國立清華大學 電機工程學系

實作專題研究成果摘要

An FPGA Deep Learning Accelerator for Potato Leaf Disease Recognition

針對馬鈴薯葉片疾病辨識的 FPGA 深度學習加速器

專題領域：系統領域

組別：B425

指導教授：鄭桂忠 教授

組員姓名：曾偉博、張仕謙

研究期間：113年1月20日至 113年11月20日止，共10個月

摘要

隨著台灣進入高齡化及少子化，未來勞動力不足將成為現實。本次的研究題目【針對馬鈴薯葉片疾病辨識的 FPGA 深度學習加速器】就是為了透過科技的力量降低農業領域所需的勞動力。考慮到產業特性及成本，我們會希望模型可以在遠端計算能力較弱的設備上推理，而不需每次都將圖片傳回雲端進行分析，為了達成這個目的，我們會將原先使用 Pytorch 建構的神經網路改成用硬體描述語言(Verilog)的方式實作，最終的目標是放置於 FPGA 板之上，之所以不基於 ASIC 設計電路，則是考量到該電路就算未來能夠面向市場，需求的數量也不會太多，所以我們最後決定以放在 FPGA 板上為目標設計深度學習加速器的數位電路。

Abstract

As Taiwan faces an aging population and declining birth rates, future labor shortages are becoming a reality. Our research project, **"An FPGA Deep Learning Accelerator for Potato Leaf Disease Recognition,"** aims to reduce the labor required in the agricultural sector through technological means. Considering industry characteristics and costs, we hope the model can perform inference on remote devices with limited computational capabilities, eliminating the need to transmit images back to the cloud for analysis each time. To achieve this, we will implement the neural network originally built using PyTorch in a hardware description language (Verilog). The ultimate goal is to deploy it on an FPGA board. We have opted not to design the circuit based on ASIC technology because, even if the circuit could be marketed in the future, the demand would not be high. Therefore, we decided to design a digital circuit for a deep learning accelerator with the goal of implementing it on an FPGA board.

Table of Contents:

一、 報告內容

1. 研究背景/動機

隨著台灣勞動力不足，政府開始大力推行智慧農業，本次專題就是基於這個理念孕育而生，並且我們決定從最容易實行的農作物疾病防治下手，選擇的農作物則是常見的糧食作物—馬鈴薯，在我們的日常生活中隨處可見，並藉由公開的訓練資料集進行軟體模型訓練以及對應的硬體模型設計。在我們這次的專題中，馬鈴薯葉片會有三種狀態，分別是 Healthy (健康)、Early blight (感染茄鏈隔孢菌)、Late blight (感染致病疫霉)。

2. 研究目的

為了兼具成本效益及實際用途，我們會希望最終用來推理的模型能運行於遠端，而不是回傳雲端運算，主要的原因在於我們預期應用該技術的農地占地應該會非常大，而且不一定是在溫室當中，假如不想額外花費成本在農地周遭建立網路連線，那麼勢必需要在遠端設備，計算能力較弱的裝置上進行邊緣運算，所以最終我們認為利用 FPGA 承載這個模型最為合適。此外，我們的數位電路是針對 PYNQ-Z2 板進行設計，選擇這個板子最主要的原因有兩個，第一點在於價格親民，第二點則是其承載 ARM Cortex-A9 處理器，剛好提供我們除了在 FPGA 軟體上確認結果之外，實際運行並查看電路是否正確運行的機會。

3. 研究方法與結果

3.1. 研究流程

資料預處理→模型訓練→模型量化→參數提取→硬體描述語言實作數位電路

以上是本次實驗的具體流程，前四部份是軟體的實作，最後是硬體的實作。在第一部分中，我們根據不同模型調整資料集圖片的大小和顏色通道數目，處理訓練資料集。在第二部分中，我們用 Pytorch 建構一些常見的神經網路模型，再根據我們的目標調整模型中的池化層、卷積層與線性層的數目、卷積層的 input 與 output channel 等等，最後決定對哪一個模型進行硬體實作。在第三部分中，我們將建構並訓練過的模型以 Post-Training Quantization(PTQ)的方式進行量化。在第四部分，我們從選定並量化完的模型中提取參數，如權重、量化尺度(scale)、零點(zero-point)等等硬體實作時需要的參數，並且依照特定資料儲存格式輸出至 txt 檔中，以初始化 FPGA 的權重。在第五部分，我們則會根據之前選定並量化的模型開始設計硬體結構。以上五點就是我們這一次專題的研究流程。

3.2. 軟體模型設計與結果

3.2.1. 前處理

在前處理的部分，我們使用了 torchvision 提供的工具來對影像進行處理，包括圖像的色彩空間轉換與圖片大小調整，目的是讓影像資料符合機器學習模型的輸入要求。首先，根據模型需求，選擇是否透過 transforms.Grayscale 函式，將彩色圖像轉換為灰度空間圖像，並設定輸出的通道數。此外，透過調整函式 transforms.Resize((x, y)) 中的參數(x, y)決定影像大小，將每張影像調整為指定的尺寸，如 LeNet 輸入影像為 28*28 的圖片。

3.2.2. 模型訓練、量化方式與參數提取

我們先挑選常見的 CNN 模型作為參考，然後再根據我們的需求客製化最符合我們這個任務的神經網路。為了對模型進行比較，在訓練模型的階段中，我們統一將 loss function 設為 nn.CrossEntropyLoss，選用 torch.optim.Adam 作為 optimizer，將超參數設置為 batch size = 32、learning rate = 0.00001，並且採用 early stop 的方式防止模型發生過擬合。

我們選取的模型共有以下十個，分別是 LeNet、AlexNet、VGG11、VGG13、VGG16、VGG19、ResNet50、DenseNet169、MobileNetV1、GoogleNet。同時，我們也對其進行量化，採用的量化設置為 fbgemm 是 X86 CPU 建議的量化設置，最終的模型訓練及量化後結果，按照尚未量化之模型正確率由高至低排序如下表(Table 1)所示:

Table 1: 各模型量化前與量化後正確率

Model	**DenseNet169**	**GoogleNet**	**VGG19**	**VGG11**	**MobileNetV1**
Original accuracy	98.58%	98.44%	98.44%	97.30%	97.30%
Quantized accuracy	98.44%	98.01%	96.73%	97.02%	91.48%
# of parameters	12.5M	10.3M	139.5M	128.7M	3.2M
Model	**RestNet50**	**VGG13**	**VGG16**	**AlexNet**	**LeNet**
Original accuracy	97.16%	97.15%	97.02%	93.32%	80.68%
Quantized accuracy	96.45%	97.02%	96.88%	93.04%	80.54%
# of parameters	23.5M	128.9M	134.2M	58.1M	43k

模型建構並訓練完成後，我們接下來會討論正確率的結果，從上表來看，除了 LeNet 以外，其餘 9 個常見的 CNN 模型正確率都超過 90%，甚至大部分精確度都來到 97%左右，這樣看起來我們似乎不需要額外設計我們的模型，但事實上仍有需要權衡之處，我們量測模型中的參數數量(Table 1)。

在這些模型中，假如我們要原始模型和量化後的模型正確率都大於 90%，使用最少參數的模型為 MobileNetV1，不過依舊有三百萬的參數，再將其量化後參數資訊儲存起來，得參數檔案大小約 3.6Mb。然而，考慮到 PYNQ-Z2 上的 block ram 容量為 4.9Mb，單單儲存參數就佔了超過 70%的 block ram 空間資源。為了避免此情況，我們的目標是客製化一個神經網路模型，並且模型量化前後的正確率都可以維持大於 90%，而參數使用量則是越少越好，以達到有效率節省資源但維持正確率的目的。我們先以 LeNet 為基準，藉由調整 LeNet 每一層結構來達成我們的目的。LeNet 模型共有兩層卷積層和三層線性層，卷積層之間有池化層和激活函數，線性層之間有激活函數層。

我們發現參數大多來源於線性層，所以降低參數總量最有效的方法就是透過增加池化層來達成減少進入線性層的輸入特徵和輸出特徵。至於模型正確率無法提升到 90%以上的主要原因在於卷積層數量不夠多，在卷積層夠多的情況下，我們就可以提升正確率，並且透過池化層來減少每層 input activation 的尺寸，以降低模型參數量。此外，正確率上不去的另一原因就是 LeNet 模型使用的是灰階圖片。最後我們透過建構不同的模型驗證前面的假設，結果如下表(Table 2):

Table 2

驗證假設方法	參數量	正確率
不使用灰階圖片	略為增加	增加約 6%
增加卷積層(CONV2D)	小幅度增加	增加約 7%
增加池化層(MAXPOOL)數目	大幅降低參數數量	減少約 3%
訓練時使用 DROPOUT	不影響	增加約 1%

最終，透過上述結論以及反覆嘗試，我們成功設計我們的 Custom 神經網路模型(圖一)，再將結果與之前建構常見的模型進行對比(Table 3)。

Table 3

Model	# of parameters	Original accuracy	Quantized accuracy
VGG19	139.5M	98.44%	96.73%
VGG16	134.2M	97.02%	96.88%
VGG13	128.9M	97.15%	97.02%
VGG11	128.7M	97.3%	97.02%
AlexNet	58.1M	93.32%	93.04%
RestNet50	23.5M	97.16%	96.45%
DenseNet169	12.5M	98.58%	98.44%
GoogleNet	10.3M	98.44%	98.01%
MobileNetV1	3.2M	97.3%	91.48%
LeNet	43k	80.68%	80.54%
Our Design(Custom)	7861	90.63%	90.48%

圖一: Custom 神經網路的結構

```
M(
  (features): Sequential(
    (0): Conv2d(3, 6, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (1): ReLU(inplace=True)
    (2): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (3): Conv2d(6, 6, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (4): ReLU(inplace=True)
    (5): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (6): Conv2d(6, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (7): ReLU(inplace=True)
    (8): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (9): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (10): ReLU(inplace=True)
    (11): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
    (12): Conv2d(16, 16, kernel_size=(3, 3), stride=(1, 1), padding=(1, 1))
    (13): ReLU(inplace=True)
    (14): MaxPool2d(kernel_size=2, stride=2, padding=0, dilation=1, ceil_mode=False)
  )
  (classifier): Sequential(
    (0): Linear(in_features=64, out_features=24, bias=True)
    (1): ReLU(inplace=True)
    (2): Dropout(p=0.5, inplace=False)
    (3): Linear(in_features=24, out_features=10, bias=True)
    (4): ReLU(inplace=True)
    (5): Dropout(p=0.5, inplace=False)
    (6): Linear(in_features=10, out_features=3, bias=True)
  )
  (quant): QuantStub()
  (dequant): DeQuantStub()
)
```

此神經網路完美的符合我們的需求，根據我們前面所說，上表所採取的量化設定為 fbgemm，不過依然有調整的空間，為了進一步減少參數檔案大小，我們採用自定義的方式進行量化，將所有卷基層的權重與輸入輸出以同樣 scale 和 zero_point 量化，與 fbgemm 量化方式相比，減少了約 20%的檔案大小(26→21kb)，正確率則幾乎不變。

$$Quantized\_value = round\left(\frac{Dequantized\_value}{scale}\right) + zero\_point$$

$$Dequantized\_value = (\ Quantized\_value - zero\_point\ ) * scale$$

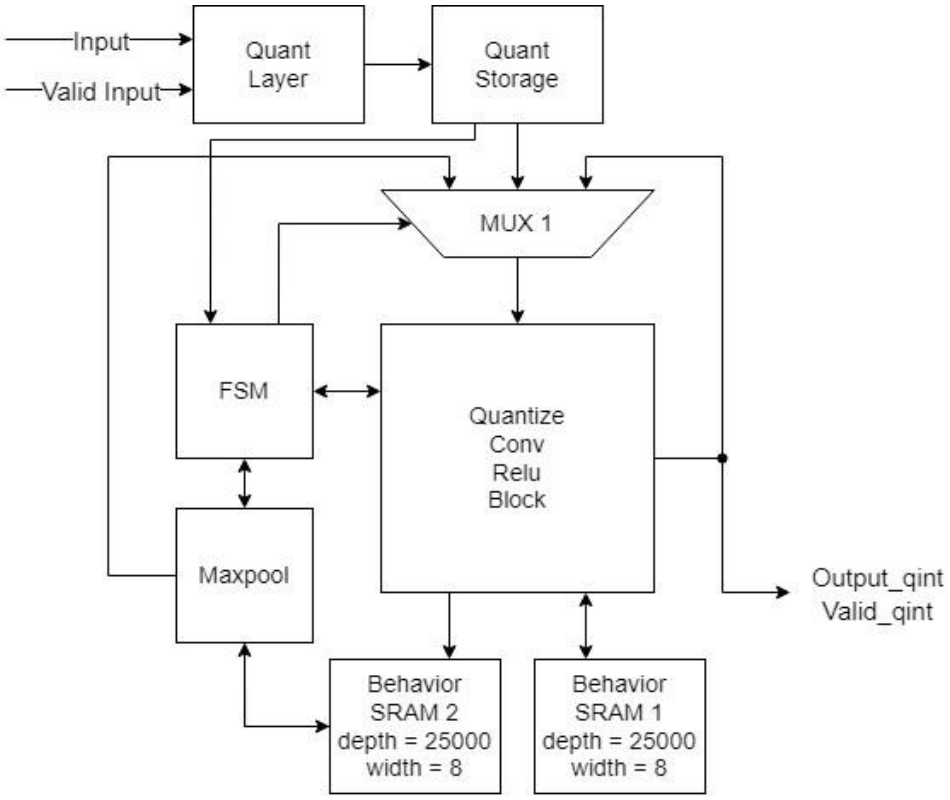
以上是量化的公式，*Dequantized_value* 是原浮點數值，*Quantized_value* 是經過量化的數值。

3.3. 硬體模型設計與結果

我們用硬體描述語言 (Verilog) 實作出相關的數位電路，共有三個版本的硬體設計。第一個版本(CNN_version1)是我們最早完成的設計，此設計的 top module 輸入和輸出並沒有考慮到是否適合放在 PYNQ-Z2 上測試，不過這個版本是最適合在 vivado 上面做 behavior simulation 和 post-synthesis simulation，此版本的數位電路較適合加入其他更大的系統內部。第二個版本(CNN_version2)則是我們考量到 PYNQ-Z2 的 PS 端和 PL 端之間的溝通，所以在第一個版本的基礎下，額外添加了一些邏輯，讓 top

module 的輸入和輸出更適合放在 PYNQ-Z2 上方測試。第三個版本(CNN_version3)則是完整的設計，我們將 CNN_version2 封裝成符合 AXI4 通訊協定的 IP，然後透過 block diagram 完整定義 PS 端和 PL 端的連接。最終，將第三個版本(CNN_version3)的設計放上 PYNQ-Z2 完成測試。

3.3.1. CNN_verion1



圖二: CNN_version1 的架構圖

3.3.1.1. Quant Layer

此 module 實現 Pytorch 模型中的 Quantize 功能，就是計算 *Quantized_value* = *round* ($\frac{Dequantized\_value}{scale}$) + *zero_point*，為了降低延遲，我們選擇稍微犧牲精準度，將 $\frac{Dequantized\_value}{scale}$ 改成 $\frac{1}{scale} * Dequantized\_value$ ，由浮點數除法變成浮點數乘法。這個 module 需要 1 個 cycle 才能輸出量化後的數值並接到 Quant storage。

3.3.1.2. Quant storage

此 module 接收來自 Quant Layer 的輸出，包含一個量化值和一個 valid 訊號。當 Quant storage 偵測到量化值是合法時，就會將這個量化值和一個 valid 訊號送到 Mux1。Quant storage 還有另一個輸出，能將合法的量化值按照時間順序儲存到 behavior SRAM 中(從地址低位開始存)，不過在 CNN_version1 中的 Quant storage 沒有

用到這個輸出。此外，Quant storage 還有一個功能，就是統計當前已經處理的合法量化值數量。在 CNN_version1 的架構圖中(圖二)， Quant storage 的輸出連接到 FSM，當 Quant storage 內部的 counter 計算到圖片的所有的 pixel 都已經處理完畢後(在這次的專題中，為 64*64*3 個合法的量化值)，則會傳輸一個訊號告訴 FSM 這件事情。

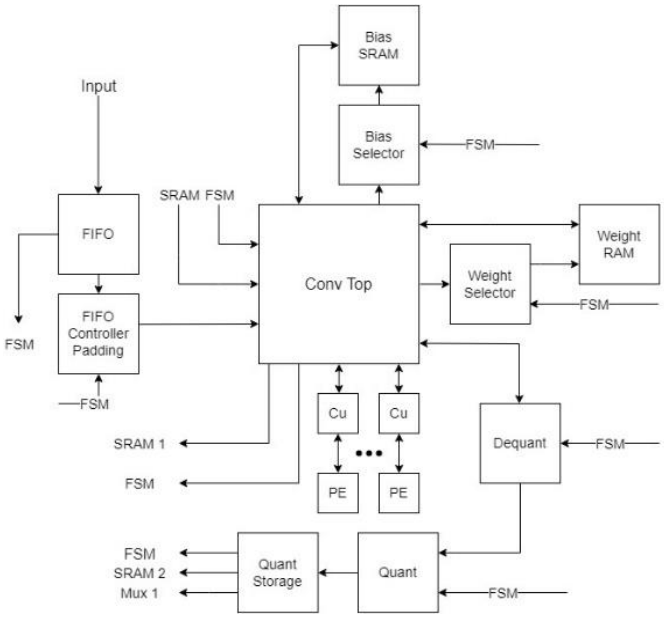
3.3.1.3. Mux1

這個 module 主要的功能是選擇進入到 QuantizedConvReLU block 的訊號，分別有從 Quant storage、QuantizeConvReLU block 以及 Maxpool 來的，然後 select signal 則是由 FSM 控制。

由於 Linaer layer 跟 Convolution layer 本質上是相同的(H = W = F = 1，input channel = in_features，output_channel = numbers of filters = out_features)，只是 Convolution 的一個特例，所以我們選擇在架構中，將軟體中的 QuantizeLinearReLU(...) 對應到硬體的 QuantizeConvReLU block 即可計算，這也就是 Mux1 三個訊號的來源。至於 FSM 的 selected signal，則是告訴 Mux1 現在神經網路進行到哪一層了，藉此讓 Mux1 正確選擇當前 QuantizedConvReLU block 的輸入訊號。

3.3.1.4. QuantizedConvReLU Block

首先，進入 module 的輸入是由 Mux1 控制的，當 write_enable ==1 時(由 Mux1 控制)，輸入會被儲存進同步 FIFO，FIFO 內部則是透過 wrtie_addr 和 read_addr 來控制讀寫的位置，不過與 block ram 不同，FIFO 的 write_addr 和 read_addr 不會由外界控制，因為 FIFO 要維持 First in, First out 的特性。我們使用 FIFO 最主要的原因是 CNN_version1 為 serial input，所以整張圖片共 64*64*3 個 input pixel 並不會同時進入設計內，為了維持合法量化值的先後順序。



圖三: QuantizedConvReLU block 的架構

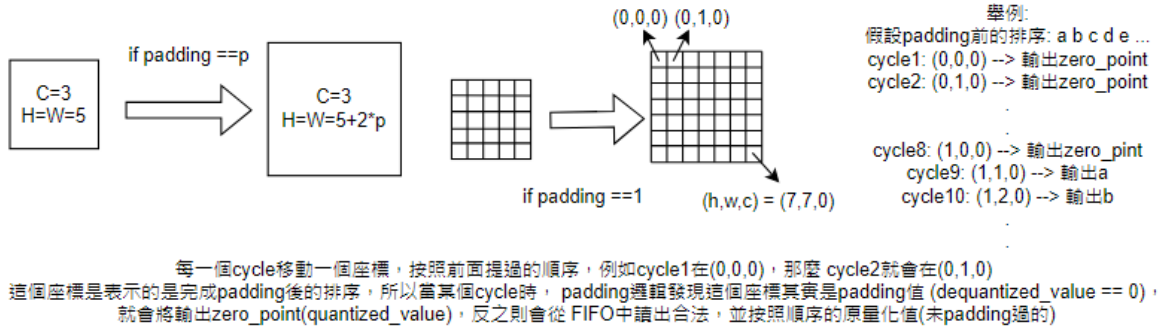
至於 input pixel (h,w,c)的順序，其中 h 代表的是「列」，w 代表的是「行」，c 代表的是「深度」。輸入 pixel 的順序是按照以下原則:「c 越小越先送入，假如 c 相同則 h 越小越先送入，假如 h 相同則 w 越小越先送入。」當量化完的 pixel 全部都存入 FIFO 時，此時從 FSM 來的訊號就會控制 PADDING 邏輯從 FIFO 中讀出資料，並進行 PADDING(圖四)，做完 PADDING 之後，將結果存入 behavior SRAM1，而 ConvTop、Quant、Dequant module 就會開始做相對應的運算(圖五)，最後我們會將運算的結果存入 behavior SRAM2。全部運算都結束之後，QuantizedConvReLU block 會將輸出一個

訊號給 FSM，讓 FSM 知道運算已經結束。

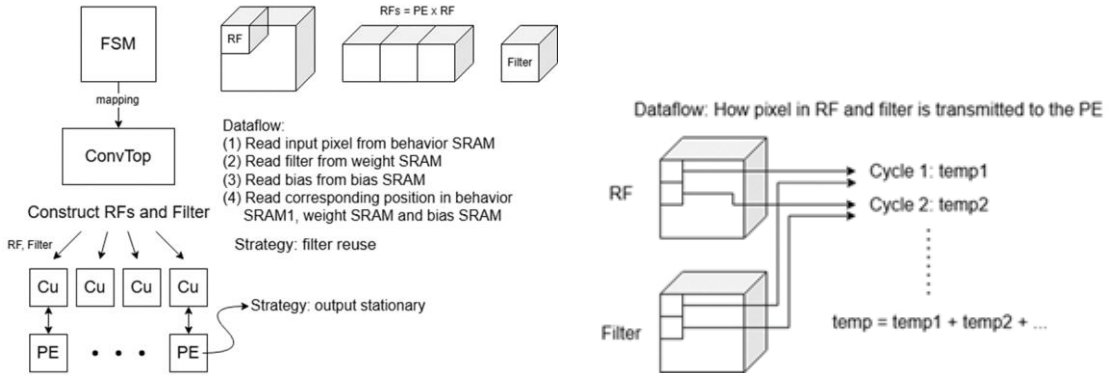
此外，我們能夠計算完成一層運算需要花費的 cycle 量(參考圖十二)，由於資料存放在 behavior SRAM1 中(實際上為 block ram，位寬為 8 bit)，代表我們一次只能從記憶體中讀取一個 pixel，那麼組建 RFS = RF*PE 需要花 PE*F*F*input_channel 個 cycle，當 RFs 讀取完成後，每個 Cu (Covunit)會分到一個 RF，由 Cu 負責將 RF 中和 Filter 中的 pixel 傳給 PE 進行累加，這樣的運算需要 F*F*input_channel cycle(就是一個 RF 有多少個 pixel，因為每次對一個 pixel 運算需要花一個 cycle)。最後，我們總共會組建 H_pad * W_pad * Numbers_of_Filters * (1/PE) 個 RFS，可得某一卷積或線性層完成運算所需的 cycle 數為下列公式:

*(PE*F*F*input_channel+F*F*input_channel)*H_pad*W_pad*Numbers_of_Filters*(1/PE)*

以第一層卷積層來說，當我們的 PE = 1 時，需花約 140 萬 cycle 完成運算，PE = 8 時，則需花約 80 萬 cycle 才能完成運算。



圖四: padding 邏輯原理



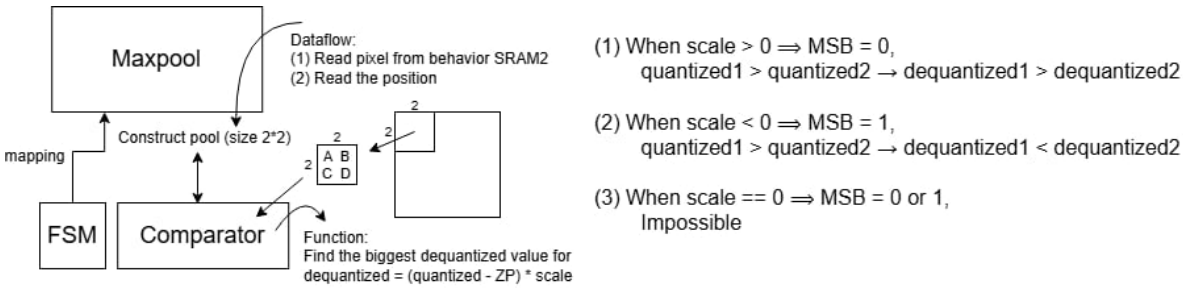
圖五: ConvTop 的作用和使用的策略(Filter reuse / output stationary)

我們定義一個 parameter PE，這個參數就是代表我們的硬體會有幾個 Processing Unit，PE 參數不同 dataflow 也會不同，這麼做的目的是為了讓使用者可以自行根據 FPGA 有的資源調整硬體配置。

我們所設計的 QuantizedConvReLU block 是 flexible，可以透過 FSM 決定不同時間點 QuantizedConvReLU block 的 H,W,C (input channel), K(output channel)的數值為何，並且我們的 dataflow 會隨著 FSM 給的數值不同而改變，藉此達到相同的硬體，重複利用的效果，我們的靈感來源自論文 efficient processing of deep neural networks - a tutorial and survey。

3.3.1.5. Maxpool Block

Maxpool Module 的功能是取一塊區域(在這裡是 2*2 個 pixel)中的最大值，得到最大值後就將此數值傳到 Mux1，當所有操作都做完之後，會輸出訊號到 FSM，讓 FSM 能夠做對應的控制。跟 QuantizedConvReLU 一樣，我們的 Maxpool block 同樣是 flexible 的，可以藉由 FSM 調整參數來產生出對應的 dataflow(圖六)。



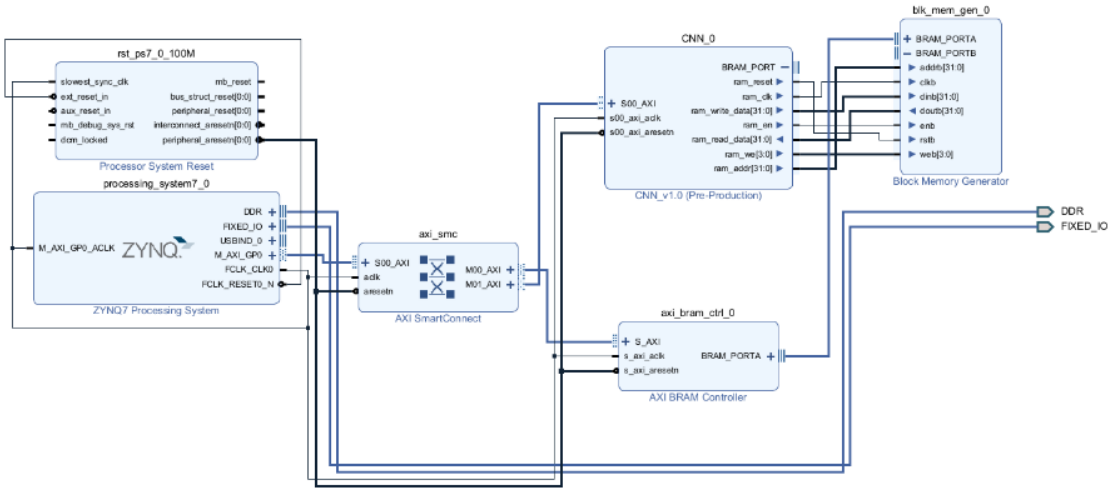
圖六: Maxpool 的比較方式和 dataflow 形式

3.3.2. CNN_version2

之所以建立 CNN_version2 而不繼續使用 CNN_version1，其實是因為我們原先的輸入輸出波型太過複雜，所以導致我們很難精準地去輸入符合要求的訊號，因此，我們改變了方法，PS 端和 PL 端之間除了 start 訊號是由 PS 端直接透過 AXI4 控制 PL 端之外，其餘的輸入輸出都會先進入到 bram ip，然後我們設計的數位電路在透過 Mem IP controller 這個 module 從 bram ip 中讀取或存入對應的數值，這樣就可以避免 PS 端需要對 PL 端進行精確的時序控制，更詳細的內容可以參考 CNN_version3 的說明。

3.3.3. CNN_version3

我們將 CNN_version2 封裝成 AXI4 IP (CNN_0)，並且讓他去控制一個 true dual port block ram，然後 PS 端則透過 bram controller 去控制。當 PS 端已經完整的把所有圖片的像素都存入 true dual port block ram 時，PS 端就會讓 start 從 0 變 1，隨後這個變數就會通過 AXI4 的協定傳遞到 AXI4 IP (CNN_0)，CNN_0 就會開始做對應的操作。此外，我們透過 AXI SmartConnection ip 區分 PS 端要給 bram controller 或 CNN_0 的 start 訊號。



圖七: 完整的 block diagram，包含 PS 端和 PL 端該如何溝通

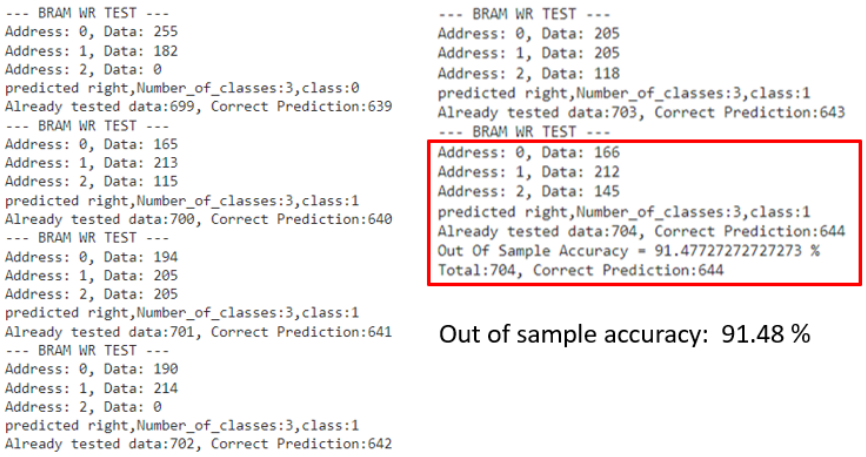
3.3.4. Design On FPGA testing result

我們將設計放到 FPGA 版上進行驗證，由於我們使用的是 PYNQ 系列的 FPGA，因此我們藉由 python 控制 PS 端向 PL 端輸送資料，並且在 PL 端運算完後再從 blk_mem_gen_0 的這個 true dual port block ram 這個 ip 中取出對應的資料，一共有 3 個 class，對應馬鈴薯葉片的三種狀態，分別是 Healthy (健康)、Early blight (感染茄鏈隔孢菌)、Late blight (感染致病疫霉)。

```
--- BRAM WR TEST ---
Address: 0, Data: 202
Address: 1, Data: 208
Address: 2, Data: 102
predicted right,Number_of_classes:3,class:1
Already tested data:1, Correct Prediction:1
--- BRAM WR TEST ---
Address: 0, Data: 205
Address: 1, Data: 205
Address: 2, Data: 117
predicted right,Number_of_classes:3,class:0
Already tested data:2, Correct Prediction:2
--- BRAM WR TEST ---
Address: 0, Data: 255
Address: 1, Data: 184
Address: 2, Data: 0
predicted right,Number_of_classes:3,class:0
Already tested data:3, Correct Prediction:3
--- BRAM WR TEST ---
Address: 0, Data: 118
Address: 1, Data: 217
Address: 2, Data: 229
predicted right,Number_of_classes:3,class:2
Already tested data:4, Correct Prediction:4
```

```
--- BRAM WR TEST ---
Address: 0, Data: 205
Address: 1, Data: 205
Address: 2, Data: 132
wrong prediction
Already tested data:9, Correct Prediction:8
--- BRAM WR TEST ---
Address: 0, Data: 174
Address: 1, Data: 214
Address: 2, Data: 76
predicted right,Number_of_classes:3,class:1
Already tested data:10, Correct Prediction:9
--- BRAM WR TEST ---
Address: 0, Data: 153
Address: 1, Data: 212
Address: 2, Data: 205
predicted right,Number_of_classes:3,class:1
Already tested data:11, Correct Prediction:10
--- BRAM WR TEST ---
Address: 0, Data: 165
Address: 1, Data: 213
Address: 2, Data: 126
predicted right,Number_of_classes:3,class:1
Already tested data:12, Correct Prediction:11
```

圖八: 藍色框框是硬體模型預測正確時的結果，紅色框框則是預測錯誤



圖九: 共有 704 筆 out of sample data (沒有參與訓練的 data)，
最終我們的神經網路電路模型從中獲得 91.48%的正確率

二、 參考資料

1. https://github.com/V0XNIHILI/parametrizable-floating-point-verilog
2. Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, Joel Emer (2017). Efficient Processing of Deep Neural Networks: A Tutorial and Survey.
3. Ahmed Abdulkader, Daniel Eskandar, Omar Essam, Omar Tarek, Ahmed Ezzat (2021). Implementation of a Convolutional Neural Network on an FPGA using Verilog

三、 心得感想與反思

在尚未進行專題研究前，我們原先對於機器學習毫無概念，但在專題中慢慢藉著自主學習網路開放式課程和閱讀相關論文，建構神經網路和深度學習加速的概念，同時藉由和實驗室學長姊討論，構思主題並深入研究。在研究過程中，從神經網路模型至電路設計不斷遭遇問題，如神經網路該如何量化並量化至何種型態、如何在有限制的 FPGA 資源裡建構可重複使用的卷積層池化層、如何存取 FPGA 的記憶體並設計適當 dataflow、PS 和 PL 端的傳輸等等，我們靠著上網查找文獻資料並反覆嘗試將困難一一排除。

針對硬體設計方面，我們在 FPGA 的 LUT 資源使用率極高(90%)，這也導致我們很難在針對目前的設計進行 pipeline 或是更進階的優化，例如透過更精準的控制 FSM，使 QuantizedConvReLU block 和 Maxpool block 可以同時進行運算等方式，因此假如未來想要更加優化數位電路，或是添加其他模組，例如攝影機控制模組之類的，或許就要選擇 LUT 資源更充沛的 FPGA 板或是研究更有效率的硬體架構。

通過專題研究，我們不僅對於機器學習和數位電路設計有了更深入的了解與研究，更寶貴的是培養了自行蒐集資料、獨立思考、解決問題的能力，能夠善用各項資

源，找出根本原因，並提出有效的解決方案，建立研究的基礎，我們深信專題的成果對於未來能有所幫助。非常感謝鄭桂忠教授給我們這個機會，也非常感謝學長姐適時的引導，提供實驗室的資源和設備，耐心地指導我們。