DEGREE PROJECT IN ,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2019

Algorithm Design and Optimization of Convolutional Neural Networks Implemented on FPGAs

ZEKUN DU

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

Algorithm Design and Optimization of Convolutional Neural Networks Implemented on FPGAs

ZEKUN DU

Master in Embedded Systems
Date: June 16, 2019
Supervisor: Anne Håkansson
Examiner: Mihhail Matskin
School of Electrical Engineering and Computer Science
Host company: Synective Labs AB

Abstract

Deep learning develops rapidly in recent years. It has been applied to many fields, which are the main areas of artificial intelligence. The combination of deep learning and embedded systems is a good direction in the technical field. This project is going to design a deep learning neural network algorithm that can be implemented on hardware, for example, FPGA. This project based on current researches about deep learning neural network and hardware features.

The system uses PyTorch and CUDA as assistant methods. This project focuses on image classification based on a convolutional neural network (CNN). Many good CNN models can be studied, like ResNet, ResNeXt, and MobileNet. By applying these models to the design, an algorithm is decided with the model of MobileNet. Models are selected in some ways, like floating point operations (FLOPs), number of parameters and classification accuracy. Finally, the algorithm based on MobileNet is selected with a top-1 error of 5.5% on software with a 6-class data set.

Furthermore, the hardware simulation comes on the MobileNet based algorithm. The parameters are transformed from floating point numbers to 8-bit integers. The output numbers of each individual layer are cut to fixed-bit integers to fit the hardware restriction. A number handling method is designed to simulate the number change on hardware. Based on this simulation method, the top-1 error increases to 12.3%, which is acceptable.

Keywords

Image Classification; Convolutional Neural Network; Top-1 Error; FPGA; Hardware Simulation

Sammanfattning

Deep learning har utvecklats snabbt under den senaste tiden. Det har funnit applikationer inom många områden, som är huvudfälten inom Artificial Intelligence. Kombinationen av Deep Learning och innbyggda system är en god inriktning i det tekniska fältet. Syftet med detta projekt är att designa en Deep Learning-baserad Neural Network algoritm som kan implementeras på hårdvara, till exempel en FPGA. Projektet är baserat på modern forskning inom Deep Learning Neural Networks samt hårdvaruegenskaper.

Systemet är baserat på PyTorch och CUDA. Projektets fokus är bild klassificering baserat på Convolutional Neural Networks (CNN). Det finns många bra CNN modeller att studera, t.ex. ResNet, ResNeXt och MobileNet. Genom att applicera dessa modeller till designen valdes en algoritm med MobileNet-modellen. Valet av modell är baserat på faktorer så som antal flyttalsoperationer, antal modellparametrar och klassifikationsprecision. Den mjukvarubaserade versionen av den MobileNet-baserade algoritmen har top-1 error på 5.5

En hårdvarusimulering av MobileNet nätverket designades, i vilket parametrarna är konverterade från flyttal till 8-bit heltal. Talen från varje lager klipps till fixed-bit heltal för att anpassa nätverket till befintliga hårdvarubegränsningar. En metod designas för att simulera talförändringen på hårdvaran. Baserat på denna simuleringsmetod reduceras top-1 error till 12.3

Nyckelord

Bild Klassificering; Konvolutionellt neuralt nätverk; Top-1 Error; FPGA; Maskinvaru Simulering

Acknowledgements

This project is carried out at Synective Labs AB in Stockholm.

I would like to express my great appreciation to professor Mihhail Matskin for his patience and guidance.

I would like to acknowledge the solid support from Mr. Jiliang Du and Mrs. Xianjun Zheng.

I would like to thank Viktor Wase for technical support and friendship. I would like to thank Gunnar Stjernberg and Niklas Ljung for their continuous help.

I would like to give my special thanks to Guanghao Guo, Dingzhi Chen and Love Olsson for helping me out of anxiety and depression.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

This master thesis degree project is about algorithm design of a convolutional neural network (CNN) implemented on field-programmable gate array (FPGAs). Basic concepts and information of the project are introduced. This part includes sections of background, problems, purposes, goals, methods, delimitations and outlines of this project.

1.1 Background

Machine learning is a part of artificial intelligence, using deduction and patterns other than specific commands to solve problems according to [1]. Deep learning, part of machine learning, has become a mature academic field since 2006 according to [2, 3]. Deep learning is always applied to fields like image classification and audio recognition with raw data. It includes two main parts which are supervised learning and unsupervised learning. Supervised learning learns through pre-labeled training inputs to distribute testing inputs into different classes, while unsupervised learning has no labels in the training inputs according to [4].

Image recognition is a specific part of computer vision which gains high-level information from images according to [5]. A small example of image recognition is to tell is there a cat or a dog inside the input picture. Computer vision has been applied to many fields, like traffic signs recognition and optical character recognition. Most computer vision tasks are handled on graphics processing units (GPUs) on which not only images but also pixels can be handled in parallel. Image recognition on GPU gives a good performance, however, it restricts tasks to computers or workstations.

The embedded system, comparing to stationary computers, has lower power consumption, smaller size, and lower unit cost according to [**apippi**]. It is used in multiple situations like robots and smartphones. With the increasing needs of computer vision technology to be applied to FPGA, combining the advantages of both computer vision and the embedded system is a great way to solve more problems in different fields.

FPGA is a good embedded system platform. Comparing to microprocessors, FPGAs have good flexibility, are able to be rewritten multiple times, and are quick to study according to [6]. It is an integrated circuit designed to be configured by a customer or a designer after manufacturing. FPGA does not have a fixed hardware structure, so it is programmable according to user applications.

This project is going to combine CNN and FPGA to produce an embedded CNN system with image recognition functionality. It is supposed to recognize the target information on the picture in a reasonable period.

1.2 Problem

Figure 1.1 shows a general picture of the image classification embedded system. Image recognition is the topic of this project. The camera is for capturing the image, and transfer captured images to the FPGA board. The FPGA board is the hardware platform which is the main part of the system. The CPU processor of FPGA board pre-handles images with the given size and data type, and then sends data to the programmable logic where the CNN is implemented and run. After CNN on board, the output of FPGA is several digits representing the image category which is previously distributed. The recognition result is shown on the screen.

This project focuses on the CNN algorithm inside the FPGA board. A typical CNN algorithm includes multiple layers of convolution, ReLU, pool and fully-connected (FC). More kinds of layers could be added along with the needs. Layers that are shown in Figure 1.1 can be repeated multiple times to get good performance.

In this project, the first thing is to decide the data set which is used for training and testing. The complexity of the data can determine the success rate of the project design. Since the hardware is an FPGA board, the performance is restricted and the bottleneck is obvious. It matters not much about what exact data set is chosen for training and testing. if the complexity is appropriate, the system is flexible to suit other data sets with similar complexity. When the network is trained to well suit the chosen data set, the structure of the network algorithm is proved suitable for the system.

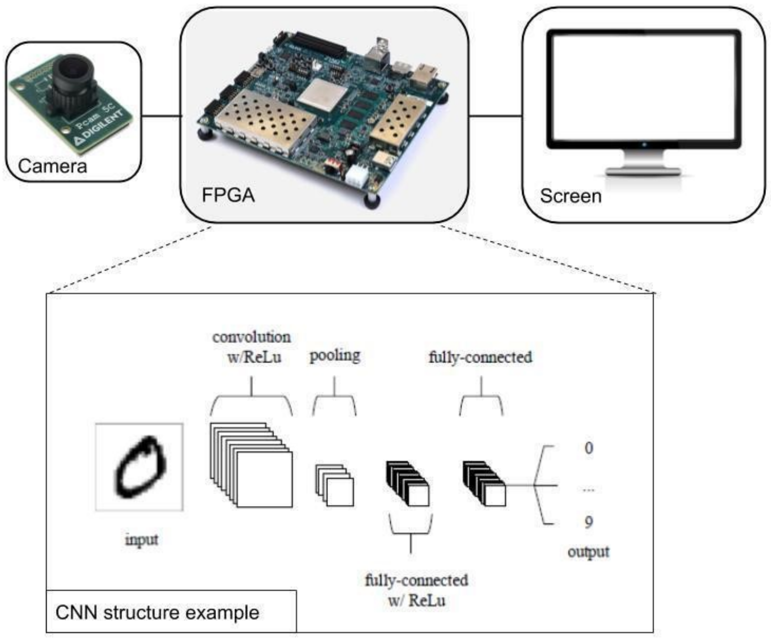

Figure 1.1: Structure of the system (pictures taken from [4, 7, 8, 9]).

The FPGA board has limited space for calculation and storage. Nowadays multiple CNN models have been designed to fix embedded systems. There are multiple models that can be chosen, like MobileNet and ResNet. Those models for embedded systems cost less spatial resources, which is a memory in this case, on the board according to [10]. Not only spatial resource is considered, but time is also a big issue. This embedded system is a real-time system, the result should be output within an acceptable period. Spatial resource and time resource is a trade-off. Although the running time from a picture is capture to the recognition result is shown mostly depends on the hardware implementation, the network design in this project should not include too many layers.

Regarding the description above before the project design starts, several related questions are released to help comprehend the problem better.

Questions: What aspect can be improved from a classical neural network to fit embedded systems? What model is chosen to implement the network? What is the maximum size of the network that would work on an FPGA? What accuracy is expected on the designed network? How to simulate the hardware performance before the network design is released? What optimization can be done to improve the network?

1.3 Application challenge

There are several challenges that may influence project design and performance.

1.3.1 Data Set Complexity

Data is a major part of the design process of the CNN algorithm. The features of data set which can influence the algorithm design are listed in Table 1.1.

Table 1.1: Features of data that may influence the project result.

Feature of data	**Description**
Data size	Data size influences the performance of the CNN whether it results in good accuracy. A common way to use data is to have around 1000 images per category [11].
Data complexity	Data complexity influences the network complexity in a positive correlation. The more complex images are, the more neural nodes this network needs to contain.
Data category	If contents from different categories are similar, the recognition result might be not correct. Having more categories leads to a higher level of difficulty.

1.3.2 Model Selection

Totally new CNN models are not possible to be designed in a short time. It is a great need to study previous work on CNN model designs. Different models have their own pros and cons. Different models are designed to suit different situations. In recent years, multiple models for CNN are designed, like AlexNet, GoogleNet, VGG-16 and so on according to [12]. There are some features shown in Table 1.2 by which models can be evaluated.

1.3.3 Hardware Memory Limit

The FPGA board on use is Xilinx Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit with a 38MB memory according to [7]. This memory size is small to implement a CNN model on it. Parameters and most of the workload is stored in the memory. It requires the designed CNN model to be small enough to suit the hardware memory restrict.

Table 1.2: Main features of models that influences the CNN performance.

Feature of models	**Description**
Top-1 error	It describes the error rate when the network gives the classification result of an image only once but it is not the correct category. Another usually used feature is the top-5 error. It represents when CNN gives the classification result of an image five times and not any answer lies in the same category.
Total parameters	It means how much space will be taken by all parameters of the CNN model. Every node of the neural network contains parameters for calculations, so the total memory for storing parameters is not neglectable.
Total workload	The unit is multiply accumulate operation (MAC) [12]. It represents the calculation size of the whole network. The workload mainly comes from the calculation buffers.

1.3.4 Calculation Precision Limit

The typical CNN is trained and tested on computers or GPUs. Both computers and GPUs have a large size of memory for data storage. It supports CNN to handle data with a high precision, which to be specific is signed floating-point numbers. However, on FPGA integers or fixed-point numbers is usually used. It causes two main problems: precision lost and overflow risk.

1.4 Purpose

Nowadays, the technology about deep learning develops fast, and it is already applied to many industry fields. Embedded system, like FPGA board, is a flexible platform for some specific work, and it has a tiny physical size comparing to traditional deep learning platform like a work station. The combination of deep learning and the embedded system is a great aspect to develop. Deep learning mostly focuses on software design because high-quality hardware and framework are not allowed to change. However, according to my acknowledgment, there is no complete method to apply the high-performance deep learning network to the target embedded system. This project is aimed at generating its own suitable method.

1.5 Goals

a) Produce a CNN model with the accuracy larger than 90% (top-1 error smaller than 10%) based on the target data set.
b) Compare different models by evaluations and to select one that is good for hardware to implement. The evaluation basically includes top-1 error, spatial resources cost and model complexity.
c) Design a method to transform the parameters of the selected model to the format (fix-point integers).
d) Generate a solution to simulate the hardware performance based on the particular FPGA board features and give a demonstration as reference. The accuracy of the simulation is supposed to be larger than 85%.

1.6 Research Methodology

There are mainly quantitative and qualitative methods in the research methodology.

Quantitative methods deal with data that is represented by numbers. It is direct to compare data which is well structured. Most parts of this project can be evaluated with numeral statistics. If some parts cannot be evaluated with numeral statistics or they are not comparable for some reasons, qualitative methods are good because they are theory generation and process based analysis.

Deduction is also used in this project. It help reject or accept the hypothesis that is raised based on early-stage theory study.

1.7 Delimitation

In this project, the data set is assumed varied enough. Diversity determines the limit of the trained network. In chapter 4, simple data is combined with a random background to increase diversity.

There are multiple CNN models in recent researches. This project only picks a few of them that are new models. We can not cover all models, only some famous models.

This project is not looking at some technology which is really new and specifically for FPGAs like binary-net.

1.8 Sustainability Questions and Ethical Problems

There are several sustainability questions. Is this project useful when the data set is changed? Yes, as long as the new data set has the similar complexity to the current data set, the CNN model could be trained to fit the new data set with the same structure. Can this project provide references for later project? Yes, the process of transforming the CNN model parameters and constraining the calculation results can provide a reference for other projects in the future that needs the process from software to hardware. It shows some significant points to pay attention which are also good references.

As for ethical problems, this project does no harm to neither the human society nor the natural environment. This project gives credits to all resources that are used as references. The data set is picked in an open source database, and the images included will not affect anyone's privacy. This project does not copy achievements from any other researchers or groups. All work without references that is mentioned later are done by the report author.

Chapter 2

Theory and Related Works

This chapter is going to talk about the theory and related works from the previous researches that can help this project.

2.1 Convolutional Neural Network (CNN) Structure and Theory

Deep learning uses machine learning skills to generate multiple layers of non-linear operation structure according to [2].

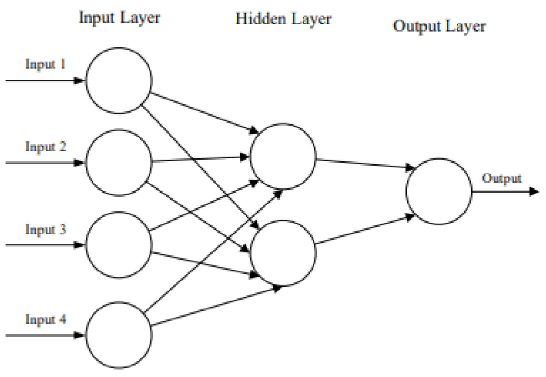

Figure 2.1: Artificial neural network model example taken from [4]

Deep learning is inspired by the structure and functionality of biological brains [13]. Its model imitates the feature of neurons and their communications. Artificial neural network (ANN) is a typical implementation method of deep learning. An ANN model includes an input layer, a hidden layer, and an output layer, which is shown in Figure 2.1. In the neural networks field, the most useful models are recurrent neural networks (RNN) and convolutional neural networks (CNN). RNN is good at natural language processing and language understanding according to [14]. CNN is good at computer vision like image recognition according to [15].

CNN has two main phases: training and inference. The training phase uses the backpropagation algorithm to update the model parameters and to improve the prediction accuracy, based on target data sets. The training process can be repeated until the prediction accuracy is acceptable. The detail of backpropagation will be discussed later in this section. The inference uses the model which is trained from the previous phase to classify a new data set. If the classification accuracy is acceptable.

Before the introduction of concrete contents of the CNN, parameters used for it need to be introduced. In the real use of CNN training and testing, data is handled with tensors which are a mathematical object of multiple dimensions with certain transformation rules according to [16]. Tensors that are transferred between layers and Table 2.1 lists parameters that are used in CNN and they are going to be talked in the following paragraphs. Batch means how many pictures are combined together as one input feature maps.

Table 2.1: Tensors in the CNN inference with parameters involved taken from [12]

Symbol	**Description**	**Contents**
B	Batch size (Number of input frames)	
W / H / C	Width / Height / Depth of Input FMs	
U / V / N	Width / Height / Depth of Output FMs	
K / J	Horizontal / Vertical Kernel size	
X	Input FMs	B×C×H×W
Y	Output FMs	B×N×V×U
Θ	Learned Filters	N×C×J×K
β	Learned biases	N

CNN has a feedforward propagation structure over multiple kinds of layers. The layers that we use in this project and discussed in this section are the convolutional layer, activation layer, pooling layer, fully connected layer, and batch normalization layer. Figure 2.2 shows a simple example of forwarding propagation including convolution, activation and pooling layer.

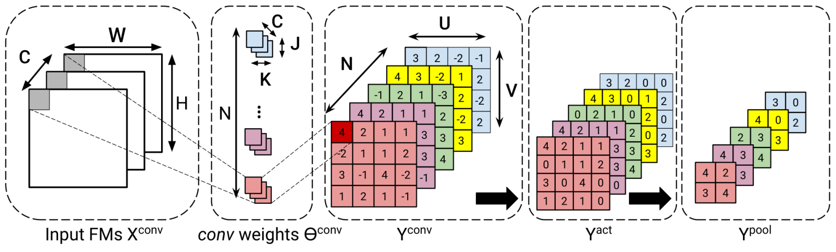

Figure 2.2: Feed forward propagation example in inference taken from [12]

2.1.1 Convolution layer

A convolution layer is a basic unit of CNN models. It is the major feature extracting part of the whole model according to [12]. It includes three major contents: input FM, output FM, and kernels. The input is a 3-dimensional image (or feature map) data, with sizes of width, height, and channel ($U \times V \times N$, a channel is also named depth). The output is a new feature map with sizes of width, height, and depth ($U \times V \times N$). An output FM is calculated by a dot multiplication between input FM and kernels in width direction and height direction. Kernels have the same depth C as the input FM. The horizontal and vertical kernel sizes (K and J) are always the same. $K = J = 3$ or $K = J = 5$ are commonly chosen. A kernel is also called a filter. One convolution layer always has multiple kernels. The number of kernels consists of the depth of the output FM of the same layer according to [12]. Kernels are also named convolution weights. The calculation rule can be represented as the following mathematical formulas according to [12].

$$\begin{aligned}
&\forall\{b,n,u,v\} \in [1,B] \times [1,N] \times [1,U] \times [1,V] \\
&Y^{conv}[b,n,u,v] = \\
&\beta^{conv}[n] + \sum_{c=1}^{C}\sum_{j=1}^{J}\sum_{k=1}^{K} X^{conv}[b,c,v+j,u+k] \cdot \Theta^{conv}[n,c,j,k]
\end{aligned} \tag{2.1}$$

As shown in Figure 2.2, when batch size is 1, the contents from the input FM with the same spatial size as kernels perform the dot multiplication and then sum up to be the content of output FM's one pixel. The block with size $K \times J \times C$ moves on the input FM. The calculation results are stored in the corresponding position of the output FM. According to the number (N) of the kernels of this layer, the output FM's depth has the same value N.

Stride and padding are two important features of convolutional functions. Stride is initially 1, so is it in Figure 2.2. When the stride changes to 2 or more, the output FM size changes. As shown in Figure 2.3, the output pixel which is circled by colored block comes from the 3×3 blocks circled by the same color. This case neglects the values of batch and depth.

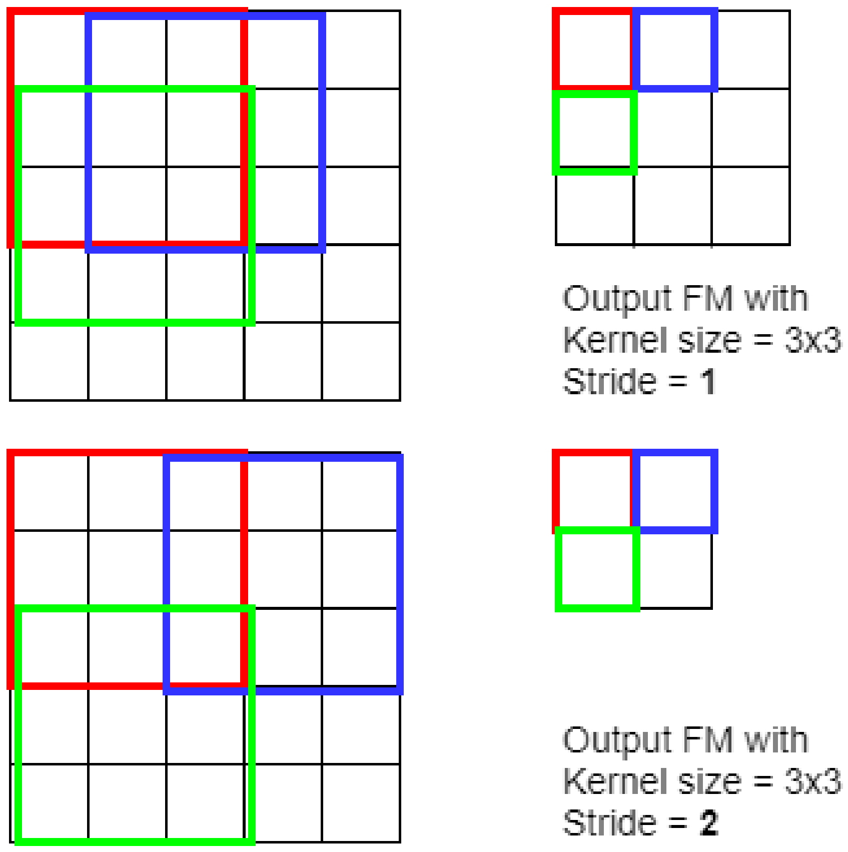

Figure 2.3: Output FMs with stride 2 and stride 3.

From Figure 2.3 we can see after a 5×5 input FM has convolution calculation with a 3×3 kernel, the width, and height both decrease by 2. This may cause the FM shrink and pixel loss according to [17]. If the CNN model contains a lot of convolution layers, the pixel loss is not neglectable. To prevent this situation, zero-padding is applied in the convolution layers. As shown in Figure 2.4, zero-padding is to extend the original input FM with zeros, and then the convolution calculation with the new FM results in an output FM with the same size. The padding width is determined by the kernel width K (kernel width and height are always the same). $W_{padding} = (K - 1)/2$.

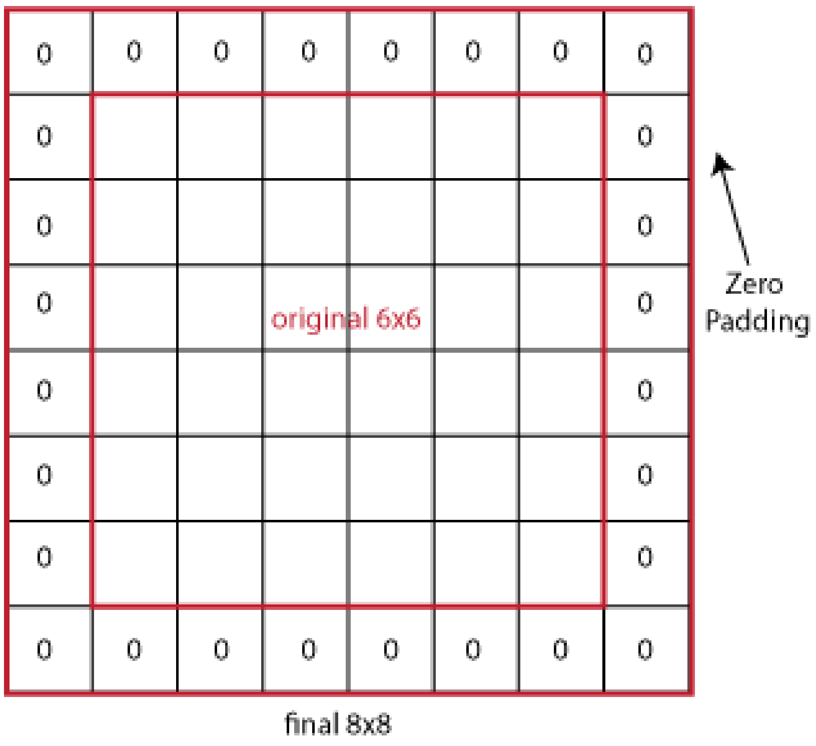

Figure 2.4: Zero padding with parameter 1 on a 6×6 image taken from [18].

2.1.2 Activation Layer

The activation layer is the layer in the CNN model which contains activation functions. It does not modify the size of input FMs but modifies the contents inside every pixel. In mathematics, activation functions are functions to map inputs to a target range, for example, between -1 and 1. The reason to use activation functions is that it is inspired by the biological action potential in neuron cells according to [19]. The mathematical formulas of this layer are shown below according to [12].

$$\begin{aligned}&\forall\{b,n,u,v\} \in [1,B]\times[1,N]\times[1,U]\times[1,V]\\&Y^{act}[b,n,h,w] = act(X^{act}[b,n,h,w]) \mid act := TanH, Sigmoid, ReLU...\end{aligned} \tag{2.2}$$

There are various kinds of activation functions. The commonly used kinds include binary step function, tanh function and rectified linear unit (ReLU) function. In Figure 2.5(a), the plot is from ReLU function with equation 2.3.

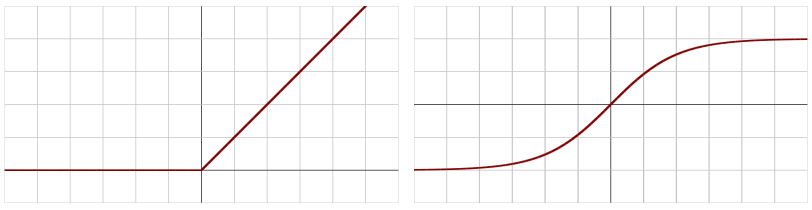
(a) ReLU (b) TanH

Figure 2.5: ReLU function and TanH function plots taken from [20].

$$f(x) = \begin{cases} 0 & for \quad x < 0 \\ x & for \quad x \geq 0 \end{cases} \tag{2.3}$$

The plot in Figure 2.5(b) is TanH functionis from equation 2.4.

$$f(x) = tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}} \tag{2.4}$$

2.1.3 Pooling Layer

The pooling layer is the last layer in Figure 2.2. Pooling layers are inserted between convolution layers, but not between every two convolution layers according to [12]. The role of pooling layers is to smaller the size in weight and height directions. Max-pooling is the only pooling function used in this project.

$$\begin{aligned} &\forall \{b, n, u, v\} \in [1, B] \times [1, N] \times [1, U] \times [1, V], \\ &Y^{pool}[b, n, v, u] = \ (X^{pool}[b, n, v + p, u + q]) \end{aligned} \tag{2.5}$$

Max pooling layers have a similar algorithm to convolution layers, which also contains filters with parameters of filter size and stride. However, the filter of pooling layers does not have attributes but the only size. As shown in Figure 2.6, the filter only picks the maximum number of numbers in the filter area and stores the maximum into the corresponding pixel of the output FM.

Figure 2.7 shows the max pooling operation is depth independent. The output FM has the same depth as the input's but with smaller size. This layer is to condense the features to reduce the following calculation cost.

2.1.4 Fully Connected (FC) Layer

Fully connected layers are the last part before the classification is done. They are linear functions, connecting every pixel of the input FM to the output. FC

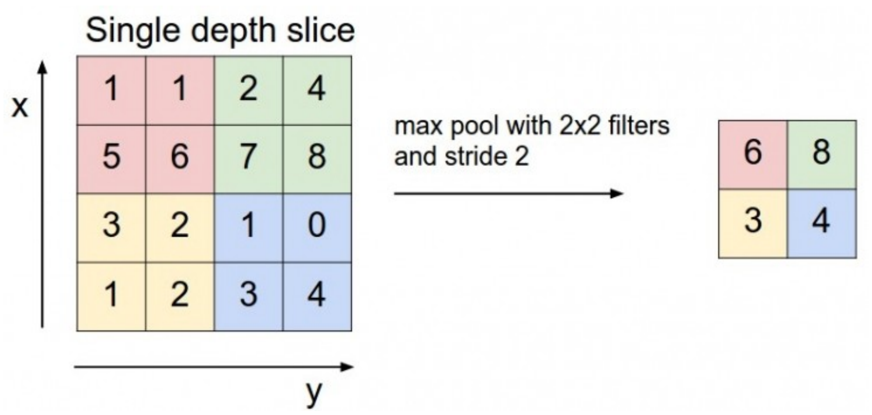

Figure 2.6: Max pooling example taken from [21].

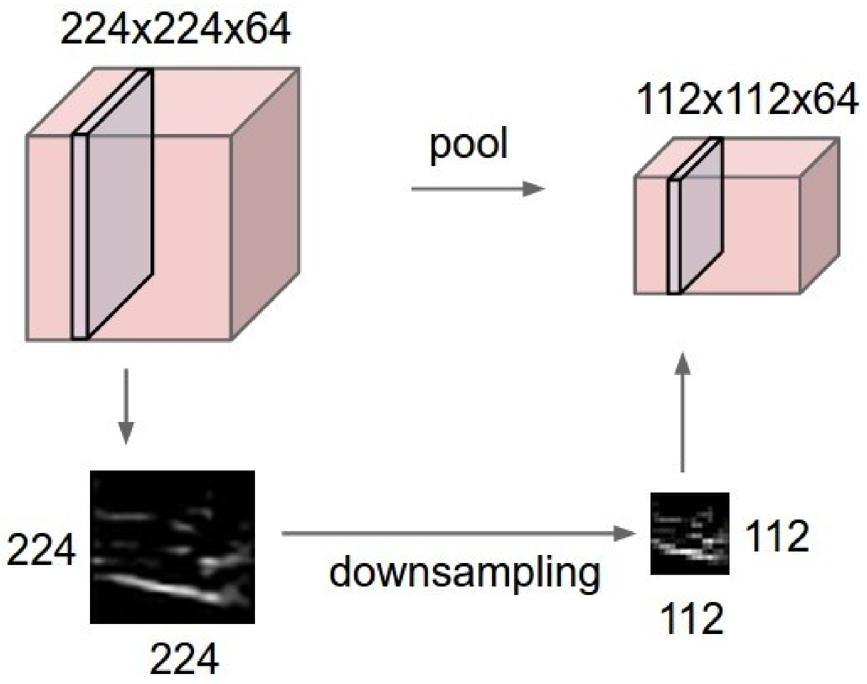

Figure 2.7: Depth independence of max pooling taken from [21].

layer can even be seen as a special convolution layer with $W = K$ and $H = J$ according to [12].

The FC layer's structure is shown in Figure 2.8. The input and output of this layer are all 1-dimension lists. Every node from an input (size N_i) and output (size N_o) is fully connected, which is how the layer name comes. Every FC layer contains a 2-dimension weight matrix with size of $N_i \times N_o$. Before every output, bias is added to the result. One CNN model can have multiple FC layers. The last FC layer result shows the possibility of which class the input image might be in. The CNN model will give the classification result with the highest possibility of the corresponding class.

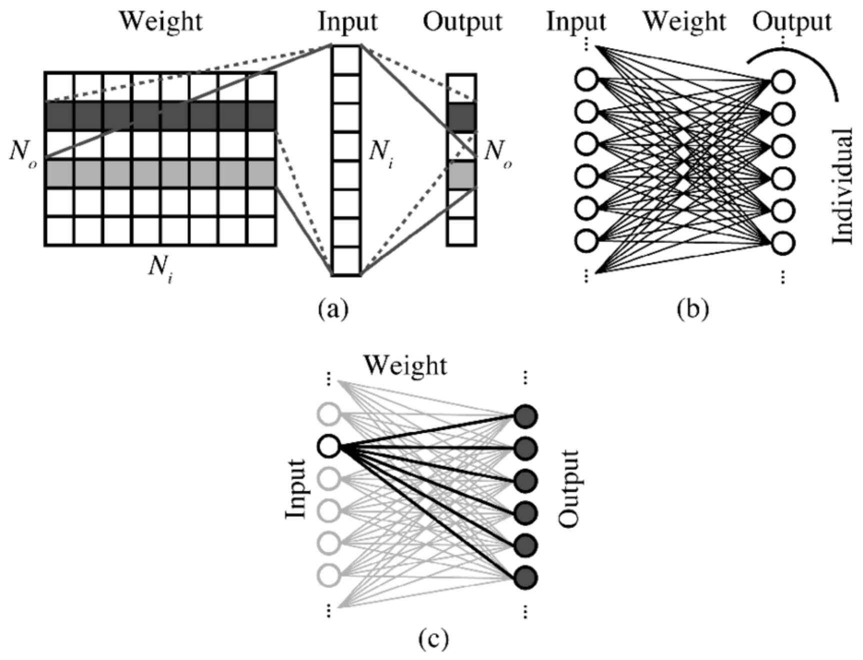

Figure 2.8: Fully connected layer structure taken from [22].

2.1.5 Back Propagation and Loss Function

Backpropagation is used during the training part of the CNN design lifetime. Inputting an image into the CNN and getting the classification result is the forward propagation process, which is also how inference works. The back-propagation is the opposite of forwarding propagation. It uses the chain rule algorithm to add the error influence back to weights of layers to improve the performance of the network according to [23]. The error represents how far the CNN model can perform classification perfectly. The backpropagation is performed layer by layer from the output port to the input port. It is the only method to modify the weights inside CNN models to get better performance.

The loss function is a component of pack propagation according to [24]. The function gives a difference by comparing CNN outputs and expected output. For example, `torch.nn.CrossEntropyLoss()` is selected for this project, whose mathematical detail is shown in equation 2.6. It is useful to train a network that classifies several classes according to [25].

$$\text{loss}(x, class) = -x[class] + \log\left(\sum_j e^{x[j]}\right) \tag{2.6}$$

2.1.6 Batch Normalization

Batch normalization is not mandatory of a CNN model. It could be a layer after a convolution layer. It can be represented as the following mathematical formula. μ represents the calculated mean of the input FM pixels. σ represents the calculated deviation of the input FM pixels. γ represents the expected deviation of output FM pixels. ϵ is a small constant to avoid the mathematical error when σ goes to zero. α represents the expected mean of output FM pixels.

$$\begin{aligned} &\forall\{b, n, u, v\} \in [1, B] \times [1, N] \times [1, U] \times [1, V], \\ &Y^{BN}[b, n, v, u] = \frac{X^{BN}[b, n, u, v] - \mu}{\sqrt{\sigma^2 + \epsilon}}\gamma + \alpha \end{aligned} \tag{2.7}$$

The batch normalization layer would speed up the training process because it allows the CNN model to be trained in a larger learning rate. Since batch normalization has a division in the formula above, it costs much more resources to be implemented on hardware. Considering batch normalizations' cost on hardware, each max pooling layer is placed before the nearest batch normalization layer before it.

2.2 CNN Models

As is acknowledged, the models of CNN take researchers much time to design and develop. In this project, we do not design new models but study different models and apply advantages from different models to our model. This project focuses on models that can be applied on the FPGA board, so the models that are known good for embedded systems are studied. The models that appeal interests are MobileNet, ResNet, ResNeXt, and ShuffNet. Different models have main differences in convolution layers. Convolution layers' cost comes from two parts: spatial connection and channel connection. As shown in Figure 2.9, if the convolution kernel size is 3×3, spatially every pixel connects to the other three pixels in width and height dimensions. Meanwhile, in the channels' view, every input channel (depth) connects to all output channels. Due to this feature, as FM size increase in all dimensions, the cost for connecting channels takes the major part. The models mentioned before dealing with this problem and save resources in different ways.

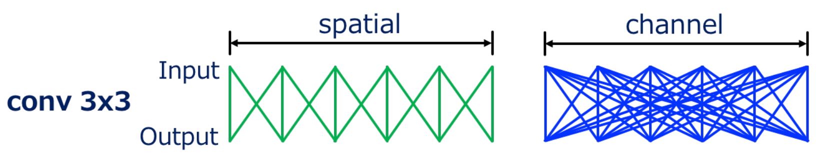

Figure 2.9: Spatial and channel connections of normal convolution taken from [26].

To improve the cost problem of the original network, pointwise convolution, grouped convolution, and channel shuffling are applied as shown in Figure 2.10. The computational cost has some parameters: height (H), width (W), input channel (N), output channel (M), kernel size (K×K) and group number (G). Conv 1×1 is also called pointwise convolution according to [27].

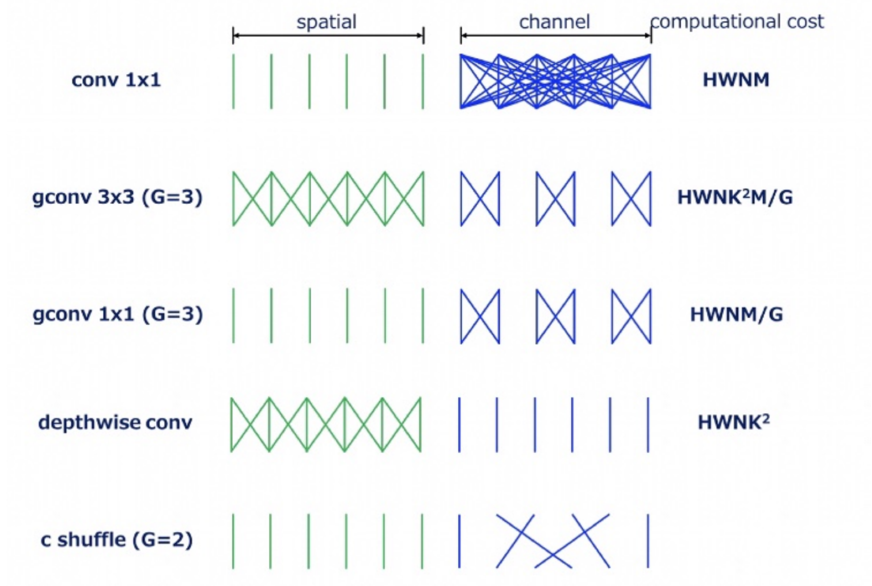

Figure 2.10: Different methods for convolution taken from [26].

Its kernel size is 1×1. That's why the spatial connection is simpler than original convolution. The functionality of pointwise convolution is to change the channel size.

Gconv 3×3 (G=3) is grouped convolution with kernel size 3×3 and group number 3. The difference between it and the original convolution is that channels are independent over groups. It does not change the spatial connection, only reduces the number of channel computation cost. Gconv 1×1 (G=3) only has spatial difference with gconv 3×3 (G=3).

Depthwise convolution has the same channel numbers for input FM and output FM. All channels are in an individual group with size 1. It separates all channels from connections, which reduces the computation cost in channel domain [28, 29, 30].

C shuffle has no computations. It only shuffles channels in a different order. For example, when group number is 2, the new channel list copies the input channel list every 2 channels.

2.2.1 ResNet

A deep residual learning network has an abbreviation of ResNet. A residual unit has a bottleneck structure as shown in Figure 2.11. The middle layer's channel number is not necessarily 2 but it is always smaller than input FM and output FM of the residual unit according to [31]. Apart from the unique layer

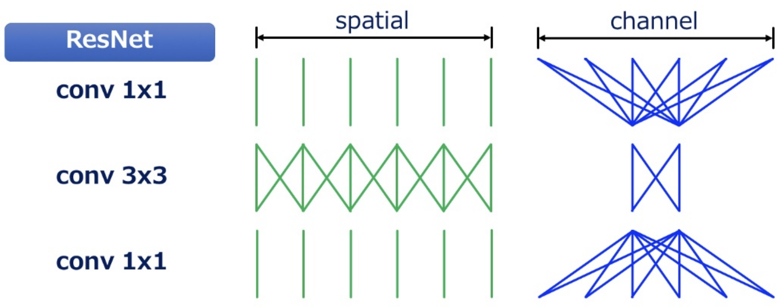

Figure 2.11: ResNet structure taken from [26].

structure, the residual network would solve the degradation problem when deep networks converge according to [32]. The solution to this problem is to map some copied layers from a shallower model to the deeper model according to [32]. Figure 2.12 shows the building idea of the additional structure. It passes the input feature map through two training convolution layers without modifications and adds the original feature map with the output.

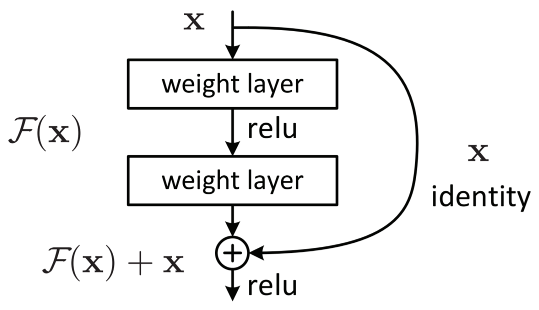
x
weight layer
relu
weight layer
$\mathcal{F}(\mathbf{x})$
$\mathcal{F}(\mathbf{x}) + \mathbf{x}$
relu
x
identity

Figure 2.12: An example block of residual learning taken from [32].

2.2.2 ResNeXt

ResNeXt is an improved model from the ResNet model. It changes the middle convolution to a grouped convolution. ResNeXt is proved to have a better classification accuracy than ResNet with the same computational cost. As

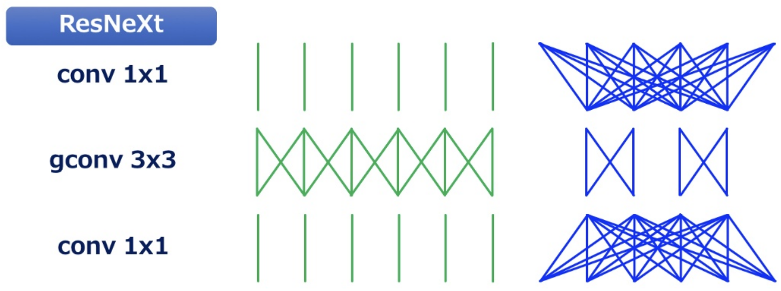

Figure 2.13: ResNeXt structure taken from [26].

shown in Figure 2.13 and Figure 2.11, if the channel numbers are set equally (n), the connection number of ResNeXt is smaller than it of ResNet as shown in the equation below.

$$N_{ResNet} = n^2 > N_{ResNeXt} = \left(\frac{n}{2}\right)^2 = \frac{n^2}{2} \tag{2.8}$$

Since channel connection influences the resource cost, ResNeXt is better than ResNet in resource consumption with the same parameters. Besides, ResNeXt also has the residual learning part of adding an identity input feature map into the output feature map.

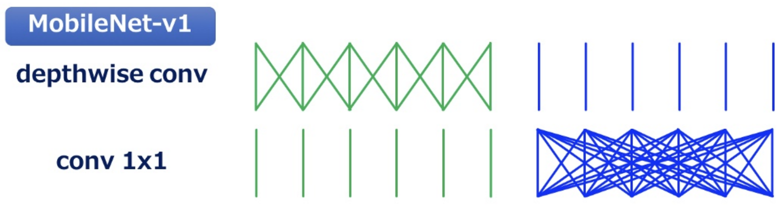

Figure 2.14: MobileNet structure taken from [26].

2.2.3 MobileNet

MobileNet has two parts: depthwise convolution and pointwise convolution, as shown in Figure 2.14 according to [10]. They are divided from original convolution. The depthwise convolution does not change the channel number. When the parameters are the same, the original convolution's computational cost is $HWNMK^2$, however, the computational cost of MobileNet is $HWN(M+K^2)$. In this example $K = 3$. M is always 32, 64 or larger. Thus, M is much larger than K^2. With K^2 neglectable, MobileNet's cost reduces to about 1/9 of original convolution.

2.3 Training CNN on GPU

Training CNN is a tough task even for stational computers. CNN models always have large sizes. The training process is repeated until acceptable results are achieved. Training has a restrict requirement for hardware and needs a lot of time. Thus, for making the full use of hardware and saving time, training CNN on GPU is a major way. There are multiple deep learning frameworks to help development, such as Caffe, CNTK, TensorFlow, Theano and Torch according to [33]. To achieve training on GPU, some tools like CUDA helps compute data on GPU.

PyTorch is a good deep learning platform based on python according to [34]. It supports multiple extended tools and libraries for deep learning work. It contains structured calculation models for tensors, like convolutions and max-poolings. Important functions are introduced in this part.

```
torch.nn.Conv2d(in_channels,out_channels,
    kernel_size,stride=1,padding=0,dilation=1,
    groups=1,bias=True,padding_mode='zeros')
```

The function `torch.nn.Conv2d()` is the most important function in the project. It supports quick access to the convolution operation. Because the feature maps contain two dimensions of information in every channel, the "2d" version of the convolution function is selected. `in_channels` represents the input feature map depth (channel number), and `out_channels` represents the kernel numbers of the particular layer which is also the depth (channel number) of the output feature map. According to different convolution methods mentioned in section 2.5, there are two kinds of convolutions in this project. For depthwise convolution, `kernel_size` is 3 and padding is 1. For pointwise convolution, `kernel_size` is 1 and padding is 0. Other parameters remain as default.

```
torch.nn.BatchNorm2d(num_features,eps=1e-05,
    momentum=0.1,affine=True,
    track_running_stats=True)
```

The function `BatchNorm2d()` normalizes the output feature maps after convolutions. All parameters except `num_features` remain default.
`num_feature` is consistent with the channel number of the previous convolution function output. The sizes of all three dimensions (height, width, and depth) do not change after this function. According to section 2.1.6, there are four important parameters of this function: calculated mean, calculated deviation, expected mean and expected deviation. They are not inputted by users. Calculated mean and deviation depend on the whole information of the input feature map. The expected mean and deviation are refined by the back-propagation process during the training process.

```
torch.nn.MaxPool2d(kernel_size,stride=None,
    padding=0,dilation=1,
    return_indices=False,ceil_mode=False)
```

The function `MaxPool2d()` shrinks the feature map size and condense the information. In this project, `kernel_size` is 2 and stride is 2. Other parameters remain default. Based on these parameters, every time the feature map goes through this function, both height and width shrink by 2 with depth unchanged.

```
torch.nn.ReLU(inplace=False)
```

The function `ReLU()` is a filter which erases all negative contents. The input is a whole tensor with all channels. The output is a tensor with the same size and every pixel inside comes from the `max(0, x)` operation. The only parameter in place remains false in this project.

```
torch.nn.Linear(in_features,out_features,
    bias=True)
```

This function `Linear()` is the base of the fully connected layer which is mentioned in section 2.1.4. The input and output are both one-dimension array. Parameters `in_features` and `out_features` represent the size of input and output. Parameter bias is default in this project.

CUDA Toolkit is provided by NVIDIA which enables “a development environment for creating high-performance GPU-accelerated applications” [35]. It transfers tensors calculation onto GPU for higher speed. The example of using CUDA is shown below.

```
device = torch.device("cuda:0"
    if torch.cuda.is_available() else "cpu")
# transfer images to GPU
images = images.to(device)
```

The learning rate would be changed during training. The learning rate is set larger at the beginning. When loss converges, the learning rate should be set smaller and keep on a train the training to get a smaller loss.

Dataloader function `torch.utils.data.DataLoader()` is a good method to handle a great number of raw images to produce a tensor stream with the expected batch size.

Chapter 3

Project Plan and Structure

In this part, the project plan is introduced and the reason to do so is explained. Useful tools and engineering methods are introduced in this section. Before the project starts, the process design is significant. Figure 3.1 shows the plan for this project.

3.1 Part A: Train Different Network Models

According to Figure 3.1, part A is the starting part of the whole project. Different networks refer to CNN models which are ResNet, ResNeXt, MobileNet, and ShuffNet. In this part, the most important thing is how good accuracy different models can achieve. It needs trying and modification to get acceptable results. After they achieve acceptable accuracy, reducing the cost with accuracy stable is a good try.

There are many attributes to represent the features of CNN models during the training part. Training time represents how long it takes the computer to train the model to get a target accuracy. Loss is generated by the loss function and represents the difference between the CNN model calculation result of the input image with the target one. Usually the lower the loss, the higher the performance of the CNN model has. The process of training a model through the whole training data set once is called an epoch. If the loss converges in several latest epochs of training, the performance of the current network is not going to improve in the training process. One method to improve is to lower the learning rate. If changing the learning rate cannot improve the performance, the network reaches the limit.

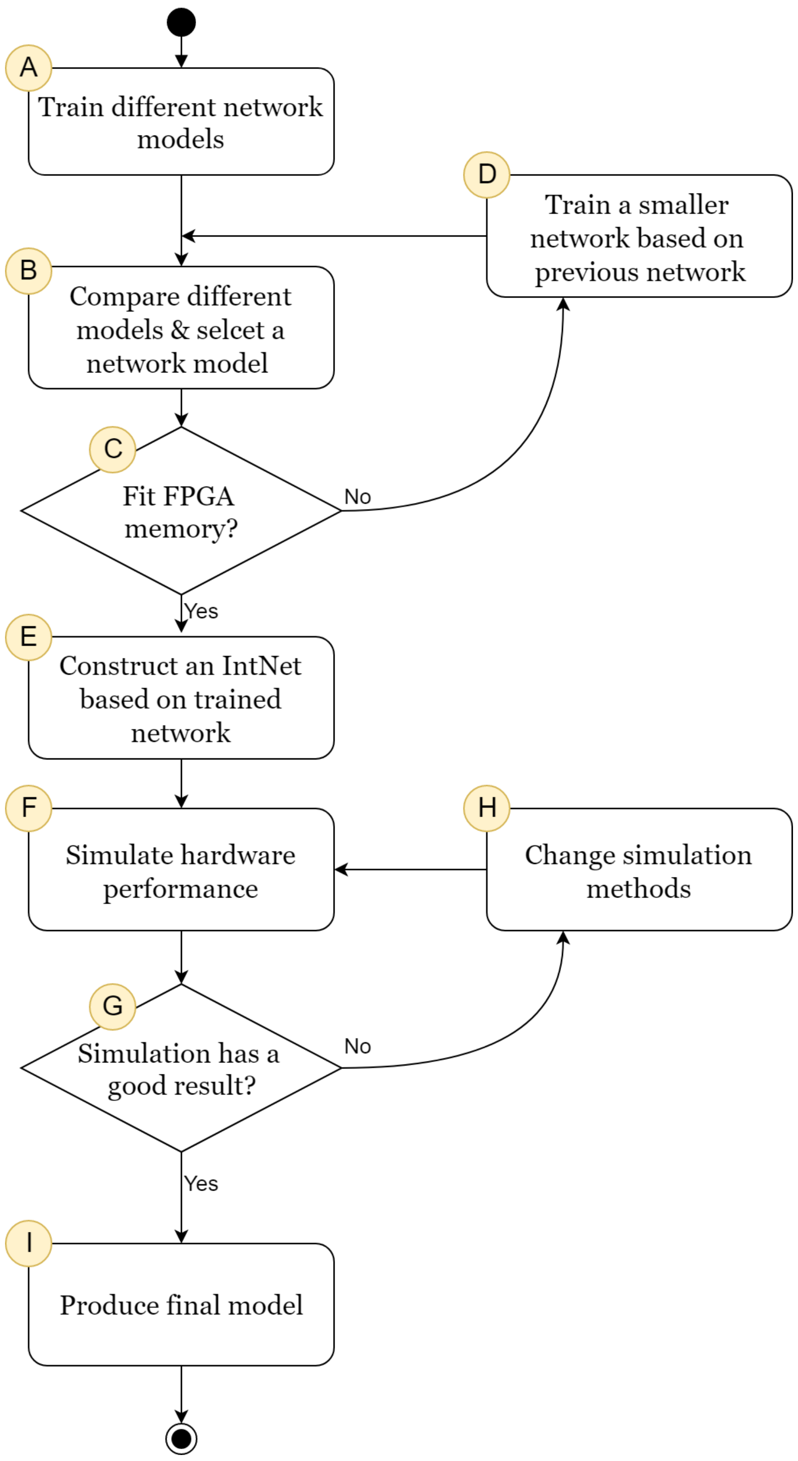
A
Train different network models
B
Compare different models & selcet a network model
C
Fit FPGA memory?
No
Yes
D
Train a smaller network based on previous network
E
Construct an IntNet based on trained network
F
Simulate hardware performance
G
Simulation has a good result?
No
Yes
H
Change simulation methods
I
Produce final model

Figure 3.1: Project plan.

3.2 Part B: Compare Different Models and Select a Network Model

Networks have different attributed to represent their features. The network model with lower spatial, lower time cost and better accuracy is great. Sometimes, the implementation difficulty on hardware is a great concern and the structural complexity of the model is considered, for example, an easily implemented model is better than a difficult one.

Floating point operations (FLOPs) represents the calculation size of a CNN model based on the floating number type. It can also show the time cost when different CNN models are compared based on the same hardware.

Parameter memory represents how much memory space is needed from the hardware. When the CNN model is implemented on a particular hardware device, the parameters including weights and bias of all layers need to be stored in hardware memory. The information data type is always decided to suit the hardware implementation, for example, all numbers should be 8-bit integers.

3.3 Part C: Fit FPGA Memory?

FPGA, as well as other embedded platforms, is strictly limited by on-chip resources. Spatial and time resources are mainly considered. The spatial resource is the memory size of the FPGA, which is always tens of megabytes (MB). The memory cost is mainly made of two parts: parameters of the network model and calculation results (feature maps) after every layer. Usually, the parameter memory cost is much smaller than feature map memory cost. The feature map memory cost can be the size of multiple feature maps when parallel processing is designed. The parallel structure requires feature maps of the same layer to be stored in the buffer at the same time, which puts much pressure on memory size. Even without the parallel structure, at least a whole feature maps are stored inside the buffer because the whole feature map is needed for the next layer of the CNN model. In conclusion, the memory should be able to handle the maximum feature map size among different layers' calculation results and all parameters.

If the training network result exceeds the memory limit, the process goes to part E, otherwise, the process goes to part F.

3.4 Part D: Train a Smaller Network Based on Previous Network

The feature of the model does not change, for example, network structure. The quantity of the model needs to be changed to solve the problem which is raised in part D. The feature map size is a product of batch size (B), output feature map width (U), height (V) and channel number (N) as shown in Table 2.1. The usual method is to reduce the channel numbers of layers. The batch size is also a flexible component, however, if a model cost exceeds the limit of hardware, its batch size is already set to 1 for the smallest cost.

If the worst condition is the case, there is always only one picture processed (batch size is 1), and no pipeline structure is used. This leads to the situation that only one layer is active at a time. So that reducing the depth of the CNN model (layer number) is not possible to solve the problem from part D.

Another method based on a special situation. When the layer that produces the maximum size of feature map is followed by a max-pooling function, putting the max-pooling function in front of the layer may solve the problem by reducing the width and height of output feature maps.

3.5 Part E: Construct an Integer-Net Based on Trained Network

This is a Boolean decision part. As mentioned in section 3.1.2, a model needs to keep the data type consistent with hardware requirements. The trained network model needs further validation after all the parameters are transformed to the required type. If parameters are out of range of integers from -128 to 127 that cover all 8-bit integers, it is better to normalize parameters into a suitable range without influences on calculation results of the network.

3.6 Part F: Simulate Hardware Performance

Part C is a good start to connect the CNN model training with hardware cost. This part focus on another important point on hardware: how does the performance change when the network is implemented on hardware. The performance includes testing accuracy and time cost.

The performance of hardware can be directly tested, however, one test for a change takes a long time. It is better to construct a software simulation with the hardware's calculation feature. If all software calculation follows the format of hardware, the accuracy result would be almost the same as it from hardware testing. The time cost is not considered here, so when performance is mentioned, it refers to the testing accuracy.

To simulate the calculation of hardware, number format, overflow, bit-cutting and division method are big issues. CNN training and validation are always processed on CPUs and mostly on GPUs. On these devices, the calculation precision is at least ensured. However, when it comes to hardware, calculation precision might be sacrificed to save memory cost.

3.7 Part G: Simulation Has a Good Result?

It is a Boolean decision part like part D. This part has an important component to judge if the CNN model produces a good result in simulation: accuracy. The same as to determine how well the trained CNN model performs, accuracy is the final achievement and the most direct attribute to reflect the quality of the model.

3.8 Part H: Change Simulation Methods

There are multiple solutions for transforming floating-point number based network to 8-bit integer number based network. The software part and hardware part are able to influence the strategy of each other, which leads to multiple ways of methods. All solutions need to follow the basic features of hardware like that calculation results need to be handled right after calculations to avoid memory exceeding. Accuracy would decrease after any solutions. A better solution decreases accuracy less.

3.9 Part I: Produce the Final Model

After all parts are finished, the CNN model is accepted. This part is responsible to produce corresponding files for the hardware part. Parameters need to be printed into the ".coe" file with the hexadecimal format. The process of how an image would be calculated including feature maps of all layers should be printed into a text file for hardware examinations.

Chapter 4

Network Training

The project process is shown in Figure 3.1, and it can be roughly separated into mainly two big parts. This section includes part A to part E in Figure 3.1 about training the network models and making the final decision.

4.1 Data set for training

The data set is supposed to be decided before the CNN model training. The complexity needs to be appropriate because the FPGA board has limited memory. The class number should also be as small as possible. A data set containing images of finger photos (representing numbers 0 5) is selected from [36]. The data image examples are shown in Figure 4.1. Data set like this is too

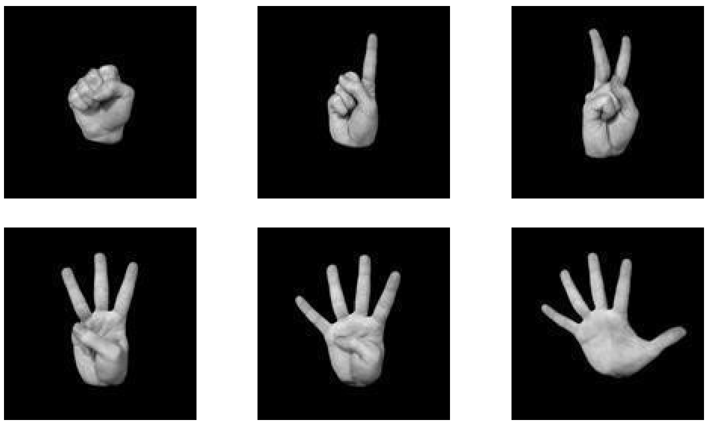

Figure 4.1: Original data set examples taken from [36].

simple to train a CNN model. There are some methods to make the data set more complicated like adding noise and background, whose details would be talked more in section 4. The images are one-channel grayscale images with size 128×128.

However, the data set is simple. Apart from images in Figure 4.1, the rest of the data set are all similar images with gestures from number zero to number five. To make the data set more representative, pure black background of the original data set need to be changed. The plan is to add random backgrounds to the pure black part of the images, which adds diversity into the data set and increase the complexity to do classification.

After the backgrounds are applied to the original data set, the images of the data set look like Figure 4.2. According to the figure, the hand images are not at the center and with the same size anymore. Hand gesture images are rotated. They are enlarged or shrink, and then they are placed at different corners of the image. Backgrounds are photos of the real world to make the training and testing data set closer to real use cases. After the handling on the images, it

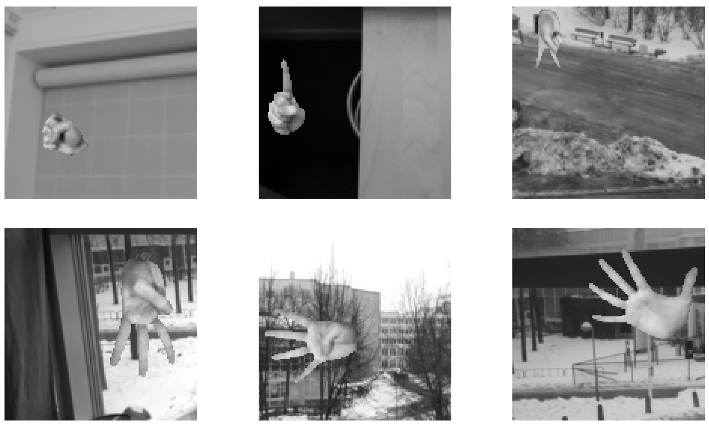
Figure 4.2: Data set examples after processing.

is important to categorize these data in a file including their file names and which class they belong to.

4.2 Restriction

When designing CNN models, the restrictions should be kept in mind. As mentioned in section 1.3.3, the FPGA kit that is used for reference has 38MB memory. Moreover, only UltraRAM of 27MB is accessible from this project's point of view. Other memories need to be connected to CPUs on FPGA which cost much time for memory reading and writing.

4.3 CNN Models Design and Modifications

The models that would be tried in this project are ResNet, ResNeXt, MobileNet, and ShuffNet. They are going to be implemented in this project with restrictions mentioned in section 4.2. The performance of the software level and the cost are shown here. The networks designed in this project are not the original ones in the articles. Their most important features are studied and the basic structure is maintained. Most importantly, the spatial cost needs to be reduced to an acceptable level by reducing the layer number or channel number.

Table 4.1: ResNet structure and FLOPs.

Layer Name		**Output Size**	**Output Channel**	**Kernel Size**	
Begin conv		128×128	8	3×3	padding = 1
Layer 1	Max pool	2×2 max pool, stride 2			
	Pointwise	64×64	4	1×1	Repeat once
	Depthwise *	64×64	4	3×3	
	Pointwise	64×64	16	1×1	
Layer 2	Max pool	2×2 max pool, stride 2			
	Pointwise	32×32	64	1×1	Repeat once
	Depthwise	32×32	64	3×3	
	Pointwise	32×32	256	1×1	
Layer 3	Max pool	2×2 max pool, stride 2			
	Pointwise	16×16	64	1×1	Repeat once
	Depthwise	16×16	64	3×3	
	Pointwise	16×16	256	1×1	
Layer 4	Max pool	2×2 max pool, stride 2			
	Pointwise	8×8	4	1×1	Repeat once
	Depthwise	8×8	4	3×3	
	Pointwise	8×8	16	1×1	
Fully connected layer		16-output fc1, 8-output fc2 and 6-output fc3			
FLOPs		1.93×10^8			

(* padding = 1)

4.3.1 ResNet

To achieve the residual deep learning, an open source ResNet model is used in this project according to [37]. In this model, Bottleneck and ResNet classes are defined. The bottleneck structure is made of two pointwise convolutions and one depthwise convolution in the middle. There are four layers after the beginning convolutional layer. Every layer begins with a max pool layer with stride 2 so that every layer begins with input size divided by 2 in width and height dimensions. Every layer contains two bottleneck structure in this project. At the end of every bottleneck structure, the input feature map is added into the calculated feature map. If the channel numbers from input and output do not match, there is an additional convolution layer which is also trained for channels to match. In 4 main layers, convolutions are followed by batch-norm layers with the corresponding channel number. All small layers including fully connected layers have ReLU functions before the feather map is output. The detail and FLOPs of the ResNet model are shown in Table 4.1.

Table 4.2: ResNet validation result.

Class	0	1	2	3	4	5
Accuracy (%)	94	93	90	94	89	97
Top-1 Error (%)	7.2					

In this model, the testing result of the 1800-image data set is shown in Table 4.2. According to this table, the performance of ResNet in this size and data set is not good.

4.3.2 ResNeXt

The ResNeXt model is similar to ResNet. They both use bottleneck structures as their main body. The only difference is there is a partially grouped depth-wise convolution instead of fully grouped depth-wise convolution in the bottleneck structure. Fully grouped depth-wise convolution is used in ResNet and different channels do not communicate during the depth-wise convolution operations.

As shown in Table 4.3, the total layer number (counting depth-wise layer as reference) is 8, which is the same as ResNet. The begin convolution, max pool, and fully connected layers are the same as ResNet.

The training and validation process uses the same data set. The validation result is shown in Table 4.4. The method of calculating the FLOPs is provided on according to [38].

Table 4.3: ResNeXt structure and FLOPs.

Layer name	**Output size**	**Output Channel**	**Kernel Size**	
Begin conv	128×128	8	3×3	padding = 1
Max pool 1	2×2 max pool, stride 2			
Pointwise *	64×64	64	1×1	Repeat once
Depthwise **	64×64	64	3×3	
Pointwise	64×64	32	1×1	
Max pool 2	2×2 max pool, stride 2			
Pointwise	32×32	128	1×1	Repeat 4 times
Depthwise	32×32	128	3×3	
Pointwise	32×32	64	1×1	
Max pool 3	2×2 max pool, stride 2			
Pointwise	16×16	32	1×1	
Depthwise	16×16	32	3×3	
Pointwise	16×16	16	1×1	
Max pool 4	2×2 max pool, stride 2			
Fully connected	16-output fc1, 8-output fc2 and 6-output fc3			
FLOPs	2.52×10^8			

(* padding = 0)
(** group = 8, padding = 0)

Table 4.4: ResNeXt validation result.

Class	0	1	2	3	4	5
Accuracy (%)	95	92	89	93	95	98
Top-1 Error (%)	6.3					

4.3.3 MobileNet

The MobileNet model structure in this project is shown in Table 4.5. Besides the beginning convolution layer, there are eight depth-point pairs of convolutions. 2×2 max pool functions with stride 2 are placed after the second, fourth, fifth and eighth pair. After the begin-convolution layer, one ReLU layer is placed afterward. Besides, each depthwise or pointwise convolution layer is followed by a batch-norm layer and a ReLU layer.

In this model, the testing result of the 1800-image data set is shown in Table 4.6 below.

Table 4.5: MobileNet structure and FLOPs.

Layer Name	**Output Size**	**Out Channel**	**Kernel Size**	**Padding**
Begin conv	128×128	8	3×3	1
Depthwise 1	128×128	8	3×3	1
Pointwise 1	128×128	32	1×1	0
Depthwise 2	128×128	32	3×3	1
Pointwise 2	128×128	64	1×1	0
Max pool 1	2×2 max pool, stride 2			
Depthwise 3	64×64	64	3×3	1
Pointwise 3	64×64	64	1×1	0
Depthwise 4	64×64	64	3×3	1
Pointwise 4	64×64	64	1×1	0
Max pool 2	2×2 max pool, stride 2			
Depthwise 5	32×32	64	3×3	1
Pointwise 5	32×32	64	1×1	0
Max pool 3	2×2 max pool, stride 2			
Depthwise 6	16×16	64	3×3	1
Pointwise 6	16×16	64	1×1	0
Depthwise 7	16×16	64	3×3	1
Pointwise 7	16×16	32	1×1	0
Depthwise 8	16×16	32	3×3	1
Pointwise 8	16×16	16	1×1	0
Max pool 4	2×2 max pool, stride 2			
Fully connected	16-output fc1, 8-output fc2 and 6-output fc3			
FLOPs	1.02×10^8			

Table 4.6: MobileNet validation result.

Class	0	1	2	3	4	5
Accuracy (%)	93	91	93	90	94	98
Top-1 Error (%)	6.8					

4.4 Comparison and Selection

According to section 4.3, the designs of ResNet model, ResNeXt model and MobileNet model are presented. The training process is recorded. The loss produced by loss functions represents how well the training goes. As shown in Figure 4.3, the loss curves of three models converge to nearly zero. It means

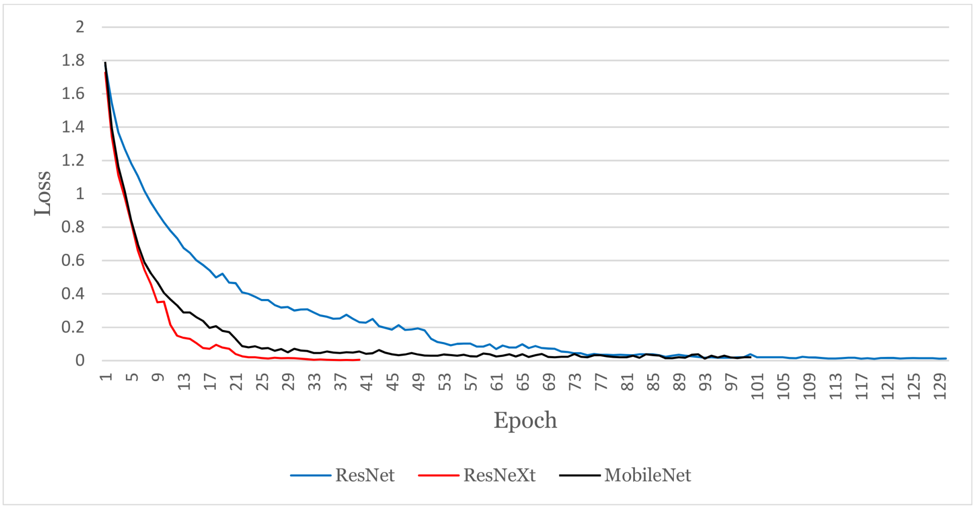

Figure 4.3: Loss curves of three models.

the training processes almost achieve the best performance of these models. These models can be compared in some aspects. The comparison in FLOPs, parameter quantity, and top-1 error is presented in Table 4.7. The method of calculating the FLOPs and number of parameters is provided on [38].

Table 4.7: Conparison among three models.

	ResNet	ResNeXt	MobileNet
FLOPs ($\times 10^8$)	1.93	2.52	1.02
# Parameters	360070	213800	43820
Top-1 Error (%)	7.2	6.3	6.8

According to Table 4.7, the choice is obvious. In FLOPs, ResNeXt is the worst because it uses the maximum channel number in some convolution layers, and gives channels more communications than ResNet. ResNet has the biggest number of parameters, while MobileNet has the fewest parameters by about 1/5. In top-1 error, the ResNet is the worst, however, their top-1 error is all acceptable.

These three models all use an 8-big-layer structure (a whole combination as a big layer, like a depthwise-pointwise layer combination in MobileNet). However, in the big layers of each model, MobileNet uses only two convolutions while ResNet and ResNeXt use one more layer of point-wise convolution. MobileNet saves resources on hardware by having fewer layers of convolutions. In other aspects and statistics, ResNet and ResNeXt have no huge advantages over MobileNet. So the MobileNet model is picked for the later process.

4.5 Improvement on Selected CNN Model

The MobileNet-based CNN model is selected for this project. It needs some modifications to fit more requirements. This model has six main modules that are going to be implemented on hardware, which are the depthwise convolution module, pointwise convolution module, ReLU module, batch-norm module, fully connected module, and max pool module.

Due to the fact that the batch-norm module is the most resource-consuming module among all modules, it is the bottleneck in this project to some degree. All modules cost resources, and most modules can reduce the cost with some methods. If the modules on the hardware are connected directly, for example, the first convolution output directly goes into the ReLU module, it saves memory space which is supposed to be a buffer to store feature maps between adjacent layers. Batch-norm module cannot save the buffer cost because it includes calculations of addition, multiplication, division and square root on input feature map pixels.

To improve the potential performance and to make the best use of hardware, the current CNN model is improved. The main frame like the order of convolution layers and corresponding parameters are not changed. The modification focuses on the batch-norm module. In the current design, as shown in Table 4.5, the batch-norm layer and ReLU layer are placed before the following max pool layer. The modification is to put four pairs of batch-norm and ReLU layers that are followed by max pool layer after the corresponding max pool layers, as shown in Table 4.8. After the improvement, the output size of

Table 4.8: MobileNet model improvement.

Before		After	
Layer Name	Output Size	Layer Name	Output Size
pointwise conv 2	64×64	pointwise conv 2	64×64
batch norm	64×64	**2×2 max pool**	32×32
ReLU	64×64	batch norm	32×32
2×2 max pool	32×32	ReLU	32×32

changed batch-norm layers are reduced by 4. The validation result after the improvement is shown in Table 4.9. Based on the situation that some layers' size is reduced, the accuracy is still in the same level as previous MobileNet model.

Table 4.9: MobileNet validation result after improvement.

Class	0	1	2	3	4	5
Accuracy (%)	96	95	92	92	95	97
Top-1 Error (%)	5.5					

Chapter 5

Simulation of Hardware Performance

In chapter 4, the CNN models are trained on GPUs with the type of floating numbers. There are no strict limits on the network. However, the hardware (FPGA in this project) is good in some specific fields. It needs the model to be handled to fit the FPGA's feature. This chapter is going to talk about generating an appropriate model based on the model mentioned in chapter 4 and simulating the performance of the new model.

5.1 Integr-Net Model construction

The MobileNet model is the final decision with a good performance in chapter 4. The Integer-Net bases on the MobileNet. It keeps the same structures of MobileNet, like the number of layers, layer types, and parameters. In this section, the only thing is to copy the model, and change the parameters from

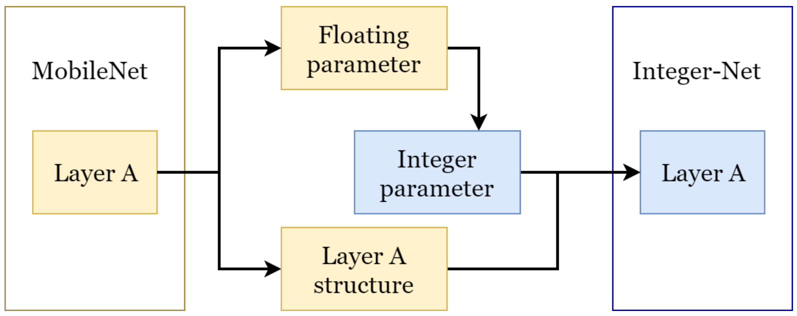

Figure 5.1: Copy process from MobileNet to Integer-Net.

floating numbers to signed 8-bit integers. The requirement of the change is to keep the final result of classification unchanged. The final result depends on the final fully connected layer output. The class with the highest number is the classification result. This requires every layer's output to be changed in the same rate within the same layer, and so do parameters. The process can be represented by Figure 5.1. It shows the example transfer of one layer, which is also called a module. The change is layer separable.

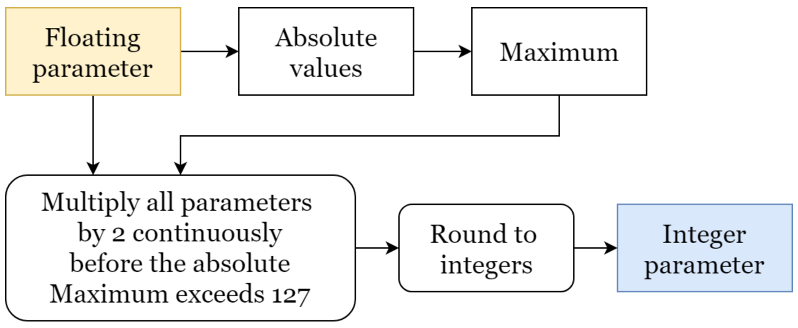
Floating parameter
Absolute values
Maximum
Multiply all parameters by 2 continuously before the absolute Maximum exceeds 127
Round to integers
Integer parameter

Figure 5.2: Parameter change process from floating point numbers to integers.

For each layer, the parameters are tensors and all parameters are smaller than 1 in this particular model. They are changed in the same way. The way of change is shown in Figure 5.2, it puts all parameters in the range from -127 to +127 and prevents parameters from exceeding the upper bound.

Parameter categories are presented in Table 5.1. For layers with only one kind of parameter like convolution layers, they only contain 3×3 kernel weights as parameters. Other layers like batch-norm layers and fully connected layers have two kinds of parameters: weights and bias. The way of dealing with these parameters is to change these two kinds of parameters in the same way (multiplied by 2 at the same time). The reason will be explained

Table 5.1: Parameter categories and sizes of MobileNet model.

Layer Name	**Parameters**
Begin conv	Weight $[1 \times channel\_out \times 3 \times 3]$
Depthwise conv	Weight $[1 \times channel\_out \times 3 \times 3]$
Pointwise conv	Weight $[channel\_out \times channel\_in \times 3 \times 3]$
Batch-norm	Weight [1]; Bias [1]
Fully connected	Weight $[channel\_out \times channel\_in]$; Bias $[channel\_out \times channel\_in]$
ReLU	(No parameters)

in later this chapter.

The reason why three convolution layers do not have a bias as a parameter is the argument `bias=False` is set for convolution layers during the network training.

From equation 2.7 in section 2.1.6, the equation can be represented as equation 5.1. All parameters are changed to fit Integer-Net requirements, so X^{BN} is changed corresponding to the change for it is the output of convolution layers. However, the attribute X_{middle} will not change. μ and σ are calculated mean and deviation from input X^{BN}, so we can consider X_{middle} remains unchanged. Based on that, γ and α should be multiplied at the same rate to make sure the output relationship unchanged.

$$Y^{BN} = \frac{X^{BN} - \mu}{\sqrt{\sigma^2 + \epsilon}}\gamma + \alpha = X_{middle}\gamma + \alpha \tag{5.1}$$

5.2 Output Feature Map Precision Treatment

Parameters are changed to fixed-bit integers after the copy from the MobileNet model to the Integer-Net model. The input image pixels initially contain 8-bit unsigned integers (0 - 255). Every layer includes the multiplication of feature map pixels and parameters. For example, if the parameters are 8-bit unsigned integers, the product of one 8-bit signed integer and one 8-it unsigned integer is a 16-bit signed integer.

It is a must to round the output feature map contents to fixed-bit numbers. There are two main reasons. The first reason is that the hardware needs to have numbers in process with the same format, which is chosen to be fixed-bit signed integers. The second reason is that if the output feature map contents are not rounded, the size of numbers will grow continuously which the FPGA board cannot handle.

So that there are many tryouts to get the best way of number handling which keeps the good performance and is possible for FPGA to achieve it.

5.2.1 Tryout 1

The first method brought out is to multiply/divide the whole feature map after each layer by 2 continuously before/until the maximum of the absolute tensor is bigger than 127. It is more clear according to the following codes.

```
while(torch.max(torch.abs(x)) > 127):
    x = torch.round(x / 2)
```

This method makes sure every layer makes the full use of the 8-bit space. If many bits representing smaller digits are ignored, the detail of numbers would be missing, which leads to the loss of precision. This method depends on the output number range. It cost extra resource on FPGA to calculate every layer's maximum of absolute feature map values.

However, the result is not acceptable. When every middle feature map is within the range from -127 to 127, the accuracy is terrible, as shown in Table 5.2. This top-1 error means using 8-bit signed integers as content pixel size

Table 5.2: The accuracy result with number range [-127, 127].

Class	0	1	2	3	4	5
Accuracy (%)	34	78	77	66	47	20
Top-1 Error (%)	46.3					

results in a bad accuracy result. The small size might swipe small numbers that influence the result due to feature map pixels' diversity. So that a slightly bigger integer size should be used to make the simulation work. 9-bit integers are chosen as the improvement of this tryout. Hence the code which controls the output feature map contents range is changed as below.

```
while(torch.max(torch.abs(x)) > 255):
    x = torch.round(x / 2)
```

This code only changes the number 127 to 255, however, the result of accuracy is totally different. As shown in Table 5.3, the top-1 error is acceptable.

Table 5.3: The accuracy result with number range [-255, 255].

Class	0	1	2	3	4	5
Accuracy (%)	92	88	88	81	87	87
Top-1 Error (%)	12.8					

5.2.2 Tryout 2

The method in section 5.2.1 based on every feature map output of each layer. It needs the platform to search the maximum number first, which requires extra cost. It is accepted in the software's point of view but not in hardware's. The hardware part needs a fixed method to deal with the repeating process. Through the process of calculating different images, the handling method in the same layer should be the same.

So that a method based on hardware features is raised. By calculating and estimating the size of each layer's number size, the output number range of each layer has an estimation. The output range is required to be an 8-bit signed integer, and the bit-cutting method from a calculation result is presented in Table 5.4. The first bit is the sign bit, so the cutting method deal with which 7 bits should be kept.

Table 5.4: Bit control of tryout 2.

Layer Name	Kept Bits [Large bit, Small bit]
Depth-wise	[11, 5]
64-channel Point-wise	[14, 8]
32-channel Point-wise	[13, 7]
16-channel Point-wise	[12, 6]
Batch-norm	[11, 5]
FC 1 (1024×16)	[22, 16]
FC 2 (16×8)	[13, 7]
FC 3 (8×6)	[12, 6]

Unfortunately, this method gives a bad result as shown in Table **??**.

Table 5.5: The accuracy result of tryout 2.

Class	0	1	2	3	4	5
Accuracy (%)	0	12	18	27	51	25
Top-1 Error (%)	77.8					

5.2.3 Tryout 3

The direction of tryout 2 is correct. The hardware is not supposed to test the maximum during each classification process. It is better to know the common number range within a different process. This tryout is a combination of tryout 1 and 2.

The first step is to repeat the process in tryout 1 with range [-255, 255] to keep good performance. During this process, the change in each layer is recorded.

The second step is to set the range for each layer like it in tryout 2. In the simulation process, it changes each layer's output to have numbers in the range [-255, 255]. The different layer has different parameters and the bit handling varies over layers.

After two steps mentioned above, each layer produces a feature map with 9-bit signed integers. The smaller bits are swiped in the `torch.round()` function. However, overflow is not talked about yet. According to the data set, the result will not exceed the range. The overflow prevention is still needed when numbers are fixed by bit size. The main idea of overflow prevention is to avoid the change between positive numbers and negative numbers, which may cause a big failure in the result.

Based on the numbers produced in the second step, the system wants to do a remainder calculation on these numbers to avoid overflow. The goal is to get a range of [-128, 127]. It needs some tricks to achieve it. The small trick is shown in the following code. This trick treats positive and negative numbers differently. It well simulates the hardware performance of cutting the bits out of range.

```
relu_mod = torch.nn.modules.activation.ReLU()
x = relu_mod.forward(x)%128-torch.ceil(
    (relu_mod.forward(0-x*2-1))%256/2 )
```

Based on the number produced by the steps mentioned above. The accuracy is not influenced much. As Table 5.6 shows, the top-1 error is almost the same as the result in tryout 1.

Table 5.6: The accuracy result of tryout 3.

Class	0	1	2	3	4	5
Accuracy (%)	91	89	83	84	88	91
Top-1 Error (%)	12.3					

Chapter 6

Comparison and Evaluation

This chapter is about the comparisons during the project that determine key selections, and evaluations that show the performance of the project.

6.1 Model Comparison and Evaluation

CNN model comparison and evaluation are described in detail in section 4.4. This comparison and evaluation happen in the middle part of the project. There are three potential CNN models to be selected for further development. There are three main attributes to compare: FLOPs, number of parameters, and top-1 error.

As shown in Table 4.7, ResNeXt is the worst at FLOPs, ResNet has the largest number of parameters, and all of the three models have similar top-1 errors. By the quantitative method, the MobileNet model is the best choice.

6.2 Hardware Simulation Evaluation

The hardware simulation also uses both quantitative and qualitative methods. The result of tryout 1 in section 5.2.1 is acceptable with the top-1 error smaller than 15%. However, due to the qualitative standard, the tryout 1 exceeds the ability of hardware. Thus tryout 1 cannot be the final decision.

Tryout 2 in section 5.2.2 puts more emphasis on hardware ability so that it uses fixed cutting rule but it produces a bad result.

Tryout 3 in section 5.2.3 combine the advantages of previous 2 tryouts. The only quantitative standard is the top-1 error after simulation. this tryout has the top-1 error with 12.3% that is smaller than 15%. According to the detail in section 5.2.3, there are no additional complicated calculations. It does not add extra recourse cost to the hardware. Hence the simulation meets the goal.

Chapter 7

Discussion and Conclusion

This project has its limits. It does not use different data set to prove the model are suitable for more cases. The algorithm size is limited in size so that it can not use some common data set like Cifar-10 as a standard.

In my acknowledgment, there is no research to fully describe how to port a software-based CNN algorithm onto the hardware platform. This project is going to produce a complete system that is supposed to be on hardware in the future, so this project should include a method of connecting the software implementation and hardware. That's why there is one chapter talking about how to simulate the algorithm performance on hardware. It's worth a try because it saves a lot of time comparing to the common hardware testing.

This project achieves what is set in the goal. This project produces CNN algorithms based on the current CNN model (ResNet, ResNeXt, and MobileNet) and has a good accuracy of 93.7% as the best. This project compares different models by restricting standards and to select MobileNet for hardware to implement. This project successfully transforms the selected model to the format that is suitable for hardware. To be specific, the parameters are changed to 8-bit integers in a way without influencing the result. This project generates a solution to simulate the network performance on hardware (particularly FPGA) and give a demonstration as reference. The simulation puts the numbers into 8-bit integers bit-wise. It well simulates the performance on hardware which helps reflect the performance on hardware. The demonstration shows the simulation performance on hardware which results in an accuracy of 87.7%. This accuracy is acceptable.

In conclusion, this project achieves the goal of the plan and meets the satisfaction of the company well.

Chapter 8

Future Work

In the future, this project could be improved in the future. Different data set can be chosen to test the performance of this model. It can show if this model works on general data. More CNN models can be chosen and be compared to the current one to get the most efficient model. There are multiple efficient models to be tried, like ShuffleNet and BinaryNet.

Bibliography

[1] John R. Koza et al. "Automated Design of Both the Topology and Sizing of Analog Electrical Circuits Using Genetic Programming". In: *Artificial Intelligence in Design '96*. Ed. by John S. Gero and Fay Sudweeks. Dordrecht: Springer Netherlands, 1996, pp. 151–170. ISBN: 978-94-009-0279-4. DOI: 10.1007/978-94-009-0279-4_9. URL: https://doi.org/10.1007/978-94-009-0279-4_9.

[2] Li Deng and Dong Yu. "Deep Learning: Methods and Applications". In: *Foundations and Trends® in Signal Processing* 7.3–4 (2014), pp. 197–387. ISSN: 1932-8346. DOI: 10.1561/2000000039. URL: http://dx.doi.org/10.1561/2000000039.

[3] Yoshua Bengio. "Learning Deep Architectures for AI". In: *Foundations and Trends® in Machine Learning* 2.1 (2009), pp. 1–127. ISSN: 1935-8237. DOI: 10.1561/2200000006. URL: http://dx.doi.org/10.1561/2200000006.

[4] Keiron O'Shea and Ryan Nash. "An Introduction to Convolutional Neural Networks". In: *CoRR* abs/1511.08458 (2015). arXiv: 1511.08458. URL: http://arxiv.org/abs/1511.08458.

[5] Dana H. (Dana Harry) Ballard and 1945- Brown Christopher M. *Computer vision*. English. Includes bibliographies and indexes. Englewood Cliffs, N.J. : Prentice-Hall, 1982. ISBN: 0131653164. URL: http://homepages.inf.ed.ac.uk/rbf/BOOKS/BANDB/bandb.htm.

[6] Lee Cloer. *FPGA vs Microcontroller – Advantages of Using An FPGA*. 2017. URL: https://duotechservices.com/fpga-vs-microcontroller-advantages-of-using-fpga.

[7] *Xilinx Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit*. 2019. URL: https://www.xilinx.com/products/boards-and-kits/zcu104.html.

[8] *Digilent, Pcam 5C Camera Module for OV5640, PB200-358 for FPGA, 410-358.* URL: https://se.rs-online.com/web/p/products/1741555/?grossPrice=Y&cm_mmc=SE-PLA-DS3A-_-google-_-CSS_SE_SE_Semiconductors-_-Semiconductor_Development_Kits%7CInterface_Development_Kits-_-PRODUCT_GROUP&matchtype=&pla-391509112982&gclid=Cj0KCQjw1pblBRDSARIsACfUG10TaB1l1Q1-7_Ls_jjY1sjwrZX_sD_o7mXs3xC1Q0L8hRKnh_JPcnQaAqhLEALw_wcB&gclsrc=aw.ds.

[9] *Vector - Computer Screen.* URL: https://www.123rf.com/photo_36565752_stock-vector-computer-screen.html.

[10] Andrew G. Howard et al. “MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications”. In: *CoRR* abs/1704.04861 (2017). arXiv: 1704.04861. URL: http://arxiv.org/abs/1704.04861.

[11] *Transfer Learning.* URL: http://cs231n.github.io/transfer-learning/.

[12] Kamel Abdelouahab et al. “Accelerating CNN inference on FPGAs: A Survey”. In: *CoRR* abs/1806.01683 (2018). arXiv: 1806.01683. URL: http://arxiv.org/abs/1806.01683.

[13] Adam H. Marblestone, Greg Wayne, and Konrad P. Kording. “Toward an Integration of Deep Learning and Neuroscience”. In: *Frontiers in Computational Neuroscience* 10 (2016), p. 94. ISSN: 1662-5188. DOI: 10.3389/fncom.2016.00094. URL: https://www.frontiersin.org/article/10.3389/fncom.2016.00094.

[14] Rafal Józefowicz et al. “Exploring the Limits of Language Modeling”. In: *CoRR* abs/1602.02410 (2016). arXiv: 1602.02410. URL: http://arxiv.org/abs/1602.02410.

[15] S. Haykin and B. Kosko. “GradientBased Learning Applied to Document Recognition”. In: *Intelligent Signal Processing*. IEEE, 2001. ISBN: 9780470544976. DOI: 10.1109/9780470544976.ch9. URL: https://ieeexplore.ieee.org/document/5265772.

[16] Todd Rowland and Eric W. Weisstein. *Tensor.* URL: http://mathworld.wolfram.com/Tensor.html.

[17] Baris Kayalibay, Grady Jensen, and Patrick van der Smagt. “CNN-based Segmentation of Medical Imaging Data”. In: *CoRR* abs/1701.03056 (2017). arXiv: `1701.03056`. URL: `http://arxiv.org/abs/1701.03056`.

[18] AARSHAY JAIN. *Deep Learning for Computer Vision – Introduction to Convolution Neural Networks*. 2016. URL: `https://www.analyticsvidhya.com/blog/2016/04/deep-learning-computer-vision-introduction-convolution-neural-networks/`.

[19] A. L. Hodgkin and A. F. Huxley. “A quantitative description of membrane current and its application to conduction and excitation in nerve”. In: *Bulletin of Mathematical Biology* 52.1 (Jan. 1990), pp. 25–71. DOI: `10.1007/BF02459568`. URL: `https://doi.org/10.1007/BF02459568`.

[20] *Activation function*. URL: `https://en.wikipedia.org/wiki/Activation_function`.

[21] *Convolutional Neural Networks (CNNs / ConvNets)*. URL: `http://cs231n.github.io/convolutional-networks/`.

[22] *Figure 3 - available via license*. URL: `https://www.researchgate.net/figure/The-structure-of-the-fully-connected-layer-a-b-and-the-data-dependency-c_fig2_318025612`.

[23] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. `http://www.deeplearningbook.org`. MIT Press, 2016.

[24] Michael A. Nielsen. *Neural Networks and Deep Learning*. URL: `http://neuralnetworksanddeeplearning.com/`.

[25] *TORCH.NN*. URL: `https://pytorch.org/docs/stable/nn.html`.

[26] *Why MobileNet and Its Variants (e.g. ShuffleNet) Are Fast*. URL: `https://medium.com/@yu4u/why-mobilenet-and-its-variants-e-g-shufflenet-are-fast-1c7048b9618d`.

[27] Min Lin, Qiang Chen, and Shuicheng Yan. “Network In Network”. In: *arXiv e-prints*, arXiv:1312.4400 (Dec. 2013), arXiv:1312.4400. arXiv: `1312.4400 [cs.NE]`.

[28] Laurent Sifre. “Rigid-Motion Scattering For Image Classification”. PhD thesis. CMAP, 2014.

[29] Laurent Sifre and Stephane Mallat. "Rotation, Scaling and Deformation Invariant Scattering for Texture Discrimination". In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2013.

[30] Francois Chollet. "Xception: Deep Learning With Depthwise Separable Convolutions". In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. July 2017.

[31] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *CoRR* abs/1512.03385 (2015). arXiv: 1512.03385. URL: http://arxiv.org/abs/1512.03385.

[32] Kaiming He et al. "Deep Residual Learning for Image Recognition". In: *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. June 2016.

[33] *Convolutional Neural Network (CNN)*. URL: https://developer.nvidia.com/discover/convolutional-neural-network.

[34] *PyTorch*. URL: https://pytorch.org/.

[35] *Develop, Optimize and Deploy GPU-accelerated Apps*. URL: https://developer.nvidia.com/cuda-toolkit.

[36] *Fingers-Classify images of 0,1,2,3,4 or 5 fingers*. URL: https://www.kaggle.com/koryakinp/fingers.

[37] fmassa. *Datasets, Transforms and Models specific to Computer Vision*. vision/torchvision/models/resnet.py. 2019.

[38] *Flops counter for convolutional networks in pytorch framework*. URL: https://github.com/sovrasov/flops-counter.pytorch.

TRITA-EECS-EX-2019:219