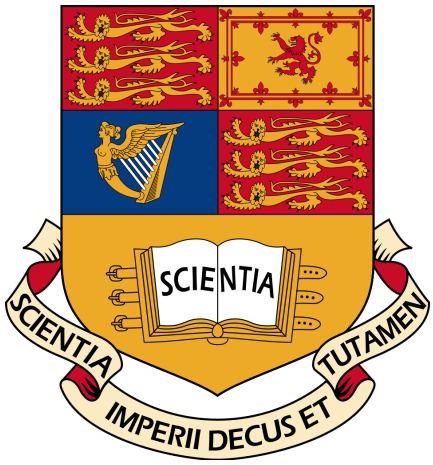
Imperial College London

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 described and analysed. 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

I would like to thank Dr Felix Winterstein for the enormous amount of help and guidance that he provided, particularly in the final stages of the project. I'm extremely grateful for the time he invested in helping me overcome the various obstacles that I encountered.

I would also like to thank Shane Fleming for allowing me to take advantage of the great work he put in to building the socDrawer, and for the crucial assistance he provided in setting it up to run my experiments.

Contents

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 optical text recognition. Neural Networks are being used in a wide range of other fields as well including speech recognition, natural language processing and text classification[3].

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 satellite 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 satellites, power is limited, and the communications equipment does not have a great data capacity. Therefore it is

expensive for the satellite to transfer full images to the base station. In addition to this, the link between the station and the satellite may only be available at certain points in the satellite'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 satellite. 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 2.6.1. These errors pose a risk that must be investigated and assessed before the CNN satellite 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 5.2.

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 development 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 - CNN Design: The design of the network is described in this section from the top level, as well important aspects of the low level design.

Section 5 - System Design: The decisions taken in the system design process are described and explained in this section. This section focuses on the embedded system, and how the software and hardware components were conceived.

Section 6 - Implementation: This section illustrate how the system was implemented, aiming to provide the reader with an understanding of the development process.

Section 7 - Testing: The testing methodology used to verify various components of the system is detailed in this section.

Section 8 - Evaluation: The focus of this section is the result of the investigation, including an evaluation of the system's performance.

Section 9 - 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, the basics of machine learning, and CNN structures will be described. FPGA implementations of CNNs, and the issues surrounding the use of FPGAs in space will also 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, controlling 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. Additionally, a lack of specialist knowledge can lead to an extremely inefficient design.

High Level Synthesis (HLS) tools are a solution to this problem. An HLS tool is a software application 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 [4]. 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 Machine Learning

The core functionality of a machine learning system is to perform tasks on new, unseen data based on a large set of previously seen training data. The training data will be split up in to different classes and the system uses this training data to build up a model that predicts which class the new

data best fits in to. This can be applied to many situations, such as classifying images, recognising speech patterns, or piloting a vehicle. Machine learning allows an electronic system to be put in a brand new situation and use past experience to make a decision.

A machine learning system takes an array of values as an input. If the input is a image, the vector values will represent pixels. This is done for a grayscale image by representing each pixel's intensity with an integer on a scale from 0, for a black pixel, to 255, for a white pixel. If the image is coloured, there will be three vector values for each pixel, with the brightness in red, green and blue in the pixel being represented as separate values. Parameters called weights are used to compare the input with the training data, and system will output a set of scores, which represent the likelihood that the input is a member of each class in the training data. The input is assigned the class with the highest score.

The performance of such a system is based on a process called training. Training is an iterative process, traditionally performed with the following method, called supervised learning. A piece of training data is fed through the system and is assigned a score at the output. The scores are compared with the expected results and the weights are adjusted according to the error. The process is repeated until the error is sufficiently small for each piece of data in the training data set. The best results are produced by training with a large set of data, with lots of variety within each class, e.g. images of the similar objects with different angles, lighting, etc.

2.2.1 Linear Classifier

One of the simplest types of machine learning structure is the linear classifier. These treat each value in the input vector as the position of the input in a separate dimension, with the length of the vector N being the total number of dimensions. The linear classifier works by forming an N-dimensional space and dividing it up. The number of subspaces resulting from these class boundaries matches the number of classes that the classifier is trained to recognise, and the class boundaries are placed by the training process. New data is treated as a point in the N-dimensional space and is assigned the class of the subspace that it falls in to.

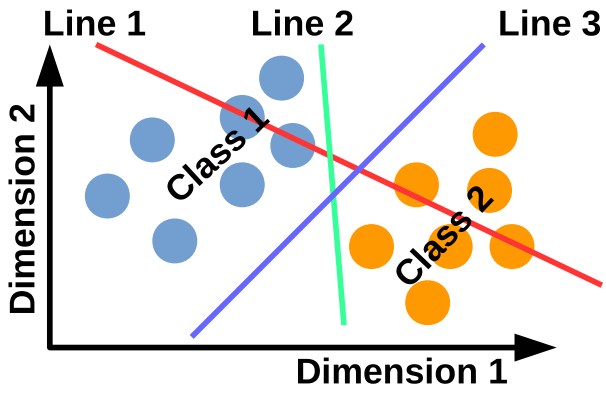


Figure 1: A linear classifier, with multiple class boundaries of differing quality shown

Figure 1 shows this process taking place with two classes and a two dimensional input space. A good class boundary should separate the classes completely, and should find the maximum margin of separation, so that the chances of new data falling on the wrong side of the boundary is minimised. Clearly, Line 1 is a poor class boundary as the classes are not separated. Line 2 is acceptable, but only separates the classes by a small margin and is therefore is far from optimal. Line 3 is the best class boundary as it separates the classes with the largest margin, allowing a wider range of inputs to be correctly classified.

As previously stated, Linear Classifiers are a simple solution and are useful for fast classification, but have drawbacks associated with them. They are only designed as a binary classifier, discerning between only two problems. Algorithms exist to apply a binary classifier to multi-class problems, but these reduce the performance significantly. Traditional linear systems such as this are unable to accurately account for high levels of variation in inputs, which led to the development of non-linear machine learning systems, such as Neural Networks.

2.2.2 Artificial Neural Networks

Artificial Neural Networks are based on the structure of biological brains. A brain is made up of billions of interconnected building blocks called neurons. The structure of a neuron and an accompanying mathematical mode are shown in Figure 2. Synapses act as the connections between neurons, supplying the dendrites with information. The dendrites act as an input, carrying the information to the cell body. The computation in the cell body is a weighted summation of the dendrite inputs, and the activation function determines the activation of the output neuron[5]. Many activation functions exist in artificial models, but usually they define a threshold that must

be surpassed by the computation result before it is passed on to the output, and these activation functions are the source of non-linearity in the network.

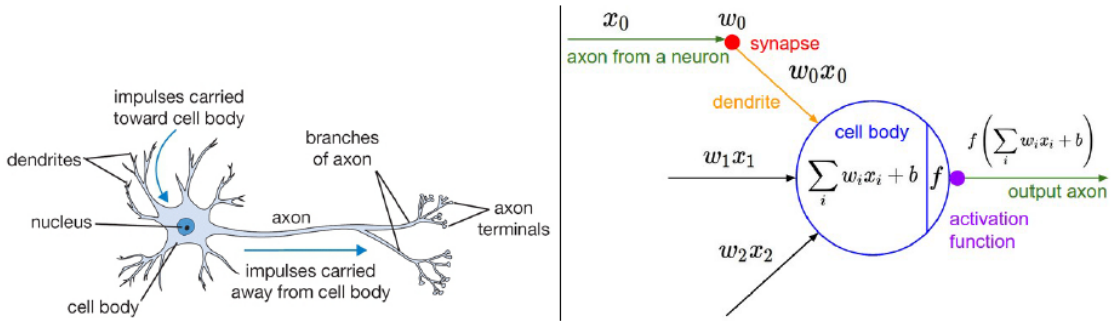


Figure 2: A neuron, and a mathematical model of a neuron[5]

These neurons are connected together into a network as shown in Figure 3. The neurons are arranged into layers, and connected to each neuron in the next and previous layers. The network has a layer for the input values and for the output values, and intermediate layers which are referred to as hidden layers since their only interaction is with the other layers. The number of hidden layers can vary, but has a big impact on how accurate the classification can be, and how many classes can be accurately classified. Deep networks with 10-20 layers have been shown to perform significantly better than shallow networks[6].

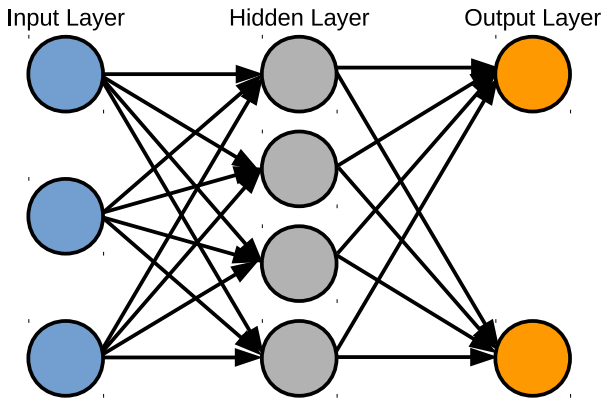


Figure 3: Simple Neural Network with one hidden layer

Neural Networks are trained in a slightly different way to other machine learning systems, using a method called back-propagation. Inputs are propagated forwards through the network until the output is reached. The resulting output values are each compared against the desired values and an error value is calculated for each output neuron. The error values are then propagated

backwards through the network and the weights of each neuron are adjusted accordingly. This process is repeated until the network is sufficiently trained.

2.3 Convolutional Neural Networks

CNNs are a specialised type of Neural Network, designed to be deep networks that model the visual perception of animals. They are usually designed to take images as inputs. 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 example, CNNs built for the ImageNet Large Scale Visual Recognition Challenge (ILSVRC) from 2012 to 2014 used a training dataset of approximately 1,281,167 images, grouped into 1000 classes[7]. In the case of 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, activation function, pooling, and fully connected layers make up a typical CNN model, with convolution and fully connected being the most important. Convolution and fully connected 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 convolution 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 [3]. 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 fully connected 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 4.

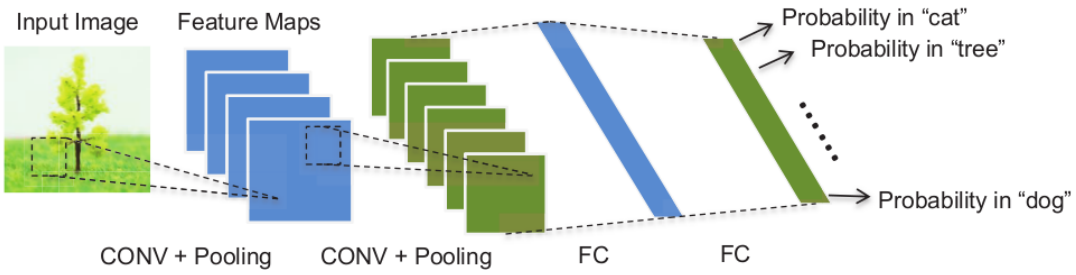


Figure 4: A typical CNN structure from the feature map perspective [8]

2.3.1 Convolution Layer

The convolution 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 feature maps with $K \times K$ convolution kernels to get an output feature neuron value as shown in Equation 1

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

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 [3].

Each position that the kernel takes corresponds to one output position and is referred to as an input region. This process can be thought of as the weights kernel scanning over the input feature map with, filtering each input region, and passing the result to the output. Many convolutions also include features called zero padding, adding layers of zeros around the input, and stride, moving the kernel over the input in steps. Zero padding can ensure that features on the edge of the feature map do not have a reduced contribution to the output, and stride reduces the computational cost of the layer by reducing the number of weights kernels needed, and reducing the dimensions of the output feature map.

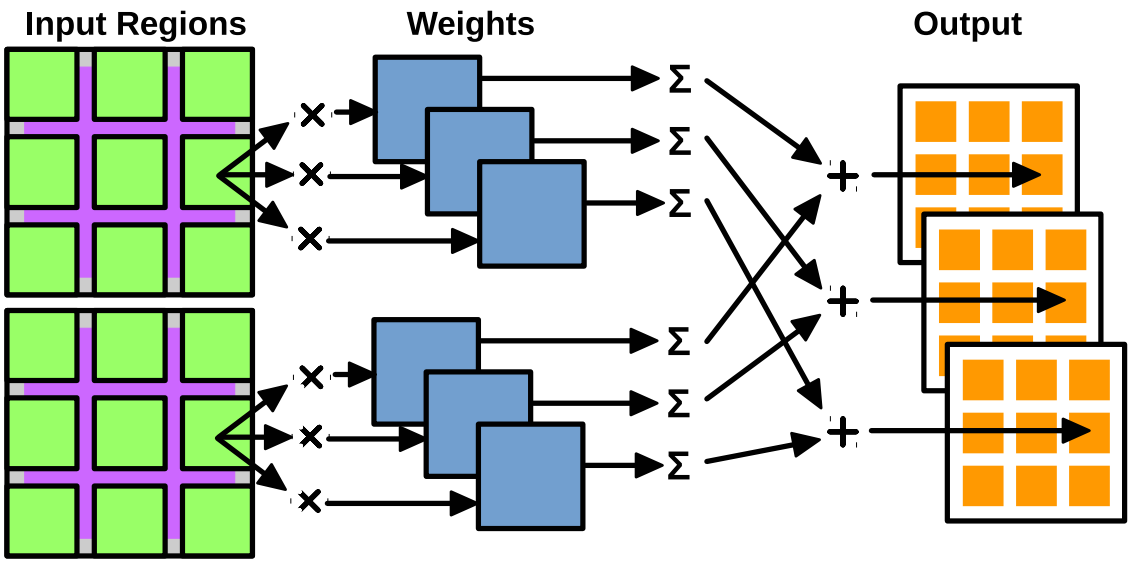


Figure 5: A visualisation of a convolution computation for one output location

Figure 5 shows a visualisation of the convolution operation. The input feature maps are shown on the left in purple, surrounded by grey zero padding. The green squares show the input regions, the positions of the weights kernel as it scans across the input. The element-wise product of the input region and the kernel is found and summed, and then added to the relevant output value. Each orange square in the output feature maps represents a single value. In this diagram, the input has two layers, and the output has three layers. It can be seen that the computation results for multiple input layers are added together to form the final output values. For multiple output layers, the computation process is repeated with new weights kernels. The number of weights is the product of the number of input layers and the number of output layers. Note that the diagram only depicts the computation for a single output location. The computation is repeated for every output location.

2.3.2 Activation Functions

Convolution layers are followed by an activation function layer. This layer can be thought of as a decision, based on the output of the convolution layer, on to what extent each neuron in the next layer has been activated. The convolution 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 [9]. Hence, Rectified Linear Unit (ReLU), defined as $y = max(x, 0)$, has become the

popular activation function. It's simplicity means that it converges faster in training, and has less computational complexity than exponent functions in tanh and sigmoid. It also simplifies hardware implementations of the layer.

2.3.3 Pooling Layer

Spatial pooling, also known as sub-sampling is used to reduce the volume of data being passed deeper into the network. As shown in Equation 2, pooling computes the maximum value within a localised $K \times K$ pooling region of the feature map, which also provides a form of translational invariance, i.e. the ability of the network to identify a feature regardless of its position in the image, in addition to reducing computational load for future layers [10].

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

The pooling layer is sometimes computed by computing the average of each input region, but in this project only maximum pooling was used. Figure 6 shows the operation of a max pooling layer graphically. The pooling layer is important as it reduces the dimensionality of lower-level features without losing important information, thus reducing the computational cost of the network [3]. A paper by J. Springenberg, A. Dosovitskiy, T. Brox and M. Riedmiller proposes an "all convolutional network" in which the pooling layers are eliminated by increasing the stride in some convolution layers without any reduction in performance[11]. Based on these results, pooling layers may soon stop appearing in new CNN architectures.

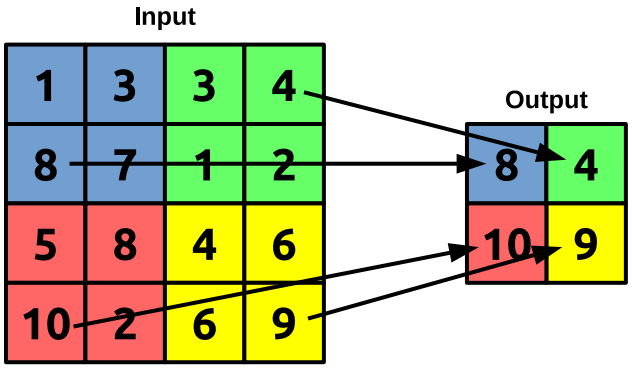


Figure 6: A visualisation of the max pooling layer

2.3.4 Fully Connected Layer

A fully connected 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 [3].

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

This equation is visualised in figure 7. As the name suggests, every input neuron is connected to every output neuron and each connection has a unique weight. In this diagram, each arrow represents a multiplication with the corresponding weight. It can be seen that the fully connected is essentially the same as a layer of the standard artificial neural network described in 2.2.2. The outputs of the fully connected layer traverse through ReLU based activation function to the next fully connected layer or directly to a Linear Classifier as described in 2.2.1. The Linear Classifier will take the output feature map of the network use it to determine the scores for each class.

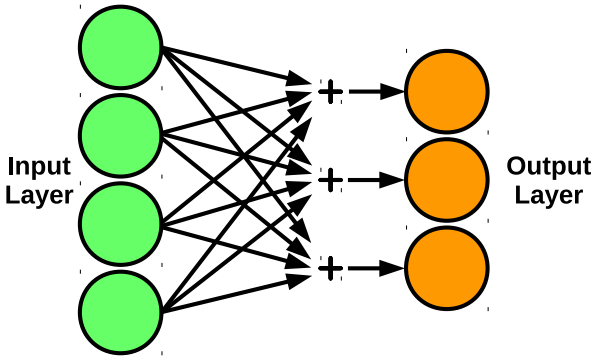


Figure 7: Connections in a fully connected layer

2.4 VGGNet

One of the most famous CNN architectures is called VGGNet, and was developed by Karen Simonyan and Andrwe Zisserman of the Visual Geometry Group at the University of Oxford in 2014[12]. This network was the runner up in the ILSVRC 2014 competition, and was showed that network depth is a key component for good performance.

The architecture, consists many convolution layers with 3x3 weights kernels, 1 zero padding and 1 stride, each followed by a ReLU Layer. These parameters were selected, in part, because they produce an output that is exactly the same size as the input. The output width or height of a convolution is calculated with the formula $outputSize = (inputSize + 2 * zeroPadding - kernelSize)/stride + 1$. By substituting in the values described above, the formula becomes $outputSize = (inputSize + 2 * 1 - 3)/1 + 1$, which reduces to $outputSize = inputSize$. This property allows convolutions to be repeated without having to manage changing layer sizes between every layer, greatly simplifying the design of a deep network.

These convolution and ReLU units are repeated in five groups of equally sized layers, each group having a smaller height and width but greater depth than the previous layer. These groups are separated by max-pooling layers, with 2x2 kernels, no zero padding and stride, in between groups. Then the network is ended by three fully connected layers and finally a softmax classifier, which converts the scores at the output of the last fully connected layer and converts them into probabilities. VGG-16, the architecture show in Figure 8, has 16 layers with weights - 13 convolution layers and 3 fully connected layers.

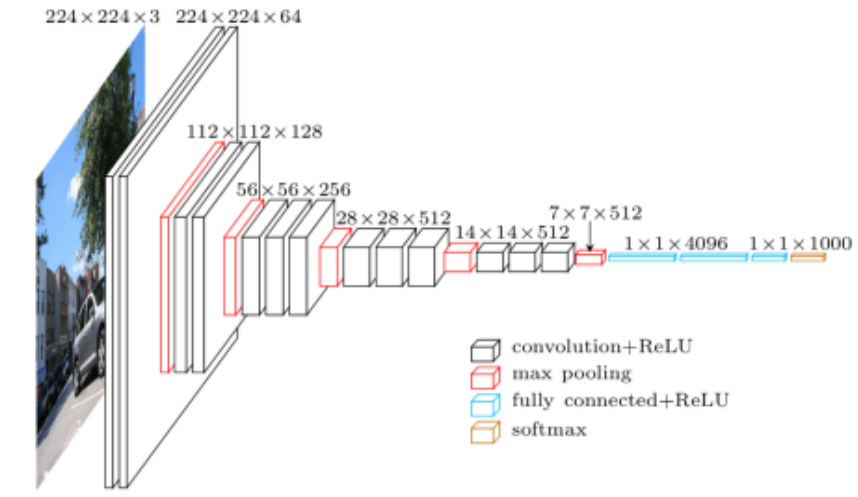


Figure 8: The VGGNet-16 Architecture[13]

The VGGNet architecture is appealing to be implemented in hardware, because of its very uniform structure, performing only 3x3 convolutions and 2x2 pooling. VGGNet's weak point are that it is very slow to train, and that it is expensive to evaluate, with fully connected layers that occupy a lot of memory. Training the network is outside the scope of this project, so this is not a major concern and the simplicity and uniformity of the network structure bring benefits that outweigh the potential drawbacks.

2.5 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.

2.5.1 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 convolution 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 convolution weights, 10-bit precision for fully connected weights, with less than 1% compared accuracy loss compared to full precision weights [3].

2.5.2 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 convolution layers. These layers typically make up around 90% of the CNN's computational workload. The multiply and accumulate nature of the convolution 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 [3]. Many CNN accelerators use 16-bit fixed-point numbers instead of floating-point numbers to represent

weights and data [14][15][16], 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.

2.6 FPGAs and Space Applications

2.6.1 Single Event Upsets

Electronic circuits are sensitive to high-energy ionizing radiation, which can induce errors called Single Event Upsets (SEUs) [17]. 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 [18].

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 read back 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 [19].

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 [18]. 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 **??**.

2.6.2 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 read back 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 [18]. 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 [19]. 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 [20]. 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.

2.6.3 Soft Error Mitigation (SEM) Core

The Soft Error Mitigation core is a Xilinx Intellectual Property (IP) core that can be used to detect, correct and classify SEUs. More importantly, it also provides error injection functionality[21]. Once added to a hardware system, it is able to emulate SEUs, allowing the evaluation of the SEU mitigation capabilities of a hardware design. This is a valuable feature as it allows a much cheaper, safer and more controlled method of testing the system than using a radiation source to induce the errors.

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 three set of functionalities - those that are necessary for the embedded CNN implementation to be considered successful, those that are necessary for the SEM testing to be considered a success, 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 - CNN

These items are necessary implementations features related to the implementation of the CNN features on the hardware system

3.1.1 Embedded System running CNN layers

The system should be able to run layers of a CNN on the target platform, described in Section 1.2 as the Xilinx Zedboard. The compilation of the program will produce an executable file in either .bin or .elf format, and this should run on the hardware platform, either printing a stream of output that can be read through a serial connection or storing the results in a file .log or .txt to be read later.

3.2 Necessary Functionality - SEM Tests

These items are necessary implementations features related to the testing of the system with SEM core

3.2.1 Convolution Layer Implemented in Hardware

As previously explained in Section 2.5.2, the convolution layer represents the biggest portion of the computational workload, and is thus the primary focus of the examination. It is therefore important that the convolution layer is at least partially implemented in hardware to allow testing with the SEM core. By analysing how the convolution layer is affected by SEUs, information about the effects on the rest of the network can be inferred.

3.2.2 SEM Testing

In order to perform the SEU emulation with the SEM core as described in Section 2.6.3, it is necessary to integrate the SEM core into the the digital hardware design. This requires the designed hardware kernels to be integrated into a larger system that includes the SEM core and the necessary interfaces to software required to set up and use the SEM core's SEU emulation functionality. The final component of the SEM tests is getting the design working with the socDrawer test rig. This would involve getting familiar with the test scripts and making changes to the existing scripts where necessary. It also requires obtaining the correct input files. This is detailed further in Section 6.3.

3.3 Desirable Functionality

3.3.1 Full CNN Implementation

It would be desirable for a full CNN model to be implemented. With this in place, it would be possible to run the neural network with emulated SEU errors and compare the performance and accuracy of the system to that with no errors, providing and opportunity to deeply investigate the effects of SEUs on the system.

3.3.2 Hardware Optimisation

Optimising the hardware blocks to run as quickly as possible is desirable. This would allow the test to be completed in a shorter period of time, potentially allowing more test cases to be investigated. It is also desirable to create hardware blocks that use the FPGA resources efficiently. The FPGA has limited resources with which to implement hardware designs, and designing compact, efficient hardware blocks allows for more functionality to be potentially implemented.

4 Convolutional Neural Network Design

In this section, the design of he Convolutional Neural Network implementation will be discussed. This includes an overview of the Caffe CNN framework, network infrastructure and finally the design of each layer type.

4.1 The Caffe CNN Framework

Caffe is an open-source framework for deep learning models, particularly Convolutional Neural Networks. This framework facilitates the easy design, training and testing of network models, and provides access to a "Model Zoo" where several popular CNN models can be downloaded and tested[22]. These models include pre-trained weights, allowing the user to bypass the time consuming process of organising a training dataset and running the training process locally. This model zoo contains a pre-trained version of the VGGNet model discussed in Section 2.4.

In order to avoid repeating the work of other academic groups, to significantly shorten the development time-frame of the CNN system, and to ensure compatibility with pre-trained network models, the Caffe framework was used as reference code for the CNN implementation. In order to ensure that pre-trained weights from the model zoo would be compatible with the CNN implementation, the Caffe's source code was analysed. The concept of "Blobs", as described in Section 4.2.1, as a homogeneous data carrier for the network was adapted from Caffe in order to guarantee compatibility, as was the generic layer structure described in 4.2.2.

4.2 Top Level Network Design

The neural network was controlled by a single class called `Net`, which contained a vector of layers, a vector of inter-layer Blobs, and two Blobs for input and output. The net was hard coded to initialise itself to the 16 layer VGGNet described in Section 2.4. Each layer was initialised with the relevant parameters, including a layer name, and the name of the bottom and top Blobs and added to a list of layers. Once this is complete, the Blobs are generated. Since the VGGNet model is made up of sequential layers without any branching, this is simply done by iterating through the layers, checking that the bottom and top Blob names match, and then constructing a new Blob instance. The Blobs are stored in a vector to be easily accessed.

Once the construction is complete, the network is initialised. Based on the construction parameters of a blob and the parameters of the input blob, the first layer is set up, and its top blob is reshaped to be the correct size. This means that the next layer's set up and top Blob reshaping can take place. The set up process is propagated through the rest of the network in this fashion.

The final stage of the network operation is to perform the forward computation. Data is read from the input blob and propagated through the network until the output Blob. This process runs the full classifier computation.

4.2.1 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 homogeneous 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. Most Blobs would only use three of these dimensions, but weights Blobs in convolution and fully connected layers use four layers. Additionally the fourth dimension allows multiple sets of input data to be propagated through the network together. This is referred to as batch processing. 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. To keep the layer designs as simple as possible, the layers were designed to accept only square Blobs as inputs and outputs. Therefore when describing a Blob, "size" refers to both the height and width of the Blob.

4.2.2 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 easily 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 would 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 family 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 do not 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 9. With a small extension, it would not 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.

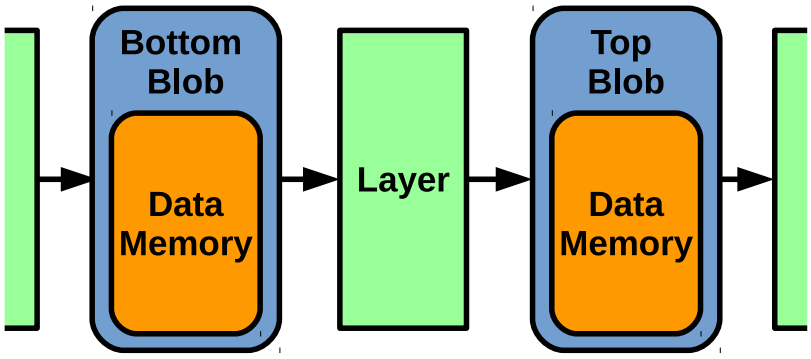


Figure 9: A layer within the network structure, showing the top and bottom Blobs and their respective DataMemory objects

The `LayerSetUp` function is run after the layer is originally constructed. Each layer class has a constructor that will initialise most of its parameters 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 output 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 2.3. 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.

4.3 Convolution Layer

The convolution layer is the most important layer in this network model for the purposes of this project, as it is where the bulk of the computation takes place, and is therefore most likely to be disrupted by SEUs. Therefore, this part of this layer had to be implemented in hardware to investigate the effects. First, two approaches to performing 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. Once implemented and compared, it was clear that the intuitive approach was much more suitable for an FPGA implementation.

4.3.1 Matrix Multiplication Method

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

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.

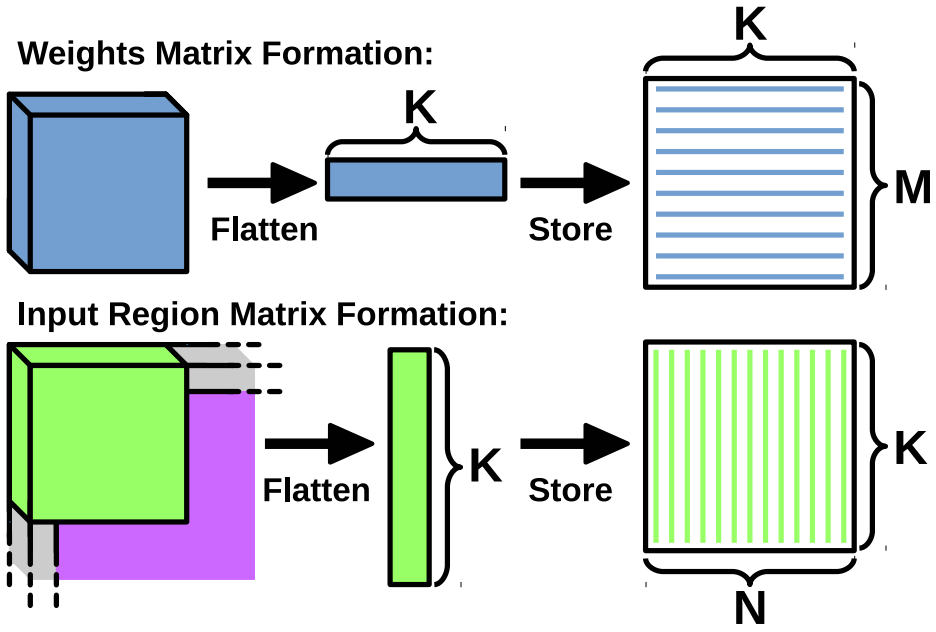


Figure 10: A visualisation of the reshaping process

Figure 10 shows 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. Once the matrices are formed, 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 β 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 down into individual row and column multiplications, but the process of moving the input regions to a new memory space will be expensive in terms of both computational cost and memory footprint, so it is not well suited for an embedded system.

4.3.2 Tiled Convolution Method

Directly interpreting the convolution algorithm as described in 2.3.1, the pseudo-code 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. The formula $inRow = stride * outRow + kernelRow - zeroPadding$ is used to calculate the input row, and the same formula is used for finding column. 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 pseudo-code 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.

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

```
for(outRow=0; outRow<OutputHeight; outRow++){
 for(outCol=0; outCol<OutputWidth; outCol++){
  for(outDep=0; outDep<OutputDepth; outDep++){
   for(inDep=0; inDep<InputDepth; inDep++){
    for(kernelRow=0; kernelRow<KernelHeight; kernelRow++){
     // calculate inRow
     for(kernelCol=0; kernelCol<KernelWidth; kernelCol++){
      // calculate inCol
      if( input is not in padded area ){
       output[outDep][outRow][outCol] +=
        weights[outDep][inDep][kernelRow][kernelCol] *
        input[inDep][inRow][inCol];
}}}}}}}
```

Code 1: Basic Convolution

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 convolution 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.

```
for(outRow=0; outRow<OutputHeight; outRow+=outRowTileSize){
 for(outCol=0; outCol<OutputWidth; outCol+=outColTileSize){
  for(outDep=0; outDep<OutputDepth; outDep += outDepTileSize){
   for(inDep=0; inDep<InputDepth; inDep += inDepTileSize){
    LoadInputTile();
    LoadWeightTile();
    LoadOutputTile();
}}}}
```

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 **??** describes how this kernel can be further optimised.

Because tiling allows for the data space to be efficiently broken down into parts, this convo-

```
for(outRow=outRowTileStart; outRow<outRowTileEnd; outRow++){
 for(outCol=outColTileStart; outCol<outColTileEnd; outCol++){
  for(outDep=outDepTileStart; outDep<outDepTileEnd; outDep++){
   for(inDep=inRowTileStart; inDep<inDepTileEnd; inDep++){
    // calculate inRow
    for(kernelCol = 0; kernelCol<KernelWidth; kernelCol++){
     // calculate inCol
     if( input is not in padded area ){
      output[outDep][outRow][outCol] +=
       weights[outDep][inDep][kernelRow][kernelCol] *
       input[inDep][inRow][inCol];
}}}}}}
```

Code 3: Computational Kernel

lution method avoids being limited by the memory capacity is much more suitable for hardware implementation. Additionally, a computational kernel of the same size can be reused for different convolution sizes. This eliminates the need to have multiple convolution hardware blocks of different sizes implemented independently on the FPGA, saving valuable hardware resources. Therefore, this algorithm was chosen to be implemented in HLS.

4.3.3 Layer Setup

To set up the layer, values for zero padding, output depth, stride and kernel size parameters are provided. Using these parameters and reading the dimensions of the input, the output height and width are calculated using Equation 4.

$$outputSize = \frac{inputSize + 2 \times zeroPadding - kernelSize}{stride} + 1 \tag{4}$$

If $outputSize$ is not an integer, then combination of the input size, zero padding, kernel size, and stride is not valid to perform a convolution. This result is checked, and then the output Blob is reshaped to have the dimensions $(outputDepth \times outputSize \times outputSize)$. Finally the weights Blob is reshaped to have dimensions $(outputDepth \times inputDepth \times kernelSize \times kernelSize)$ and is filled with zeros to be initialised.

4.4 Fully Connected Layer

The fully connected layer is the only layer apart from the convolution layer with a large computational footprint. The fully connected layer does perform as many computations, but has a larger memory footprint, partly because no weights are re-used between computations. It was hoped that the impact of SEUs on this layer would be investigated as well, so a hardware implementation was designed. Similarly to the convolution layer, two software algorithms were implemented and compared before deciding on version to implement in hardware.

4.4.1 Vector-Matrix Multiplication Method

Each output neuron of the fully connected Layer, is assigned a weighted sum of the outputs of every neuron in the input layer, as described in Section 2.3.4. This can be computed by considering the input values as a vector and the weights as a matrix, and performing a vector-matrix multiplication. The input is given as a vector of K values. Fully connected layers produce vectors as outputs, but the outputs of other layer types is flattened to create this input vector. The weights are given as a $K \times N$ matrix, where N is the number of output neurons. The output is then found with $\mathbf{C} = \mathbf{A} \times \mathbf{B}$, where $\mathbf{C}$ is the output vector of length N, $\mathbf{A}$ is the weights matrix, and $\mathbf{B}$ is the vector of input values.

If the network computation is being performed in batches (i.e. multiple sets of input data are being propagated through the network together), every input set in the batch can be computed at once by combining the input and output vectors. This results in matrices of size $B \times K$ and $B \times N$ for the input and output respectively, where B is the number of input sets in the batch.

The vector-matrix multiplication is the intuitive solution to computing the fully connected layer, as the computation follows the description of the fully connected layer's operation - each output value is a weighted sum of the inputs. By treating the vectors as matrices with size 1 in one dimension, the vector-matrix multiplication is evaluated in the same way as a standard matrix-matrix multiplication. Conveniently, this means that the method will work for batches of data without any modifications beyond changing the matrix dimensions. As described in Section 4.3.1, this method of computation can take advantage of the `gemm` function to increase performance. Once again, this is ideal for a software implementation, but is not suitable for an embedded system with limited computational resources.

4.4.2 Dot Product Convolution Method

An alternative method of computing the fully connected layer is to compute it with a convolutional kernel. For this, we consider the inputs as a vector of length N_{if}. The weights are divided into N_{of} vectors of length N_{if}, where N_{of} is the length of the output, and one vector is assigned to each output feature. Equation 4 shows the formula for computing one output value in the fully connected layer. This is equivalent to a dot product of in, the input vector, and wt_{f_o}, the weights vector corresponding to the output feature.

$$out[f_o] = \sum_{f_i=0}^{N_{if}} wt_{f_o}[f_i] \times in[f_i] = wt_{f_o} \cdot in \tag{5}$$

The next thing to consider is how to perform a convolution on these flattened layers. As previously explained, the convolution operation scans a weights kernel over the input and at each position, an output value is assigned the sum of the element-wise product of the weights kernel and the input region. Equation 6 shows a formula for finding the convolution with a flattened input and a flattened weights kernel stored in a vector of length K, using $zeroPadding = 0, stride = 1$

$$(wt * in)[n] = \sum_{k=0}^{K} wt[k] \times in[n + k - 1] \tag{6}$$

The size of the convolution output is equal to the number of input regions that the kernel scans over. Thus, by setting $K = N_{if}$, convolution will produce only one value and Equation 7 shows how the formula reduces to a dot product[23]. As shown in Equation 5, the dot product is equivalent to finding the output value of a fully connected layer. Therefore, by setting the size of the convolution kernel to match the size of the input, we can compute a fully connected layer with the convolution computation.

$$(wt*in)[n] = (wt*in)[1] = \sum_{k=0}^{N_{if}} wt[k]\times in[1+k-1] = \sum_{k=0}^{N_{if}} wt[k]\times in[k] = wt\cdot in = out[f_o] \tag{7}$$

As discussed in section 2.4, VGG net consists of five sets of convolution layers followed by three fully connected layers. Therefore, the convolution-based fully connected layers will have two kinds of inputs. The bottom of the layer will be either the output of the convolutional part of the network or the output of another fully connected layer. In order to form the dot product, the size of the weights kernel must match the size of the input Blob. At the end of the convolutional

part of the network is a pooling layer that produces 512 different 7×7 feature maps at its output. Therefore, the first fully connected layer must use a 7×7 weights kernel. The fully connected layer produces a vector of values at its output. Therefore, any fully connected layers after the first on will treat their input as a set of 1×1 feature maps and will have a matching kernel size. In the VGGNet model, this applies to the final two layers.

A convolution is an ideal way to compute the result because the fully connected layer can be easily computed in uniform chunks, perfect for FPGA computation. In addition, the convolution layer has already been designed for the convolution layer, and any optimisations implemented in the design of the convolution layer can improve the performance of the fully connected layer with no extra work. Therefore, this method was selected to be implemented in hardware.

It is important to note that although they are both valid ways of calculating the result, the Vector-Matrix Multiplication and Dot Product methods use the weights in a different order, and one method cannot use weights trained for the other method without adjusting the order in which the weights are accessed. This problem is overcome by treating the weights as a matrix as described in the Vector-Matrix Multiplication method and using the transpose of this matrix in computations. The Caffe framework uses the Vector-Matrix Multiplication method to compute the results of the fully connected layer, so the Dot Product method implementation was modified to access the weights as if they were transposed. In doing this, consistency with the Caffe framework could be maintained, allowing the network to continue using pre-trained weights from Caffe's model zoo.

4.4.3 Layer Setup

The only input parameter required for setting up the fully connected layer, apart from the input Blob, is the output length. The weights Blob is given the dimensions $(outputLength \times inputLength \times inputSize \times inputSize)$ and the output Blob is reshaped to be a vector of length $outputLength$.

4.5 Rectified Linear Unit Layer

The ReLU layer was the easiest to design. Code 4 shows that the layer output is formed by iterating through the input and copying each value. If a value is found to be less than zero, a zero is copied into the output location instead. Due to the simplicity and very low computational cost of this layer, there was no need to consider a hardware implementation. In fact the process of transferring

the data to and from programmable logic would be more expensive than simply implementing the whole layer in software.

```
1 for(index = 0; index < inputLength; index++){
2  output[index] = max(0, input[index]);
3 }
```

Code 4: ReLU layer computation

4.5.1 Layer Setup

The setup for this layer is simple. No input parameters are needed to set the layer up, apart from the dimensions of the input Blob. The output Blob is reshaped to match the dimensions of the input Blob and the setup process is over.

4.6 Pooling Layer

The pooling layer was designed with a series of nested `for` loops, as shown in code 5. The program iterates through the layers, rows and columns of the output, and for each output location, the maximum value in the corresponding input region is determined. This lower boundary of an input region is calculated as $inStart = max(0, outIndex * stride - pad)$, and the upper boundary is calculated as $inEnd = min(kernelStart + kernelSize, inputSize)$. The input region is iterated through to assign the maximum value to the output location

```
1  for(dep = 0; dep < depth_; dep++){
2   for(outRow = 0; outRow < outputSize; outRow++){
3    for(outCol = 0; outCol < outputSize; outCol){
4     // calculate boundaries of  pooling region
5     for(inRow = inRowStart; inRow < inRowEnd; inRow++){
6      for(inCol = inColStart; inCol < inColEnd; inCol++){
7       if (input[dep][inRow][inCol] > output[dep][outRow][outCol]){
8        output[dep][outRow][outCol] = input[dep][inRow][inCol];
9       }
10 }}}}}
```

Code 5: Max Pooling Computation

4.6.1 Layer Setup

The setup for this layer is similar to that of the convolution layer without any weights. The input parameters are zero padding, stride and kernel size parameters are provided, and the output size is calculated with the Equation 4 from Section 4.3. If the result is not an integer, the combination of the input parameters is not valid. The output Bloc is reshaped to $(outputDepth \times outputSize \times outputSize)$.

5 System Design

This section will provide an overview of the system design. This will include the Processor System software, the Programmable Logic and the interface between these two components of the system.

5.1 Processing System (PS) Design

The ARM CPU on the Zynq-7000 provides a processing system on the Zedboard enabling the embedded system to run software. This software consists of an operating system and Application Software, i.e. software that has been designed and built specifically for this project.

5.1.1 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 results 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-7000's 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.

5.1.2 Application Software

Within the Linux kernel, a software host code is used to run the HLS kernel. This program acted partially as a software wrapper for the kernel computation, configuring and setting the inputs, running the kernel, and receiving the output. This program also served the purpose of the test kernel, being run to perform the test for each of the SEUs introduced to the system by the SEM core. The function of the host code is summarised in Code 6.

```
1  Read linear address from input;
2  Set constants and address constants;
3  Set and initialise weights and inputs arrays;
4  Set up outputs pointer;
5  Set AXI4-Lite registers;
6
7  Initiate hardware kernel
8  while(kernel is running){
9   block;
10 }
11
12 errorCount = 0;
13 Open golden example file;
14 for(index = 0; index < outputLength; index++){
15  Get golden[index];
16  if(output[index] != golden[index]){
17   errorCount++
18 }}
19 print("<linear address>, <errorCount>");
```

Code 6: Host code for the hardware kernel

Lines 1 to 5 are setting up the inputs and interfaces of the kernel. Lines 8-10, deal with the running of the kernel itself, using a while loop to delay the program until the kernel has finished running. The final lines from 12 to 19 read a "golden" example, i.e. a ground truth example that the output can be verified against, one value at a time and compare it to the corresponding kernel output value. A counter keeps track of the number of discrepancies, and finally the code prints a message. This message shows the linear address at which the SEU was emulated in this test run, and the number of incorrect caused by this SEU, an indication of the severity of the SEU fault. Section **??** will describe how these output messages are collected and analysed.

5.2 PS and PL Interfacing

In order 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 available 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 AXI-MM and AXI4-Lite are both bi-directional. The data that needed to be transferred to and from the tiled convolution kernel, as described in Section 4.3.2 was categorized as either array data, which would be transferred using the AXI-MM, or single values, which would be transferred using the AXI4-Lite. The information carried in each interface is shown in Figure 11.

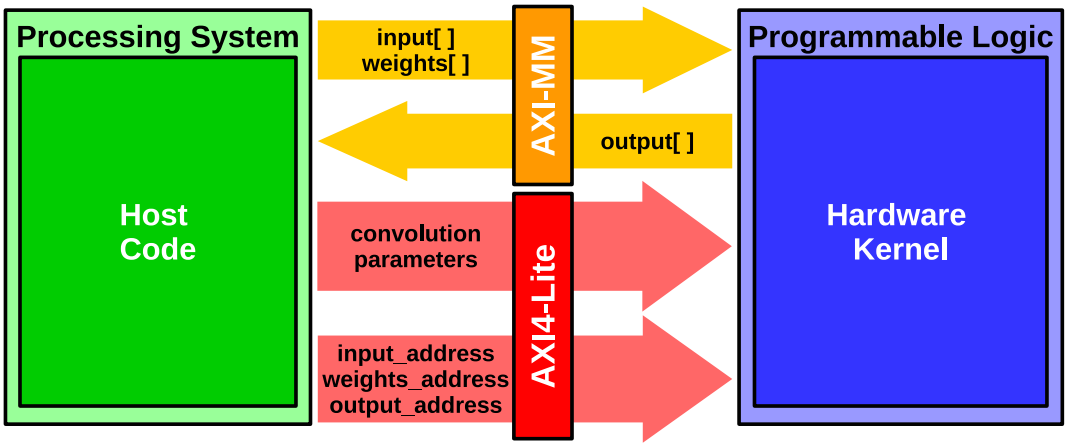


Figure 11: A diagram showing the transfer of data between the PS and PL using the AXI interfaces

5.2.1 AXI-MM Master Bus

The AXI-MM stores data sequentially, so that it can be easily accessed with a pointer. In order to transfer multiple arrays of data through the master bus, the arrays are concatenated and stored on the master array. This allows each array to be accessed by setting a pointer at the correct offset with

respect to the master bus base address. In order for the hardware kernel to take advantage of this, the offset of each array on the master bus is sent to the FPGA a single value over the AXI4-Lite interface. The AXI-MM master bus is configured in the HLS code using `#pragma` commands, shown in Code 7 which specify the length of the bus, and the variable name of a pointer to the base address.

```
1 #pragma HLS INTERFACE ap_bus port=master_portA depth=2147483648
2 #pragma HLS resource variable=master_portA core=AXI4M
```

Code 7: Commands to configure AXI-MM master bus

Unnecessary master bus accesses harm the efficiency of a hardware block. To limit this, the hardware kernel loads the input and weights data from the interface into buffers at the start of its operation, and stores the output data in a buffer until the end of its operation, when it loads the output buffer into the AXI-MM interface.

5.2.2 AXI4-Lite Interface

The top level function of the HLS code takes these values as function parameters, and configures the AXI4-Lite interface with the use of `#pragma` commands, shown in Code 8. The commands are used to specify which variables are being transferred and also to group them together in a "bus bundle". The bus bundle bundles the variables into a common AXI4 slave interface, allowing the same interface to be used for several variables.

```
1 #pragma HLS INTERFACE ap_none register port=outputName
2 #pragma HLS RESOURCE core=AXI4LiteS variable=outputName metadata="-
    bus_bundle CONFIG_BUS"
```

Code 8: Commands to configure AXI4-Lite bus

On the other side of the interface, the software accesses the variables by writing to and reading from specific kernel registers that are mapped to the AXI4-Lite bus. The bundled variables are accessed through sequential registers.

5.3 Hardware Kernel Design

The programmable logic in the system is designed using Xilinx's Vivado design suite, as described in 2.1. It would have been possible to implement the design with and hardware description lan-

guage such as VHDL or Verilog instead, but use of the HLS was chosen to speed up development. The hardware kernel used for the evaluation is an implementation of the computational kernel described in Section 4.3.2. A single hardware block was generated, as shown in Figure 12, and integrated with a larger hardware design. The hardware block show the AXI4-Lite bus on the left side with the label `S_AXI_CONFIG_BUS`, and the AXI-MM master bus is shown on the right side with the label `M_AXI_MASTER_BUS`.

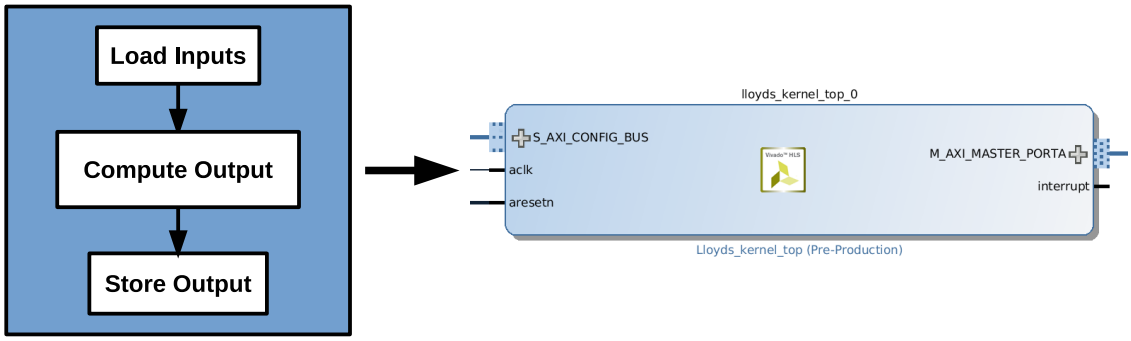


Figure 12: A diagram of the hardware kernel and the resulting Vivado block

An important decision to make was deciding which parts of the CNN system were appropriate for a hardware implementation. Transferring data and setting up a hardware kernel has significant computational overhead, so the best candidates would need to have a significant computational workload in order to make the costs worthwhile. This eliminated the ReLU layer and the pooling layer from consideration, as they essentially move data with no significant computational workload. The convolution layer is the best candidate for hardware implementation, as it reuses input data, i.e. each input and weights value will be used multiple times in the computation. This is ideal as the computational cost of the convolution layer is by far the bulk of the network's workload. The fully connected layer is also a good candidate for hardware acceleration, as it has a large computational workload as well. However it does not reuse data and takes up a large amount of memory with its weights, because each input and output neuron have an individual weight between them. The data transfer cost of this means that the fully connected layer would not be as efficient as the convolutional layer in hardware, but would still bring significant benefits.

6 Implementation

This section will explore any relevant details that were not mentioned in Section 5: Design. This will include pseudo-code 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.

6.1 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.

6.2 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 delays in the process of setting up the socDrawer for testing prevented this from taking place.

The convolution function hardware was then integrated 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 6.3 and the SEU tests were performed. More details on the results gathering and analysis are given in Section 8.

6.3 The SocDrawer

The SocDrawer is a test rig set up to facilitate parallel testing on an array of Zedboards, which hold the Zynq SoCs, in order to speed up exhaustive hardware tests such as error injection testing. It is designed modularly so that any number of Zedboards can be added and tested simultaneously, and if one SoC is causing problems, it can simply be removed from the testing set up. This testing rig was already set up with scripts to perform error injection testing, which meant that the test set up only required the hardware kernel to be integrated with the existing system. The most time-consuming part of this process was configuring the AXI interfaces, as described in Section 5.2, to work with the host code, detailed in Section 5.1.2, on the SocDrawer. It was eventually discovered that aggressive optimisation by the HLS compiler was removing key parts of the interface and the problem was resolved by redesigning and rearranging the HLS code.

A diagram of the testing SocDrawer's testing scripts are shown in Figure 13. These scripts were already on the socDrawer, but were re-purposed for the tests of this project. The testing process is takes two files as inputs. The first of these is a `.bin` file, which stores the hardware configuration in a bitstream. This file is used to load the hardware design into the FPGA. The second important file is a `.bin` file. This file contains a list of binary values, corresponding with the bits in the bitstream. The values are set to 1 if they are deemed to be essential for the kernel to function properly, and are set to 0 otherwise. It is these essential bits that will be flipped in the error injection process. `generateLinearAddresses.py` converts the contents of the .ebd file in to linear addresses, i.e. for each essential bit, a hexadecimal number is generated to represent the address of that bit. These linear addresses are stored in a .la file.

The next script is called `createChunks.py`. The tests are performed in increments, so that if the test is interrupted, it can be restarted from the most recently completed increment. In order to achieve this, the .la file is split into chunks that are tested on an individual basis.

The final script of the test is the most important. `kmeans_evaluateChunks.py` generates a script to run tests for a particular chunk, send the scripts and the chunk to a SoC, runs the test and retrieves the results. The script will do this repeatedly for each SoC in parallel until the tests are over or one of the SoCs crashes due to a "problem chunk". Problem chunks are chunks that test a bit which causes the SoC to fail completely, requiring it to be rebooted before testing can continue. As described in 5.1.2, each time the host code runs, it will echo the input value it was given as a command line argument, which will be the linear address being tested, and then it will print the number of discrepancies between the kernel output and the golden example. `kmeans_evaluateChunks.py` concatenates all of these outputs into a comma separated file, created one column of linear addresses, and one column showing the number of errors caused by the error in that address.

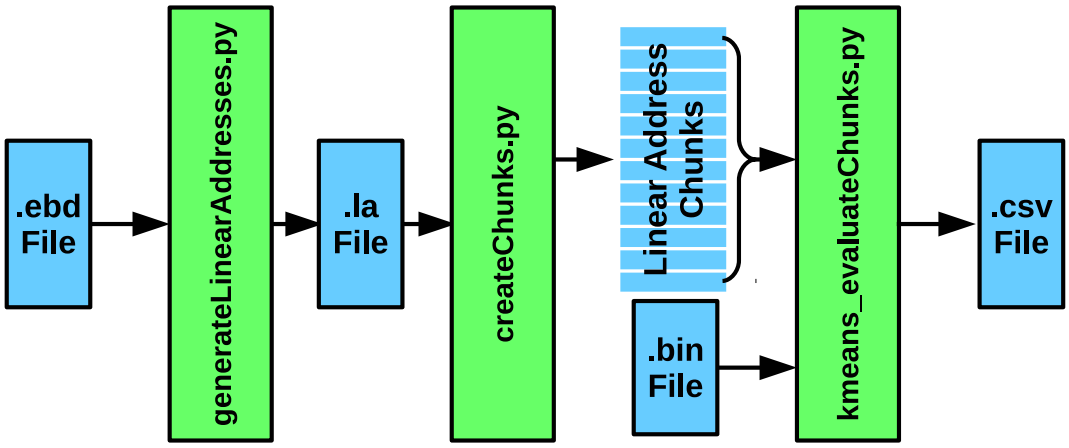


Figure 13: A block diagram showing the top level process of performing a test series on the SocDrawer

6.4 Software Implementation

The software implementation of a full CNN was nearly completed, but was abandoned in favour of the hardware implementation as time was running out. Everything described in Section 4 was implemented successfully, except for the full network. Most of the network functionality was implemented, but unfortunately there were still key components missing from the network. The input and output sections ended up being too time consuming to attempt. One solution considered

for the input was to pre-process an input image, using an application such as MATLAB to convert it into a file of raw numbers to be put straight into the input of the network but there was not time to implement this. The output of the network was supposed to be a classifier called SoftMax. This was deemed too time consuming to attempt, as the project was running behind schedule at this point.

6.5 Hardware Implementation

To implement the hardware system, an existing Vivado hardware design used for SEM error injection was used in order to save time. The board design is shown in Figure 14. The design includes the SEM IP core which is the small block at the bottom. It also includes two HLS kernels, shown on the middle left with the Vivado logo, that can be configured using Vivado HLS. There are two blocks controlling the AXI interface on the middle right. Finally on the far right is the board's interface with Zynq processing system, which runs the system's software. The convolution kernel replaced the bottom HLS block, which is named `lloyds_kernel_top_0`. By changing the source code of the kernel, it was possible to very quickly adapt the hardware block to run the convolution kernel.

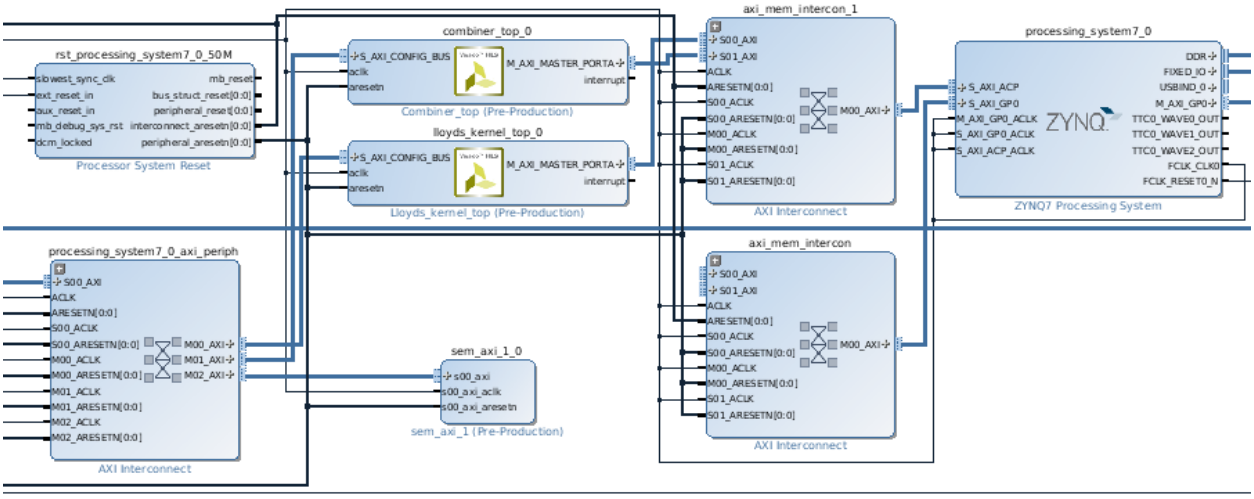


Figure 14: Board Design for the digital system

The next step in the process was to set the compiler to generate a .ebd file for the relevant hardware, as discussed in Section 6.3. This was done by opening the implementation design in Vivado, shown in Figure 15. This provides a visualisation of the FPGA fabric and all of the

hardware designs implemented upon it. The orange section in the top left shows the part of the board responsible for the control and configuration of the FPGA. Setting SEUs up in this region would result in configuration upsets, as described in 2.6.1. The part of the board that is occupied by the convolution kernel is highlighted in white. This section was selected for testing, configuring the compiler to generate a .ebd file that contain bit addresses relevant to this block.

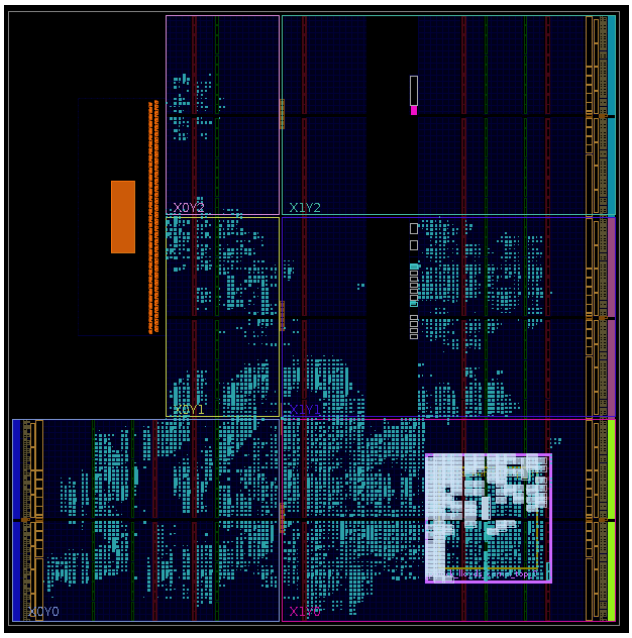


Figure 15: A view of the implementation design

Due to time pressures, only the convolution layer was ultimately integrated into the hardware design. A design for a hardware kernel was built and tested in software, but delays arising from debugging the AXI interfaces, mentioned in Section 6.3, pushed the project plan back significantly, leaving no time for the design to be implemented in hardware and integrated into the rest of the hardware design.

7 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 hardware/software interface and for the hardware kernel.

7.1 Software Layer Testing

For each layer that was implemented in soft, a short test function was written to verify its functionality. Due to the time pressures of this project, these tests were kept simple and were designed to verify that the basic computations were correct, rather than testing corner cases and checking completeness.

For the ReLU layer, this test consists of simply filling the input with uniformly distributed values in the range $[-255, 255]$ and comparing the input with output. If the input value is negative, the output value is expected to be 0. If the input value is positive, the corresponding output value should be equal to that value.

For the pooling layer, this test was slightly more challenging. Simply running through the same algorithm to check the result is not a reliable test, and redesigning the code was deemed an inefficient use of the project time, particularly given the fact that the pooling layer is not computationally expensive enough to warrant a performance comparison between different implementations, so the second version would only be used for verification. Therefore the input Blob is filled with values in ascending order from the lowest index, creating a matrix with where for any given sub-matrix, the highest value is in the location with the highest index. Therefore, the test could be performed by comparing the input and output, and checking that the value selected by the max pooling layer was indeed the value with the highest index. This test is obviously far from exhaustive, but as previously explained, it was deemed sufficient to verify basic functionality given the tight schedule of the project.

The fully connected and convolution layers are both much more complex than the ReLU and pooling layers, so devising a test function in C would have been quite a challenge. However, each of these functions was implemented twice, each using two different algorithms, as described in Sections 4.3 and 4.4. In each of these cases, the matrix multiplication based implementation was verified using MATLAB. The testing was performed by running a set of inputs through the layer implementation, and using MATLAB's matrix multiplication functionalities to perform the same

computation. The results were compared, and if they were correct, the implementation was correct. This was done for a few different input sets and computation sizes to make the testing slightly more thorough. Then, when the second implementations of the fully connected and convolution layers were complete, they were simply compared against the first implementation to check their correctness.

7.2 PS/PL Interface Testing

A crucial part of the system that needed to be tested is the AXI4 bus that transferred data between the Programmable Logic and the Processing System, as described in Section 5.2. Errors in the setup of this interface could result in incorrect values being passed to and from the kernel, or could mean that the hardware kernel has no way to input or output data, rendering it completely unusable.

To test the interface, a simple testing kernel was designed and built. This kernel was designed to have the same interface design as the convolution kernel described in 5.3. By testing a kernel that had the same external interfaces as the desired hardware kernel, it could be confirmed that required interfaces were working properly before proceeding with the system implementation. The kernel simply read some values from the input and weights arrays, and from each of the AXI4-Lite variable inputs. Each of these values was then assigned to a location in the output array. The host code controlling the kernel was set up to assign values to each of the input arrays and variables, initiate the kernel, and print the output buffer. This allowed the operation of the AXI interface to be verified by inspection.

7.3 Hardware Kernel Testing

Once the full hardware kernel had been designed, it was tested to verify that it achieved the ground truth result computed with the pure software implementation. This was a very simple case of running the test kernel, as shown in Section 5.3, and checking by inspection that the result was correct. The test was repeated for several iterations and it was discovered that the output values were accumulating the results of each run. Each time the kernel was tested, the output, rather being set to the result of the convolution, would be the sum of all runs up to that point. With a minor adjustment in the HLS code, this potentially disruptive error was corrected.

8 Evaluation

This section will analyse whether each of the following deliverables were achieved: An Embedded System running CNN layers, a Convolution Layer Implemented in Hardware, SocDrawer Testing

8.1 Embedded System running CNN layers

The individual layers of the CNN were implemented and run on the embedded system. The system was implemented with a testing interface, reading commands that could run test cases for individual layers. The convolution, fully connected, ReLU and pooling layers were all implemented and tested successfully on the Zedboard. These results were verified by the test functions described in Section 7.1, and were compared against the results of the pure software implementation to ensure that they match. Therefore this requirement was fulfilled. The full source code for this can be found in the `src_hw/` folder in github repository given in Appendix A

8.2 Convolution Layer Implemented in Hardware

The convolution kernel was implemented in hardware using Vivado HLS and combined into a hardware block. This using the host code described in Section 5.1.2, the kernel was run successfully. It was successfully generated and synthesised in Vivado and thoroughly tested. The convolution test used a 14x14 input feature map, and a one 3x3 weights kernel, and producing a 14x14 output feature map. This small size was chosen in order to make the SocDrawer testing as quick as possible.

Resource Name	Number Used	Total Available	% Used
LUT:	1092	53200	2.0
FF:	1403	106400	1.3
DSP:	5	220	2.3
BRAM:	10	280	3.6

Table 1: A table comparing the used resources against the total resources available in the Zynq-7000

The FPGA resources used by the kernel are given in Table 1. The LUTs are Look Up Tables, used to efficiently encode boolean function. These are the foundation of all FPGA designs, so it is very beneficial to use them efficiently. FF stand for Flip Flops, which are just registers that store

values on a per-clock-cycle basis. DSPs are Digital Signal Processing components. They are used for additions and multiplications, which is the bulk of the convolution work. Finally the BRAM, or Block Random Access Memory is used to rapidly load or store information. These are used by the AXI interfaces. It can be clearly seen that the usage on each of these is extremely low. While this is a good thing, and is partially because the convolutional kernel implemented was extremely small, the small size also means there could have been performance benefits in further parallelising the design. There would be a trade-off with resource usage, but that is fine given the low resource usage. This requirement is fulfilled, but not as well as it could have been. The source code for the HLS kernel and the accompanying host code can be found in github repository described in Appendix A.

8.3 SEM Testing

The hardware kernel was added to a existing board design that included the SEM core, as described in section 6.5. The overall hardware block built successfully, and worked fine with the SEU inactive. Then the system was integrated into the SocDrawer experiment setup.The SoCs were all configured and individually ran the host code perfectly, so the hardware kernel and host code were integrated into the SocDrawer. Everything designed as part of this project worked on the SocDrawer setup. Using the scripts described in Section 6.3, the results shown in Section **??** were achieved. Unfortunately, this was completed very late, and as a result, an unforseen error manifested and caused the tests to fail unexpectedly. This was fixed only at the very last moment, and only 12 test results were captured. These are shown in Figure 16 and Appendix B. It can be seen that in each of these 12 tests, the convolution kernel was able to produce the correct result for all 196 values of the 14*14 output. Given a little bit more time, a much more conclusive range of results would have been achieved.

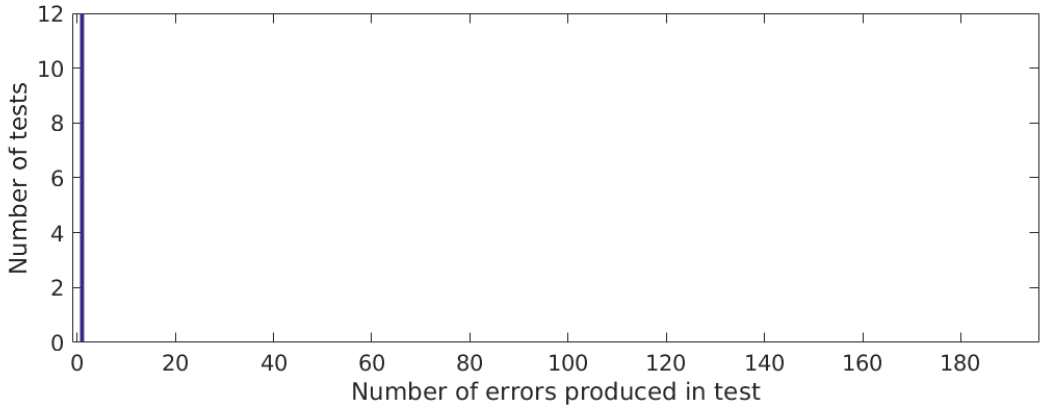


Figure 16: A view of the implementation design

8.4 Desireables

The CNN was only partially implemented in software, so this requirement was not completed. Additionally, there was not time to complete hardware optimisations on the kernel, so this requirement was also not completed.

9 Conclusion and Future Work

In summary, this report describes the investigation of the effects of Single Event Upsets upon the operation of a Convolutional Neural Network. After the introduction, Background information about neural networks and single event upsets was provided. The Project Specification laid out what the project aimed to achieve. The CNN Design section described the process of creating the CNN and all of its layers. The System Design section described how the rest of the system was conceived. The Implementation section provided a summary of how the setup and execution of the design was achieved, including what wasn't completed. The Testing section showed how the systems constructed were validated and corrected. Finally the Evaluation gave an overview of what was achieved, and what was not accomplished in the given time frame.

The results of our testing take an extremely small sample. It's impossible to determine from 12 results whether a convolution kernel is guaranteed to work in a high radiation environment. However, what can be safely determined is that there are definitely cases where the convolution kernel does work properly. This is demonstrated by the fact that all 12 of the tests performed gave the correct output.

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] 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.

[4] 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.

[5] http://cs231n.github.io.

[6] Y. Bengio, "Learning deep architectures for ai," tech. rep., Dept. IRO, Universite de Montreal, 2009.

[7] 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," *International Journal of Computer Vision (IJCV)*, vol. 115, no. 3, pp. 211–252, 2015.

[8] 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.

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

[10] 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.

[11] J. T. Springenberg, A. Dosovitskiy, T. Brox, and M. Riedmiller, "Striving for simplicity: The all convolutional net," in *ICLR*, 2015.

[12] K. Simonyan and A. Zisserman, "Very deep convolutional network for large-scale image recognition," in *ICLR*, Visual Geometry Group, University of Oxford, April 2015.

[13] M. Cord, "Deep cnn and weak supervision learning for visual recognition." https://blog.heuritech.com/2016/02/29/a-brief-report-of-the-heuritech-deep-learning-meetup-5/.

[14] 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.

[15] 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.

[16] 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.

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

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

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

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

[21] *Soft Error Mitigation Controller v4.1*.

[22] 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.

[23] http://tech.gilt.com/deep/learning/2016/05/18/fully-connected-to-convolutional-conversion, 2016.

[24] 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.

[25] 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.

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

[27] 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.

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

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

[30] 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.

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

A Source Code

All of the source code for the project is stored in a public git repository and can be found at the following URL:

https://github.com/JHertz5/space_brain

B Results

This show the results gathered. The first column is the linear address of the error, and the second column is the number of error out of a potential 196 produced.

```
28505810,       0
28612501,       0
28662336,       0
28689688,       0
28762927,       0
28801388,       0
28821924,       0
29255163,       0
29849546,       0
30153000,       0
30303300,       0
30496777,       0
```