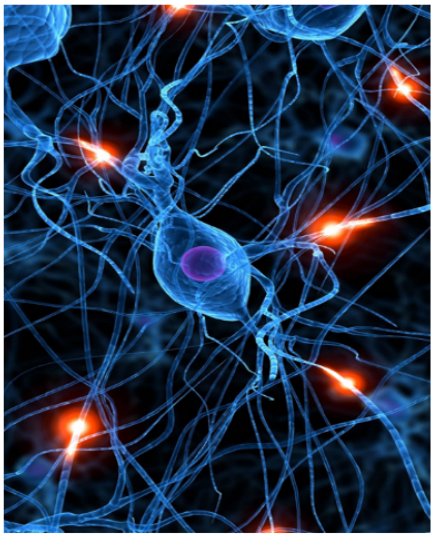
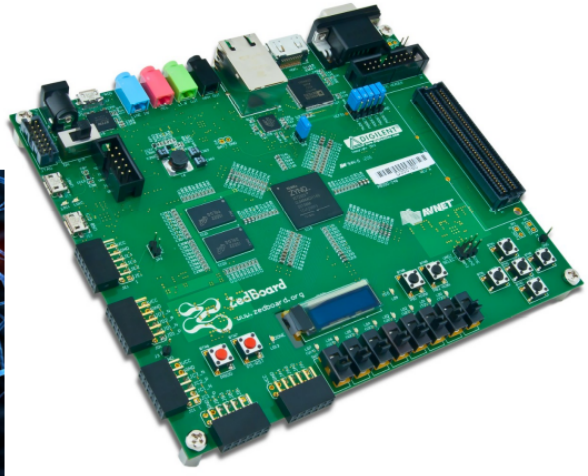
Imperial College London

Department of Electrical and Electronic Engineering

Final Year Project Report 2017





Project Title:	**Space Brain - Investigating the Suitability of FPGA based Convolutional Neural Network for Space Applications**
Student:	**Jacobus (Jukka) Johannes Hertzog**
CID:	**00828711**
Course:	**EE4T**
Project Supervisor:	**Dr. Felix Winterstein**
Second Marker:	**Dr. David Thomas**

Abstract

Over the last decade, advancements in the design of Convolutional Neural Networks (CNNs) have led to significant improvements in the accuracy of image classification systems. A potential application of a CNN image classifier would be processing data on board a satellite. In such a situation, with limited power and computational resources, it would be beneficial to make use of an FPGA in order to alleviate the resource demands of the computationally expensive CNN. However, FPGAs are sensitive to ionizing radiation, and at orbital altitudes, the high radiation environment can induce a variety of errors in the FPGA fabric, casting doubt on the suitability of a commercial FPGA for CNN applications in space.

This project implements an FPGA based CNN, and simulates radiation-induced errors. These results are analyzed to determine the impact on the system's performance. [The tests are yet to be performed, and as such, this part of the abstract is not yet completed.] The implementation target platform is the ZedBoard Zynq-7000 ARM/FPGA SoC. The hardware and software portions of the embedded system are both implemented using the Xilinx SDSoC development environment.

Acknowledgements

TODO

Contents

1 Introduction **6**

1.1 Project Motivation 6

1.2 Target Platform 7

1.3 Report Structure 8

2 Background **9**

2.1 High Level Synthesis 9

2.2 Convolutional Neural Networks 9

2.2.1 Convolution Layer 10

2.2.2 Activation Functions 11

2.2.3 Pooling Layer 12

2.2.4 Fully Connected Layer 12

2.3 FPGAs and CNN Implementations 12

2.3.1 Implementation Challenges 13

2.3.2 FPGA CNN Accelerators 13

2.4 FPGAs and Space Applications 14

2.4.1 Single Event Upsets 14

2.4.2 SEU Mitigation Techniques 14

3 Project Specification **16**

3.1 Necessary Functionality 16

3.2 Desirable Functionality 16

3.3 Testing Specification 16

3.4 Evaluation Specification 17

4 Design **18**

4.1 Network Design 18

4.1.1 Data Objects 18

4.1.2 Layer Design 19

4.2 Convolution Layer 20

4.2.1 Matrix Multiplication Method 20

4.2.2 Tiled Convolution Method 21

4.3 Fully Connected Layer 23

4.4 Rectified Linear Unit Layer 24

4.5 Pooling Layer 24

4.6 Processing System (PS) Design 24

4.6.1 Zynq-7000 Operating System 24

4.6.2 Software 25

4.7 Programmable Logic (PL) Design 25

4.8 PS and PL Interfacing 25

4.9 Soft Error Mitigation (SEM) Core Integration 26

5 Implementation **27**

5.1 Infrastructure Setup 27

5.2 Programming Language 27

5.3 Development Methodology 27

5.4 Hardware Optimisations 28

6 Testing **29**

6.1 Layer Testing 29

6.2 Network Testing 29

7 Evaluation **30**

7.1 FPGA Implementation of a CNN 30

7.2 Fault Tolerance Investigation 30

8 Conclusion and Future Work **31**

1 Introduction

Inspired by the structure of the optical nervous system in animals, a neural network is made up of layers of artificial neurons that recognise certain features from the input data. The concept of an artificial neural network was first introduced in 1980 [1], and driven by increases in computing power and a growth in machine learning applications, neural networks have become very powerful tools. In the last decade, the development of Convolutional Neural Networks (CNNs) has lead to great advancements in image classification accuracy.

CNNs are a powerful Deep Learning tool that can be used to solve extremely complex computational problems. In particular, they have gained popularity in image classification applications [2]. Other popular applications within machine vision include video classification, face detection and gesture recognition. They are also being used in a wide range of other fields including speech recognition, natural language processing and text classification.

However, whilst achieving exceptional performance, CNNs are extremely computation and resource heavy. Typically, they will be implemented on large servers or on GPU based systems to accommodate the need for computational power. As a result, implementing them on embedded systems, which typically have very limited resources, presents many challenges. A promising solution to this problem is the use of an FPGA, which provide a very high computational efficiency with low power usage, in addition to other benefits.

Now suppose that the FPGA based CNN will be implemented on a satellite. In orbit, the satellite will be exposed to intense radiation, and as a result, errors will be produced on the radiation-sensitive FPGA. Before such a satellite can be designed and launched, it is important to know how these errors would affect the performance of the CNN, and if the CNN design could be modified to mitigate the impact of these errors. Therefore, the aim of this project is to implement a CNN on an FPGA-based system, and then to investigate the suitability of this system for space applications.

1.1 Project Motivation

CNNs are an extremely valuable and powerful tool in modern machine learning, with countless applications. One of these potential applications is on board a sattelite orbiting the Earth, using a camera to gather image data. Currently, these images would have to be transferred to a terrestrial base station in order to be processed and analysed for information. On small sattelites, power is limited, and the communications equiment does not have a great data capacity. Therefore it is

expensive for the sattelite to transfer full images to the base station. In addition to this, the link between the station and the sattelite may only be available at certain points in the sattelite's orbit, and the channel capacity may be dependant on factors such as weather interrupting signals.

In this scenario, the rate at which data can be gathered is heavily limited by the rate at which the data can be transferred to Earth. A way to alleviate this problem would be to implement a CNN on board the sattelite. The CNN could discern the usefulness of an image and discard uninteresting data, saving the cost of transferring it to Earth. The CNN could even analyse the images and send only the results to Earth, eliminating the need to transfer the images altogether.

However, implementing such a system is not without challenges. The aforementioned radiation induced errors can completely disrupt electronic devices, and FPGAs are particularly vulnerable. The impact of these errors, referred to as Single Event Upsets (SEUs), is discussed in more detail in Section 2.4.1. These errors pose a risk that must be investigated and assessed before the CNN sattelite can be attempted. If it is found that the system is never going to be feasible, then the design and construction of such a system would be an avoidable waste of time and resources. This is the motivation behind this project.

1.2 Target Platform

The hardware used in this project is a product from Xilinx called a Zedboard, a development board for the Zynq-7000 System On a Chip (SoC). This board was chosen because of its useful hardware features and its integrated support for valuable Xilinx proprietary tools. These tools include High Level Synthesis systems, SoC design tools and the Soft Error Mitigation IP core. The Zynq-7000 consists of Xilinx FPGA as its Programmable Logic(PL), and a dual-core Cortex-A9 ARM processor as its Processing System(PS) integrated together. Data transfer between these performed by the AXI4-Lite interface, which is discussed in more detail in Section **??**.

The ARM processor facilitates easy control of the system with a Linux kernel. This is then able to run software and initiate hardware processes in order to perform computations. Software for the processor was designed using the Xilinx SDSoC devlopment environment.

Vivado HLS was used to design and build a hardware block from C++ code. Then Vivado was used to integrate this hardware block into an complete hardware architecture, allowing it to be connected to and controlled by the software. This was then synthesised and the generated bitstream was used to configure the FPGA.

1.3 Report Structure

Section 2 - Background: In order to guarantee that the reader has the contextual knowledge required to understand the project, this section will High Level Synthesis, and CNNs. FPGA implementations of CNNs and the difficulties of using FPGAs in radiation-heavy environments will also be explored.

Section 3 - Project Specification: This section details the project requirements that should be met by the final system.

Section 4 - Design: The decisions taken in the design process are described and explained in this section. The high level design is described, as well important aspects of the low level design.

Section 5 - Implementation: This section illustrate how the system was implemented, aiming to provide the reader with an understanding of the development process.

Section 6 - Testing: The testing methodology used to verify various components of the system is detailed in this section.

Section 7 - Evaluation: The focus of this section is the result of the investigation, including an evaluation of the system's performance.

Section 8 - Conclusion and Future Work: This section will summarize the results of the project, highlight areas for improvement and further work, and present concluding remarks.

2 Background

This section will provide some details and information that will provide context to the project, and will be useful for understanding key concepts in the rest of the report. High Level Synthesis tools, CNN structures, FPGA implementations of CNNs, and the issues surrounding the use of FPGAs in space will all be discussed.

2.1 High Level Synthesis

Typically, digital circuits on reconfigurable hardware architectures, such as FPGAs, are designed using a Hardware Description Language (HDL) such as Verilog or VHDL. While these methods allow the designer to have a great level of control over the system, and can produce extremely efficient designs, the designer is forced to specify functionality at a very low level of abstraction, specifying cycle by cycle behaviour. Use of HDL tools requires hardware expertise and, when designing a complex system, makes for a long and cumbersome development process.

High Level Synthesis (HLS) tools are a solution to this problem. An HLS tool is a piece of software which can interpret the desired behaviour of code written in a programming language like C or C++, and generate an HDL implementation of that behaviour, allowing the designer to work at a higher level of abstraction. This means that a software engineer can create FPGA designs without having to build up hardware expertise, allowing them to utilise the speed and efficiency of hardware designs. It also benefits hardware engineers, allowing them to design the system more quickly and reliably, facilitating the development of more complex systems [3]. Popular HLS tools include academic, open-source software like LegUp, and commercial tools like Catapult-C and Xilinx's Vivado. In this project, the embedded system will be designed using SDSoC from Xilinx, which utilises Vivado HLS to produce a configuration for the Programmable Logic in the Zynq SoC.

2.2 Convolutional Neural Networks

CNNs can be used to classify images in a forward inference process. But before using the CNN for any task, the CNN should be trained on a set of training data. The training of a CNN is often implemented on large servers with a significant computational capacity, with an enormous set of training data, or on a GPU when such a server is not available. For the embedded FPGA platform,

this project will only focus on implementing and utilising the inference process of a CNN, and not on the training process.

A typical CNN consists of a number of layers that run in sequence. Convolution (CONV), activation function, pooling, and Fully Connected (FC) layers make up a typical CNN model, with CONV and FC being the most important. CONV and FC layers have parameters of called weights, which are set by the training process. These weights will determine which patterns each layer of the CNN will be activated by, and ultimately, what the CNN is able to recognise and classify.

The first layer of a CNN reads an input image and outputs a series of feature maps. The input image will be three dimensional - the height and width of the image make up two dimensions, and colours make up the third layer. For an RGB image, the input will have a depth of 3. Then there will be CONV layers interspersed by activation function and pooling layers, which will make up most of the CNN. These layers will decompose the image into feature maps, varying from low-level features such as edges, lines, curves, etc., in the initial layers to high-level features in the deeper layers [4]. Each subsequent layer reads the feature maps generated by preceding layers and generates new feature maps at its output. Finally a classifier, consisting of at least one FC layer, reads the final feature maps and determines the probability of the input imaging belonging to each category of the training data. An example of a CNN model is shown in Figure 1.

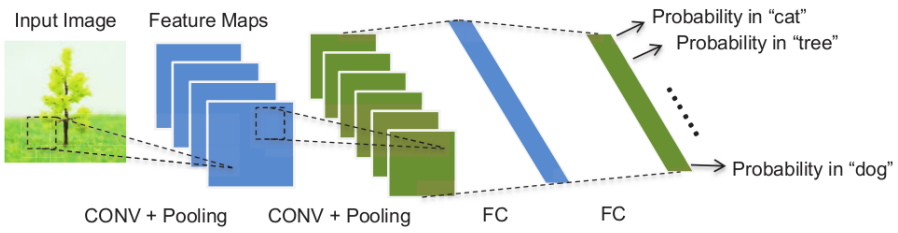


Figure 1: A typical CNN structure from the feature map perspective [5]

2.2.1 Convolution Layer

The CONV layer takes a series of feature maps as input and convolves with convolutional kernels to obtain the output feature maps. The simplest version of the convolution operation a involves 3-dimensional multiply and accumulate operation of N_{if} input features with $K \times K$ convolution kernels to get an output feature neuron value as shown in Equation 1

$$out(f_o, x, y) = \sum_{f_i=0}^{N_{if}} \sum_{k_x=0}^{K} \sum_{k_y=0}^{K} wt(f_o, f_i, k_x, k_y) \times in(f_i, x + k_x, y + k_y) \quad (1)$$

where $out(f_o, x, y)$ and $in(f_i, x, y)$ represent the neurons at position (x, y) in the feature maps f_o and f_i, respectively and $wt(f_o, f_i, k_x, k_y)$ is the weights at position (k_x, k_y) that gets convolved with input feature map f_i to get the output feature map f_o [4].

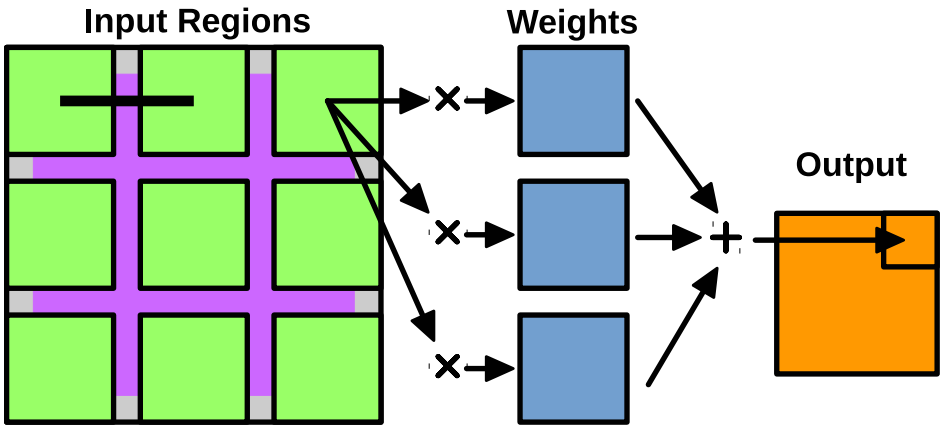


Figure 2: The Convolution Computation. The input feature map is shown in purple surrounded by grey zero padding, and the input regions are shown in green. The stride is shown as the thick black line in the input feature map.

This can be visualised as scanning over the input feature map with the weights kernel and computing the sum of the elementwise products of the weights kernel and a section of the input feature map that matches the size of the kernel (an input region). Each position that the kernel takes corresponds to one output position. The more widely used version of the convolution includes zero padding, adding layers of zeros around the input, and stride, moving the kernel over the input in steps. The full operation is depicted in Figure 2

2.2.2 Activation Functions

CONV layers are followed by an activation function layer. This layer can be thought of as a decision, based on the output of the CONV layer, on to what extent each neuron in the next layer has been activated. The CONV layer output is a linear combination of the inputs and the weights at a position in the network, and role of the activation function is to produce a non-linear decision boundary. The commonly used activation functions in traditional neural networks are non-linear functions such as tanh and sigmoid, which require a longer training time in CNNs [6]. Hence, Rectified Linear Unit (ReLU), defined as $y = max(x, 0)$, has become the popular activation function among CNN models as it converges faster in training, and has less computational complexity than exponent functions in tanh and sigmoid, also aiding hardware design.

2.2.3 Pooling Layer

Spatial pooling or sub-sampling is utilized to reduce the feature dimensions as we traverse deeper into the network. As shown in Equation 2, pooling computes the maximum or average of neighbouring $K \times K$ neurons in the same feature map, which also provides a form of translational invariance [7]. The pooling layer helps abstract higher-level features without redundancy, as well as reducing the dimensionality of lower-level features without losing important information [4].

$$out(f_o, x, y) = \underset{0 \geqslant (k_x, k_y) < K}{max/average}(in(f_o, x + k_x, y + k_y)) \tag{2}$$

2.2.4 Fully Connected Layer

An FC layer is a classification layer where all the input features (N_{if}) are connected to all of the output features (N_{of}) through synaptic weights (wt). These are at the end of the CNN and perform the final classification, based on the features that have been recognised by the rest of the network. Each output neuron is the weighted summation of all the input neurons as shown in Equation 3 [4].

$$out(f_o) = \sum_{f_i=0}^{N_{if}} wt(f_o, f_i) \times in(f_i) \tag{3}$$

The outputs of the inner-product layer traverse through ReLU based activation function to the next inner-product layer or directly to a Softmax function that converts them to probability in the range (0, 1). The final accuracy layer compares the labels of the top probabilities from softmax layer with the actual label and gives the accuracy of the CNN model [4].

2.3 FPGAs and CNN Implementations

CNNs are computationally demanding and consume a lot of resources, and thus are difficult to implement on an embedded systems. FPGAs are the most promising platform for the implementation of an embedded CNN, due to their high computational efficiency and low power consumption. FPGAs are also more attractive than other digital hardware platforms, such as ASICS, because of their reconfigurability and fast development time, aided by HLS tools.

2.3.1 Implementation Challenges

One of the main challenges faced by FPGA implementations is memory bandwidth. A typical CNN model may have more than 60 million weights, which requires over 200MB of memory when represented with a 32 bit number. A standard commercial FPGA does not have enough on-chip memory to store this volume of data, so external memory must be utilised. Having to transfer this information to the FPGA can introduce a computational bottleneck. However, these weights are used in CONV layers multiple times. Thus, the system can be optimised by reducing frequent memory access. Chen et. al achieves this with a tiling method and dedicated buffers for data re-use. Another approach, is data quantization, which is discussed below. By quantizing the 32-bit floating-point weights into 16-bit fixed-point values, the pressure on the memory system can be alleviated. Suda et al. performed a study on the optimal precisions to balance accuracy and efficiency. It was found that the best choices were 8-bit precision for CONV weights, 10-bit precision for FC weights, with less than 1% compared accuracy loss compared to full precision weights [4].

2.3.2 FPGA CNN Accelerators

An FPGA implementation of a CNN, intended to improve the speed and efficiency of the CNN computation, is referred to as an FPGA CNN accelerator. Most research into FPGA CNN accelerators has focussed on using hardware to accelerating the computation of the CONV layers. These layers typically make up around 90% of the CNN's computational workload. The multiply and accumulate nature of the CONV layer's operation makes them ideal candidates for hardware acceleration, as FPGAs are able to perform these operations with great efficiency.

FPGAs also provide flexibility to implement the CNNs with limited data precision [4]. Many CNN accelerators use 16-bit fixed-point numbers instead of floating-point numbers to represent weights and data [8][9][10], especially because FPGAs. Using a shorter, fixed-point data representation can make significant reductions to the memory footprint and the use of computational resources. The CNN's data must be quantised in order for the fixed-point implementation to work, which will introduce a quantization error into the result. However, Chen et al. showed that using a 16-bit fixed-point implementation rather than a 32-bit fixed-point implementation only added an extra 0.26% to the error rate when using the MNIST dataset.

2.4 FPGAs and Space Applications

2.4.1 Single Event Upsets

Electronic circuits are sensitive to high-energy ionizing radiation, which can induce errors called Single Event Upsets (SEUs) [11]. In space, this is a particular problem. The radiation shielding provided by the Earth's magnetosphere, and enjoyed by systems at lower altitudes, is no longer present, leaving systems open to radiation from a variety of sources. A voltage pulse from an SEU can cause errors including altering digital values. SEU induced voltage pulses are problematic for all circuits in space, but the nature of reconfigurable circuits means that they are particularly vulnerable. These SEUs are typically manifested as a bit flip in a memory cell or a transient logic pulse in combinational logic. Habinc et al. describes three main categories of SEU [12].

The first category is the configuration upset. FPGAs are configured by loading the configuration bitstream into an internal configuration memory. This memory controls the configurable logic elements and how they connect together. A configuration upset occurs when an SEU alters a value in the configuration memory, and thus changes the function of the configuration.

The second category is the functional upsets in user logic are SEUs within the FPGA's logic. The FPGA's user logic cannot be tested through a readback of the configuration memory, because the logic contents will change as part of normal operation. These errors can cause transient glitches in combinatorial logic, and can upset sequential logic where the state of the system is vital to its function [13].

The final category is the architectural upset. This is when an SEU occurs in the control elements of the FPGA, e.g. the configuration circuits or reset control [12]. This can have an enormous range of consequences. For example, an SEU in the reset control could cause the FPGA to be unintentionally reset and all state information would be lost due to the FPGA being reconfigured.

Architectural upsets are outside the scope of this project, but configuration and functional upsets will be investigated. This is explained in more detail in Section 3.3.

2.4.2 SEU Mitigation Techniques

As a result of the problems that can be caused by SEUs, many techniques have been developed to mitigate the impact of SEUs in space. A few popular design techniques for overcoming SEU errors

are described below. Although this is outside the scope of the project, some of these techniques could potentially be used to increase the SEU resilience of the FPGA CNN implementation.

A very effective way of detecting configuration upsets is to readback and verify the configuration memory of the FPGA. As long as the external memory holding the bitstream is uncompromised, any discrepancies caused by SEUs can be detected and corrected [12]. The disadvantage of this method is that any sequential logic within the FPGA will lose all state information.

Functional Triple Modular Redundancy is another popular method of mitigating SEU errors [13]. This involves making three copies of a given design within the FPGA fabric, and having them 'vote' on the result through a combinatorial circuit. This is very effecting at reducing the error rate from combinatorial and user upsets, as two of the three circuits need to be compromised in the same way in order for a false result to be given as the output of the system.

Radiation hardening is the act of physically making a chip less susceptible to radiation, typically by encasing it in a substance that will shield the internal electronics [14]. Many FPGA vendors sell radiation hardened versions of their products, and it should greatly reduce the impact of SEUs, but it is often a very expensive solution due to the cost of the 'rad hard' hardware.

3 Project Specification

In this section, the requirements and design specifications captured during the planning of the project will be discussed. The requirements will be divided into two set of functionalities - those that are necessary for the project to be considered successful, and those that are not essential but would greatly enhance the system and the results gathered - and briefly described. Following this, the testing and evaluation specifications are laid out and discussed.

3.1 Necessary Functionality

- Runs on Zedboard
- Split Hardware and Software intelligently
- Conv Layer Implemented
- Implementation must be efficient enough to run tests within reasonable time
- SEU module is incorporated into design

3.2 Desirable Functionality

- Full CNN implemented, able to process and classify images with reasonable accuracy
- System is optimised to run as efficiently as possible

3.3 Testing Specification

- Compare control results against pure software implementation to verify correctness
- Where numerical approximations are required, trade-off must be justifiable

3.4 Evaluation Specification

- Analyse SDSoC's estimate of hardware resources used, as well as actual resources used.
- Compare computation time/performance against those in literature
- If applicable, compare CNN accuracy to those in literature
- Assess impact of SEUs with Failure In Time measurement

4 Design

This section will provide an overview of the system design from the top level, highlighting the distribution of computational load between software and hardware.

4.1 Network Design

IF APPLICABLE

- Describe Network organisation and design (If full network is implemented)
- Describe Caffe, Justify use as basis for design (pre-trained network models, etc)
- mention blobs

4.1.1 Data Objects

In order for data to be easily processed by network layers, a class called `DataMemory` was created. This is a simple wrapper around dynamically allocated memory that provides homogenous storage of all data in the network. The data is accessed through functions called `getConstData` and `getMutableData`, which both return a pointer that can be used to access the data as an array. A **`const`** pointer is returned by `getConstData` for situations where the memory is only going to be read, and `getMutableData` returns a normal pointer which can be used to read and write values to and from the memory space.

In order to make a universal, easily used data handler, the aforementioned `Blob` class was created. This class acts as a wrapper around `DataMemory` instances, and is used to store all data passing through the software portions of the network, including the weights data for Convolution and Fully Connected layers. Blobs are given four dimensions (batch number, depth, height and width) and allow the stored data to be accessed as a four dimensional matrix of data, as opposed to a flat array. The Blob class also includes useful functions that simplify data handling, such as a `count` function that returns the length of the data stored, and the `CopyFrom` function that copies data from one blob to another.

4.1.2 Layer Design

Each layer type was implemented as a class, which provided several benefits. Every layer would be self contained, with all functions and data related to the class being easiy accessible. The class data could also be protected from being changed in an unintended way by storing information in private variables, and allowing external access to the data only through dedicated functions that will not misuse it. Finally, using classes allows multiple, identical instances of the same layer types to be easily created.

In addition to multiple version of the same layers types, several very diverse layer types are required for a CNN to be properly implemented. If these were all implemented in vastly different ways, each with a unique interface for external code, setting up and running the network woud be challenging, if not impossible. Such a network would be very difficult to control automatically, and the interaction with each individual layer would likely have to be hard coded.

To overcome this problem, a class template was implemented for the network layers. A class template is a structure that defines a famiy of classes, providing a standardised set of functions and variables that will be inherited by all classes that use the class template as a base. The `Layer` base class is therefore used to define three main functions that create an interface for the layer. These functions have different purposes and perform varying tasks depending on the layer, so these functions don't define any of the layer functionality, but define an interface and usage is standardised for every layer in the network. Each of these functions takes pointers to the top and bottom Blobs of the layer as inputs, as shown in Figure 3. With a small extension, it wouldn't be difficult to implement layers that could handle multiple top and bottom Blobs, but this addition is unnecessary to compute the VGGnet CNN model, and was rejected in favour of keeping the implementation as simple as possible.

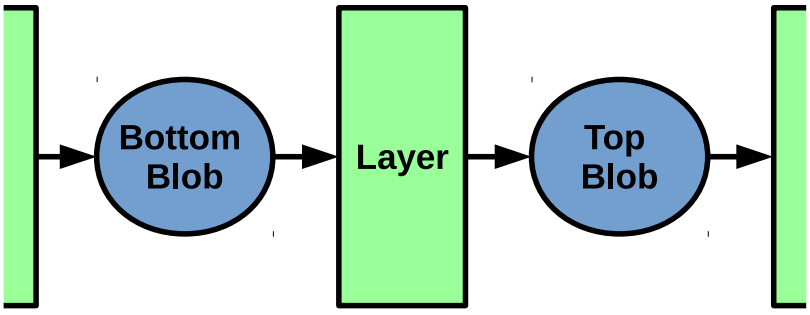


Figure 3: General Layer Architecture

The `LayerSetUp` function is run after the layer is originally constructed. Each layer class has a constructor that will initialise most of its paramters when a layer instance is created, but

sometimes of these parameters are dependant on the shape of the bottom Blob. These parameters are then set by the `LayerSetUp` function. This function also performs a sanity checks on the layer parameters to ensure that nothing is amiss.

The `Reshape` function is run after `LayerSetUp`. As the name suggests, it reshapes the top Blob to match the bottom Blob. A ReLU layer must produce an ouput of the same dimensions as the input, while a Convolution layer's output is determined by a combination of the input size and kernel size, and the padding and stride parameters, but in every layer type, the output shape is a critical part of the layer's operation. This function therefore adjusts the top Blob so that the layer operates correctly.

The most important function in this interface is `Forward`, which is run last and computes the output of each layer as described in Section 2.2. This simple interface allows a set of input data to be run through the entire network with a simple **`for`** loop that iterates through the layers and uses the top Blob of a layer as the bottom Blob of the next layer until the final layer output is calculated.

4.2 Convolution Layer

Two approaches to implementing the convolution were implemented in software and compared. The more intuitive approach uses nested **`for`** loops to iterate through the dimensions of the weights, inputs, and outputs and compute the results incrementally. The other approach reshapes the input and weight data, and performs the convolution in one large matrix multiplication.

4.2.1 Matrix Multiplication Method

The matrix multiplication method is requires the data to be in a specific shape. Thus, the data is flattened and organized into a matrices, as shown in figure 4.

In this approach, the weights kernels are considered to be three dimensional, where the height and width are given by the kernel size, and the depth is given by the depth of the input feature map. The each three-dimensional weights kernel is flattened into a row and the each row is stored sequentially. Thus the result is an $M \times K$ matrix where M is the number of weights kernels, and K is the volume of an individual kernel. If the weights are stored sequentially in memory, this reshaping process can be done without copying any data to new locations. In such a situation, it is a simple case of changing how the memory will be accessed, by setting the depth and height of the weights Blob to 1, and the Blob width to K.

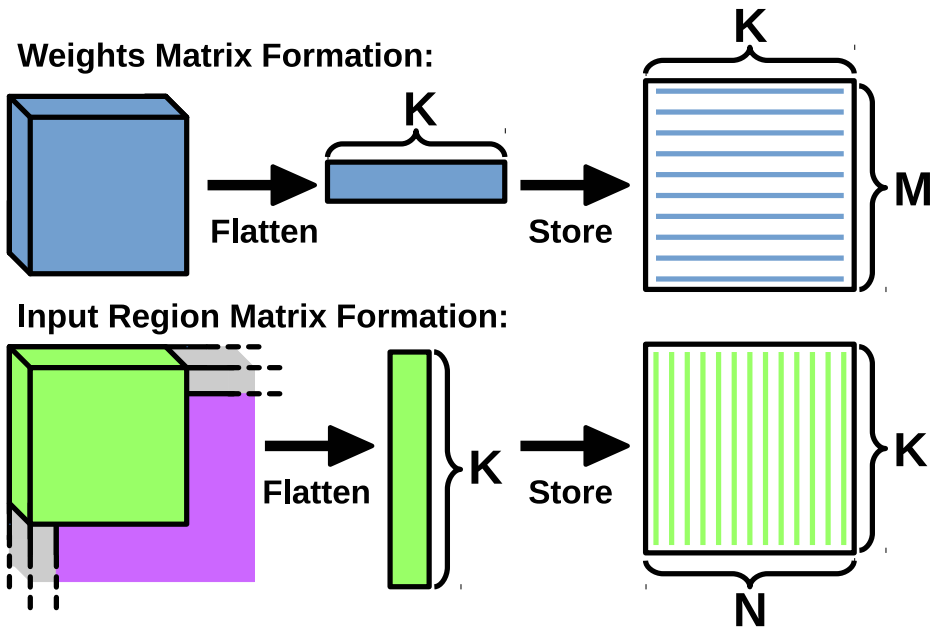


Figure 4: A visualisation of the reshaping process. The weights data is shown in blue, and the input region data is shown in green, embedded in a purple feature map with grey padding

Then the input feature map is considered in terms of input regions, i.e. each area that will be multiplied with weights kernels. The size of these regions matches the weights kernel, and the number is dependent on the input size, kernel size, padding and stride. These input regions are flattened and stored as columns. However, this is not as simple as changing the data access. If the convolution has padding, the input regions may contain extra zeros. Additionally, the stride and kernel size may result in input regions are overlapping or are not continuous. Therefore, each input region is individually mapped and copied from the input data into columns. This forms a $K \times N$ matrix, where K is the volume of an input region, and N is the number of input regions in the input feature map.

The output of the convolution is then found with $\mathbf{C} = \mathbf{A} \times \mathbf{B}$, where $\mathbf{C}$ is the output matrix, $\mathbf{A}$ is the weights matrix, and $\mathbf{B}$ is the matrix of input regions. In order for this computation to be valid, it should be clear that the volume of an input region is the same as the volume of a kernel. By changing the output Blob dimensions, the $M \times N$ output matrix can be treated as feature map with depth M, equal to the number of weights, and width and height that give a product N, equal to the number of input regions.

By reducing the convolution to a single matrix multiplication, the algorithm can take advantage of a software library called Basic Linear Algebra Subprograms, or BLAS. This can bring significant performance improvements, as the BLAS library can provide highly optimised functions for

a given platform. One such function is called `gemm()`, and performs a general matrix multiplication, $\mathbf{C} \leftarrow \alpha\mathbf{AB} + \beta\mathbf{C}$, where α and $beta$ are constants. By setting $\alpha = 1$ and $beta = 0$, the `gemm()` function can be used to compute the output of the convolution. This method is ideal for a software implementation of the convolution that will not be running on an embedded system. However, the weights matrix and input matrix will both be occupy a lot of memory in most practical CNNs, and a typical FPGA will not have enough on-chip memory to perform the entire convolution at once. This can be mitigated by breaking the matrix multiplication is broken down into individual row and column multiplications, but the process of moving the input regions to a new memory space will take up a lot of memory, so it is not well suited for an embedded system.

4.2.2 Tiled Convolution Method

Directly interpreting the convolution algorithm as described in 2.2.1, the pseudocode in Code 1 can be formed. This code iterates through the rows, columns and depth layers of the output, as well as the rows and columns of the weights kernels and the depth layers of the input. The input row and column for the given kernel and output locations are found in lines 6 and 8. In the centre of the loops, the value in the output feature map is incremented by the product of the corresponding weight and input values, but if the input location is in a padded region, the input value is 0. Therefore, the output only changes when the input is not in padding. It can be noted that this pseudocode is functionally identical to the Matrix Multiplication method, except that the computation is performed value at a time, and the input data is not moved to a new location in memory.

```
1  for(outRow=0; outRow<OutputHeight; outRow++){
2    for(outCol=0; outCol<OutputWidth; outCol++){
3      for(outDep=0; outDep<OutputDepth; outDep++){
4        for(inDep=0; inDep<InputDepth; inDep++){
5          for(kernelRow=0; kernelRow<KernelHeight; kernelRow++){
6            inRow = stride * outRow + kernelRow - pad;
7            for(kernelCol=0; kernelCol<KernelWidth; kernelCol++){
8              inCol = stride * outCol + kernelCol - pad;
9              if( input is not in padded area ){
10               output[outDep][outRow][outCol] +=
11                 weights[outDep][inDep][kernelRow][kernelCol] *
12                 input[inDep][inRow][inCol];
13 }}}}}}}
```

Code 1: Basic Convolution

Put straight on to an FPGA, this algorithm will require the entire input feature map and all of the weights kernels to be loaded onto the FPGA at once, which is not feasible for a typically sized convolution layer. However, Zhang et al. describes a method of tiling the data so that the convolution can be performed in chunks and is therefore suitable for FPGA computation. The code is split into two parts. Code 2 is responsible for managing the data tiling and loading the tiles of data into the computation kernel. Code 3 shows the computation kernel that performs the tiled convoution computation.

Code 2 iterates through the same dimensions as the outer four loops in the Code 1, but is incremented by the tile size for that particular dimension. If an appropriate set of tile sizes can be found, the tiling approach can allow convolutions of various sizes to be performed using the same kernel, which leads to very efficient use of FPGA resources. Tiling also allows the computation to be performed using multiple hardware kernels in parallel.

```
1 for(outRow=0; outRow<OutputHeight; outRow+=outRowTileSize){
2   for(outCol=0; outCol<OutputWidth; outCol+=outColTileSize){
3     for(outDep=0; outDep<OutputDepth; outDep += outDepTileSize){
4       for(inDep=0; inDep<InputDepth; inDep += inDepTileSize){
5         LoadInputTile();
6         LoadWeightTile();
7         LoadOutputTile();
8 }}}}
```

Code 2: External Data Transfer

The computational kernel shown in Code 3 iterates through the tile and performs the convolution for the tiled region. The code is nearly identical to Code 1, but is performed only on one tile of data. Section 5.4 describes how this kernel can be further optimised.

```
1 for(outRow=outRowTileStart; outRow<outRowTileEnd; outRow++){
2   for(outCol=outColTileStart; outCol<outColTileEnd; outCol++){
3     for(outDep=outDepTileStart; outDep<outDepTileEnd; outDep++){
4       for(inDep=inRowTileStart; inDep<inDepTileEnd; inDep++){
5         inRow = stride * outRow + kernelRow - pad;
6         for(kernelCol = 0; kernelCol<KernelWidth; kernelCol++){
```

```
7           inCol = stride * outCol + kernelCol - pad;
8           if( input is not in padded area ){
9             output[outDep][outRow][outCol] +=
10              weights[outDep][inDep][kernelRow][kernelCol] *
11              input[inDep][inRow][inCol];
12  }}}}}}
```

Code 3: Computational Kernel

Because tiling allows for the data space to be broken down into parts, this convolution method avoids being limited by the memory capacity of the FPGA.

this convolution method is much more suitable for hardware implementation. Therefore, this algorithm was chosen to be implemented in HLS.

4.3 Fully Connected Layer

- Describe layer code
- Explain FC->CONV conversion
- Explain redesign, benefits of new implementation

4.4 Rectified Linear Unit Layer

- Describe layer code

4.5 Pooling Layer

- Describe layer code

4.6 Processing System (PS) Design

4.6.1 Zynq-7000 Operating System

When using SDSoC, there are three options for controlling the processor. The Zynq 7000 can run a Linux SMD kernel, another operating system called FreeRTOS, or the application can be run on "bare metal", meaning no OS is in place. Linux SMD (Symmetric MultiProcessing) is a basic Linux architecture, specialised for high performance on multi-core processors, and FreeRTOS is also designed to run multiple concurrent threads or tasks.

Running the application on bare metal would be the simplest option, and would use much less memory. However, an operating system would greatly simplify the transfer of data necessary for the planned SEU tests. Rather than having to transfer and parse data packets through a serial connection, an OS would be able to copy and store data in an organised file structure, making the process of feeding input data and extracting resultst data much more convenient.

The Linux kernel is very powerful, can take advantage of the huge library of Linux based tools, and makes it easy to write scripts that allow tests and other processes to be automated. FreeRTOS has a very small memory footprint and can provide faster performance. Both operating systems are specialised to run efficiently on the Zynq-7000s dual-core ARM processor. The Linux kernel was ultimately chosen, because it would be easier to use and the ability to easily write scripts was deemed a priority.

4.6.2 Software

TODO

- Describe software used to control system
- Perhaps describe various socDrawer scripts and programs

4.7 Programmable Logic (PL) Design

- Describe HLS, and alternative (VHDL/Verilog)
- Justify using HLS rather than VHDL/Verilog (faster development, etc)

4.8 PS and PL Interfacing

In order to for the Processing System to control and utilise the Programmable Logic, an interface is needed to transfer data between the two. The Zedboard offers bus architectures based on Advanced Microcontroller Bus Architecture (AMBA) protocol, designed by ARM. AMBA is commonly used as an interconnect in SoCs and was created for multi-processor designs. The particular specification used in Vivado and SDSoC is called Advanced Extensible Interface 4 (AXI4).

There are three different AXI4 interfaces availablle for selection. AXI-Memory Mapped (AXI-MM) master bus uses the full AXI4 interface and is able to transfer large amounts of data. AXI4-Lite is a version of AXI4 with a smaller memory footprint and is used for moving single values. AXI4-Stream is used for transferring data in bursts. In this project, the AXI4-Stream was not used as there was not found to be a need for burst data streaming, but the AXI4 and AXI4-Lite were both used.

The data that needed to be transferred to and from the hardware kernel were categorized as array data, which would be transferred using the AXI-MM, or single values, which would be transferred using the AXI4-Lite. The array data consists of the weights tile, the input tile, and the output tile. The single values are parameters of the convolution, including the start and end values of the tiles in each dimension.

The AXI-MM stores data sequentially, and is accessed with a pointer. In order to transfer multiple sets of data on the master bus, they can be stored sequentially at different offsets, and then the data can be accessed by setting the pointer at the correct offset with respect to the master bus base address. In order for the hardware functions to take advantage of this, the relevant offsets are sent to the FPGA as single values over the AXI4-Lite interface.

- Justify division of tasks
- Describe interface and related design choices

4.9 Soft Error Mitigation (SEM) Core Integration

- Describe SEM IP core (a core to simulate SEUs)
- Justify using SEM IP core rather than creating own test (simplicity, time, etc)

5 Implementation

This section will explore any relevant details that were not mentioned in Section 4: Design. This will include pseudocode or fragments of code where they are useful, and by the end of the section, the reader should have an approximate understanding of the code structure and the development process that took place.

5.1 Infrastructure Setup

The work was performed on a Ubuntu computer, set up with two main tools. The first tool to setup up was the Xilinx SDSoC development environment, which was used to design the system. SDSoC is able to optimise and compile C, C++ or OpenCL source code for a Zynq SoC. The compiler generates software for the ARM core and, using an HLS tool, a bitstream for the FPGA. This allows the user to design the entire system, easily accelerating functions with the FPGA. This facilitated rapid development and design of firmware for the Zynq SoC.

The other important tool to set up is called Caffe (or Convolutional Architecture for Fast Feature Embedding). This is a deep learning framework that can be used to design, train, optimise and test CNNs, with a focus on computer vision [15].

- Describe use of Caffe with reference to Section 4.1

5.2 Programming Language

All original code used in the project was written in C++. The hardware code was written in C++, with the addition of `#pragma` pre-processor commands to control the HLS. The software controlling the socDrawer consisted of python and C code, along with some shell scripts. Most of the software that needed to be edited in order to reconfigure the socDrawer was C code, so the python and shell scripts were largely left unchanged.

5.3 Development Methodology

Git version control was used extensively for the code in this project. When changes were made to the code, they were committed to the Git repository, along with comments briefly describing the

changes had been made. This was done for a number of reasons. Firstly, the repository serves as a backup. A copy of all code is stored remotely in the git meant that even if the physical machines crashed and lost all of their memory, the code could still be easily recovered, with little or no progress lost. Secondly, it allows the code to be shared between computers, meaning that the code can be reviewed or built upon on a personal computer away from campus whenever convenient Finally, the git repository acts as a record of all of the work completed. Using `github.com`, the history of commits and the corresponding messages is easily accessed and visualised.

The majority of the work was done in a computing lab on the 9th floor of the Electrical and Electronic Engineering building. Here, a Zedboard was set up so that it could boot from an SD card. The executable files generated by SDSoC were loaded on the SD card and transferred to the Zedboard. When the Zedboard had been power cycled and had finished computing, and serial connection between the Zedboard and a desktop computer facilitated commands to be sent to control the system, and results data to be sent back.

Since the CNN implementation was going to be implemented in software with one or two hardware functions, it was decided that a pure software implementation would be useful, both as an easy way to test the algorithms used in the full version, and to serve as a reference for results coming from the hardware accelerated implementation. Once this was complete, the Convolution function was moved into hardware. It was planned for the Fully Connected Layer to also be hardware accelerated, but time constraints prevented this from taking place.

The convolution function hardware was then intergrated into a hardware design with other IP blocks, including the SEM core, which facilitated interfacing with external software. This hardware design was then loaded into the socDrawer, described in Section **??** and the SEU tests were performed. More details on the results gathering and analysis are given in Section 7.

5.4 Hardware Optimisations

- Describe optimisations performed with HLS, i.e. pipelining, partitioning arrays, etc

6 Testing

The focus of this section is on the verification methods used to confirm that the system implementation was correct. The testing methods for individual layers will be analysed first, followed by tests for the whole network.

6.1 Layer Testing

- Describe test functions for layers
- Describe comparison to pure C++ implementation

6.2 Network Testing

- Describe test functions for Network
- Describe comparison to pure C++ implementation

7 Evaluation

- Summarise project specification requirements, and evaluate success on each

7.1 FPGA Implementation of a CNN

- How much of CNN was implemented
- Hardware resources used
- CNN performance
- Layer computation time

7.2 Fault Tolerance Investigation

- Describe implementation of data collection, i.e. SocDrawer, test scripts, data collection scripts, etc
- Analysis of data

8 Conclusion and Future Work

- Summary of report sections
- Summary of results and conclusion
- Suggestions for future investigations

References

[1] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biol. Cybernetics 36*, 1980.

[2] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein, A. C. Berg, and L. Fei-Fei, "Imagenet large scale visual recognition challenge," *Int. J. Computer Vision*, 2015.

[3] R. Nane, V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels, "A survey and evaluation of fpga high-level synthesis tools," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2015.

[4] N. Suda, V. Chandra, G. Dasika, A. Mohanty, Y. Ma, S. Vrudhula, J. sun Seo, and Y. Cao, "Throughput-optimized opencl-based fpga accelerator for large-scale convolutional neural networks," *FPGA*, 2016.

[5] J. Qiu, J. Wang, S. Yao, K. Guo, B. Li, E. Zhou, J. Yu, T. Tang, N. Xu, S. Song, Y. Wang, and H. Yang, "Going deeper with embedded fpga platform for convolutional neural network," *FPGA*, 2016.

[6] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," *NIPS*, 2012.

[7] Y.-L. Boureau, J. Ponce, and Y. LeCun, "A theoretical analysis of feature pooling in visual recognition," in *Proceedings of the 27th International Conference on Machine Learning*, 2010.

[8] C. Zhang, P. Li, G. Sun, Y. Guan, B. Xiao, and J. Cong, "Optimizing fpga-based accelerator design for deep convolutional neural networks," *FPGA,*, 2015.

[9] T. Chen, Z. Du, N. Sun, J. Wang, C. Wu, Y. Chen, and O. Temam, "Diannao: A small-footprint high-throughput accelerator for ubiquitous machine-learning," in *ASPLOS*, 2014.

[10] C. Farabet, B. Martini, P. Akselrod, S. Talay, Y. LeCun, and E. Culurciello, "Hardware accelerated convolutional neural networks for synthetic vision systems," in *ISCAS*, 2010.

[11] F. Wang and V. D. Agrawal, "Single event upset: An embedded tutorial," *21st International Conference on VLSI Design*, 2008.

[12] S. Habinc, "Suitability of reprogrammable fpgas in space applications," *Gaisler Research*, 2002.

[13] S. Habinc, "Functional triple modular redundancy (ftmr) - vhdl design methodology for redundancy in combinatorial and sequential logic," *Gaisler Research*, 2002.

[14] L. Rockett, D. Patel, S. Danziger, B. Cronquist, and J. Wang, "Radiation hardened fpga technology for space applications," *EEEAC paper 1305*, 2006.

[15] Y. Jia, E. Shelhamer, J. Donahue, S. Karayev, J. Long, R. Girshick, S. Guadarrama, and T. Darrell, "Caffe: Convolutional architecture for fast feature embedding," *arXiv preprint arXiv:1408.5093*, 2014.

[16] S. I. Venieris and C.-S. Bouganis, "fpgaconvnet: A framework for mapping convolutional neural networks on fpgas," *IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines*, 2016.

[17] B. Dutton, M. Ali, C. Stroud, and J. Sunwoo, "Embedded processor based fault injection and seu emulation for fpgas," *International Conf. on Embedded Systems and Applications*, 2009.

[18] M. Peemen, A. A. A. Setio, B. Mesman, and H. Corporaal, "Memory-centric accelerator design for convolutional neural networks," *ICCD*, 2013.

[19] V. Gokhale, J. Jin, A. Dundar, B. Martini, and E. Culurciello, "A 240 g-ops/s mobile coprocessor for deep neural networks," in *CVPR Workshops*, 2014.

[20] S. Chakradhar, M. Sankaradas, V. Jakkula, and S. Cadambi, "A dynamically configurable coprocessor for convolutional neural networks," in *ISCA*, 2010.

[21] Xilinx, *LogiCORE IP Soft Error Mitigation Controller v4.0*, 2013.

[22] S. Cadambi, A. Majumdar, M. Becchi, S. Chakradhar, and H. P. Graf, "A programmable parallel accelerator for learning and classification," in *19th international conference on Parallel architectures and compilation techniques*, 2008.

[23] Xilinx, *Zynq-7000 All Programmable SoC Overview*, 2016.