the adapter network in PyTorch, the manual development of the network and the backpropagation process in C was undertaken. This approach was chosen to ensure that all aspects of the network and the training process, including the adapter, were coded from scratch. The goal was to design the adapter with backpropagation in hardware, and manual implementation allowed for a fair and equal comparison with hardware results during testing.

In contrast to the PyTorch model, which provides abstractions for designing the SSCNN and the adapter architectures, the C implementation required the manual coding of the entire network. This also applies to the adapter architectures. Similar to Rosenfeld et al.'s DAN network [**36**], the network incorporates a control signal to toggle between the original network configuration and the network with the adapter activated.

The C Simulator involved developing the computation for the inference and training portion of the network. The inference process involves performing convolution using the filter maps from the convolutional layer, executing max-pooling, and subsequently passing the output through a fully-connected layer. These operations are carried out using for loops and element-wise operations. During the backpropagation process, the equations presented in Section 2.3.5 are employed to develop the code. The output error is computed using the cross-entropy loss function, and this error is propagated backward through the network to calculate the gradients. Finally, the network parameters are updated with a learning rate of 0.01.

To clarify the training process, Figure 3.9 illustrates the workflow of the C Simulator for a single training instance.

As mentioned in the PyTorch section, the same procedure is undertaken to pre-train the network on the Reduced MNIST dataset and adapt it to the original MNIST networ.

The C Simulator serves another crucial purpose beyond testing correctness. It provides the data that the hardware adapter modules will use as inputs to ensure the correctness of the hardware adapters. During training in the C Simulator, the data required by the adapter is saved into memory files. These memory files can be read into the Vivado testbench and then fed into the hardware adapter modules during testing. Additionally, the expected output of the adapter for both forward and backward propagation is saved into memory files. This enables a comparison between the hardware adapter's output and the expected output. A detailed explanation of the interaction between the C Simulator and the hardware adapters can be found in the following section.

As the adapter is being implemented in hardware, consideration is given to the numbering system used. The network uses IEEE floating point and custom floating-point representations with various bit widths, including single precision, half precision, custom 12-bit, and custom 8-bit. For single precision, the adapter network is simulated in C using the data type "float." For the other bit widths, a custom floating-point library found in [**38**] is utilized. The results

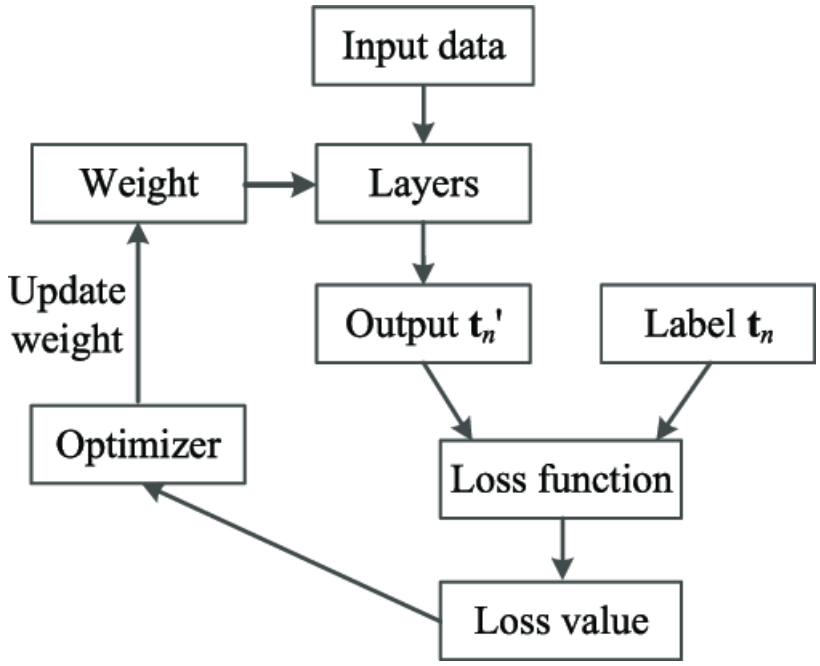


FIGURE 3.9. Training process of the SSCNN

of the adapter network with each of these floating-point bit widths in the C simulator are presented in the results section.

3.3 Hardware Design

This section delves into the design and FPGA implementation of the series and parallel adapters in hardware. Recall, our adapter is integrated into a convolutional neural network known as SSCNN, and it is tested using the MNIST and Reduced MNIST datasets for multi-domain learning. In the software domain, we implemented both the adapter and the network. However, in the hardware domain, we focus solely on designing the adapter as a standalone module. This adapter module takes inputs as if it were part of the SSCNN network and produces outputs as it would if integrated into the network

3.3.1 RTL Design

Figure 3.2 illustrates the architecture of the series adapter when integrated into the SSCNN network. For the inference phase of the network, the series adapter takes the output feature maps (6 in total) from the convolutional layer of size 24x24 and performs cross-correlation convolution using 1x1 filter banks. It then adds its output to the output of the convolutional layer, resulting in a convolutional adapter layer. In the hardware implementation, the inputs to the series adapter for the inference phase consist of the convolutional layer's output feature maps and the 1x1 filter banks (weights and biases). The series adapter performs cross-correlation convolution between the provided feature maps and filter banks and produces the convolutional adapter layer as output.

Meanwhile, Figure 3.4 illustrates the architecture of the parallel adapter when integrated into the SSCNN network. For the inference phase of the network, the parallel adapter takes the convolutional layers input which is the single channel input image of size 28x28. Since it is a single channel, cross-correlation is not performed and the 1x1 filter banks can perform normal convolution with the image to produce the output of the adapter which is then added to the convolutional layers output to produce the convolutional adapter layer. In the hardware implementation, the inputs to the parallel adapter also requries the input image for that training instance.

Online training is a crucial part of the adapter's functionality. During the training of the network with the adapter, error gradients are propagated backward through the network. The adapter convolutional layer receives error gradients from the max-pooling layer, following the equations described in Section 2.3.5 to compute the error gradients at the convolutional adapter layer. The calculated error is also used to update the weights and biases of the 1x1 filter. Therefore, in the hardware implementation, the series and parallel adapter modules also requires, as inputs, the error from the max-pooling layer. They perform backpropagation and generates the error at that layer for further backward propagation, along with the changes in weights and biases for that specific training instance.

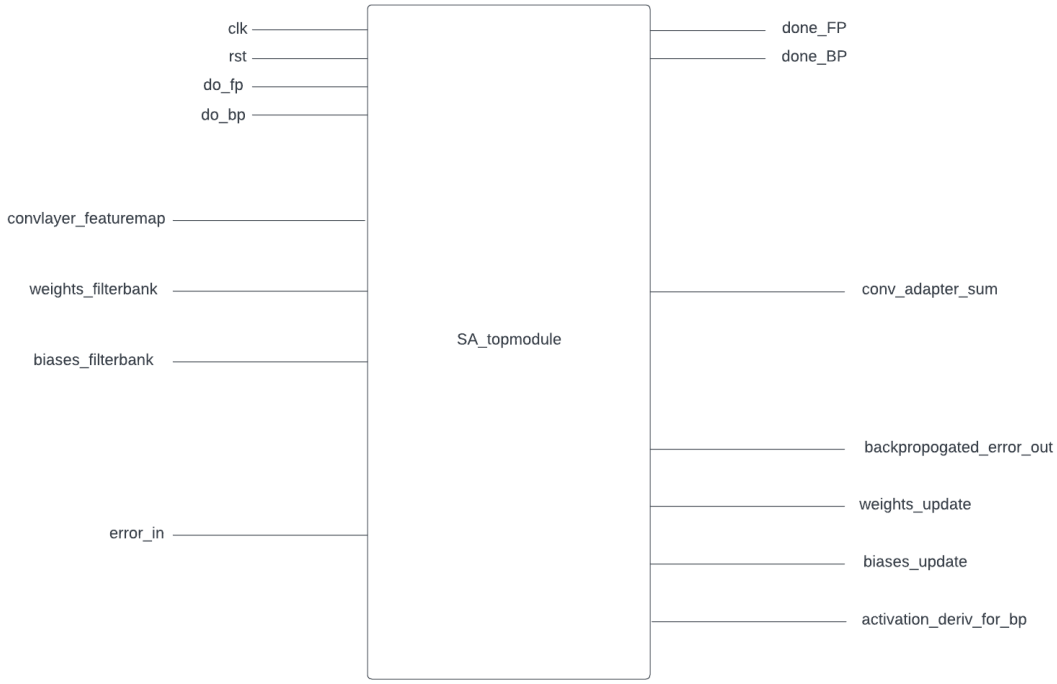


FIGURE 3.10. RTL Block Diagram of serial adapter

The dataflow within the adapter modules is controlled by a finite state machine (FSM), which is defined by the state transition diagrams in Figures 3.13 and 3.14 for the series and parallel adapter modules, respectively. The FSM is responsible for managing the flow of operations during both forward propagation (inference) and backpropagation (training). It has different states to handle these two phases.

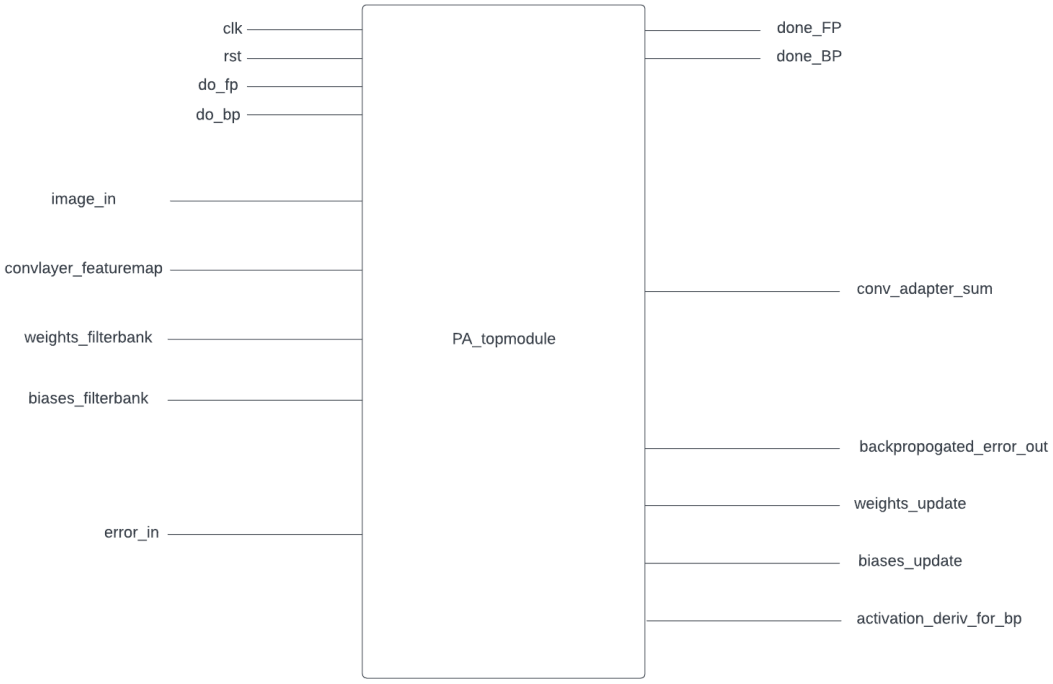


FIGURE 3.11. RTL Block Diagram of parallel adapter

In the idle state, the FSM waits for a start signal, which triggers the beginning of the forward propagation process. During this phase, the adapter module computes the convolutional adapter layer output. Once the forward propagation computations are complete, the FSM transitions to the backpropagation state. In this state, the adapter module calculates the error for the convolutional adapter layer, which can be propagated backward through the network. Additionally, it computes the changes in the adapter filter bank parameters, including weights and biases.

Since the convolutional layer produces 6 feature maps, the adapter module is also designed to output 6 channels, as these channels are summed together to create the convolutional adapter layer. During backpropagation, there are 6 sets of errors that need to be calculated, and 6 sets of filter bank parameters that need to be updated. In the hardware implementation, there are two approaches to handle this: sequentially and in parallel.

The sequential approach involves looping the FSM over 6 times for each training instance. In this scenario, the adapter module processes one channel at a time. Although this method takes more clock cycles to complete the backpropagation for all 6 channels, it has lower hardware usage. On the other hand, the parallel approach handles all 6 channels simultaneously but requires more hardware resources. For this project, the sequential version was used to conserve hardware resources, despite taking longer in terms of clock cycles.

3.3.2 Neuron Module

Central to the adapter network is the neuron module, a critical component for both forward and backward propagation processes. Figure 2.3 provides a visual representation of a perceptron and the underlying calculations it carries out. These calculations are outlined in Equation 2.1, which involves the multiplication of a weight by an input value, followed by the addition of a bias term. This operation is performed iteratively, typically for each neuron connected to the module.

In hardware implementations, to efficiently execute these repetitive multiply-accumulate operations, specialized hardware units known as Multiply-Accumulate Units (MACs) are employed. MAC units are designed to handle the intricate computations required in neural networks, making them a crucial element in the hardware realization of such networks.

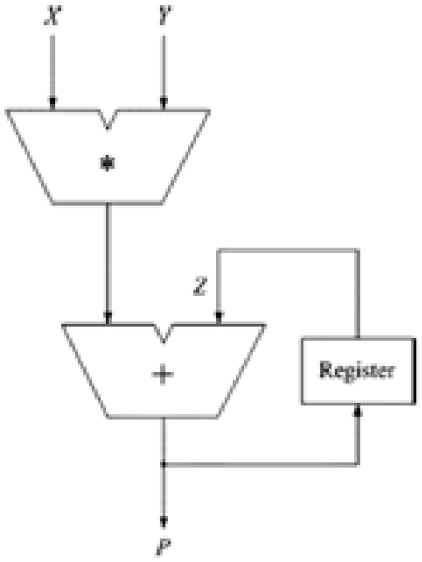


FIGURE 3.12. Multiply-Accumulate Unit

Figure 3.16 presents a block diagram illustrating the MAC (Multiply-Accumulate) unit. The inputs, denoted as X and Y, represent the weight and input data. Prior to entering the adder, a bias is incorporated into the product. This is then added with previous calculations which are stored in a register.

3.3.3 Series Adapter

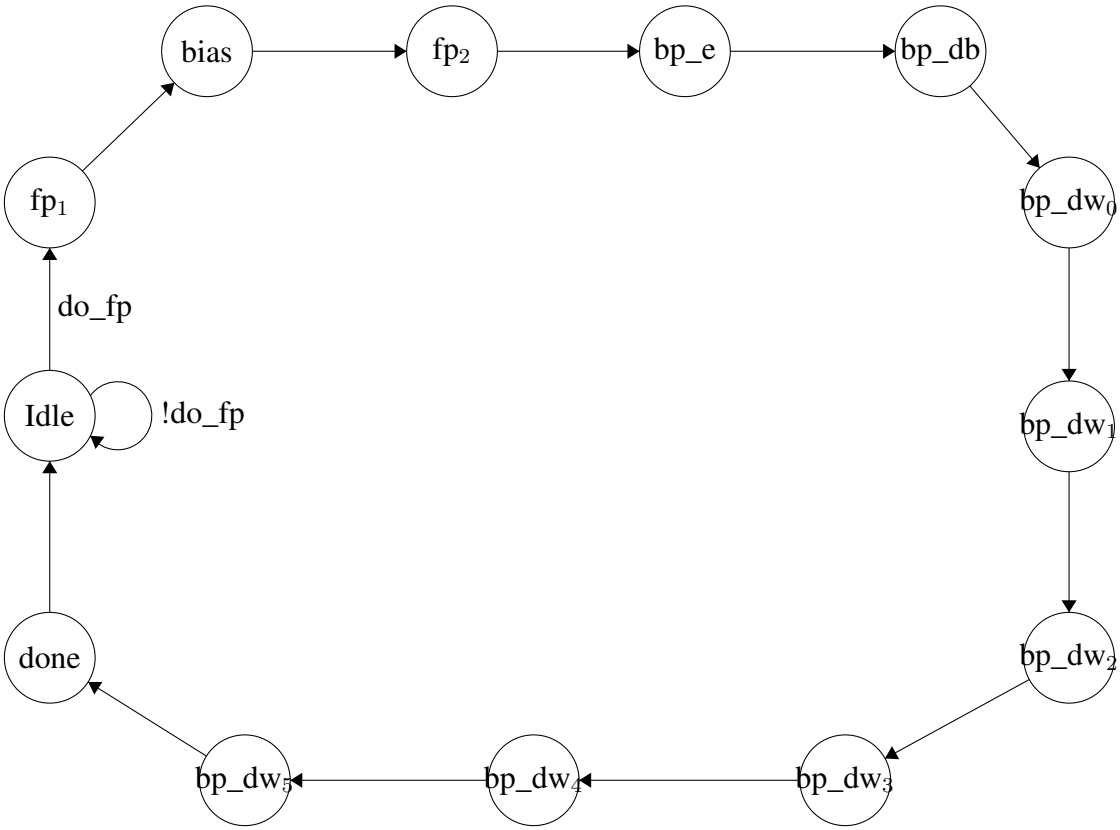


FIGURE 3.13. Serial Adapter finite state machine (FSM) with sequential feature channel

Figure 3.13 displays the finite state machine (FSM) which controls the series adapter's dataflow. The FSM manages both inference and backpropogation of the adapter module.

In the idle state, the FSM waits for the "do_fp" signal, which begins the series adapter's forward propagation. The "fp_1" state represents the phase where the series adapter executes a 1x1 convolution on the input data using the adapter's 1x1 filter banks. As discussed earlier in Section 3.1.1, the series adapter takes the 6 output feature maps from the convolutional layer as input to conduct its own 1x1 convolution.

Because the input data involves multiple channels, cross-correlation is required. The "fp_1" state operates over 6 clock cycles to accomplish its task. Each clock cycle involves multiplying an input feature map by a single 1x1 filter and accumulating the result at the corresponding neuron. After these 6 clock cycles, cross-correlation for a single output channel reaches completion, and the FSM transitions to the "fp_2" state.

In "fp_2," a single clock cycle is required to add a bias to the result of the 1x1 convolution. This step concludes the computation for one output channel of the convolutional adapter layer. With feature maps measuring 24x24, and the adapter performing 1x1 convolutions, this results in 24x24 hardware neurons. Additionally, 24x24 adders are essential for applying bias, and an additional set of 24x24 adders handles the complete convolutional+adapter operation.

Subsequently, the FSM progresses to the initial stage of backpropagation, focusing on error calculation at the convolutional adapter layer. This process, governed by equation the equation found in 2.3.5, involves multiplying the incoming error by the derivative of the activation function corresponding to that layer's accumulation value. It takes a single clock cycle. This operation results in the generation of 24x24 ReLU derivative modules and requires 24x24 multipliers.

Upon the completion of error calculation for all neurons, the FSM advances to its second stage, aimed at computing the changes in bias for the 1x1 filter bank. This state consumes a total of 576 clock cycles (24x24). According to the equation found in 23.5, the change in bias equals the sum of the errors across all neurons at that layer. This particular stage requires the use of 24x24 multipliers and a single adder.

The following six stages are dedicated to calculating the weight changes. Given the cross-correlation with six input channels, 6 weights are needed to be updated. Each of these states takes 24x24 cycles to complete, following the principles of the equation found in 2.3.5. These states generate 24x24 multipliers and also a single adder. To conserve resources, the hardware modules are shared among these states.

For each run through the FSM, the hardware conducts inference and backpropagation calculations for one channel. To cover all six channels, the FSM iteration is repeated six times to address a single training instance effectively. This approach ensures resource efficiency while handling multiple channels.

3.3.4 Parallel Adapter

Figure 3.14 illustrates the finite state machine (FSM) responsible for governing the dataflow within the parallel adapter module. This FSM manages both the inference and backpropagation processes.

The primary distinction between the series and parallel adapter modules lies in the "fp_1" state and the weight change state. The parallel adapter takes the input image as its primary input. Since the image is a single channel, there is no need for cross-correlation, and only one 1x1 filter bank is necessary. Consequently, the "fp_1" state in the parallel adapter only requires a single clock cycle, as it performs the multiplication of the filter bank with the input image. Since only 1 weight is necessary, this also means just one weight needs to be updated during backpropagation, significantly reducing the number of clock cycles.

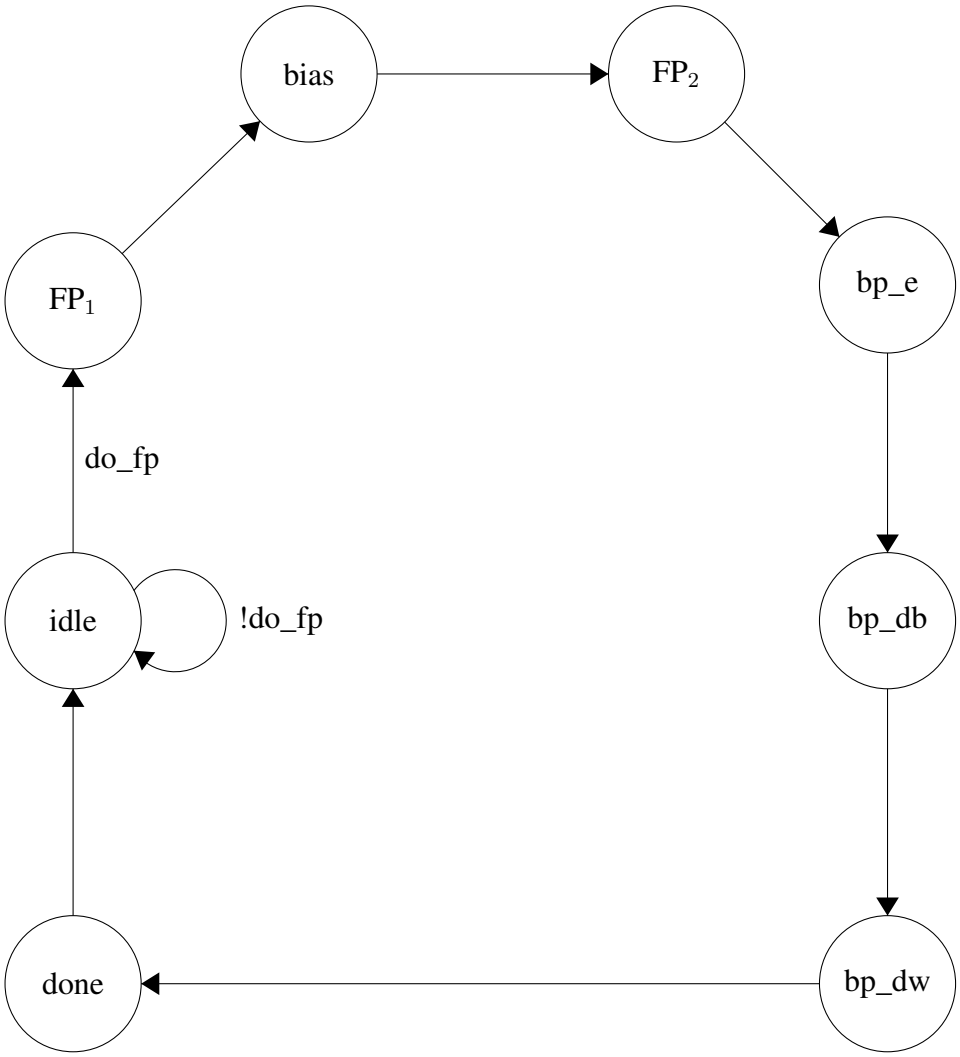


FIGURE 3.14. Parallel Adapter finite state machine (FSM) with sequential feature channel

However, the parallel adapter generates a larger number of hardware modules, including neurons, multipliers, and adders. This is due to the size of the input image, which measures 28x28. To ensure that the convolutional layer can be combined with the adapter layer at the convolutional adapter layer, 28x28 1x1 convolutions must be performed. Also, the bias and weight update states in the parallel adapter take 784 cycles (28x28) to accommodate the size of the input image.

3.3.5 Floating Point IP Cores

Up to this point, we've mentioned the use of multipliers and adders in the network. However, as previously noted, the precision of the network includes IEEE single precision floating point and custom floating point with bit widths of 16, 12, and 8. Since HDL lacks built-in floating-point operations, we employ a custom floating-point core. For this purpose, we

utilize Flopoco, a versatile tool for generating arithmetic cores, including Floating-Point Cores, designed for FPGAs [**39**]. These cores are highly parameterized in precision and offer precision accuracy down to the last bit.

Flopoco supports two primary floating-point formats: IEEE floating-point formats and a simple floating-point format. The simple format includes additional encoding for exceptional cases (zero, infinity, NAN) in two extra bits, as illustrated in Figure 3.17. A comparison between these two formats can be seen in Figure 3.18

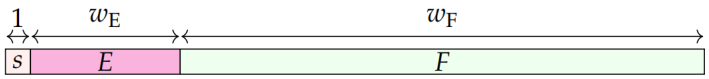


FIGURE 3.15. Bit fields of an IEEE754-like floating-point number of parameters (wE, wF)

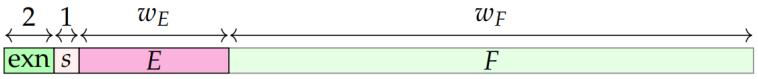


FIGURE 3.16. Bit fields of a custom floating-point number of parameters (wE, wF)

exn	meaning
00	zero (there are two zeroes, noted $+0$ and -0, according to s)
01	normal numbers
10	infinity ($+\infty$ or $-\infty$, according to s)
11	NaN (Not a Number)

FIGURE 3.17. Encoding of exceptional cases in the FloPoCo floating-point format

	IEEEfloat(w_E, w_F)	Nfloat(w_E, w_F)
bias value	$E_0 = 2^{w_E-1} - 1$	
Total size	$w_E + w_F + 1$ bits	$w_E + w_F + 3$ bits
$e_{\min}$	$-2^{w_E-1} + 2$	$-2^{w_E-1} + 1$
$e_{\max}$	$2^{w_E-1} - 1$	2^{w_E-1}
Smallest	$2^{e_{\min}-w_F} = 2^{-2^{w_E-1}+2-w_F}$	$2^{e_{\min}} = 2^{-2^{w_E-1}+1}$
Largest	$(2 - 2^{-w_F}) \cdot 2^{e_{\max}}$	$(2 - 2^{-w_F}) \cdot 2^{e_{\max}}$

FIGURE 3.18. Comparison between IEEE floating-point format and custom floating point format

CHAPTER 4

Results

This section outlines the results obtained during the course of the project, divided into software and hardware outcomes.

In the software results, the adapter network developed in PyTorch and the custom C simulator were evaluated. The process involved pre-training the original SSCNN network on the Reduced MNIST dataset (which lacks the number 9). Then, the adapter was activated, and the network with the adapter was trained on the full MNIST dataset, which includes the digit 9. During this process, the convolutional layer of the original network remained frozen to retain knowledge from the Reduced MNIST dataset. The performance was assessed using single precision floating-point numbers in PyTorch. In the custom C simulator, tests were conducted using various bit widths, including single and half precision, as well as custom 12-bit and 10-bit floating-point formats.

In the hardware results, With the series and parallel adapter modules designed in hardware, verifying their correctness through simulation before performing synthesis and implementation is crucial. Vivado simulator was used to perform the simulations. Testbenches were developed for both the series and parallel adapters. For each training instance, the C simulator saved the data required by the hardware adapter modules as input into memory files. The testbenches then read these memory files and provided the inputs to the adapters. AFter each simulation, the outputs of the adapters are stored in text files and compared with the outputs from the C simulator files for the same training instance. If they matched, it indicated that the adapter modules were functioning correctly. These correctness tests were performed for the 32-bit single precision adapter modules. There were no correctness tests performed on the other bit width adapter modules as the focus was more on implementation on a board and testing one (32-bit single precision) and ensuring correctness was sufficient.

After the correctness tests for the 32-bit single-precision adapter modules, the hardware adapter modules are synthesized and implemented on a target board, the Zynq UltraScale+ RFSoC ZCU111 Evaluation board [**40**] using Vivado. Resource utilization, power consumption, and timing tests are conducted on these modules. This evaluation is carried out for adapter modules in four different bit widths: 32-bit single-precision floating point, 16-bit half-precision floating point, custom 12-bit floating point, and custom 10-bit floating point.

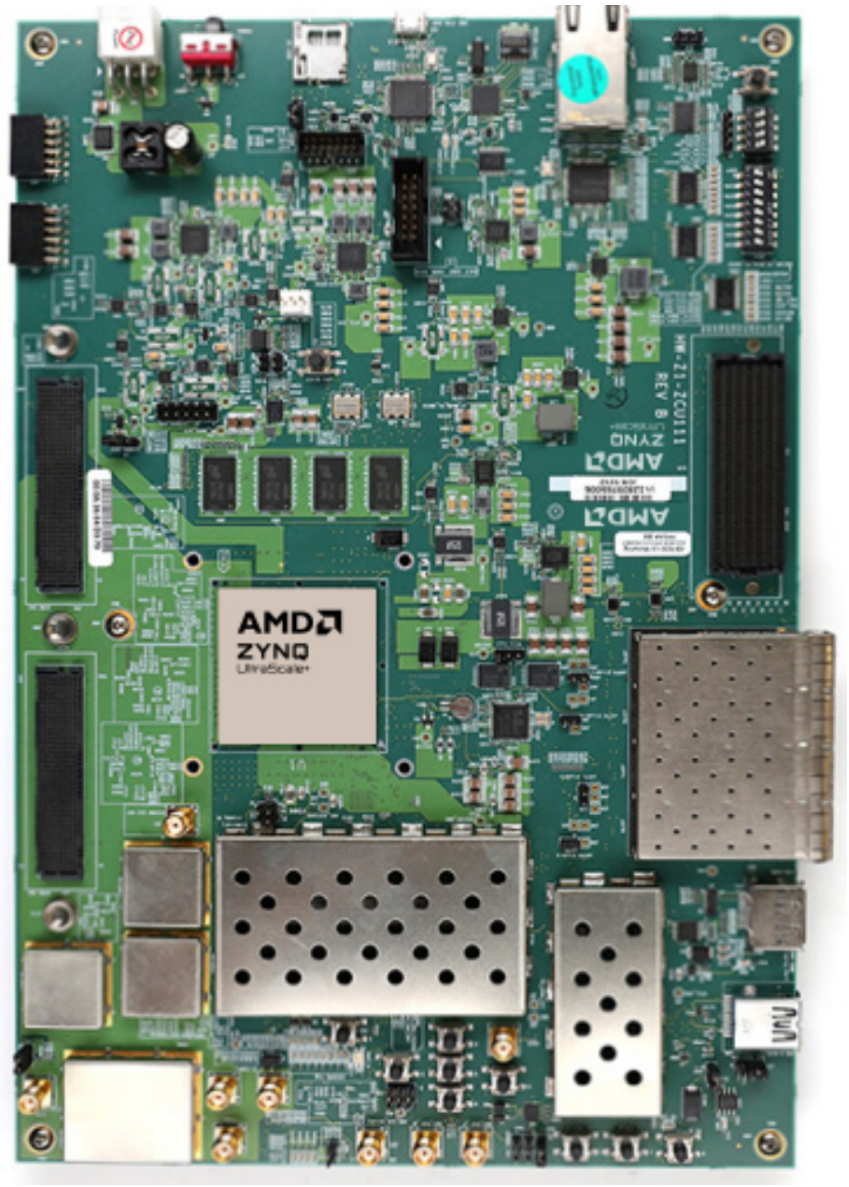


FIGURE 4.1. Zynq UltraScale+ RFSoC ZCU111 Evaluation Board

4.1 Software Resuts

Table 4.1 summarizes the PyTorch simulation results for the SSCNN adapter network. The initial table provides the accuracy score for the original SSCNN network trained on the Reduced MNIST dataset, achieving a base performance of 98.35% accuracy. The network underwent training for 15 epochs with a batch size of 100.

For the subsequent experiments, where the pretrained SSCNN network on Reduced MNIST was further fine-tuned, both full finetuning and fine-tuning with the series and parallel adapters were applied to a new dataset, the Original MNIST. All three methods maintained high levels of accuracy on the Original MNIST dataset. Full finetuning achieved a slightly higher accuracy

of 98.27%, but it required the retraining of the entire network's parameters. In contrast, the series and parallel adapters only introduced a small number of additional parameters for training.

As observed in the literature [**35**, **37**], the parallel adapter slightly outperformed the series adapter, however, only by a small margin. The experiment was conducted using 32-bit single precision floating point.

Model	Reduced MNIST
SSCNN	98.35

Model	Original MNIST
Finetune	98.27
Series Adapter	97.9
Parallel Adapter	98.2

TABLE 4.1. PyTorch multiple-domain learning. The first table reports the accuracy (%) score for the original SSCNN network trained on Reduced MNIST (number 9 removed). The second figure reports the accuracy (%) score for the SSCNNN network pre-trained on Reduced MNIST adapted to the Original MNIST.

After the PyTorch simulation, similar tests were conducted using the Custom C simulator, which involved building the adapter network from scratch, including the inference and back-propagation algorithm. These tests covered various bit-width floating-point representations, including 32-bit single precision and 16-bit half precision. Table 4.2 illustrates the accuracy score for the original SSCNN network trained on the Reduced MNIST dataset. These tests also spanned 15 epochs, although no batch size was utilized. The results from the Custom C simulator yielded slightly lower accuracy compared to the PyTorch results, but they still maintained high accuracy for the dataset. Furthermore, the reduction in floating-point precision led to a decrease in accuracy, which was expected, as a decrease in precision leads to less accuracy when performing backpropagation.

Model	Reduced MNIST
SSCNN Single Precision	97.43
SSCNN Half Precision	96.05

TABLE 4.2. Custom C Simulator multiple-domain learning. The figure reports the accuracy (%) score for the original SSCNN network trained on Reduced MNIST (number 9 removed)

Model	Reduced MNIST
Single Precision	
Series Adapter	96.17
Parallel Adapter	96.60

TABLE 4.3. Custom C Simulator multiple-domain learning. The figure reports the accuracy (%) score for the SSCNNN network pre-trained on Reduced MNIST trained adapted to the Original MNIST

Like the PyTorch, for the pre-trained SSCNN network on Reduced MNIST, fine-tuning with the series and parallel adapters was conducted on the original MNIST dataset. These experiments were performed for the 32-bit single-precision network model. Similar to the PyTorch results, fine-tuning with the adapters achieved a high level of accuracy, specifically 96.17% and 96.60%, on the new dataset. The parallel adapter once again outperformed the series adapter.

Both the PyTorch and Custom C simulator results highlight the adapter modules' ability to perform multiple-domain learning and adapt a pre-trained network (SSCNN in this case) to a new domain. Importantly, this was demonstrated on a relatively small network, the SSCNN, which comprises a single convolutional, max-pooling, and fully connected layer, and small datasets—the original and Reduced MNIST. The success of these experiments indicates that the concept of multiple-domain learning and adapter modules can be applied to not only large networks and datasets but also smaller ones. These promising results provide a green light to proceed with the design and implementation of the adapter modules in hardware, given their proven functionality.

4.2 Hardware Results

From Figures 3.11 and 3.10, the adapter modules have five outputs. One of these outputs, the convolutional adapter layer is the result of the 1x1 convolution of the input with the adapter. This makes up the forward propogation logic of the adapter module. The other outputs are

from the backpropogation logic of the adapter modules. These are the error at the adapter layer to be backpropogated, and the change in weights and biases for the adapter filter banks.

During simulation, the adapter modules are tested for individual training instances. The inputs are sourced from memory files generated by the Custom C simulator and the outputs of the adapter module simulation are stored into txt files. These outputs are then compared with the expected output from the custom C simulator for the same training instance. This process is performed using a diff command between the expected output and hardware simulation outputs. To ensure that the adapter modules work for multiple instances, the above process was performed for 4 consecutive training instances.

Appendices A and C provide a detailed comparison between the first output, namely the convolutional adapter layer output, obtained from the C simulator and the hardware simulation. Given the substantial size of the convolutional adapter layer output (6x24x24 for the series adapter and 6x28x28 for the parallel adapter), only a section of the outputs is shown for visualization. The comparison reveals that the outputs from the C simulator and the hardware adapter are for the most part similar, affirming the correct functionality of the hardware adapter forward propogation logic. However, there are slight variations in some outputs, with the last fractional number differing mostly by 1. This difference can be attributed to minor differences in how the custom floating-point IP core from flopoco, utilized in the adapter modules, and the C programming language round floating-point numbers. Despite these negligible differences, it is evident that the adapter is performing as anticipated.

The diff comparisons presented in Appendices B and F thoroughly assess the outputs from both the C simulator and the hardware simulation for the error, change in weight, and change in bias, encompassing both the series and parallel adapters. These comparisons exhibit identical results, affirming that the backpropagation logic of the adapter modules is functioning as intended.

Following the correctness test, the adapter modules were synthesised and implemented on the test board, the ZCU111 Evaluation Board. The board has a large number of logic blocks including LUTs and FFs. It also has moderately high amount of DSP blocks. A snapshot of the resources available on the board can be seen in Figure 4.4. Figures 4.2, 4.3 and 4.4 show the LUT, FF and DSP resource usage of the adapter modules for 4 different bit-widths.

LUT	FF	BUF	DSP	IO
425280	850560	696	4272	347

TABLE 4.4. Available resources on ZCU111 Evaluation Board

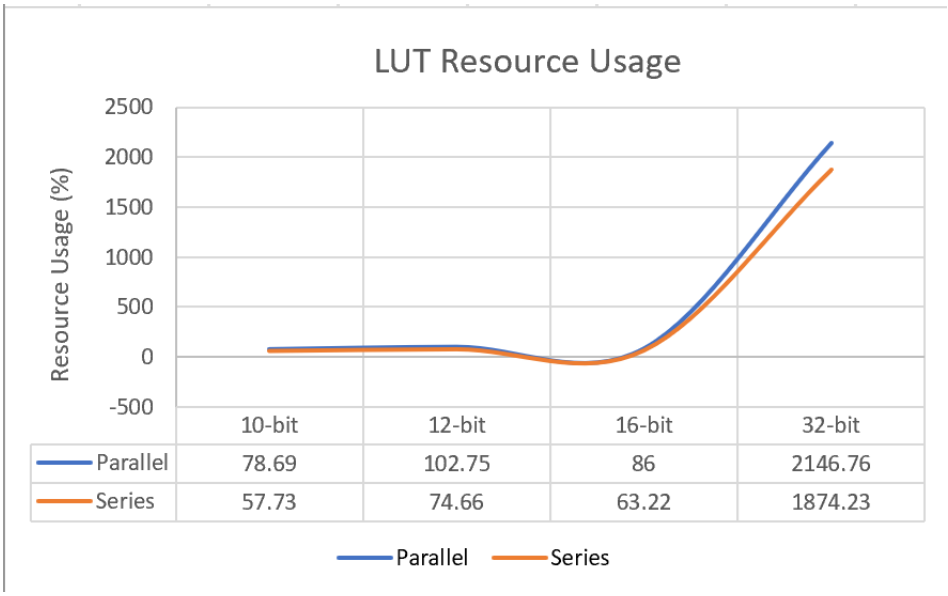


FIGURE 4.2. LUT usage of Series and Parallel Adapter on ZCU111 Evaluation Board. For 12-bit parallel adapter, and for 32-bit series and parallel adapter, results are from synthesis only as these modules do not fit on board for implementation

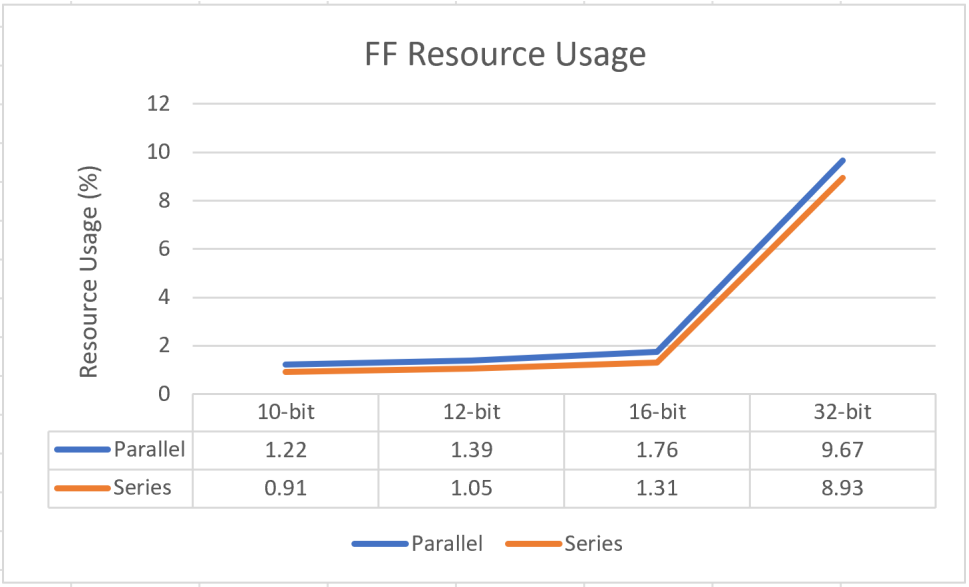


FIGURE 4.3. FF usage of Series and Parallel Adapter on ZCU111 Evaluation Board. For 12-bit parallel adapter, and for 32-bit series and parallel adapter, results are from synthesis only as these modules do not fit on board for implementation

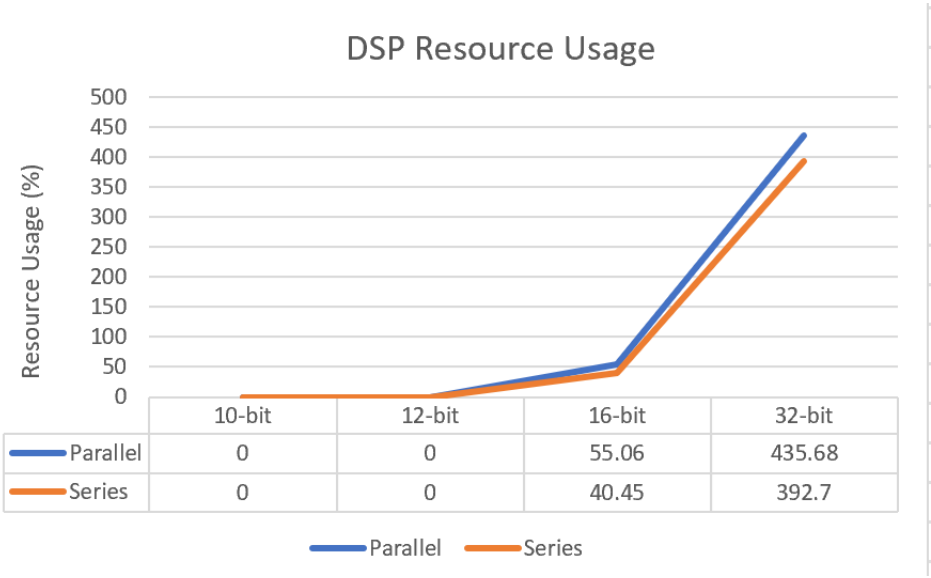


FIGURE 4.4. DSP usage of Series and Parallel Adapter on ZCU111 Evaluation Board. For 12-bit parallel adapter, and for 32-bit series and parallel adapter, results are from synthesis only as these modules do not fit on board for implementation

Looking at the figures above, the assessment of the 32-bit width adapter modules' resource utilization reveals that they are impractical for FPGA implementation. Both the series and parallel adapter modules far exceed the available resources of the FPGA board. For instance, the series adapter uses 2147% of the available LUTs and 435% of the available DSPs, while the parallel adapter uses 1874% of the available LUTs and 393% of the available DSPs. These percentages substantially surpass the FPGA's capacity, making the implementation unfeasible.

Interestingly, even though the adapters' LUT and DSP usage is excessive, they utilize efficient usage of flip-flops (FFs), with only 9.7% and 8.9% utilized for the 32-bit width series and parallel adapters, respectively. This can be attributed to the prevalence of combinatorial logic and lack of sequential the design, as well as the absence of pipelining in the modules. The impracticality of the 32-bit width adapters can be explained by the resource-intensive nature of performing floating-point arithmetic in hardware, as floating-point IP cores for single precision consume a significant amount of hardware resources. Given these findings, it is evident that the 32-bit width single precision adapter modules are not a viable design for FPGA implementation.

The resource utilization trends in relation to bit-width indicate that as the adapter modules use smaller bit-widths, the hardware resources on the target board decrease accordingly. This observation aligns with expectations, as smaller bit-width floating-point numbers inherently require less hardware for arithmetic operations. Specifically, the 10-bit width custom floating-point adapter modules demonstrate the most efficient resource usage, utilizing only 78% and 57% of LUTs, 1.22% and 0.91% of flip-flops (FFs), and no DSPs for the parallel and series adapters, respectively. This represents a significant reduction in resource utilization compared to the original 32-bit single precision floating-point adapter modules.

An interesting point to note is that the 16-bit half precision floating-point adapter modules consume fewer LUTs than the 12-bit custom floating point and a similar number to the 10-bit adapters. This anomaly is explained by the fact that the 16-bit and 32-bit networks utilize digital signal processors (DSPs), as depicted in Figure 4.4. For the higher bit-widths, flopoco's floating-point IP cores employ both LUTs and DSPs due to the increased complexity of the computations. Consequently, the number of LUTs decreases because the arithmetic operations rely on DSPs as well.

Another significant trend in resource utilization is that, across all bit-widths, the series adapter modules consistently uses fewer hardware resources compared to their parallel counterparts. This can be attributed to the architectural differences of the adapter modules within the SSCNN network. As explained in section 3, the series adapter takes the convolutional layers' output as input, which is 24x24 for 1 channel. In contrast, the parallel adapter takes the input image which has a size of 28x28. Since the adapters perform 1x1 convolution, resulting in the same number of neurons as the input, the parallel adapter generates more neurons in hardware, necessitating more resources to perform the required computations.

The results suggest several key conclusions. Firstly, it is evident that employing lower bit-width adapter modules can lead to substantial resource savings on the target FPGA. However, it is important to ensure that the chosen bit-width can still support online training. Recent work, such as the paper "Single-Batch CNN Training using Block Minifloats on FPGAs" [**41**], has introduced innovative 8-bit block minifloat formats capable of facilitating online training for CNNs on FPGAs. Thus, combining the adapter modules with lower bit-widths such as the above holds significant promise for adapter modules capable of multi-domain adaptation in hardware.

Additionally, different configurations of the adapter modules can result in varying levels of hardware utilization. In this case, the series adapter configuration consistently requires fewer hardware resources than the parallel adapter. This suggests that exploring alternative configurations may result in even greater reductions in hardware resource consumption, providing flexibility and efficiency in adapting neural networks for diverse tasks.

One main goal of this project was to determine the resource cost when implementing and integrating an adapter module with online training into an existing hardware network. With the above methods and additional optimization techniques, the required additional resources for an adapter module decrease. As this reduction takes place, hardware-based adapter modules for networks already implemented in hardware become more feasible and beneficial.

Model	Total On-Chip Power (W)
Single Precision 32-bit	
Series Adapter	N/A
Parallel Adapter	N/A
Half Precision 16-bit	
Series Adapter	2.841
Parallel Adapter	3.258
Custom 12-bit	
Series Adapter	3.506
Parallel Adapter	N/A
Custom 10-bit	
Series Adapter	3.172
Parallel Adapter	3.677

TABLE 4.5. Total On-chip power usage of series and parallel adapter on ZCU111 Evaluation Board. For 12-bit parallel adapter, and for 32-bit series and parallel adapter, only synthesis was performed as these modules do not fit on board for implementation which means no power usage data.

Table 4.5 presents the total on-chip power usage of the different bit-width series and parallel adapters on the target board. The 32-bit single precision adapter modules and the custom 12-bit parallel adapter module do not have any power usage data as they were too large to be implemented on the board. The on-chip power refers to the power consumed internally within the FPGA, equal to the sum of device static power and design power. From the table, it is evident that the total on-chip power for all the designs are quite large. This is due to a large portion of the target board's logic elements being used. There are various methods to decrease power usage on an FPGA board. The following techniques can be utilized to reduce dynamic power:

- Decrease the average logic-switching frequency.
- Reduce the amount of logic switching at each clock edge.
- Reduce the propagation of the switching activity.

Model	WNS (ns)	WHS (ns)	WPWS (ns)
Single Precision 32-bit			
Series Adapter	N/A	N/A	N/A
Parallel Adapter	N/A	N/A	N/A
Half Precision 16-bit			
Series Adapter	13.099	0.013	16.391
Parallel Adapter	7.717	0.017	16.391
Custom 12-bit			
Series Adapter	7.779	0.039	13.238
Parallel Adapter	N/A	N/A	N/A
Custom 10-bit			
Series Adapter	8.276	0.017	13.238
Parallel Adapter	10.702	0.046	13.238

TABLE 4.6. Timing analysis (Worst Negative Slack, Worst Hold Slack, Worst Pulse Width slack) of series and parallel adapter on ZCU111 Evaluation Board. For 12-bit parallel adapter, and for 32-bit series and parallel adapter, only synthesis was performed as these modules do not fit on board for implementation which means no power usage data.

All synthesis and implementations were conducted with a clock speed of 37MHz. This choice of clock speed aligns with the clock frequency employed in the hardware implementation of the SSCNN network, as documented in [**1**]. Given this clock speed and the associated timing constraints, the analysis reveals that there are no timing violations within the network. Table 4.6 provides a concise summary of the worst negative slack (WNS), worst hold slack (WHS), and worst pulse width slack (WPSW)for the various adapter modules of different bit widths.

Worst negative slack (WNS) relates to the setup slack of the critical path in the design. The positive WNS values in the table signify that the design comfortably meets the setup timing requirements for a clock speed of 37MHz. Similarly, worst hold slack addresses the hold slack of the critical path, and here too, the positive values indicate that the design complies with the hold timing requirements for a clock speed of 37MHz. Lastly, worst pulse width slack ensures that the design adheres to the minimum and maximum period, high pulse, and low pulse time requirements for each instance clock pin. The positive values across the board affirm that the designs meet these criteria.

During the adapter module development, various clock speeds were explored for synthesis and implementation. It was observed that no timing violations occurred at clock speeds of up to 100MHz. However, at clock speeds exceeding this threshold, timing violations emerged. To enhance the design's speed, the introduction of pipelining is a feasible implementation technique. Pipelining can lead to performance improvements by dividing the critical path

into multiple stages, reducing the overall critical path length, and allowing for higher clock speeds.

CHAPTER 5

Conclusion

In this research, a hardware adapter module in series and parallel configuration with online training was designed, built and tested for four different floating point bit-width formats. It was then synthesised and implemented on the ZCU111 Evaluation Board in order to explore resource utilization, power consumption and timing analysis. The adapter modules were used to adapt a small convolutional neural network, the Super Skinny Convolutional Neural Network (SSCNN) composed of one convolutional layer, one max-pooling layer and one fully-connected layer. The dataset used on the network to perform multiple-domain adapition is the Reduced MNIST dataset, which has the number 9 removed, and the Original MNIST dataset.

Software versions of the adapter network were first designed and tested in python using the machine learning module PyTorch, and then from scratch in C. These tests were conducted at various floating-point bit-width formats. Pre-training the original SSCNN network on the Reduced MNIST data yielded high levels of accuracy for both the PyTorch model and C simulator model. For the 32-bit network, PyTorch produced an accuracy of 98.35% and the C network an accuracy of 97.43%. The 16-bit half precision and custom 12-bit floating point precision also trained the network to similar accuracies. Following this, the network was fine-tuned with the series and parallel adapters and compared with full fine-tuning for a new dataset, the Original MNIST dataset. The series and parallel adapters maintained similar levels of accuracy as full-finetuning on the new dataset while only introducing a small number of trainable parameters. They also retain the knowledge of the original domain the network is pretrained on. Meanwhile, full fine-tuning retrains the whole network and adjusts all parameters.

The adapter modules were then designed and implemented in hardware. Four different floating point bit-widths were designed and resource usage examined. 32-bit width single precision utilized the most resources on the target board and was too large to fit on the board. 16-bit half precision, custom 12-bit precision serial, and custom 10-bit precision all fit on the board and the resource usage decreases as the bit width decreased. The series adapter required less hardware usage then the parallel adapter, making it a more viable option in hardware, despite the parallel adapter performing slightly better in software.

As highlighted in section 1, one of the primary objectives of this project was to gauge the resource requirements when integrating an adapter module into an existing hardware network. The experiments conducted have revealed that the current resource demand of the adapter modules exceeds or use a large portion of the available hardware resources. However, these

experiments have also demonstrated promising avenues for optimizing and reducing hardware usage.

One of the notable strategies for resource reduction is the reduction of the bit-width format, while still ensuring that the adapter modules can effectively support online training. Recent research, such as the development of innovative 8-bit block minifloat formats capable of enabling online training [**41**], provide valuable approaches for implementing these formats within the adapter module networks. This can significantly cut down hardware requirements, making the integration of adapter modules more feasible. Moreover, exploring different adapter module architectures may also lead to reductions in hardware utilization.

In summary, FPGA-based adapter modules with online training functionality show great promise and represent viable alternatives to software solutions. They can be effectively employed for multi-domain learning and be integrated into various networks. This project serves as a successful proof-of-concept for these capabilities.

5.1 Future outlook

Looking ahead, a primary objective for future work should be the optimization of adapter modules to minimize hardware resource usage and power consumption. As demonstrated in this project, several techniques can be employed to reduce the hardware demands of adapter modules. The next steps would involve implementing and fine-tuning these techniques to achieve significant reductions in hardware usage. Additionally, exploring different adapter architectures and their hardware implementations could offer further insights into resource usage reduction.

In summary, this project paves the way for more efficient and resource-conscious hardware-based adapter modules, offering a promising avenue for enhancing adaptability in neural networks.

References

[1] J. Si, "Neural networks in hardware," Ph.D. dissertation, University of Nevada, Las Vegas, 2019.

[2] A. Samuel, "Some studies in machine learning using the game of checkers," *IBM Journal*, 1959.

[3] K. Rowe, "How search engines use machine learning: 9 things we know for sure," Available at https://www.searchenginejournal.com/ml-things-we-know/408882/#close (2023/10/21).

[4] "How machine learning is used in autonomous vehicles," Available at https://www.rinf.tech/how-machine-learning-is-used-in-autonomous-vehicles/ (2023/10/21).

[5] K. R. Davenport T, "The potential for artificial intelligence in healthcare," *Future Healthc J*, 2019.

[6] "Machine learning (in finance)," Available at https://corporatefinanceinstitute.com/resources/data-science/machine-learning-in-finance/ (2023/10/21).

[7] V. Ford and A. Siraj, "Applications of machine learning in cyber security," in *Proceedings of the 27th international conference on computer applications in industry and engineering*, vol. 118. IEEE Xplore Kota Kinabalu, Malaysia, 2014.

[8] H. Chen and C. Hao, "Hardware/software co-design for machine learning accelerators," in *2023 IEEE 31st Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, 2023, pp. 233–235.

[9] e. a. Norman P. Jouppi, "In-datacenter performance analysis of a tensor processing unit," in *EEE 44th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, 2017.

[10] C. Wang, L. Gong, X. Li, and X. Zhou, "A ubiquitous machine learning accelerator with automatic parallelization on fpga," *IEEE Transactions on Parallel and Distributed Systems*, vol. 31, no. 10, pp. 2346–2359, 2020.

[11] K. Kara, D. Alistarh, G. Alonso, O. Mutlu, and C. Zhang, "Fpga-accelerated dense linear machine learning: A precision-convergence trade-off," in *2017 IEEE 25th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*.

IEEE, 2017, pp. 160–167.

[12] K. Nagarajan, B. Holland, A. D. George, K. C. Slatton, and H. Lam, "Accelerating machine-learning algorithms on fpgas using pattern-based decomposition," *Journal of Signal Processing Systems*, vol. 62, pp. 43–63, 2011.

[13] M. Iman, H. R. Arabnia, and K. Rasheed, "A review of deep transfer learning and recent advancements," *Technologies*, vol. 11, no. 2, 2023. [Online]. Available: https://www.mdpi.com/2227-7080/11/2/40

[14] X. Yin, W. Chen, X. Wu, and H. Yue, "Fine-tuning and visualization of convolutional neural networks," in *2017 12th IEEE Conference on Industrial Electronics and Applications (ICIEA)*, 2017, pp. 1310–1315.

[15] H. B. Sylvestre-Alvise Rebuffi and A. Vedaldi, "Learning multiple visual domains with residual adapters," *31st Conference on Neural Information Processing Systems*, 2017.

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

[17] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," *CoRR*, vol. abs/1810.04805, 2018. [Online]. Available: http://arxiv.org/abs/1810.04805

[18] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich, "Going deeper with convolutions," in *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2015, pp. 1–9.

[19] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "Imagenet: A large-scale hierarchical image database."

[20] "Cifar-100 (canadian institute for advanced research)," https://www.cs.toronto.edu/~kriz/cifar.html, 2009, accessed: 2023/10/21].

[21] A. Wang, A. Singh, J. Michael, F. Hill, O. Levy, and S. R. Bowman, "Glue: A multi-task benchmark and analysis platform for natural language understanding," 2019.

[22] X. Zhai, J. Puigcerver, A. Kolesnikov, P. Ruyssen, C. Riquelme, M. Lucic, J. Djolonga, A. S. Pinto, M. Neumann, A. Dosovitskiy, L. Beyer, O. Bachem, M. Tschannen, M. Michalski, O. Bousquet, S. Gelly, and N. Houlsby, "A large-scale study of representation learning with the visual task adaptation benchmark," 2020.

[23] "Ieee standard for floating-point arithmetic," *IEEE Std 754-2019 (Revision of IEEE 754-2008)*, pp. 1–84, 2019.

[24] S. Andresen, "John mccarthy: father of ai," *IEEE Intelligent Systems*, vol. 17, no. 5, pp. 84–85, 2002.

[25] O. Sharma, "Exploring the statistical properties and developing a non-linear activation function," in *2022 International Conference on Automation, Computing and Renewable Systems (ICACRS)*, 2022, pp. 1370–1375.

[26] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *Nature*, vol. 323, pp. 533–536, 1986.

[27] M. Nielsen, "Neural networks and deep learning," http://neuralnetworksanddeeplearning.com/chap2.html, 2012.

[28] "Zynq ultrascale+ rfsoc zcu111 evaluation kit," https://www.xilinx.com/products/boards-and-kits/zcu111.html#specifications, accessed: 2023/10/23].

[29] "Fpga design: A comprehensive guide to mastering field-programmable gate arrays," https://www.wevolver.com/article/fpga-design-a-comprehensive-guide-to-mastering-field-programmable-gate-arrays, 2023, accessed: 2023/10/23].

[30] Y. Chen and W. du Plessis, "Neural network implementation on a fpga," in *IEEE AFRICON. 6th Africon Conference in Africa,*, vol. 1, 2002, pp. 337–342 vol.1.

[31] N. B. Gaikwad, V. Tiwari, A. Keskar, and N. C. Shivaprakash, "Efficient fpga implementation of multilayer perceptron for real-time human activity classification," *IEEE Access*, vol. 7, pp. 26 696–26 706, 2019.

[32] J. Eldredge and B. Hutchings, "Rrann: a hardware implementation of the backpropagation algorithm using reconfigurable fpgas," in *Proceedings of 1994 IEEE International Conference on Neural Networks (ICNN'94)*, vol. 4, 1994, pp. 2097–2102 vol.4.

[33] S. Zagoruyko and N. Komodakis, "Wide residual networks," 2017.

[34] A. A. Rusu, N. C. Rabinowitz, G. Desjardins, H. Soyer, J. Kirkpatrick, K. Kavukcuoglu, R. Pascanu, and R. Hadsell, "Progressive neural networks," 2022.

[35] S.-A. Rebuffi, H. Bilen, and A. Vedaldi, "Efficient parametrization of multi-domain deep neural networks," 2018.

[36] A. Rosenfeld and J. K. Tsotsos, "Incremental learning through deep adaptation," 2018.

[37] H. Chen, R. Tao, H. Zhang, Y. Wang, W. Ye, J. Wang, G. Hu, and M. Savvides, "Conv-adapter: Exploring parameter efficient transfer learning for convnets," 2022.

[38] M. Fasi and M. Mikaitis, "Cpfloat: A c library for simulating low-precision arithmetic," vol. 49, no. 2, 2023. [Online]. Available: https://doi.org/10.1145/3585515

[39] F. de Dinechin and B. Pasca, "Designing custom arithmetic data paths with FloPoCo," *IEEE Design & Test of Computers*, vol. 28, no. 4, pp. 18–27, Jul. 2011.

[40] https://www.xilinx.com/products/boards-and-kits/zcu111.html#specifications, accessed: 2023-10-31.

[41] C. Guo, B. Lou, X. Liu, D. Boland, and P. H. Leong, "Single-batch cnn training using block minifloats on fpgas." New York, NY, USA: Association for Computing Machinery, 2023. [Online]. Available: https://doi.org/10.1145/3543622.3573171

APPENDIX A

Comparison of Series Adapter Module in C Simulator and in hardware simulation for forward propogation

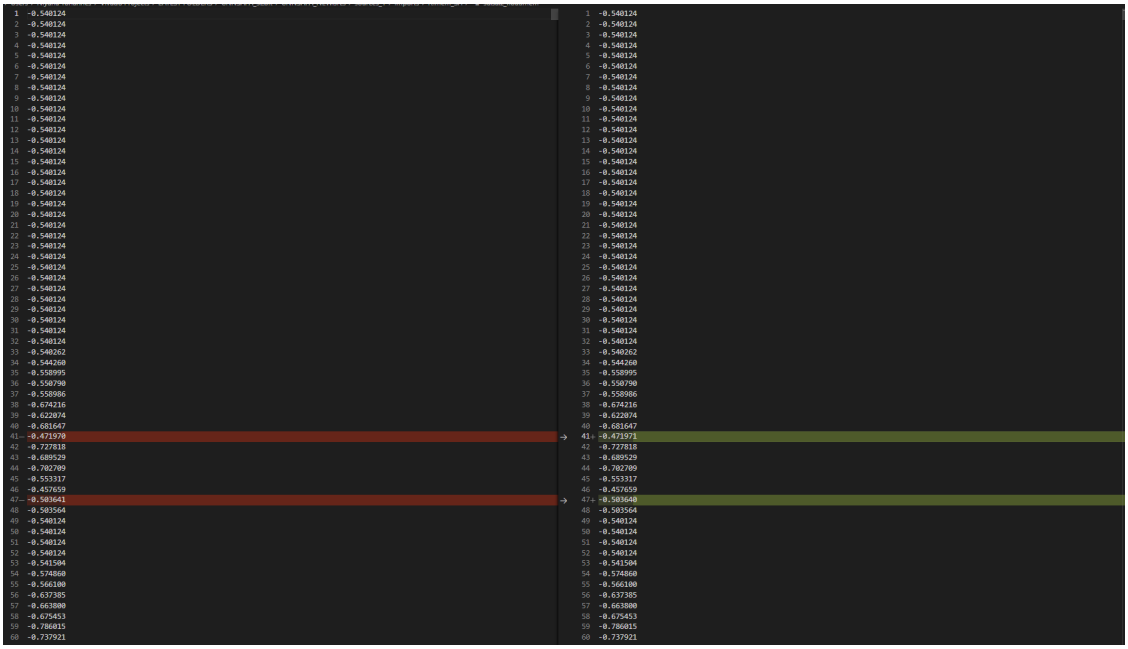


FIGURE A.1. Convolutional Adapter Layer output for series adapter. C simulator on the left and hardware simulation on the right

APPENDIX B

Comparison of Series Adapter Module in C Simulator and in hardware simulation for backpropogation

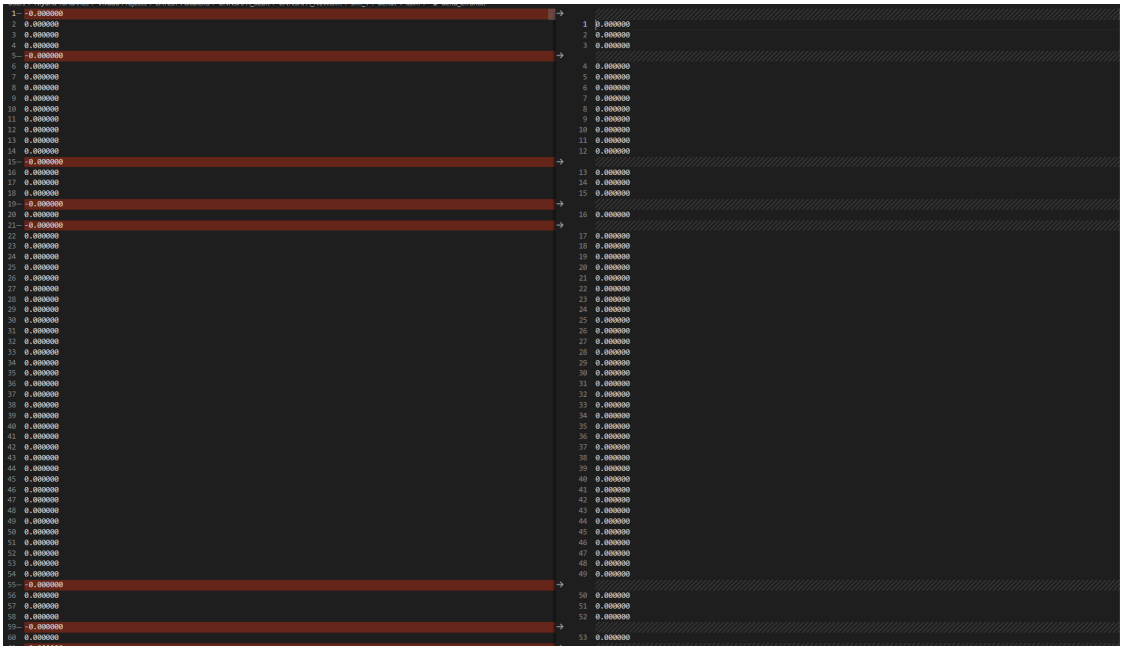


FIGURE B.1. Error at adapter layer output for series adapter. C simulator on the left and hardware simulation on the right

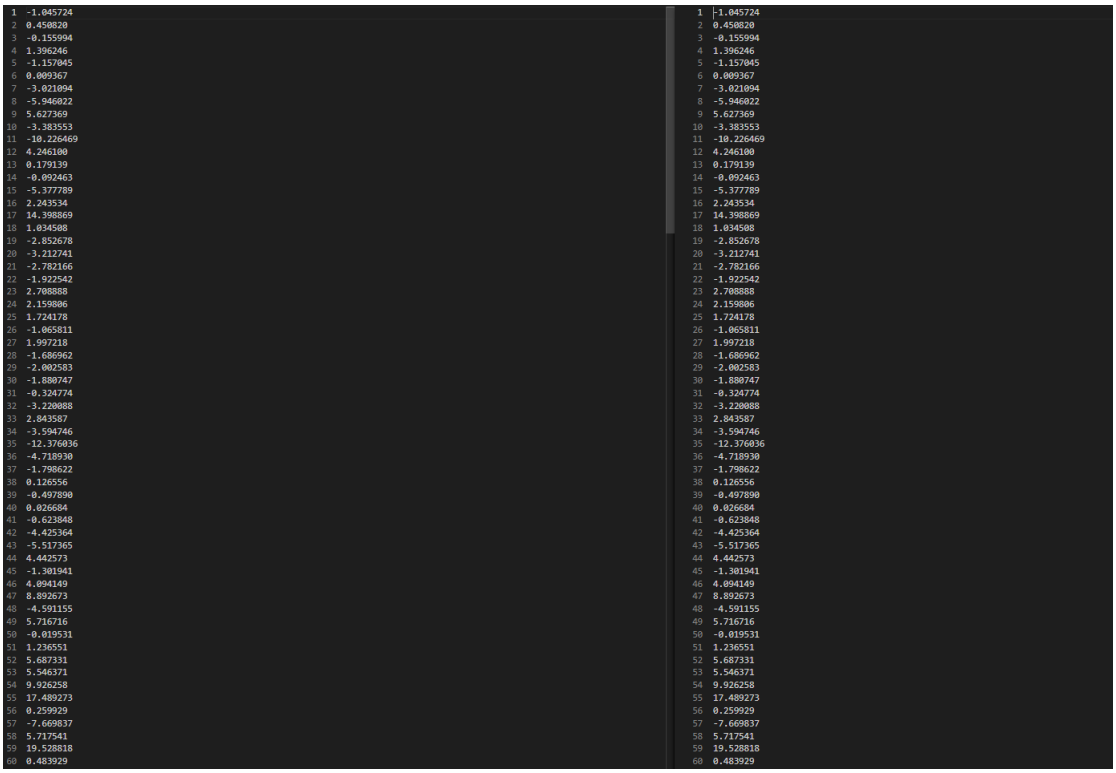


FIGURE B.2. Change in weight at adapter layer output for series adapter. C simulator on the left and hardware simulation on the right

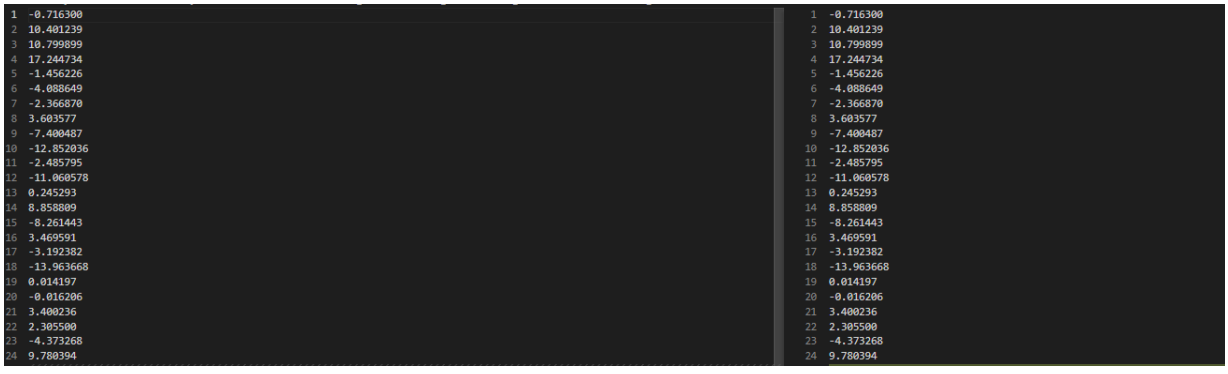


FIGURE B.3. Change in bias at adapter layer output for series adapter. C simulator on the left and hardware simulation on the right

APPENDIX C

Comparison of Parallel Adapter Module in C Simulator and in hardware simulation for forward propogation

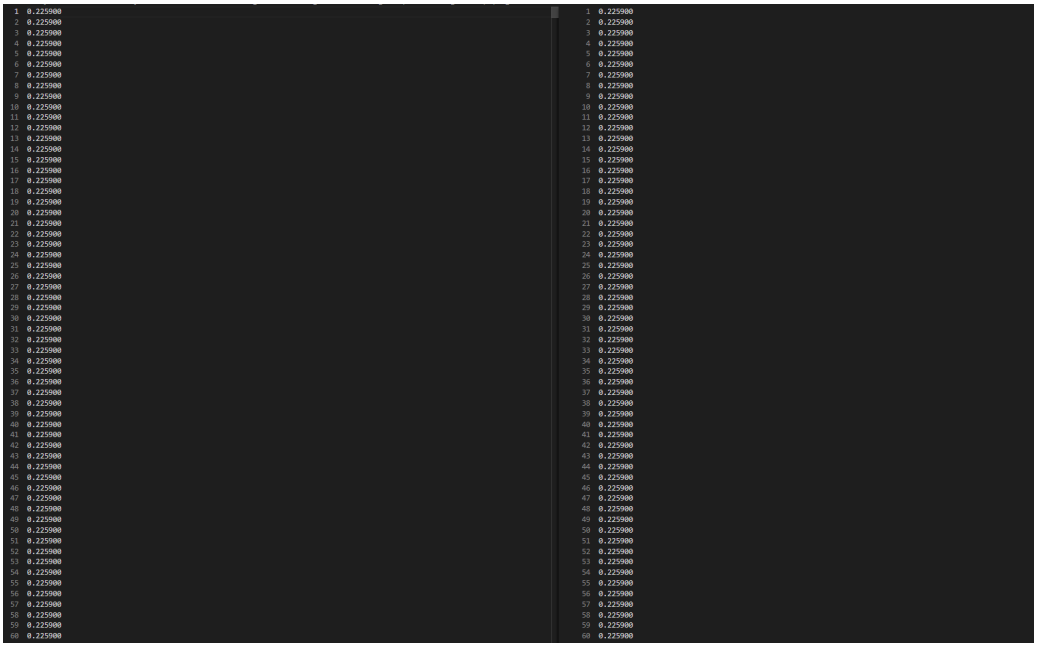


FIGURE C.1. Convolutional Adapter Layer output for parallel adapter. C simulator on the left and hardware simulation on the right

APPENDIX D

Comparison of Parallel Adapter Module in C Simulator and in hardware simulation for backpropogation

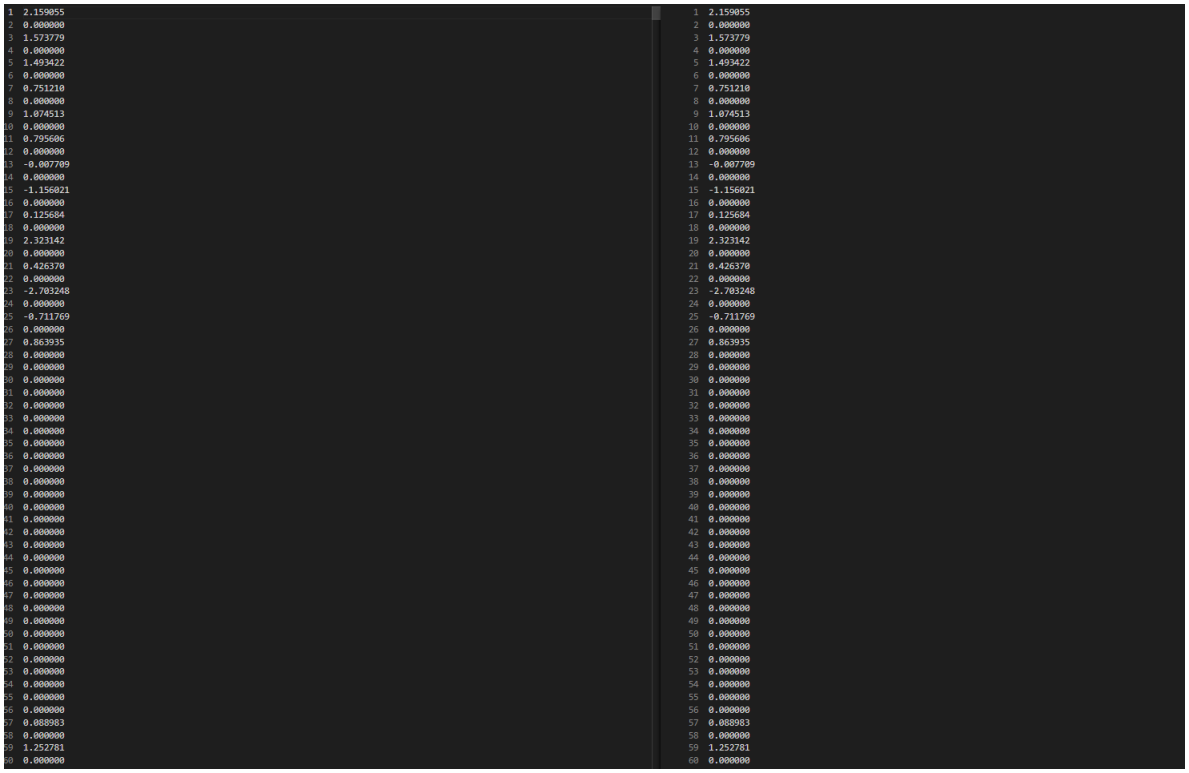


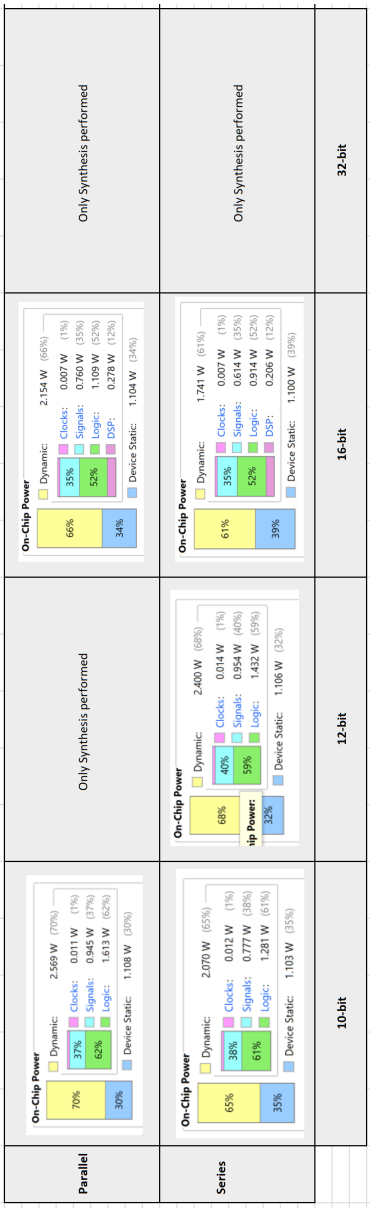
FIGURE D.1. Error at adapter layer output for parallel adapter. C simulator on the left and hardware simulation on the right

FIGURE D.2. Change in weight at adapter layer output for parallel adapter. C simulator on the left and hardware simulation on the right

FIGURE D.3. Change in bias at adapter layer output for parallel adapter. C simulator on the left and hardware simulation on the right

APPENDIX E

Complete Power usage summary of Series and Parallel Adapter on ZCU111 Evaluation Board



APPENDIX F

Complete timing summary of Series and Parallel Adapter on ZCU111 Evaluation Board