
THE UNIVERSITY OF BRISTOL

FINAL YEAR PROJECT

# A complete FPGA-based pipeline for object classification using binarized neural networks

*Roy Miles*

School of Electrical and Electronic Engineering

Supervised by

Dr Jose NUNEZ-JANEZ

School of Electrical and Electronic Engineering

9 April 2018

# Acknowledgments

I would first like to thank my supervisor Dr Jose nunez-janez. Jose has always been there to support me through my research, and has been instrumental to my progress. I was very lucky to have my project under his supervision.

I would also like to thank Xilinx and their help through the PYNQ forums. They provided support on porting the PYNQ framework.

Finally, I would like to thank Moslem Amiri at the University of Bristol for helping me setup the reduced BNN topology.

# Contents

# List of Figures

# List of Tables

# Abstract

The emergence of ground-breaking results [1] from deep neural networks has been due to the increase in computational performance of graphical processing units (GPUs) and the accessibility of large training data sets. Although GPUs offer a good platform for deep neural network deployment, due to their architectural design they impose significant limitations in terms of latency and power consumption. This limitation is due to their performance with non-standard data types and irregular computation, it is for this reason that FPGAs have been considered as a viable solution.

Redesigning neural network models for their application in programmable logic has been demonstrated with very good results [2]. Binary neural networks (BNNs) can achieve the same accuracy on standard datasets (MNIST, CIFAR1) with minimal latency and maximum throughput. This stems from the significant redundancy in typical floating-point models [3], which are less portable onto high performance programmable logic. An FPGA design can exploit the minimal latency of binary computational blocks and also reduce the overall classification latency by encapsulating the state of the network into the design, therefore removing the need for off-chip memory access.

# 1 Introduction

There is often a trade-off between the accuracy and latency of classification for video streams at high frame rates. However, recent research has demonstrated that, by utilising low latency computational logic blocks and a pipelined approach, a hardware accelerated BNN can achieve classification latencies as low as 0.31us [2] and also achieve state of the art accuracy on standard data sets.

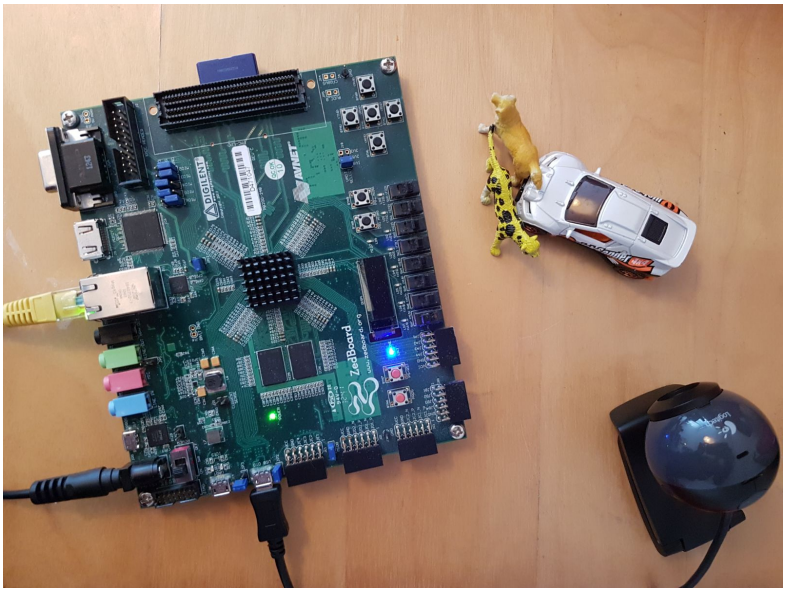


Figure 1: Classifying objects using the Zedboard with a webcam feed

This paper studies the usage of a fully binarized neural network implemented in hardware. The physical environment (see figure 1) integrates a Logitech web camera for the video feed and extracts regions of interest, which are then passed into the BNN for real-time object recognition. Later chapters will study the comparative latency of a hardware implementation of a binary neural network in relation to that defined in software. The BNN will then be tested in various environments to evaluate its performance and resilience to noise and occlusion. Timing bottlenecks in the overall design are then identified and addressed individually to achieve higher frame rates.

# 2 Convolutional Neural Networks

## 2.1 Overview

Convolutional neural networks (CNNs) have proven to be very successful across multiple domains, spanning from computer vision to natural language processing [4]. Unlike conventional neural networks, CNNs use what are known as convolutional layers. These layers convolve a kernel over an image and the weights inside these kernels define the features that are extracted. The model can then be trained, using back-propagation, to update these weights in order to minimise the output error on a given training data set using a supervised learning approach.

A convolutional neural network will often employ multiple hidden layers which hierarchically extract more abstract features from the image. Each layer will typically consist of a convolutional layer followed by an activation layer and a pooling layer. All of which will be discussed in this chapter.

This paper will focus on a convolutional neural network trained to classify images from the CIFAR10 dataset. These are 32x32 images that are to be classified into 10 categories ranging from cars to frogs and airplanes. The following sub chapters will discuss some of the theory behind the components of a CNN, as this will be needed before moving onto the BNN.

## 2.2 Convolutional layer

The convolutional layer is the main component that distinguishes a CNN from other deep neural network models. The principle operation involves sequentially multiplying and striding a kernel over an image. The size and weights inside the kernel dictates what features are extracted from the original image. The Sobel filter, for example, uses 3x3 kernels to emphasize edges in the image. Unlike a neural network, which learns the weights in the kernel, the sobel filter is precalculated using the an approximation to the derivative in two dimensions.

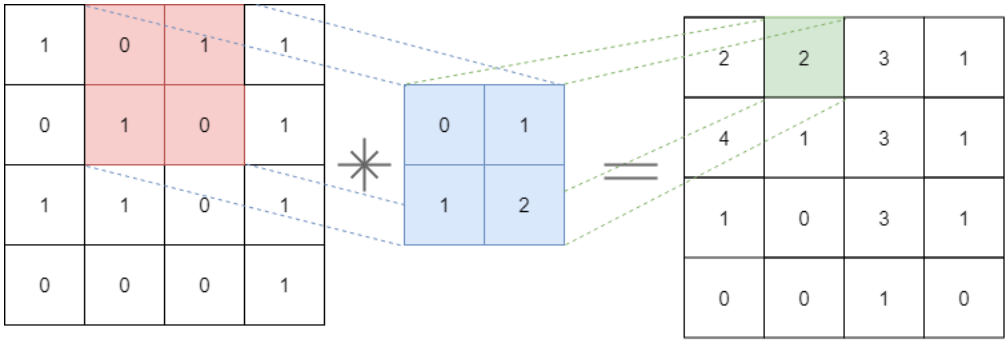


Figure 2: Operation of a convolutional layer using a 2x2 kernel and zero padding

As a convolutional neural network consists of multiple hierarchical convolutional layers, they can effectively extract more abstract features from the image. Historically, these kernel weights had to be pre-calculated by hand, however, through the use of back-propagation, these weights can iteratively converge to extract appropriate features for a given data set.

## 2.3 Pooling layers

The pooling layers are responsible for progressively reducing the size of the input image. This has the effect of reducing the computation through the network and reducing over-fitting by converting the input to a more compact representation.

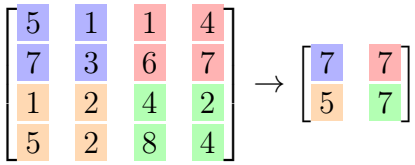


Figure 3: Using max pooling to downsample a 4x4 matrix

A typical pooling layer is often designed to calculate the maximum, minimum or average value of the input inside a window. These layer can then be implemented in software by comparing the individual elements inside each window or summing them up and dividing by the area of the window. An example of max pooling can be seen in figure 3.

## 2.4 Activation function

The activation layer is responsible for introducing a non-linearity into the model, so that the network can learn a non-linear mapping. Various non-linear functions have been used, such as tanh, ReLU or a sigmoid.

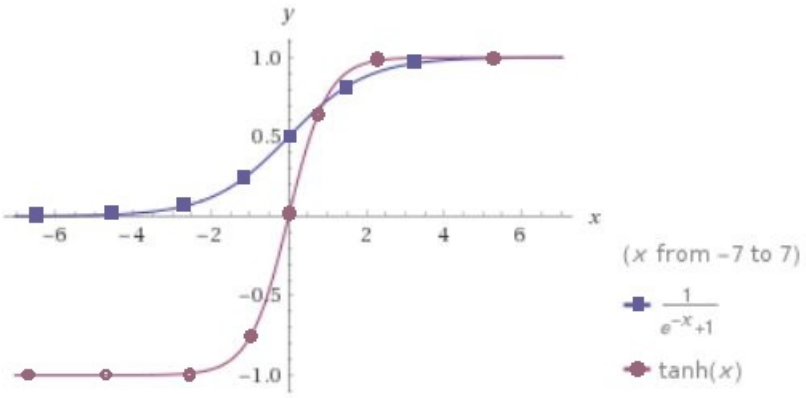


Figure 4: Plot of the hyperbolic tangent and the sigmoid function

Calculating the hyperbolic tangent or the sigmoid of an input is often computationally expensive, especially when it is performed lots of times during training and inference. It is for this reason that the computationally simpler ReLU operation is more often used, which has a minimal comparative impact on the networks accuracy [5].

## 2.5 Batch normalisation

Batch normalisation is a technique used to compensate for covariance shift. This is when the test data set follows a different distribution to that which the model is trained on, and can lead to the output activations shifting by too much. By normalising these output activations, the training rate can be increased with less of this internal shift [6].

# 3 Binary Neural Networks

## 3.1 Overview

Conventional CNNs incur some redundancy due to their floating-point model [2]. It is for this reason that a fully binarized approximation can be used to reduce the memory footprint and latency without any significant reduction in accuracy. The consequence of this model is that the vector multiplication and summations can be reduced to simple logic operations and pop-counts, which are easily ported onto FPGA fabric. Having a model that is suited for an FPGA, can see its applications in low-latency, power constrained applications.

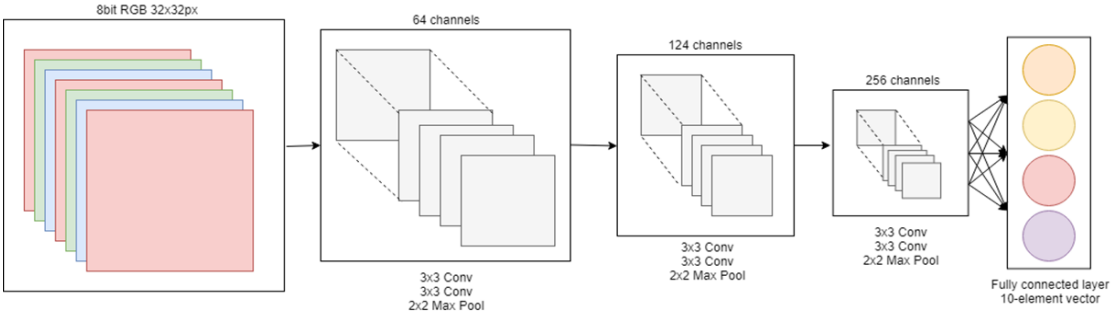


Figure 5: Outline of the FINN BNN implementation

A typical RGB image will consist of 3 channels (red, green and blue) with an 8-bit representation. A fully binarized neural network uses the same architecture of a CNN, but with the pixels reduced to a binary format, along with the kernel weights and the activation functions. This new model minimizes the memory footprint and offer favorable advantages for embedded systems, which will be discussed later in this paper.

The four network components/techniques described in the previous chapter (convolutional layers, pooling layers, activation functions and batch normalisation) are each addressed in the FINN paper.

1. Convolutional layer: The convolutional layers are implemented as matrix-vector products, which can be effectively parallelized on the FPGA fabric.
2. Pooling layer: Due to the internal binary representation of the network, the max-pooling operation is reduced to a binary-OR operation.
3. Activation function: The activation used is the $Sign(x)$, which is implemented as a binary-OR operation too.
4. Batch normalisation: Batch normalisation is implemented prior to the activation function and is computed using thresholding.

## 3.2 Motivation

Very deep neural networks (DNN) are becoming very computational demanding, especially as the demand for performance increases. Research has demonstrated [3] that, by rethinking the fundamental numerical representation used, the networks can be scaled more efficiently and with little degradation on accuracy.

There are many embedded applications which are strictly limited in latency and memory, and the BNN implementation outlined in the FINN paper [2] [3] is heavily based on these requirements. The neurons and synapses are mapped onto processing elements (PE) and SIMD lanes respectively. From this, the number of processing elements and SIMD lanes are then used as parameters to control the folding of the matrix-vector products, which effectively allows the resource utilisation and FPS requirement to be dynamically adjusted. This approach proves to offer a platform for scaling the BNN topology in accordance with the hardware resources available.

OpenCL has also been demonstrated as an effective platform to build efficient hardware architectures for platforms with multiple FPGAs [7]. OpenCL applications run on heterogeneous platforms that scale well for multiple accelerators,

including GPUs, CPUs and FPGAs. The PipeCNN [7] project addresses the low-power applications of a CNN using OpenCL, and demonstrates very low power consumptions over conventional GPU platforms.

# 4 Porting PYNQ Framework

This paper uses a Zynq-7000 Zedboard, which is an evaluation and development board with an integrated dual-core ARM cortex processor and a Xilinx FPGA. The integration of the PS (Programmable software) with the PL provides significant performance benefits that cannot be matched by two-chip solutions and is ideal for software and hardware accelerated designs.

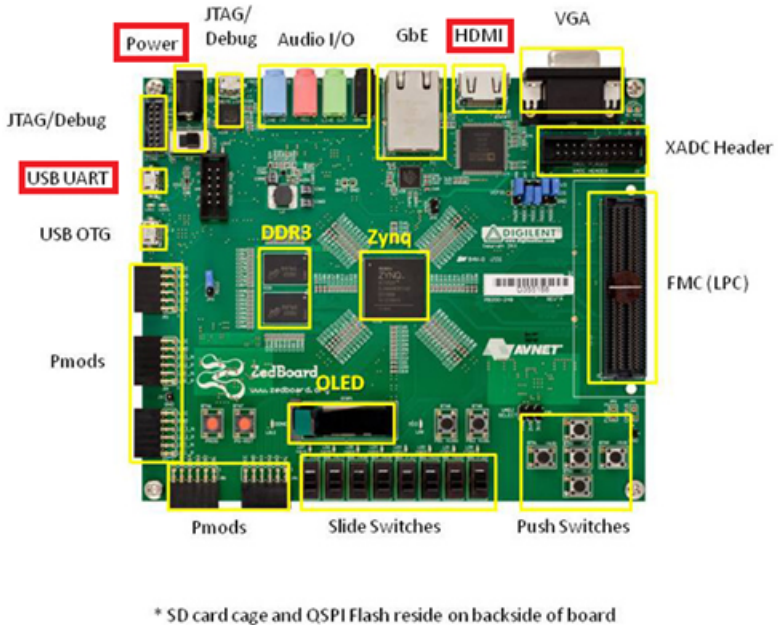


Figure 6: Physical layout of the zedboard, with the important ports highlighted in red.

This chapter will discuss how to port the PYNQ BNN design onto the Zedboard. This process involves partitioning the SD card, generating the boot files and compiling the Linux kernel. Parth parikh's blog [8] describes some of these steps in more detail however does not cover all the necessary steps for use of the PYNQ

framework with the BNN overlay, or the general setup of the Zedboard. This chapter aims to cover all these pre-requisites for using the BNN design. The terms "host" and "client" will be used to correspond to the computer/laptop and the Zedboard respectively, and all the appropriate pre-built files can be found in my GitHub repository at https://github.com/iyop45/Final-year-project.

## 4.1 Binary Neural Network Overlay

The PYNQ framework [9] uses high level abstraction to allow developers to take advantage of the benefits of programmable hardware, without having to use HDL design and synthesis tools. All the hardware designs are implemented with a common overlay to offer a simple software interface and to improve portability. However, this method of abstraction makes it difficult to make significant design optimizations at the driver level. It is for this reason that the PYNQ framework is not fully utilized in this project, and instead a new custom framework abstraction has been built up in C++.

BNN-PYNQ [10] is the package that implements the binarized neural network design described in the FINN Paper [2]. The package contains two overlays, namely CNV and LFC, which are topologically designed for the CIFAR-10 and MNIST datasets respectively. This paper will primarily focus on the CIFAR-10 network and its real-time application however the techniques discussed can be equally applied for the LFC network.

## 4.2 Partitioning the SD card

Two partitions will need to be created, namely for the root file system and the boot files. These partitions can be created manually or simply by flashing the SD card using the supplied PYNQ ISO image, which can be downloaded at http://www.pynq.io/. At the time of writing, the most recent version is the PYNQ-Z1 v2.1 image.

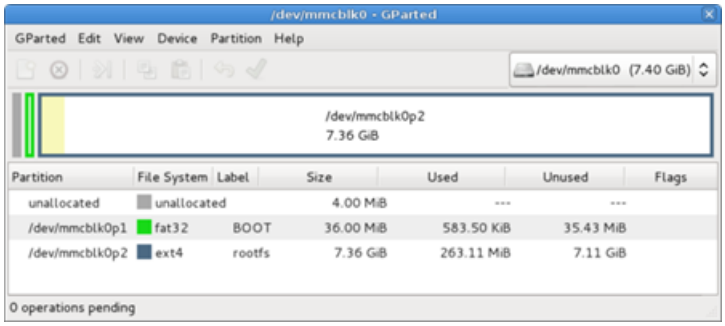


Figure 7: SD card partition

The SD card will need to have least 8GB and, for a manual partitioning, the boot partition will need to laid out as shown in figure 7.

## 4.3 Communicating over UART

PuTTy (https://www.chiark.greenend.org.uk/~sgtatham/putty/latest.html v0.7) can be used to communicate with the Zedboard over a USB serial port with a baude rate set to 115200. Boot logs will be printed to the console as soon as the Zedboard is powered on, which allows any errors to be traced back to their source. This cannot be achieved when communicating over SSH.

## 4.4 Communicating over SSH

After the board has booted into Linux, an SSH connection can be made. Communicating over SSH has many benefits over using the UART. Specifically, files can be transferred using SFTP, WiFi connections can be shared, and graphical applications can forwarded to the host screen. Each of which will be discussed in this chapter.

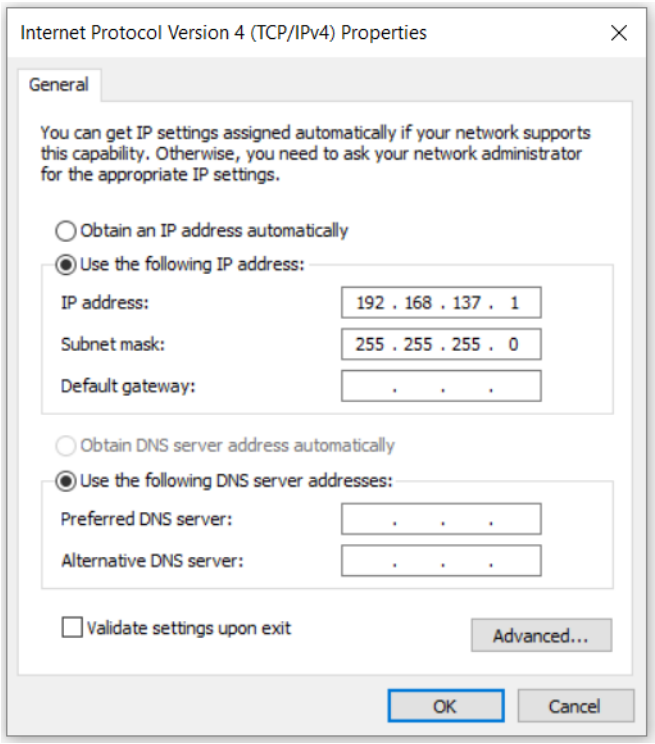


Figure 8: Ethernet properties to set static IPv4 address

To setup the SSH connection over Ethernet, a static IP needs to be set on the host under the same subnet as the Zedboard. On windows this can be achieved by navigating to the Ethernet IPv4 properties through the control panel (see figure 8). The static IP address of the Zedboard can be changed by editing the interfaces file found at `/etc/network/interfaces.d/eth0`. The subnet mask was set to `255.255.255.0`, and the static IP addresses set to `192.168.137.1` and `192.168.137.99` for the host and client respectively.

The windows firewall can block communication from the Zedboard and so the firewall may need to be disabled for public networks, or alternatively an exception will need to be added. To verify the connection is setup correctly, ensure both devices can ping each other. Then, once able to reach the login prompt, the default username and password will be "xilinx".

### 4.4.1 Transferring files and sharing Wifi connection

It is useful to be able to easily transfer files to and from the Zedboard. For example, the scripts and source files can be developed on the host laptop before being offloaded onto the Zedboard for compilation. This is the general workflow for if the laptop offers an easier development environment. With regards to sharing an Internet connection, this is useful when installing a variety packages on the Zedboard.

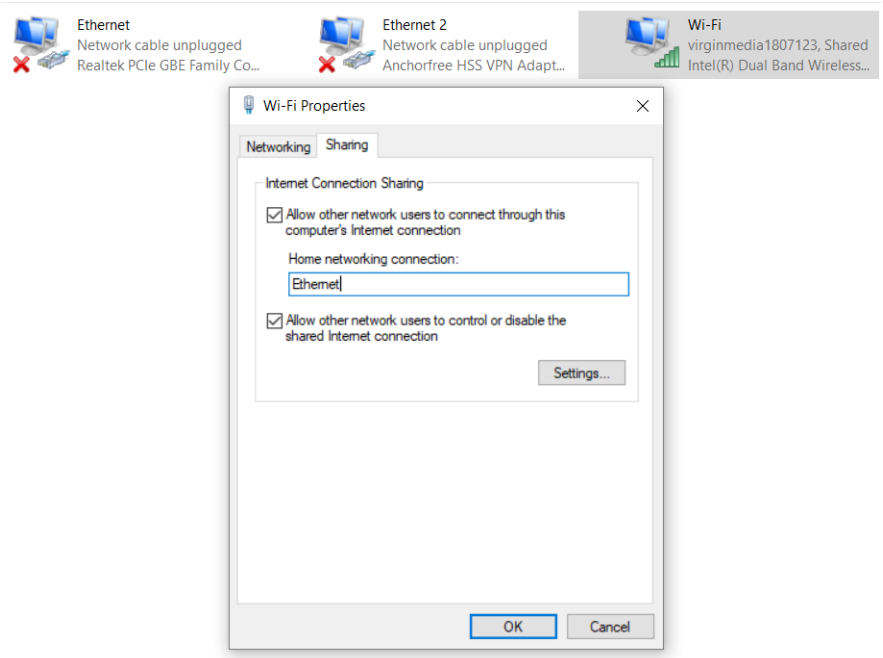


Figure 9: Sharing Wifi connection with the Zedboard Ethernet connection

To enable internet access for the Zedboard through the host computer, the hosts internet connection will need to be shared with the Ethernet connection. For windows, this can be achieved using the network connection utilities found at Control Panel\Network and Internet\Network Connections. The sharing properties for Internet connection will need to be modified, as shown in figure 9.

With regards to file transfers, it is convenient to use the SFTP protocol, which is commonly integrated into most SSH applications. For this project WinSCP (v5.13) was used, which can be found at https://winscp.net/eng/download.php.

### 4.4.2 X11 forwarding

X11 forwarding under SSH is a form tunneling by which the display on the client can be viewed on the host. This allows GUI applications (running on the Zedboard) to be shown on the host computer. Windows does not run an X11 display server by default and so an alternative, such as Xming (which can be found at https://sourceforge.net/projects/xming/), will need to be setup and running before the GUI application is executed on the client. In this case, the version corresponding to Xming-6-9-0-31-setup.exe is used. If logged in as root, the .Xauthority file will also need to be copied into the root/ directory. This ensures xauth will work with the root user. Note that the default root password is "xilinx".

### 4.4.3 Updating packages with an archived Linux kernel

The Linux kernel used by Xilinx is based on an old repository, which is now archived. To handle and update packages using apt-get, the repository sources must be changed to search from the archives. This is done by changing the networks.d file to the following (corresponding to the Wily Linux archive):

```
## EOL upgrade sources.list
# Required deb http://old-releases.ubuntu.com/ubuntu/ wily main restricted universe multiverse
deb http://old-releases.ubuntu.com/ubuntu/ wily-updates main restricted universe multiverse
deb http://old-releases.ubuntu.com/ubuntu/ wily-security main restricted universe multiverse
# Optional
#deb http://old-releases.ubuntu.com/ubuntu/ wily-backports main restricted universe multiverse
```

## 4.5 Building the devicetree.dtb

The devicetree is responsible for describing the hardware components for the operating system. The devicetree blob (DTB) is a binary format that represents the hardware components, and can be generated from a devicetree string (which is in a human readable text format).

The devicetree string can be created by modifying the pre-existing zynq-zed.dts file, as shown in figure 10, which is found in linux-xlnx/arch/arm/boot/dts/ of the Xilinx Linux kernel repository (v4.16 is used here). The first node is for the

boot sequence, and enables communication through the UART port, whereas the other two nodes are for the BNN. They outline the hardware offset address and the XLNK staging driver details, which are used to allocate contiguous memory (and can be found in `/dev/xlnk`). Note: The hardware offset address is evaluated after synthesizing the BNN hardware design.

```
...
chosen {
        bootargs = "console=ttyPS0,115200 root=devmmcblk0p2 rw earlyprintk rootfstype=ext4 rootwait
devtmpfs.mount=1 uio_pdrv_genirq.of_id=\"generic-uio\"";
        linux,stdout-path = "/amba@0/serial@E0001000";
};
amba {
        fabric@43c00000 {
                compatible = "generic-uio";
                reg = <0x43c00000 0x10000>;
                interrupt-parent = <&intc>;
                interrupts = <0x0 0x1d 0x4>;
        };
};
xlnk {
        compatible = "xlnx,xlnk-1.0";
        clock-names = "xclk0", "xclk1", "xclk2", "xclk3";
        clocks = <&clkc 0xf &clkc 0x10 &clkc 0x11 &clkc 0x12>;
};
...
```

Figure 10: Modified Zedboard devicetree string

```
export CROSS_COMPILE=arm-xilinx-linux-gnueabi-
./scripts/dtc/dtc -I dts -O dtb -o <devicetree name>.dtb <devicetree name>.dts
```

Figure 11: Compiling the devicetree string

The devicetree will need to be cross compiled using the ARM toolchain, as shown in figure 11. A different command will be used for any version of Vivado more recent than 2016.3 [8].

## 4.6 Configuring and compiling the Xilinx Linux kernel

The Xilinx Linux kernel is used and can be found at: https://github.com/Xilinx/linux-xlnx, where v4.16 is used here.

```
make ARCH=arm xilinx_zynq_defconfig
make ARCH=arm menuconfig
make ARCH=arm UIMAGE_LOADADDR=0x8000 uImage
```

Figure 12: Compiling the Xilinx Linux kernel with the Zynq configuration

The default Zynq configuration will be used as a basis for the build. After which, the menu config will need to be opened to enable/edit the appropriate drivers and parameters (see figure 12).

There are a couple of drivers that need to enabled for the PYNQ framework. These are the xlnk and uio platform drivers, which can be found in Device Drivers->Staging drivers->Xilinx APF Accelerator driver and Device Drivers->Userspace I/O drivers->Userspace I/O platform driver with generic IRQ handling respectively [11].

Logitech cameras are also not natively supported on Linux, and so the USB Video Class (UVC) will also need to be enabled too (Drivers->Media->USB->UVC).

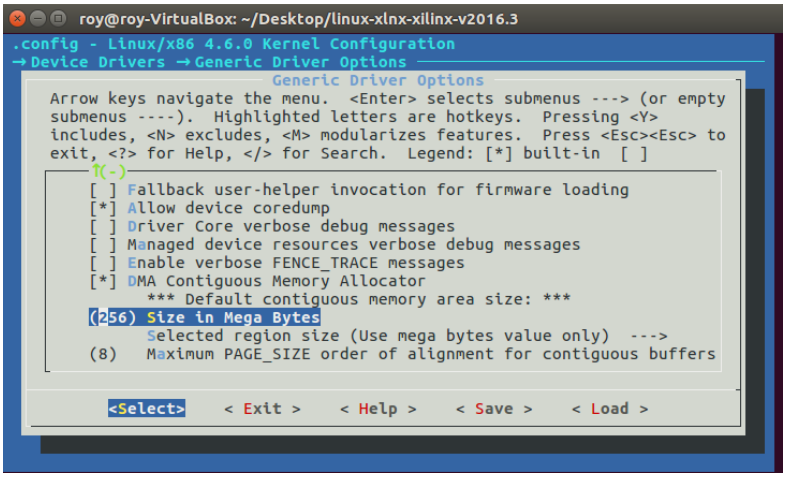


Figure 13: Setting a sufficient memory size for the contigious memory allocator

The BNN reads and writes from contiguous memory on the host, and so sufficient space must be specified in the Linux build. The other accelerated hardware blocks will make use of the physical addresses in contiguous memory too, and so at least 256MB should be set. This process can be shown in the menuconfig of figure 13.

## 4.7 Synthesising the BNN hardware design

The hardware design was generated using Vivado 2016.3 (WebPACK edition), as the later versions proved to have some compatibility issues.

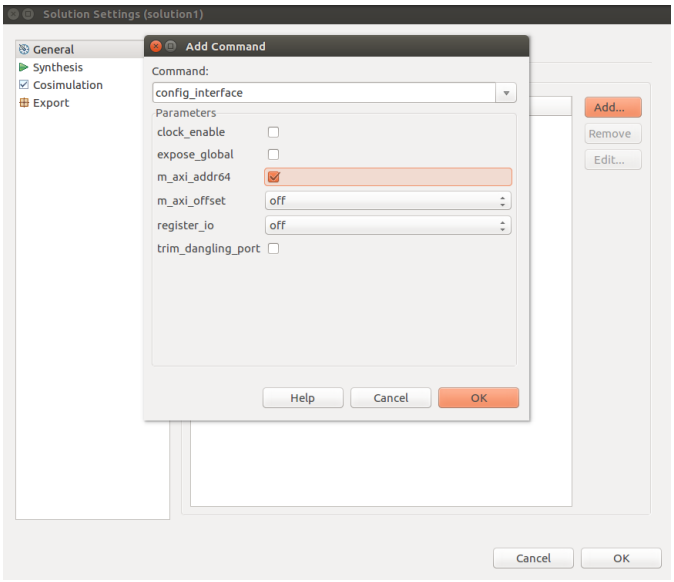


Figure 14: Enabling 64bit bus address widths in the solution settings

Firstly, the BNN HLS source had to be synthesised and exported as an IP block. This involves copying all the source files into a new Vivado HLS project that is targeted at the Zedboard. Before synthesizing and exporting the design, the address bus widths will need to be set to 64 bits, which is a command that can be set in the solution settings (as shown in figure 14).

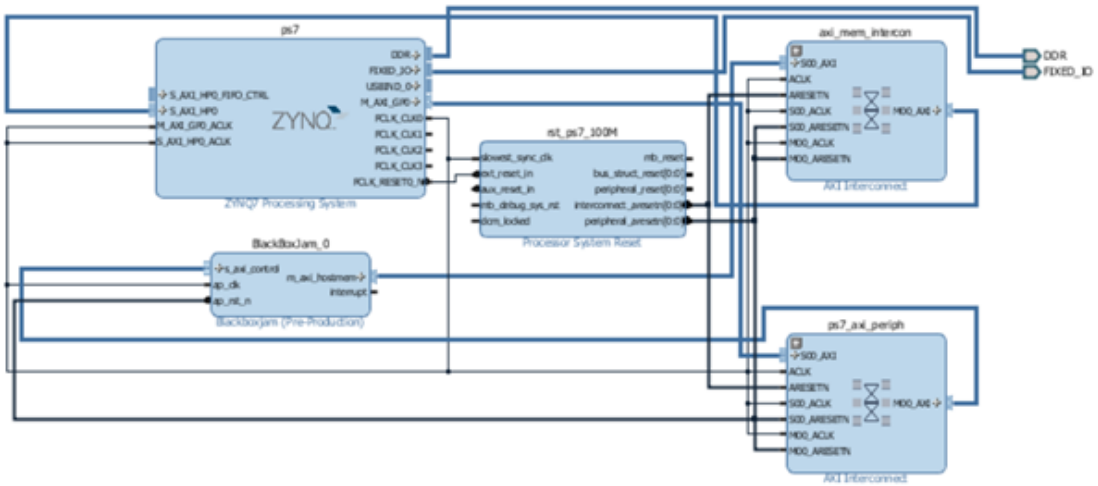


Figure 15: Block diagram showing the BNN IP with the ZYNQ processor system

Once the BNN IP is generated, it can then be imported into the supplied BNN-PYNQ/bnn/src/library/script/make-vivado-proj.tcl layout and connected using the built-in connection automation tool (see figure 15).

| Name | BRAM_18K | DSP48E | FF | LUT |
|---|---|---|---|---|
| DSP | - | - | - | - |
| Expression | - | - | - | - |
| FIFO | - | - | - | - |
| Instance | 81 | 26 | 27032 | 42903 |
| Memory | 189 | - | 3104 | 416 |
| Multiplexer | - | - | - | 1243 |
| Register | - | - | 159 | - |
| Total | 270 | 26 | 30295 | 44562 |
| Available | 280 | 220 | 106400 | 53200 |
| Utilization (%) | 96 | 11 | 28 | 83 |

Table 1: Resource utilisation for the BNN with the default configuration

Table 1 shows the resource utilisation for default BNN design. As you can see, the utilisation is near 100%, especially in terms of BRAM usage. This means that there will be little room for other accelerated blocks, however will achieve maximum classification FPS for the Zynq FPGA.

## 4.8 Creating the boot image

The boot image (BOOT.bin) wraps the first stage bootloader, the PL logic bitstream, and the second stage bootloader (u-boot.elf) into a single binary file. The method for compiling the bootloaders can be found in Parth Parikh's blog [8].

Once the bootloaders have been compiled, the boot image can be generated from a .bif file using the bootgen utility (see figure 16 and figure 17).

```
//arch = zynq; split = false; format = BTN
the_ROM_image:
{
        [bootloader]zynq_fsbl.elf
        cnv-pynq-pynq.bit
        u-boot.elf
}
```

Figure 16: Contents of the zynq.bif file required to build the boot image

```
bootgen -image /<path to bif>/zynq.bif -o /<path to output>/BOOT.bin
```

Figure 17: Bootgen command to build the boot image

# 5 Extracting regions of interest

For each frame the regions of interest (ROI) need to be extracted and individually classified using the BNN. This section will discuss the various techniques that can be employed to segment the frames such that the regions of interest can be identified.

## 5.1 Setting up OpenCV

OpenCV is an open source image/video manipulation library for C++ and Python. This library will be used to implement all the following ROI extraction methods. This chapter discusses the steps required for integrating the OpenCV library into an existing project.

There are some pre-requisites libraries needed before compiling the OpenCV library:

```
sudo apt-get install build-essential
sudo apt-get install cmake git libgtk2.0-dev pkg-config libavcodec-dev libavformat-dev libswscale-dev
```

Figure 18: Installing the pre-requisite libraries

To setup and use OpenCV in an existing project, it must first be downloaded and saved locally.
The releases can be found at: https://github.com/opencv/opencv/releases, where v3.4.1 is the most recent at the time of writing, however v3.1.0 is used here.

```
git clone https://github.com/opencv/opencv.git
cd opencv
git checkout 3.1.0
cd ..

git clone https://github.com/opencv/opencv_contrib.git
cd opencv_contrib git checkout 3.1.0
cd ..

cd opencv
mkdir build
cd build
```

Figure 19: Downloading OpenCV v3.1.0 along with the extra modules

Once all the source files have been downloaded, the build files will need to be generated using CMake.

```
cmake -D CMAKE_BUILD_TYPE=RELEASE \
        -D CMAKE_INSTALL_PREFIX=/{Install path}/ \
        -D WITH_TBB=ON \
        -D WITH_V4L=ON \
        -D WITH_OPENGL=ON \
        -D OPENCV_EXTRA_MODULES_PATH=../../opencv_contrib/modules \
        -D BUILD_EXAMPLES=ON
```

Figure 20: Generating the OpenCV build files using CMake

The final step is to build the OpenCV library. This is done by running "make" and then "sudo make install" within the `build/` directory - This will create the required shared object files.

The compiled libraries can then be individually linked using the -l flag or, alternatively, using the pkg-config tool, which will do this automatically (figure 21).

```
g++ main.cpp -L/opt/opencv/build/lib -lopencv_calib3d -lopencv_core -lopencv_features2d -lopencv_flann
-lopencv_highgui -lopencv_imgcodecs -lopencv_imgproc -lopencv_ml -lopencv_objdetect -lopencv_photo
-lopencv_shape -lopencv_stitching -lopencv_superres -lopencv_ts -lopencv_video -lopencv_videoio -
lopencv_videostab

// Using pkg-config
export PKG_CONFIG_PATH=/opt/opencv/lib/pkgconfig g++ ‘pkg-config –cflags –libs opencv‘
```

Figure 21: Linking the OpenCV libraries directly, and using the pkg-config tool

## 5.2 CIFAR10 format

The BNN design accepts images in the CIFAR10 data format. This data format is different from that used in OpenCV, and there is no built in conversion function. The CIFAR10 format requires the red, green and blue channels to be flattened and a label prepended as shown in figure 22.

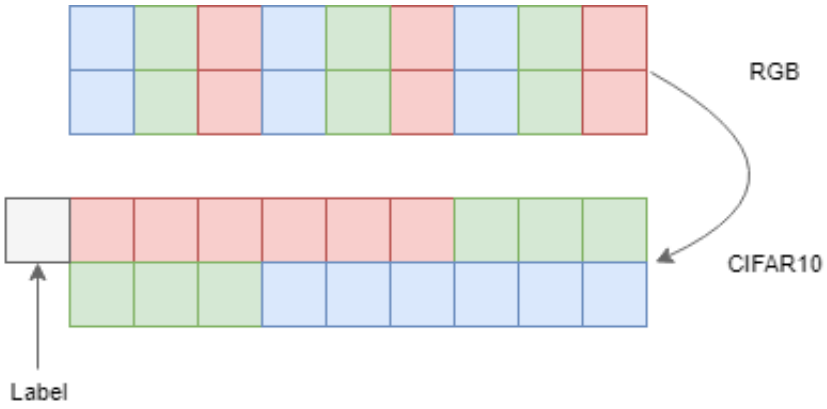


Figure 22: CIFAR10 and RGB formats

A function had to be implemented for converting to and from the CIFAR10 format for classifying regions using the BNN. Although the conversion latency is not significant on its own, when performed multiple times on a single frame, it can prove to be a performance bottleneck. The CIFAR10 format allows multiple images to be contained into a single file, which is used later to exploit the pipelined architecture of the BNN.

## 5.3 Relative size of bounding box

The full system should be able to extract all the regions of interest from a video feed for each frame. These regions are then converted to CIFAR10 format and passed into the BNN for classification, after which the results are then overlayed on the video. The performance of classification is not only dependent on the accuracy of the BNN, but also on the correctness of the regions and relative sizes of the boxes that cover the objects in the scene. This section evaluates the classifica-

tion correctness for using bounding boxes that are too large and too small for the object in view. By evaluating these results, the miss-classifications for the overall system can be rooted to either the BNN or the ROI extraction method.

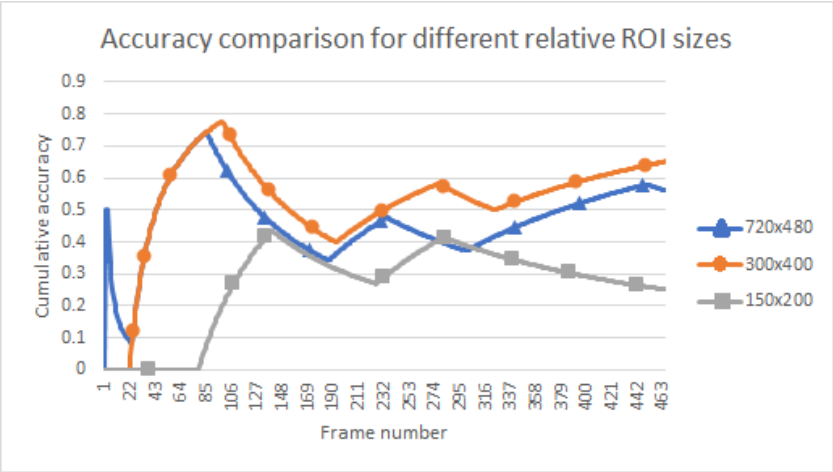


Figure 23: Affects of overestimating and underestimating the region of interest

Figure 23 shows the accuracy for classifying an object using a fixed ROI of different sizes (where 720x480 is the full sized frame). Generally, choosing a larger ROI than needed can still classify the object but with a lower asymptotic accuracy than if a more appropriately sized rectangle is used (which is roughly 300x400). However, undercutting the ROI size dramatically affects the accuracy as there is not enough distinct features for the small rectangle in some of the frames. The edges of the object are often the dominant features and so if they are not captured in the ROI, it will often lead to an incorrect classification.

It is for this reason that a region of interest extraction method should aim to at least cover the entire object, and then try and minimize the area to improve the overall accuracy. If the regions are too small relative to the object, the accuracy performance of the overall system will be significantly degraded.

## 5.4 Motion segmentation using background subtraction

This segmentation approach removes the static background from a frame by subtracting the current frame from the previous. The algorithm works by initially grayscaling the original frames and then applying a Gaussian blur. These two operations simplify the frames and remove high frequency content, so to improve the resilience to noise. The difference between these resultant frames is then computed, and then thresholded to black and white. A hard or adaptive threshold can be set, and will have implication on the effectiveness of this approach. Generally, in a scene where objects are darker, an adaptive threshold is more effective.

The black and white difference frame often includes fragmented regions and small residuals which should not be identified as regions of interest. The residual clusters can be suppressed and the fragmented regions combined by setting an area threshold on the ROI and performing an appropriate morphological operation on the image respectively.

**Algorithm 1** Motion segmentation

```
1:  procedure BackgroundSubtraction
2:      curFrame ← Current Frame
3:      prevFrame ← Previous Frame
4:
5:      // Performing a blur operation helps smoothen the edges in the frames
6:      GRAYSCALE(curFrame)
7:      GRAYSCALE(prevFrame)
8:      BLUR(curFrame)
9:      BLUR(prevFrame)
10:
11:     // Calculate the absolute frame difference and then threshold the result
12:     // and isolate the contours
13:     frame ← |curFrame - prevFrame|
14:     frame ← (frame > threshold)
15:     frame ← dilate(frame)
16:     frame ← erode(frame)
17:
18:     return findContours(frame)
19: end procedure
```

Algorithm 1 shows the implementation of this method for extracting regions of interest. The dilation followed by erosion is known as a morphological close operation and is used to combine fragmented regions.

A more thorough description of the motion segmentation is outlined below. This shows the sequential steps required to segment the image and then identify the regions of interest.

1. The current and previous frames are gray-scaled. This simplifies the algorithm and removes colour dependence.
2. A 20x20 Gaussian blur is then applied to the current and previous gray-scaled frames. The blur operation has the effect of smoothening out the noise in the original frames.
3. The difference frame is computed through the absolute difference of the two blurred frames. The result highlights regions of the frame where there is movement (difference).
4. The difference frame is then thresholded. This is either a hard threshold or an adaptive threshold. Thresholding removes the areas of small movement and strengthens others. Generally, a threshold set to 70 was experimentally found to be effective in a variety of scenes.
5. Due to the fragmented nature of the previous result, the frame is then morphologically transformed to connect regions together and remove small residuals. This is done using a "close" operation, which consists of a dilation followed by an erosion. The dilation expands on clusters of white pixels, whereas the erosion contracts these regions. Combining the two operations effectively connects the clusters together.
6. Finally, the clusters are then contoured to identify the regions of interest.

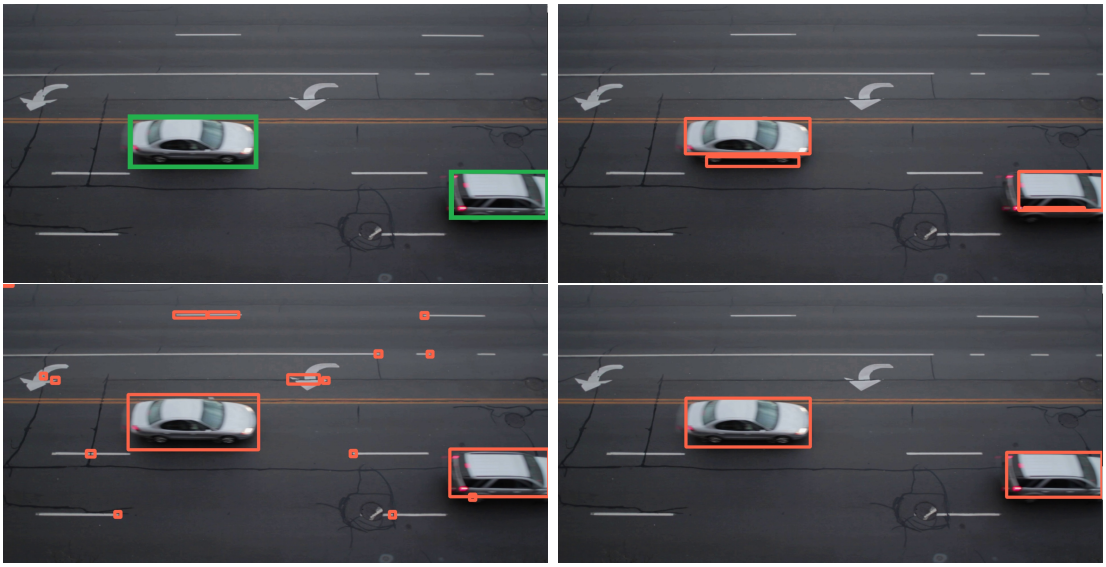

Figure 24: Contouring the difference frame to identify the regions of interest. a,b,c,d from top left to bottom right. a.) Ground truths b.) No morphological transform c.) Dilating the difference frame d.) Closing the difference frame

Figure 25: Dilation introduces artifacts that can be mistaken as regions of interest. a.) Before dilation b.) After dilation

Without applying any morphological transforms, and simply contouring the difference frame, will result in false positives surrounding the regions of interest (see figure 24a). This is caused by the difference frame fragmenting the regions of interest into multiple contours. Having regions that only partially cover the object will lead to significant classification errors, as demonstrated in the previous chapter.

Dilating the image combines the fragmented regions, however, can lead to small, incorrect clusters arising (figure 25c). These small clusters can be partially suppressed by setting an appropriate area threshold. A more effective technique is to apply the morphological close operation. This involves a dilation followed by an erosion, and effectively cancels the introduction of new clusters that are introduced through the dilation. The close operation proved to be very effective at identifying the contours and therefore regions of interest correctly. This is evident by figure 24b which shows the identified regions correctly covering the cars in the frame.

## 5.5 Colour segmentation using k-means clustering

Alternatively known as "image simplification", this method works by clustering regions of the frame that are closely related in terms of colour. This problem aids itself well for use with the kmeans clustering algorithm. Kmeans clustering is an iterative machine learning algorithm that converges on a solution for clustering data into groups, or in this case regions of interest in the frame.

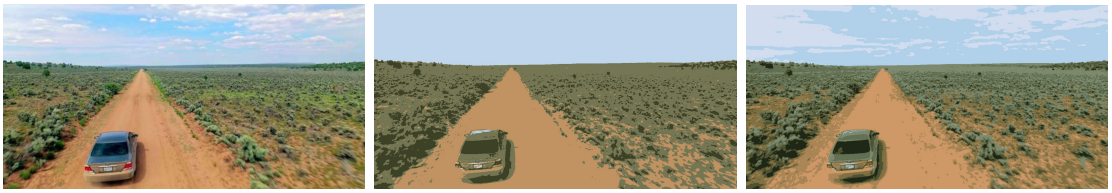

Figure 26: Applying kmeans clustering to segment an image by colour. a) original image b) 4 clusters c) 9 clusters

Computationally demanding methods are often the bottleneck in real-time applications. The kmeans algorithm will significantly degrade the FPS of the system since it needs to operate on a large data set (720x480) with upwards of 10 clusters. An FPGA based implementation of the kmeans clustering algorithm is considered later in this paper, and promises a significant improvement in latency.

Unfortunately, colour segmentation performs poorly on some example frames, as can be seen in figure 26. The sky and the road can easily be contoured and identified as a region of interest, however the car will not be. This is because the regions of interest, under the CIFAR data set (in this case a car), consist of a range of colours, which will be fragmented during the clustering.

## 5.6 Image striding

"Image striding" involves dividing a frame up into pre-defined regions. This approach is indifferent on the activity within a frame, however has little to no computational overhead.

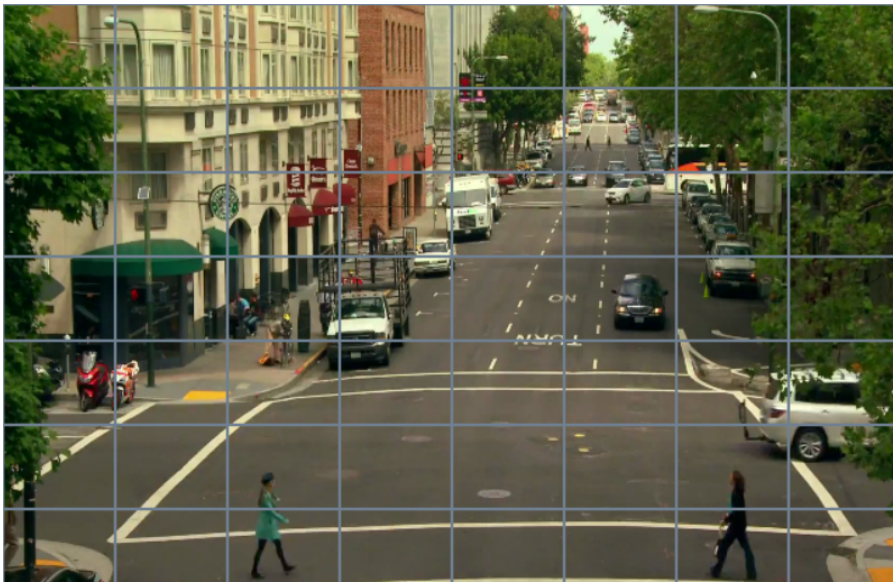

Figure 27: Dividing frame into a grid of regions

Figure 27 shows how the frame can be subdivided into regions of equal size. All the regions are then classified and a certainty threshold is set. This method assumes prior knowledge on the threshold certainty and the ROI sizes, however can be an effective fall-back if other segmentation methods fail to extract any ROIs. This method relies heavily on the classification certainty metric. Unfortunately, chapter 6.1 demonstrates that these metrics can often be misleading and unreliable.

# 6 Accuracy Evaluation

The BNN must successfully classify objects in a real-life environment and still maintain the state of the art accuracy suggested by the results from the test CIFAR10 data set (see Table 2).

| Test Batch | 100 images | 1.000 images | 10,000 images |
|---|---|---|---|
| data_batch_1.bin | 94% | 95.8% | - |
| data_batch_2.bin | 95% | 94.8% | 95.15% |
| test_batch.bin | 78.5% | 78.52% | - |

Table 2: Test batch classification results

This chapter looks at the classification accuracy of the BNN under various test conditions and studies various methods for improving the overall systems accuracy and resilience to noise.

## 6.1 Classification certainty

The BNN returns a vector of probabilities for the classification of an input image. The index of the highest element is associated with the most likely classification. For a given class, the higher output will be indicative of more prominent features in the image, as learnt by the network. Therefore, the absolute and relative magnitude of the output is correlated with the certainty of the classification. There are various metrics that can be used to evaluate the certainty of a given classification.

A certainty metric should pass a few tests. Specifically, the certainty of a classification should degrade as the object is occluded, transformed, or if noise is added. The certainty should also be low on regions where there is no object present.

Mathematically, the classification certainty should be highest when the output vector consists of a single non-zero entry and be lowest when all entries are the same (highest uncertainty). Using these two bounds, a function can be derived

with a sensitivity parameter (as shown in figure 28).

The sensitivity parameter $\lambda$ has the affect of shifting the results and increasing its spread. Generally, it will not change the classification trend.

$$v = \{r_1\ r_2\ r_3\ ...\ r_n\}$$

*Normalise the output vector*

$$v_{norm} = v / \sum_{n=1}^{N} v_n$$

*Define a sensitivity parameter* $\lambda$

$$S = \sum_{n=1}^{N} v_n^{\lambda}$$

*Certainty from 0 to 1,* $\gamma$

$$\gamma = \frac{1}{S}$$

Figure 28: Calculating classification certainty - Metric A

*Assume the results are in ascending order*

$$v = \{r_1\ r_2\ r_3\ ...\ r_n\}$$

$$x = v_n - v_1$$

$$y = v_n - v_{n-1}$$

*Define a sensitivity parameter* $\lambda$

$$\gamma = \frac{y^{\lambda}}{x}$$

Figure 29: Calculating classification certainty - Metric B

Another classification certainty metric is considered here (Metric B - figure 29) that focuses on the absolute value of the maximum element with respect to the minimum element and its neighboring classes (in terms of the next most likely classification). This metric aims to overcome the shortfalls of the other previously derived method (Metric A) that focuses on the spread of results, rather than the

absolute values that are returned. By considering the relative "strength" of the classification to the next most likely class, this certainty should effectively degrade as the classification begins to fail. A sensitivity parameter $\lambda$ is also used to adjust the certainty spread.

The practical results of both these methods will be discussed in the following chapters under noise and occlusion.

## 6.2 Orientation

This test is very dependent on the diversity of images used to train the network. Fortunately, the CIFAR10 training data set contains a very large sample of images which allows the network to learn the features of an object at different angles and orientations.

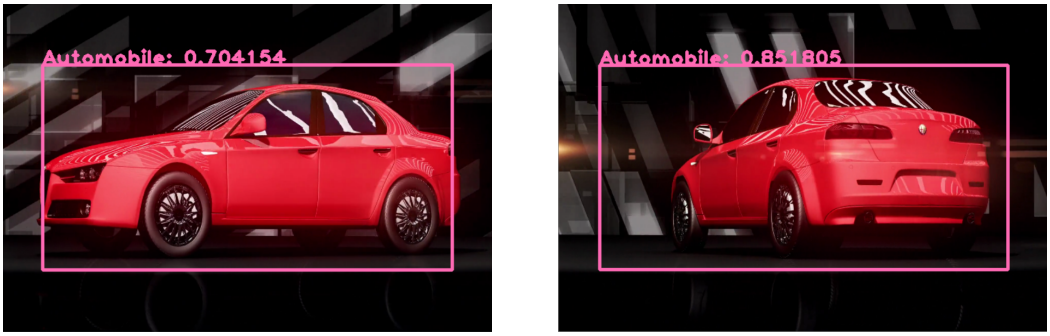


Figure 30: Classifying a video of a rotating car

The BNN was tested on a video of a rotating car (see figure 30). The results were very successful and the BNN managed to achieve 88.16% classification accuracy across 142 frames. When the car was frontward facing, the BNN would occasionally classify it as a "Truck", which is not significantly far from its correct classification.

## 6.3 Occlusion

An image was progressively occluded (one column at a time) and classified by the BNN. This test was to see when the BNN would fail to classify an object and to evaluate this performance both subjectively and objectively using the certainty metrics.

Figure 31: Progressively occluding and classifying an object. a) original image of car b) point at which the BNN fails to classify the object

Figure 31 show the original image of a car and the point at which the BNN fails to classify the car under occlusion. From a subjective standpoint, the BNN performs relatively well as it only fails when 42% of the object is occluded. Similar results can be achieved on different images.

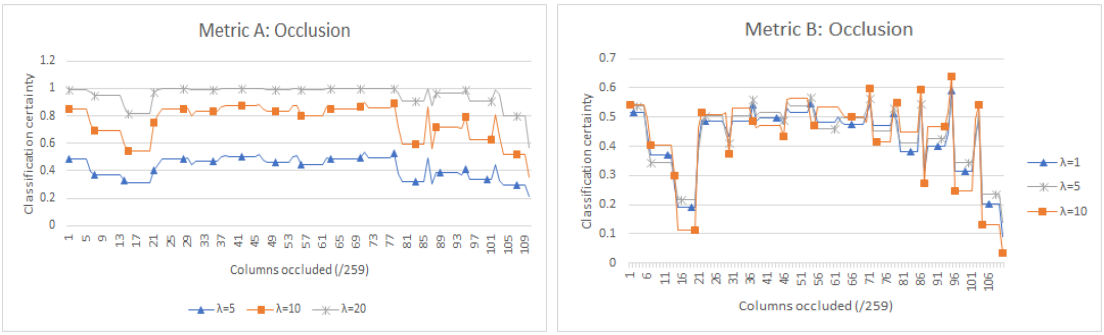


Figure 32: Classification certainty of an image under progressive occlusion for metric A (left) and metric B (right)

Figure 32 shows the classification certainty of the car as it is progressively occluded. The expectation is that the certainty would gradually degrade as the object is more occluded, up until the point at which it fails classification. Both graphs show a downwards trend in certainty after 80 columns of the image have been occluded, however for metric B this is more significant.

Increasing the sensitivity parameter $\lambda$ for metric B significantly increases the spikiness of the certainty as the objects features are removed (through occlusion). There are relatively few sharp spikes at the beginning of the test and so this spikiness nature could be used as a feature for an adaptive system to evaluate whether the BNN is performing well or not.

## 6.4 Noise

Noise is often introduced into a scene through reflections, image quality or the focus of the camera. This experiment looks at the implication of adding artificial noise on the classification of an image. Additive white gaussian noise is used with an increasing standard deviation, up until the classification fails.

Figure 33: Adding Gaussian noise with an increasing standard deviation. a) original image of a deer b) point at which the BNN fails to classify the deer

The BNN performs poorly on images under the application of noise. This is evident by the failed classification show in figure 33b. This stems from the difference

in how neural networks and humans extract features for classification. As a kernel is convolved over an image, the individual pixels can have a significant role in the activation of a given layer. It has been shown that CNNs can suffer from individual pixel attacks [12], and this is likely a problem BNNs can face too. Later chapters on classification windowing discusses methods for suppressing this problem and improving the overall systems resilience to pixel level noise.

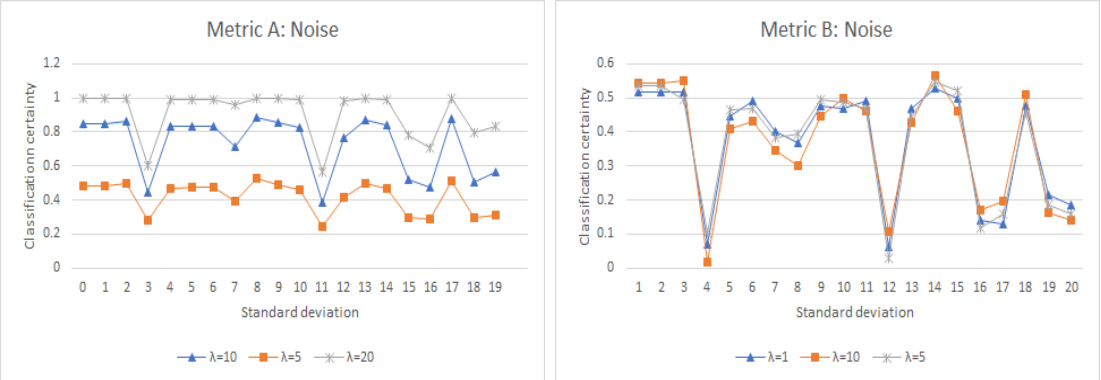


Figure 34: Classification certainty of an image under application of noise for metric A (left) and metric B (right)

Although a gradual decline in certainty as noise is added would be preferable, due to its very nature, this is unlikely to happen. When noise is added, the evaluated certainty does not yield any significant information (as shown in figure 34). These results show the weaknesses in using the classification certainties to correlate with our perception of an objects classification. However, they are sufficient in showing the general trend of classification of a region over multiple frames in a real-life scene which involves a combination of occlusion and noise.

## 6.5 Classification window

When using the Logitech camera as the video feed, and a fixed ROI covering an object, the classification was not fixed across multiple frames. This stems from the quality of the camera, which introduces some noise into the classification, and the

manual focusing required. The overall system would occasionally fail to correctly classify the object in front of the camera, despite no apparent changes to the scene. The classification window approach aims to address this problem.

A classification window averages the BNN results for a given ROI over multiple frames. This has the affect of smoothening out any spurious/wrong classifications. The approach is implemented by summing up the weighted probability vectors of the past N frames. The frame results are weighted from current to oldest frame with an exponential decay. The rate of this weighting decay can be adjusted to smoothen out more erroneous results at a trade-off for classification responsiveness. The classification responsiveness is how quickly the system can react to a sudden change in the object under an ROI. A large window with a low decay constant, will mean that the window will need to be significantly filled up with new results to offset the previous classifications.

**Algorithm 2** Classification window

**procedure** ClassificationWindow
    $\lambda = decay\ constant = 0.2$
    $w = e^{-\lambda n} = \{$ 1 0.818731 0.67032 0.548812 0.449329 0.367879 ... $\}$

    *// Store last N results for C classes*
    $previousResults[N][C] = \{r_{1n},\ r_{2n},\ r_{3n}\ ...\ \}$
    $adjustedResult[C] = \{\}$

    *FOR n = 1 to N*
        $adjustedResult = previousResults[n] * w[n]$
    *END FOR*

    ***return*** *adjustedResult*
**end procedure**

Algorithm 2 shows the pseudo-code for implementing the classification window. The previous N classifications are stored in an $N \times C$ matrix, where C is the number of classes, in this case 10 for the CIFAR10 dataset. The current classification is weighted in accordance with the decay constant $\lambda$. The implementation is very fast in comparison to the other computational bottlenecks of the system, and so it won't see any benefit in being accelerated on the FPGA, even if the implementation can be fully parallelized.

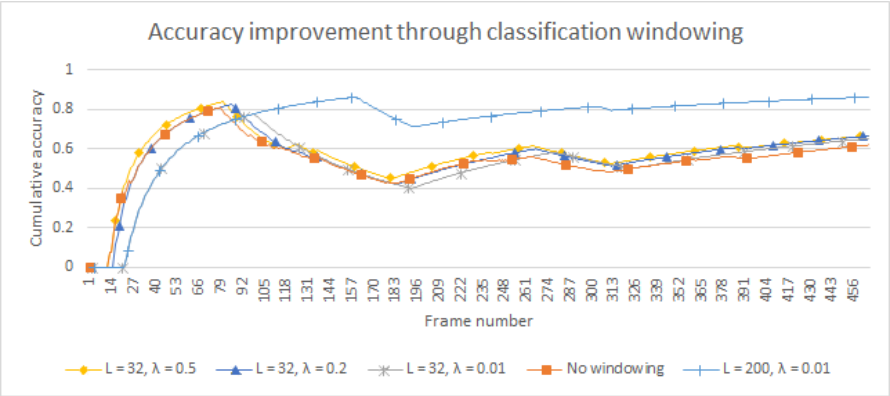


Figure 35: Smoothing classification errors and improving overall accuracy using windowing methods. $\lambda$ is the decay constant and L is the size of the window.

The windowing method was run through lots of examples. Figure 35 shows the classification results for a video with lots of occlusion, camera movement, and changes in lighting conditions. The windowing method will always asymptotically achieve a higher cumulative accuracy (over the entire video) and the size of the window (L) and weight decay($\lambda$) can be adjusted depending on the dynamic nature of the video. If error bursts are only expected to last a few frames, the size of the window can be reduced. However, in the example above, there is $\sim$ 100 frames where the object is completely obscured and so a large window size of $\sim$ 200 is needed to smoothen the errors out. The classification windowing results demonstrate a fundamental correlation between the overall accuracy of the system and the FPS. By increasing the FPS of the camera feed, the window can be increased with less degradation on responsiveness and significant improvements on the overall accuracy.

## 6.6 Object tracking

To be able to use the windowing algorithm alongside the motion segmentation, the system will need to be aware of what past classifications correspond to what current regions of interest. Unfortunately, these objects will typically move around and disappear in and out of the scene. Object tracking aims to address this problem by matching previous and current ROIs based on their relative positions in the frame.

**Algorithm 3** Object tracking

```
1:  procedure ObjectTracking
2:      sceneObjects s = {}
3:      FOR each frame in video
4:
5:          flaggedObjects f = {}
6:          FOR each region of interest r
7:
8:              FOR each sceneObjects sn, j = 0 to N
9:                  IF sn overlaps r AND f[j] = 0
10:                     newObject = false
11:                     f[j] = 1
12:                     s[j].update(sn)
13:                 END IF
14:             END FOR
15:
16:             IF newObject
17:                 s.add(r)
18:             END IF
19:
20:         END FOR
21:     END FOR
22: end procedure
```

By introducing object tracking, the windowing approach can be integrated into a dynamic ROI video (for example on videos with motion or colour segmentation as the approach for extracting ROIs). Algorithm 3 shows a basic implementation for tracking multiple objects across frames. The method works by comparing the

previously identified regions of interest boxes and checking if any of them overlap with the currently identified regions of interest. Those which do overlap correspond to the same object, whereas those which don't will be marked as new objects.

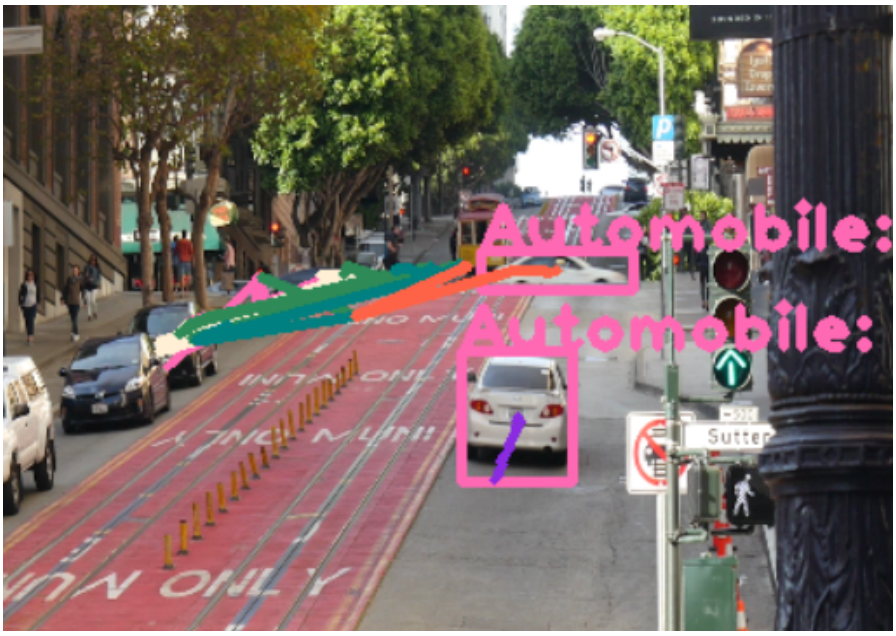


Figure 36: Frame showing the objects in the scene being tracked into and out of the frame using the distinct coloured lines

Figure 36 shows the object tracking algorithm correctly identifying objects and tracking their movement across multiple frames. The purple line shows the previous trajectory of the white car and the orange lines show how a car moved across the frame from the junction at the top of the street. The implementation was proven to be very effective in conjunction with the classification windowing.

# 7 Performance Evaluation

The FPGA based BNN addresses the significant latency of conventional CNN implementations in software. It is for this reason that the overall design should still be able to maintain high FPS and low latency for its application in embedded devices.

## 7.1 Software comparison

Table 3 shows the significant throughput and latency improvements of the hardware and software implementation of the BNN. This shows how limiting software classification can be for a high performance system.

| Implementation | Latency(us/image) | Throughput(images/s) |
|---|---|---|
| Software | 780515 | 1.28 |
| Hardware | 328 | 3050 |

Table 3: Software and hardware latency and throughput comparison

## 7.2 Increasing throughput by filling the pipeline

The BNN implementation uses overlapping computation and communication to maximise throughput. To make full use of this design, the pipeline must be filled on each frame, and this can be achieved by combining multiple input images into batches for the BNN to accept, rather than passing them in one at a time.

| # Images | Latency(us/image) | Throughput(images/s) |
|---|---|---|
| 10 | 512 | 1952 |
| 100 | 346 | 2889 |
| 1,000 | 329 | 3035 |
| 10,000 | 327 | 3050 |

Table 4: Maximising throughput by filling the pipeline

Table 4 shows that there is diminishing returns for more than 100 images. This is when the pipeline will be completely filled. Therefore, to ensure maximum performance from the BNN, there should be at least 50 images for classification under each frame. Unfortunately, this is often very dependent on the activity of the video and how many regions of interest are extracted during segmentation.

The BNN-PYNQ framework passes in images one at a time. This involves individually converting each region to the CIFAR10 format and waiting on its classification before moving onto the next region. Fortunately, the CIFAR10 format allows multiple images to be saved as a single file. By ensuring all the regions of a given frame are saved in a single CIFAR10 file, they can be passed into the BNN through a single call. This significantly reduces the classification latency for a given frame.

The PYNQ examples save the images to a file and then individually pass the paths to a C function that interleaves the image and calls upon the accelerator. Better performance can be achieved by also removing the need to read and write each image to a file for classification, and instead calling directly upon the accelerator after interleaving and converting the OpenCV Mat to CIFAR10 format. The PYNQ framework relies on these read/write operation due to the abstraction of python framework. Fortunately, the system designed in this paper is written in C++, and so communication directly with the drivers is easily achieved.

| Frame Number | Multiple BNN calls with I/O operations | Single BNN call with no I/O operations |
|---|---|---|
| 1 | 282420 | 64522 |
| 2 | 1237166 | 60996 |
| 3 | 208756 | 53755 |
| 4 | 218402 | 53581 |
| 5 | 169165 | 68082 |
| 6 | 163483 | 68129 |
| 7 | 255516 | 75089 |
| 8 | 274572 | 68059 |
| 9 | 298617 | 82244 |
| 10 | 252939 | 89715 |

Table 5: Frame latency improvement by pipelining classification and communicating directly with the BNN drivers.

Table 5 shows the significant frame latency improvement by addressing the pipelining and I/O operations. Generally, the frame latency is reduced by $\sim 4\times$, and this

only becomes even more significant for more active frames with lots of regions of interest for classification.

## 7.3 Reducing BNN resource utilisation

By adjusting the number of SIMD lanes and processing elements (PE), the resource utilisation of the BNN design can be controlled. This will sacrifice the throughput of classification, however, will allow other computation to be accelerated on the FPGA, alongside the BNN.

The number of SIMD lanes (S) and processing elements (P) determines how the matrix-vector products are partitioned. The number of clock cycles required to multiply an $X \times Y$ matrix is given by $\frac{X}{P} \times \frac{Y}{S}$. This leads to a trade-off between resource utilisation and throughput.

For each layer, the number of SIMD lanes and processing elements must be calculated for a given FPS. This gives the notion of a synapse fold $F^s$ and a neuron fold $F^n$, from which the total fold $F$ can be obtained.

$$F^n = \frac{X}{P}$$

$$F^s = \frac{Y}{S}$$

$$F = F^n.F^s = \frac{F_{clk}}{FPS}$$

For an FPS of 200 images/s, the configuration parameters of each layer can be calculated:

$$P = \#peCounts = \{4, 8, 4, 4, 1, 1, 1, 1, 4\}$$

$$S = \#simdCounts = \{3, 8, 8, 8, 8, 2, 1, 1, 1\}$$

After evaluating the required number of SIMD channels and PE blocks, WMEM and TMEM need to be adjusted and the training weights need to be generated

by retraining the BNN using the supplied PYNQ training script found in bnn/src/training/.

| Name | BRAM_18K | DSP48E | FF | LUT |
|---|---|---|---|---|
| DSP | - | - | - | - |
| Expression | - | - | - | - |
| FIFO | - | - | - | - |
| Instance | 81 | 26 | 18908 | 32236 |
| Memory | 157 | - | 824 | 388 |
| Multiplexer | - | - | - | 537 |
| Register | - | - | 159 | - |
| Total | 238 | 26 | 19891 | 33161 |
| Available | 280 | 220 | 106400 | 53200 |
| Utilization (%) | 85 | 11 | 18 | 62 |

Table 6: Utilization Estimates for a BNN design with the reduced number of SIMD lanes and PE elements

| Topology | Accuracy(%) | Inference time (usec per image) | Classification rate (per second) |
|---|---|---|---|
| Original | 95.02 | 328.057 | 3048.25 |
| Reduced | 95.8 | 4621.78 | 216.367 |

Table 7: Latency comparison between the two BNN topologies over 5000 test images

As shown in table 7, the classification rate is sacrificed in return for reducing the number of resources. This is because there is less processing elements in the design and therefore less parallelism. However, the classification rate is still very fast and does not impose any significant bottleneck on the overall design.

## 7.4 Accelerating computation

Table 8 shows that the dilation operation and the kmeans clustering algorithm are the most significant latency bottlenecks for the motion and colour segmentation respectively. By using a reduced BNN topology, these algorithms can be accelerated alongside the BNN.

| Name | Latency (us) - averaged over 10 frames (720x480 frame) |
|---|---|
| Background subtraction | 3910 |
| Dilation | 131662 |
| Contouring | 6901 |
| Kmeans clustering | 13922440 |

Table 8: Latency bottlenecks in design

### 7.4.1 Drivers for the separate IP blocks

To accelerate computation alongside the BNN, the HLS design has to be exported as a seperate IP block and then mapped to the AXI interconnects of the zynq processor system.

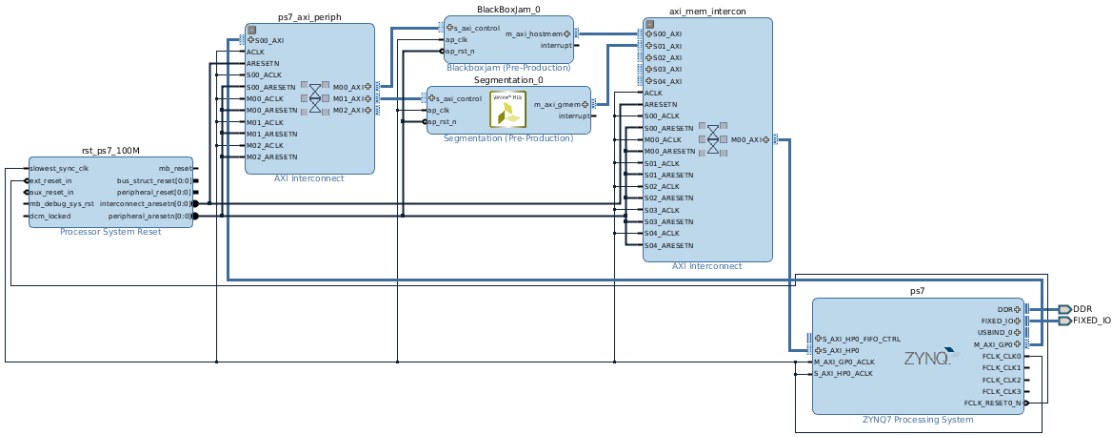


Figure 37: Block diagram of the BNN and the motion segmentation algorithm as seperate IP blocks

Figure 37 shows the "Segmentation" IP block connected to the AXI peripheral and AXI memory interconnect. The top level function of the design should use AXI4 interfaces for the arguments, which can be bundled together into a single port using the INTERFACES pragma. The Vivado software will automatically connect the AXI ports to the processor system when block automation is run.

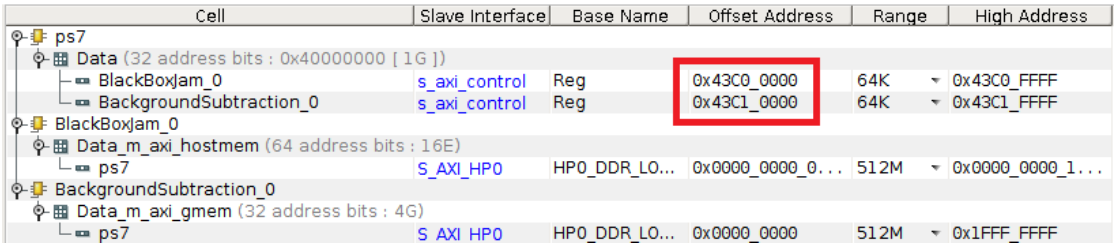


Figure 38: Hardware offset addresses for IP blocks

The two IP blocks will have different base offset addresses (see figure 38), which will need to be noted for the drivers. The hardware addresses of the individual IP block top level arguments can be found in the impl/ip/drivers/{project_name}/src folder of your HLS project. Once the complete design has been synthesized, implemented and exported as a bitstream, the bit file can then be used to create the boot image (as described in an earlier chapter on porting the PYNQ framework). As the new IP block uses a different offset address, but has the same AXI interface, the PYNQ supplied Xlnk driver class can be used. This class has some utility functions to initiate the accelerator and check if it is idle, ready, or busy.

### 7.4.2 Dilation algorithm using HLS

The dilation algorithm expands the white pixels of a binary image. A white pixel in the original image results in all neighboring pixels being changed to white over a window, which is either a preset ellipse or a rectangle. Due to the nature of this operation, there are lots of loops and sub loops, which is the reason behind the significant software latency.

The HLS video library [13] provides a set of synthesize-able OpenCV functions to accelerate image/video processing on the FPGA. However, the dilation algorithm considered here is implemented directly in HLS, and the host driver communicates with the FPGA using contiguous memory allocation with a physical pointer. The implementation then reads/writes to the host memory in accordance with the algorithm. This approach incurs a significant overhead, since there will be lots of read/write operations from the FPGA, especially when the input image is large.

The dilation algorithm itself is not significantly computationally expensive in terms of arithmetic operations. Along with tight constraints on available FPGA resources, it is difficult to achieve significant latency improvements. However, through carefully pipelining the implementation, the FPGA can demonstrate near software level performance.

The Linux kernel must also have sufficient memory area for contiguous memory allocation of the extra accelerated image processing block. This can be set when compiling the Linux kernel in the menuconfig (256MB was sufficient here).

| Frame size | Hardware latency | Software latency |
|---|---|---|
| 720x480 | 57944448 | 98936 |
| 360x240 | 11475790 | 35516 |
| 180x120 | 318209 | 139726 |
| 90x60 | 2423 | 1955 |
| 45x30 | 934 | 1024 |

Table 9: Latency comparison for dilation in software and hardware in us

Table 9 shows the latency for dilating an image in software and in hardware. Both the algorithms use a 20x20 window and the hardware implementation requires that the OpenCV matrix frame is converted into an appropriate data format before being offloaded. This conversion overhead is not a significant factor, however reading and writing to host memory from the FPGA is. Also, the design can also be optimised to mitigate the need for a conversion at all, and so only the latency without this conversion will be considered for this analysis. The dilation implementation uses the DATAFLOW pragma to increase concurrency, and the loops and sub-loops are unrolled as much as possible to utilize all the remaining LUTs.

A typical video stream may use a 720x480 frame size and so this will be chosen as the reference for comparison. The accelerated dilation algorithm stores the frame in host memory and so there is a latency overhead through the use of off-chip memory access (as mentioned earlier). It can be seen in table 9 that this overhead is the dominant factor for any frame size above 45x30, and will lead to a significantly higher latency than its software implementation. The offloaded

computation is only beneficial for small frame sizes, however the results can be combined to determine the full size dilated frame. For example, the 45x30 dilation will need to called upon 256 times, and combined, in order to determine the full 720x480 dilated frame. It is clear that the software implementation will still outperform the hardware implementation on a 45x30 frame, even when considering the hardware latencies lower bound - A 45x30 hardware implementation will require $934 \times 256 = 239104$, which is greater than the software implementation for a 720x480 frame.

If a larger FPGA is made available, more of the algorithm could be parallelized such that the computational latency is reduced to clock rate. The dilation algorithm latency can be significantly improved when ported onto the FPGA fabric, however to practically realize this accelerated block, a different approach would need to be used for offloading memory onto the FPGA. The blocks could be converted into local streams, before being written back to the host memory after computation. This would improve the overall design flow and is similar to how the HLS video library works, which will be discussed in the next chapter.

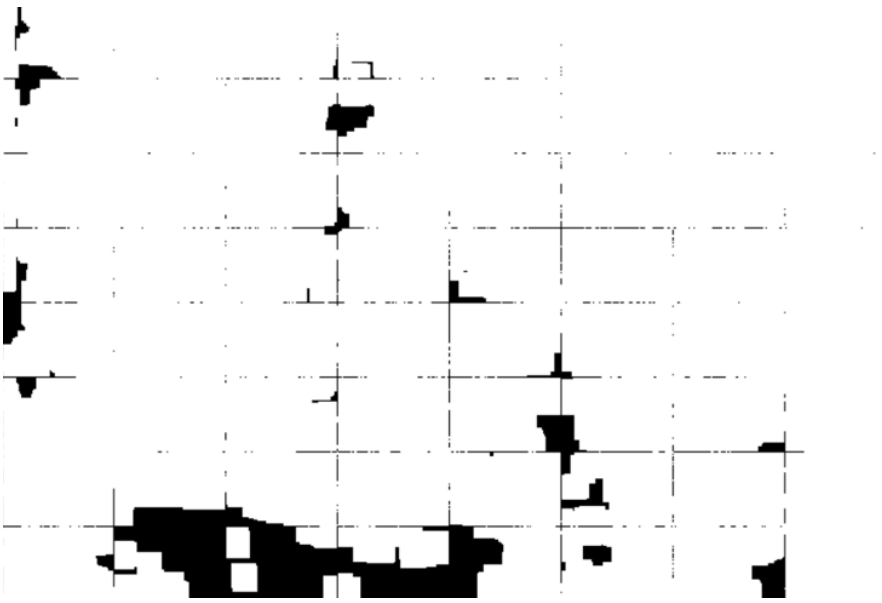

Figure 39: Block dilation artifacts when recombining to form the frame

Dilating the image in blocks also introduces artifacts at the boundaries of these blocks when combined together to form the complete dilated 720x480 frame (as shown in figure 39). These artifacts arise from the clusters that cross over between the boundaries of the blocks. They can be suppressed by removing the boundary checks in the dilation algorithm implementation and appropriately interleaving the data.

### 7.4.3 Motion segmentation using HLS video library

The HLS video library promises a means of accelerating each step of the motion segmentation algorithm, not just the dilation operation. The HLS library is designed to heavily utilize the DSP units of the FPGA for operations such as blurring and for which the current design makes very little use of. The HLS design will accelerate the blurring, absolute difference, thresholding, dilation and eroding of the current and previous frame. The HLS video library uses streams, which not only offer an improved data-rate, but allows the resource utilisation to relaxed. The entire frame can be offloaded onto the accelerator, without any significant degradation in latency performance (like with the custom dilation HLS implementation). This is because this implementation operates on internal streams, rather than constant reads and writes to host memory.

The current and previous frame interfaces cannot be bundled into the same port, else the synthesized design can have deadlocks. Both the current and previous AXI interfaces must have their own AXI mem slave address and must be mapped accordingly to the ps7_axi_periph and axi_mem_intercon blocks.

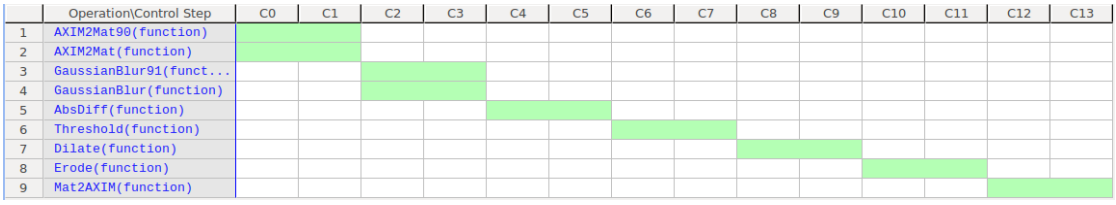


Figure 40: Overlapping computation to improve latency

Figure 40 shows the data flow of the motion segmentation using overlapping communication for AXIM2Mat and Gaussian Blur operations. This is achieved using the DATAFLOW pragma. However, it was found that the DATAFLOW pragma cannot be applied at the AXI interface (AXIM2Mat), as the current and previous cannot be read at the same time else it will invalidate the algorithm design. Only the Gaussian blur operations can be overlapped because of this interface restriction and the sequential nature of the algorithm.

| Implementation | Latency (us) |
|---|---|
| Software | 68020 |
| Hardware (after conversion and offloading) | 495 |
| Hardware (after conversion, before offloading) | 1837 |
| Hardware (before conversion and offloading) | 4783 |

Table 10: Final latency improvements of the motion segmentation algorithm through hardware acceleration (without erosion on a 150x100 frame)

Table 10 shows the significant latency improvements through the hardware acceleration using the HLS video library. The table shows the upper and lower bound latencies for the hardware accelerator from before converting the image through to after the result has been read out and converted back to a final image.

These results show that there is a significant improvement in the motion segmentation when offloaded effectively onto the FPGA. The algorithm performance shows a 125× reduction in latency and a practical 15× improvement. The design could easily be extended to include the erosion algorithm or a larger frame, depending on the BRAMs available on the FPGA.

Figure 41 shows the synthesized schematic of the motion segmentation algorithm implemented using the HLS video library. Highlighted in red (and covering much of the design view further down) shows the blocks used to read in the current and previous frame data for the internal stream. The size of these frames directly corresponds to the number of blocks that are inferred and explains the trade-off between resource utilisation and the maximum frame size that can be supported for the implementation.

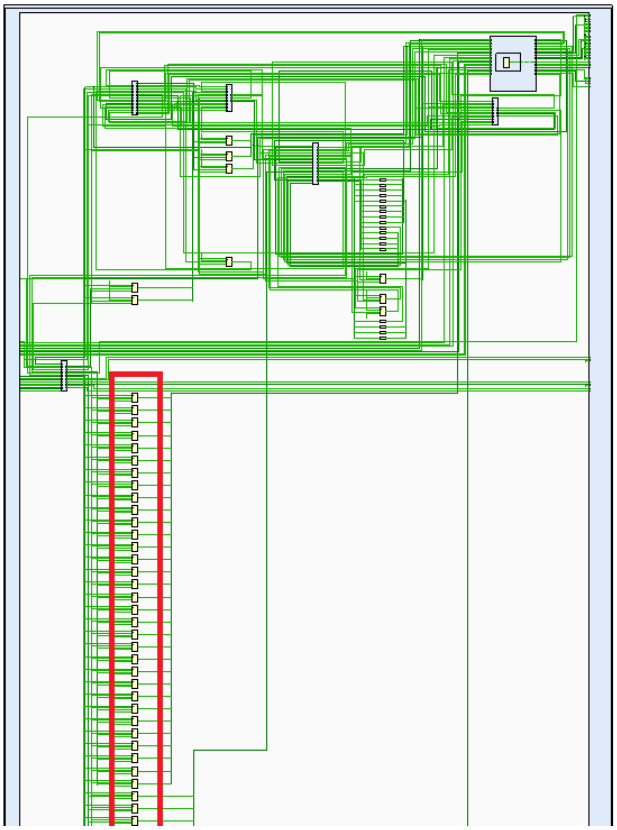

Figure 41: Schematic for the motion segmentation implementation

### 7.4.4 Tree-based k-means clustering

Recent research has demonstrated a significant latency improvement by using a tree-based kmeans implementation on an FPGA [14].

This implementation was synthesized alongside the reduced BNN for a data dimensionality of 1, 5400 data points and 5 clustering iterations. This paper does not cover the complete integration of this accelerated block alongside the BNN but it shows promise on improving the colour segmentation latency. Writing the drivers would be difficult as the hardware design uses complex tree based data structures, which may not offer an easy interface for the Zynq processor system.

## 7.5 Complete design

The overall design covers camera capture to object classification. This includes lots of intermediate steps with various computation being offloaded onto the programmable logic. A full pipelined design can be seen in figure 42 and shows the use motion segmentation, with the dilation algorithm being accelerated on the FPGA. However, it has been demonstrated that more of the design flow can be accelerated on the FPGA depending on the BRAM resources available.

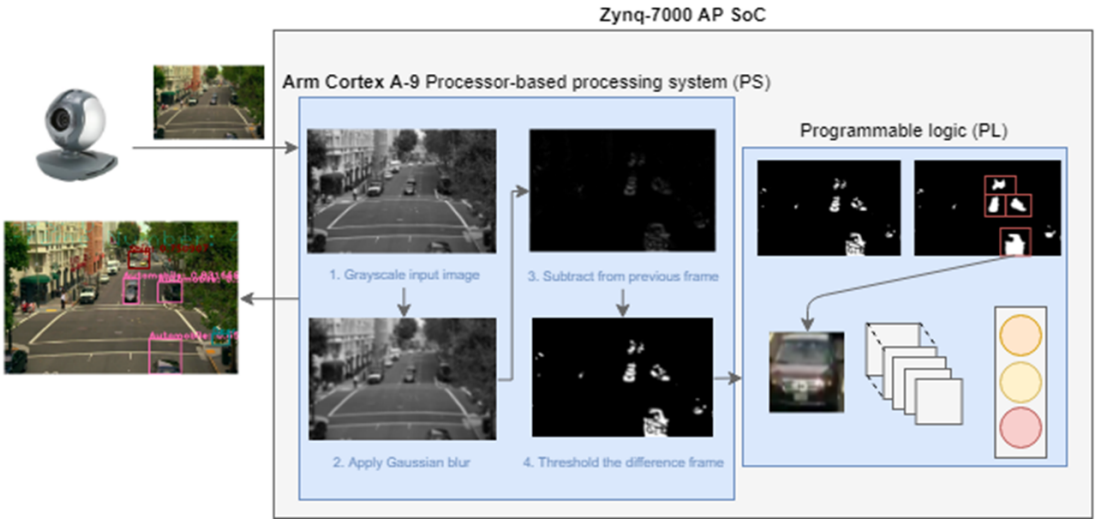


Figure 42: Pipelined design for camera feed to classification, using background subtraction

| System | FPS (640x480) | FPS (176x144) |
|---|---|---|
| Raw camera feed | 40 | 450 |
| Camera + single ROI classification with no windowing | 12 | 18 |
| Camera + single ROI classification with windowing and object tracking | 11 | 15 |

Table 11: FPS using the camera feed at different resolutions (reduced BNN topology)

Table 11 shows the FPS of the system using a Logitech camera. The raw FPS of the camera is very high, however the classification incurs an upper-bound on the FPS that can be achieved. This is not limited by the BNN implementation or the segmentation accelerator, but instead by the overhead of resizing and converting

the region to CIFAR10 format. The practical FPS of the system can be dramatically increased by optimizing this area of the design and looking into retraining the network on RGB images, to avoid the need for any CIFAR10 conversion.

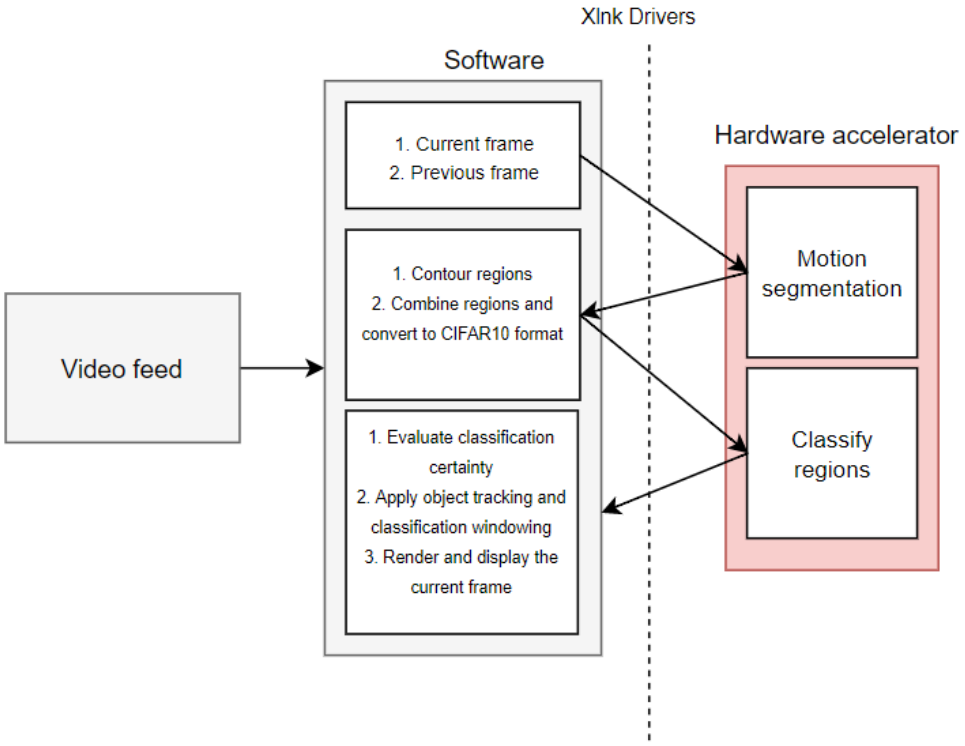


Figure 43: Final block diagram of the system.

Figure 43 shows the complete flow of the system in more detail. This is from video capture, through frame segmentation to classification, and finally the rendering and display of the frame with the overlayed results. The current bottlenecks that have yet to be addressed lie in the CIFAR10 conversion (as mentioned previously) and the Xlnk driver interface, specifically the unnecessary copying of data to and from contiguous memory. After addressing these final bottlenecks, the overall design will only be capped by the Logitech cameras FPS. All other aspects of the design will not significantly benefit from hardware acceleration, specifically the classification windowing and the object tracking - The classification windowing is very simple to compute and the object tracking manipulates an array of structs, which will not be easily ported onto the FPGA fabric.

| Component | Latency(us) |
|---|---|
| 1. Read in and resize current frame | 78806 |
| 2. Motion segmentation | 4236 |
| 2a. HLS algorithm | 508 |
| 2b. Segmentation, including offloading | 2312 |
| 3. Pipelined ROI classification | 47141 |
| 3a. Inference | 27568 |
| 3b. Inference, including offloading | 39610 |
| 4. Motion tracking and classification windowing | 33 |
| 5. Rendering and drawing to screen | 2190 |
| Total | 132406 |

Table 12: Latency of the complete pipelined design using a 150x100 frame on 3 ROIs using a video file source

The complete pipelined design demonstrates very low latency for frame capture through to image classification. The design uses a 150x100 frame, since the typical QCIF (176x144) resolution marginally overuses the available FPGA resources (BRAM usage 285/280) for the segmentation IP. Although reading in the frame takes up a considerable portion of the overall pipeline (see Table 12), it cannot be addressed directly without improving the hardware of the current setup. When using a video file source, the design can achieve $\sim 8FPS$ and when using the Logitech camera as the source, the FPS can be increased significantly to around $\sim 160FPS$, although this drops to $\sim 36FPS$ on frames where there are ROIs for classification. The complete design will easily scale well for larger frames and demonstrates superior performance over other current designs that rely solely on software computation. For example, the "YOLO Real-Time Object Detection" [15] demonstrates $\sim 30FPS$ but relies on high performance GPUs, such as the Pascal Titan X.

# 8 Conclusions and Future Work

This paper shows the benefits of using a BNN in a practical application. Methods have been proposed that demonstrate an effective complete pipelined system for capturing, identifying and classifying objects in real time. The full framework of which is available on GitHub. The system can achieve high FPS on a relatively slow processor, which suggests very high frame rates on less restrictive hardware.

The current framework has addressed the significant latency bottlenecks in the process of extracting regions of interest using motion segmentation along with efficiently pipelining these regions for classification by the BNN. The main factors now limiting the maximum FPS come down to the drivers and the CIFAR10 conversion. The BNN could be retrained on RGB images to remove the need for this conversion. To maximise computational performance, the CPU could also perform extra computation while waiting on the FPGA accelerators. This feature could be integrated through the use of interrupts or an extra software thread.

Future research could look into the integration of the kmeans algorithm to be accelerated alongside the BNN. The design could then be used to segment and classify objects on still images, with very low latency.

The kmeans algorithm could also be integrated into the motion segmentation to improve the overall correctness of the ROI extraction. The colour segmentation and motion segmentation both have independent strengths and weakness in terms of extracting interesting regions from a frame, and so combining the two techniques could significantly improve the robustness of the system. This could also pave way for an adaptive system, which could use the certainty metrics as feedback on the correctness of the current ROI method.

# 9 References


[1] G. E. H. Alex Krizhevsky Ilya Sutskever, *Imagenet classification with deep convolutional neural networks*, https://papers.nips.cc/paper/4824-imagenet-classification-with-deep-convolutional-neural-networks, 2012.

[2] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre and K. Vissers, 'Finn: A framework for fast, scalable binarized neural network inference', in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ser. FPGA '17, ACM, 2017, pp. 65–74.

[3] *Rethinking numerical representation for deep neural networks*, https://openreview.net/pdf?id=BJ_MGwqlg, 2017.

[4] *Applications of convolutional neural networks*, http://ijcsit.com/docs/Volume7/vol7issue5/ijcsit20160705014.pdf, 2016.

[5] *Rectified linear units improve restricted boltzmann machines*, http://www.cs.toronto.edu/~fritz/absps/reluICML.pdf, 2010.

[6] *Batch normalization: Accelerating deep network training by reducing internal covariate shift*, https://arxiv.org/abs/1502.03167, 2015.

[7] *Pipecnn: An opencl-based open-source fpga accelerator for convolution neural networks*, https://github.com/doonny/PipeCNN/blob/master/documents/FPT2017-PipeCNN.pdf, 2017.

[8] P. parikh, *Pynq linux on zedboard*, https://superuser.blog/pynq-linux-on-zedboard/, 2017.

[9] *Open-source project from xilinx for building flexible application for zynq platforms*, http://www.pynq.io/, 2018.

[10] *Bnn implementation for the pynq platform*, https://github.com/Xilinx/BNN-PYNQ/, 2018.

[11] *Zynq-7000 porting tutorial*, http://nbviewer.jupyter.org/gist/PeterOgden/9a5054a06408d2bd711d6de563281930.

[12] *One pixel attack for fooling deep neural networks*, https://arxiv.org/abs/1710.08864, 2018.

[13] *Hls video library*, http://www.wiki.xilinx.com/HLS+Video+Library, 2018.

[14] *Fpga-based k-means clustering using tree-based data structures*, http://ieeexplore.ieee.org/document/6645501/, 2013.

[15] *Yolo: Real-time object detection*, https://pjreddie.com/darknet/yolo/.

# 10 Appendices

The C++ framework developed and used for this paper can be found at: https://github.com/iyop45/Final-year-project2