A thesis submitted in partial satisfaction of the requirements
for the degree of Master of Computer Science and Engineering
in the Graduate School of the University of Aizu

Animal Recognition and Identification with Convolutional Neural Networks for Farm Monitoring System

by

Ryunosuke Murakami

September 2019

The thesis titled

Animal Recognition and Identification with Convolutional Neural Networks for Farm Monitoring System

by

Ryunosuke Murakami

is reviewed and approved by:

Main referee

Professor

Ben Abdallah Abderazek 2019/08/19

Associate Professor

Yuichi Okuyama 2019/08/17

Senior Associate Professor

Satoshi Nishimura Aug 9, 2019

THE UNIVERSITY OF AIZU

September 2019

Contents

List of Figures

List of Tables

Abstract

Currently, research on AI technology using deep learning is actively conducted. The main applications are object recognition, speech recognition, language translation, and so on. Compared with the conventional machine learning method such as SVM, it exhibits high robustness in object feature extraction, and its recognition accuracy is close to human-level. One of the problems with such technology is high computational complexity. In the CPU operation, there are delays due to the restriction of sequential processing and an increase in power consumption due to the use of redundant computational units. On the other hand, GPUs consist of many cores and are good at parallel processing; however, the increase in power consumption in proportion to the number of cores is raised as a problem. As the implementation of deep learning applications for embedded systems, these limitations become a major problem. In FPGAs, it is possible to design an application specific circuit with reconfigurable logic elements, so that optimization in such a problem can be expected. In this thesis, we developed a power efficient object recognition system for farm monitoring system based on convolutional neural network, and explain an FPGA implementation that applies to speed up and low power consumption design. This thesis only covers the implementation of the object recognition unit in the proposed farm monitoring system. The recognition accuracy of the implemented classifier without optimization is 98.18%. As optimization in the implementation, pipeline design and quantization by the fixed-point implementation are applied. As implementation of quantization, we implemented with 16 bit and 8 bit. In the 16-bit implementation, the processing was faster by 3.67 times, and the recognition accuracy remained the same. On the other hand, in the 8-bit implementation, the processing was faster by 4.66 times, and the recognition accuracy dropped to 94.53%. As future work, we will take measures against further quantization and the accompanying precision loss.

Chapter 1

Introduction

Due to the increasing demand in the agricultural industry, the need to effectively grow and protect a plant and increase its yield is necessary. It is essential to monitor the plant during its growth period and protect it from animals at the time of harvest. Monitoring the plants from plantation to harvesting is necessary for better productivity. Smart farming needs a right decision and monitoring tools for better productivity, quality, and profit. Artificial neural network concept is efficient for image processing. Convolution Neural Network (CNN) appeared as a very effective way for an object recognition system. We adopted this technology for an efficient farm monitoring system. This system came from the actual human brain, and it has demonstrated that performance is very close to human-level. Our research goal is to develop a power efficient object recognition system for farm monitoring system based on artificial neural network concept. The following two main methods were adopted to achieve this goal. We aimed to do hardware implementation of CNN on FPGA for higher computation speed and lower power consumption than CPU based system. Evaluation of real hardware complexity (power and area) and performance (recognition accuracy, time) showed a better result compared with CPU based system.

1.1 Background

Recently, deep learning citelecun2015deep technologies grows rapidly, and it has performed very well on many application like object recognition citeren2015faster, voice recognition citevenayagamoorthy1998voice , and language translations citecho2014learning. That method achieved incredibly high accuracy on each specific regions, and its accu-

racy is close to human-level. Deep learning methods achieved most of the highest methods in one large image classification competition ILSVRC citedeng2012ilsvrc, and those research fields expanding more and more.

1.2 Motivation

The conventional method for farm monitoring system adopts straightforward and brief sensors like an infrared sensor to detect vermin. This can be easily implemented, but it not flexible in detecting the specific targets. Those systems always make alerts whatever the things touch the infrared sensors. To solve this problem, we focused on efficient and reliable farm monitoring system by using deep learning algorithms. While its classification accuracy is high, like human level, it consumes much cost like time and electrical power for its huge computational requirements. To care these defects, we decided to use FPGA citemittal2018survey, which can design any specific computation unit for this system.

1.3 Purpose of the Thesis

In this thesis, we propose one efficient object recognition system which uses deep learning method. We made a human-level object recognition system in software, and then we did hardware implementation and accelerated the network by several optimization methods. As a result, we could achieve high computation speed, and high accuracy object recognition system, which is expected to be used in our farm monitoring system. This thesis introduces some efficient optimization method that we adopted and well suited for the high computation speed and high classification accuracy.

1.4 Purpose of the Thesis

In this thesis, we proposed one efficient farm monitoring system which uses deep learning method. We made a human-level object recognition system in software, and then we did hardware implementation and accelerated the network by several optimization methods. As a result, we could achieve high computation speed, and high accuracy

object recognition system, which is expected to be used in our farm monitoring system. This work belongs to a large scale neuromorphic system [1–7], which is being developed at the Adaptive Systems Laboratory [8].

1.5 Thesis Organization

The organization of this thesis is as follows. Chapter 2 is an explanation of the neural network, which is the basis of the proposed method. Chapter 3 introduces the idea of the proposed farm-monitoring system architecture. Chapter 4 explains hardware implementation and applied optimization methods. Chapter 5 shows the evaluation results, which indicated that our design improved the performance compared with software implementation. Finally, chapter 6 is dedicated to the conclusion and future work.

Chapter 2

Neural Network Architecture

The idea of a Neural Network [9] is derived from the actual human brain, which consists of more than 10 billion nerve cells [10]. Each of those neuro nerve cells is connected with other neuro-cells, and it forms one large network. This method turned out that it performs much better on information processing tasks than the conventional method which uses predefined features and compare. [11] Those methods are currently used in the field of image recognition, natural language processing, and other pattern recognition tasks. An artificial neural network is a mathematical model of human brain structure as explained, and it shows outstanding performances in various fields. This chapter introduces a brief idea of the Neural Network.

2.1 Biological Neuron

To help understand the basic behavior of artificial neurons, the basic structure of biological neurons is explained here. Figure 2.1 shows a structure of the biological neuron. Neuron cell is composed of 3 main parts, cell body, axon, and dendrite. The cell body is the parts where axon and dendrite are connected, and protein synthesis occurs here. Axon is used for transmitting the material which is synthesized in neuron body. Dendrite is a tree structure branches, and it collects signals from other neuron cells. The functionality of neuron is transmitting information to other neurons when it gets signals from others by generating an action potential. All its transmissions are controlled by bias to change their propagation.

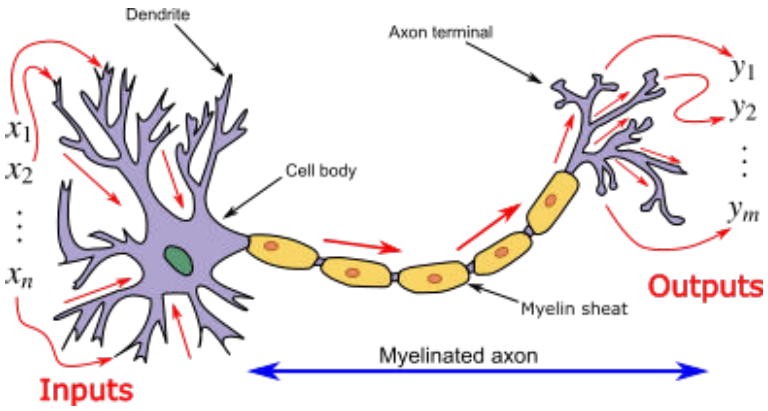

Figure 2.1: Model of Biological Neuron [12]

2.2 Artificial Neuron

The idea of an Artificial Neural Network is from the biological neuron. Biological neuron behavior is represented by mathematics. Input signals are from other adjacent neurons output. Artificial neuron input the summation of all other connected neurons output signals. If the input values, the summation of all input signals, are bigger than a bias value, it will fire and transmit the output signal to other neurons. Digital values represent firing, if its value is 0 it indicates no firing, and if its value is 1 it indicates firing. And weights of input signals from other neuron is considered, which can assign importance of other connected neurons signals. Each neurons' weight valu is determined by learning process which is described in Chapter 2.4. This function is mathematically equated in Equation (2.1) and (2.2), and this artificial neuron is drawn in Figure 2.2.

$$u = \sum_{k=1}^{n} x_k w_k \, (n \; = \; number \, of \, input) \tag{2.1}$$

$$y = f(u - \theta) \tag{2.2}$$

2.3 Activation Function

As you can see the mathematical representation of the neuron model, this model has only linearity in its expression. However, to solve complicated and challenging classification problems, it needs to have non-linearity for more flexible expression abil-

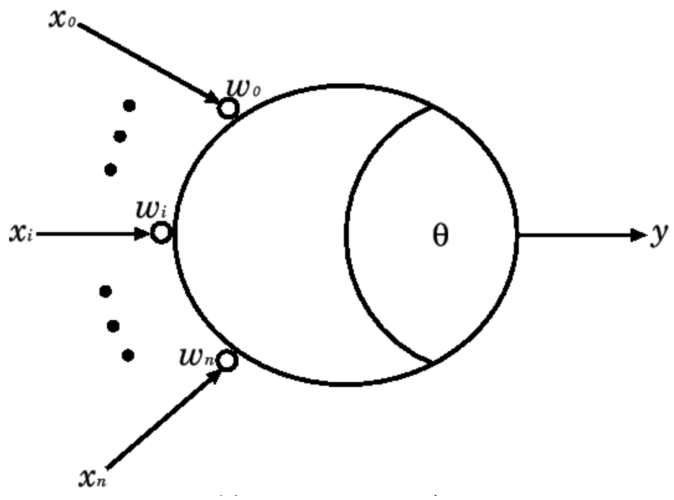

Figure 2.2: Model of Simple Artificial Neuron

ity. Activation Function [11] has the role of giving non-linearity to the artificial neuron model by enhancing or simplifying its mathematical expressions. This section provides a brief explanation of representative activation functions.

2.3.1 Sigmoid Function

Sigmoid function [11] is an S-shape and smooth function. Figure 2.3 shows the Sigmoid function and this equation is represented in Equation 2.3. This function has 2 main characteristics. The one is all output values becomes in the range of 0 to 1. The second is that this equation can be differentiable at any point in this function. However, as you can know from Equation 2.3, the amplitude of function input values must be carefully considered because it mostly becomes 0 or 1 if the input is too large. The gradient of this function is represented in Equation 2.4 and Figure 2.4

$$sigmoid(z) = \frac{1}{1 + e^{-}z} \tag{2.3}$$

$$Sigmoid'(z) = (1 - f(z))f(z) \tag{2.4}$$

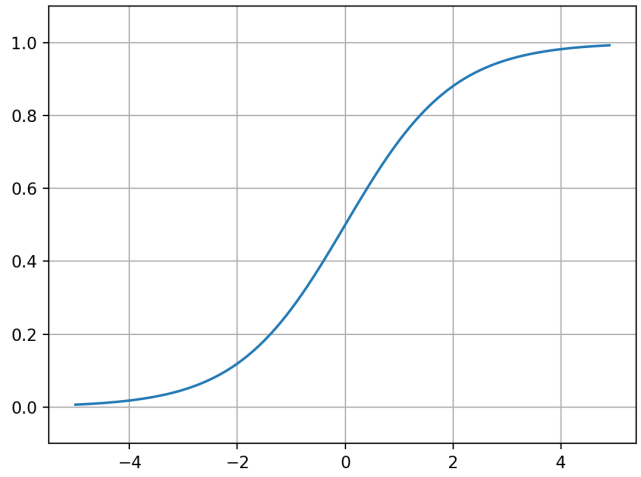

Figure 2.3: Sigmoid

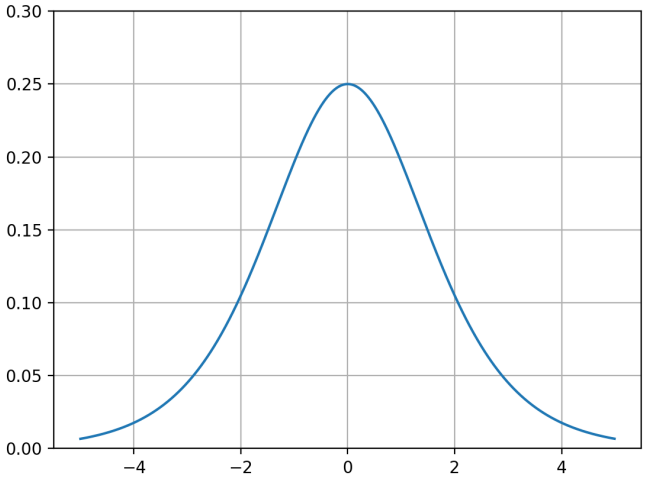

Figure 2.4: The gradient of the sigmoid function

2.3.2 ReLU Function

ReLU [11] function is a straightforward function, as shown in Figure 2.5 and Equation 2.5. This simple function is achieved by just replacing the minus value (under 0) with 0 of the linear input function. This function is not only an elementary and low amount of computation but also it can achieve higher performance in the learning of Neural Network. We dont need to care for the amplitude of function input values, while it must be in Sigmoid function. Because of less complexity compared with Sigmoid function, it can compute faster than Sigmoid function. Furthermore, since the gradient is 1 when the input is larger than 0, the possibility of the gradient disappearance is extremely low. Therefore, this function is the most widely used activation function in many Neural Network system. The gradient of ReLU function is represented in Equation (2.6) and Figure 2.6.

$$ReLU(z) = max(0, z) \tag{2.5}$$

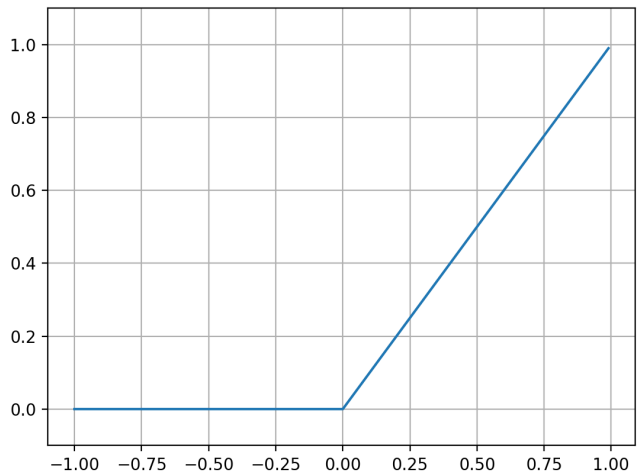

Figure 2.5: ReLU: Rectified Linear Unit

$$ReLU'(z) = \begin{cases} 1 & (x > 0) \\ 0 & (otherwise) \end{cases} \tag{2.6}$$

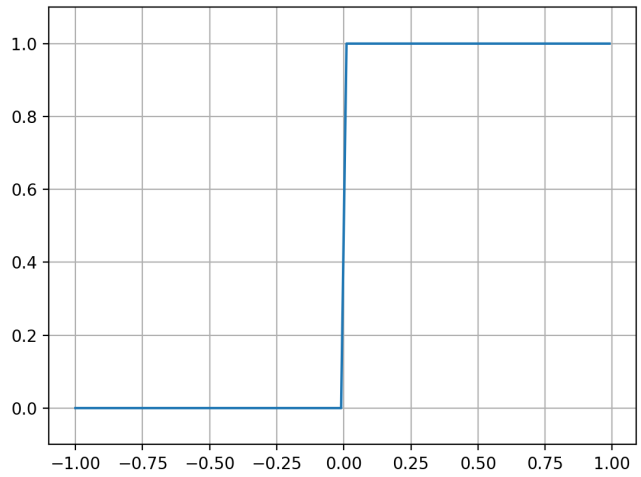

Figure 2.6: The gradient of the ReLU

2.3.3 Softmax Function

Softmax [11] function is a generalization of logistic regression, and it is used for estimating the classification score of each class in Multi-Class Classification [13] problem. Multi-Class Classification is a task of classifying the limited number of classes regarding input contents. Equation 2.7 shows this function equation. This function computes the classification score of each class, and summation of all class scores always becomes 1. The equation of the the loss function in Softmax is represented in Equation 2.8.

$$Softmax(z_k) = \frac{e^{z_k}}{\sum_{i=1}^{n} e^{z_i}} \, (z_i = the\, potential\, of\, one\, output\, neuron) \tag{2.7}$$

$$CrossEntropy(z) = -\sum_{i=1}^{K} \log Softmax(z_i) \tag{2.8}$$

2.4 Artificial Neural Network

Artificial Neural Network is made of neurons such as our human brain. This model turned out that functions such as the human brain, and it could perform human intelli-

gence in my application to some extent. Here we introduce you two Artificial Neural Network paradigms. One is an essential and common Neural Network, which is called Multilayer Perceptron. [14] And the other is an advanced Neural Network which is called Convolutional Neural Network. [15]

2.4.1 Multilayer Perceptron

The basic structure of Multilayer Perceptron is composed of one Input and Output layer and more than one of the hidden layer. Figure 2.7 shows an example of MLP, which performs handwritten numeral recognition. As shown in the figure, signals are processed in the separated layers neuron units, and then the estimated classification result is obtained in the Output layer. Each layer has a specific functionality. Input layer input the given data to the network. This layer is very simple, it does not have input connections, and it just outputs the given data. The hidden layer has both input and output connections. All neurons in this layer have connections of both input and output with other layers neurons, it has the functionality of information processing in its neuron units. Neurons in the hidden layer repeat the following two functions to process signals. It gets the input signals, which is multiplied with weight values from other neurons. And it outputs the values after it adds bias value and applies activation functions. The output layer has the same functionality with hidden layers information process. However, its output values become the estimated class values of its whole network. This Network structure is also known as Feedforward Neural Network. [15]

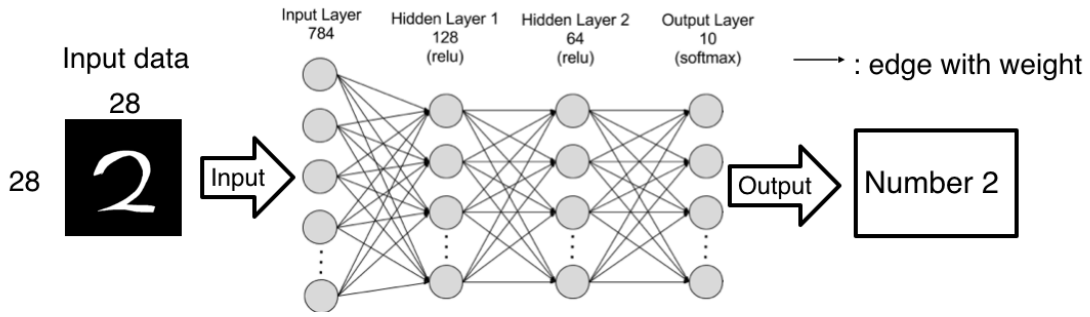

Figure 2.7: An example of Multi-layer perceptron

2.4.2 Stochastic Gradient Descent

Stochastic Gradient Descent (SGD) [14] is the most widely used optimization algorithm in deep learning. This optimization is done by minimizing the loss function, which is calculated in the network. SGD batches part of data from learning data, and use them for learning. In this time, the learning process is conducted to each data from batched data at every iteration.

2.4.3 Convolutional Neural Network

Convolutional Neural Network has 2 different layers, Convolution and Pooling layer, besides Feedforward Neural network. This network is mainly used in image recognition application. Similar to Feedforward Neural Network, this network is also trained by Stochastic Gradient Descent optimization. CNN classifies the class by FFNN with the extracted features from a series of Convolution and Pooling operation. Figure 2.8 shows one of the most popular CNN architecture, which is named Le-Net-5 [16].

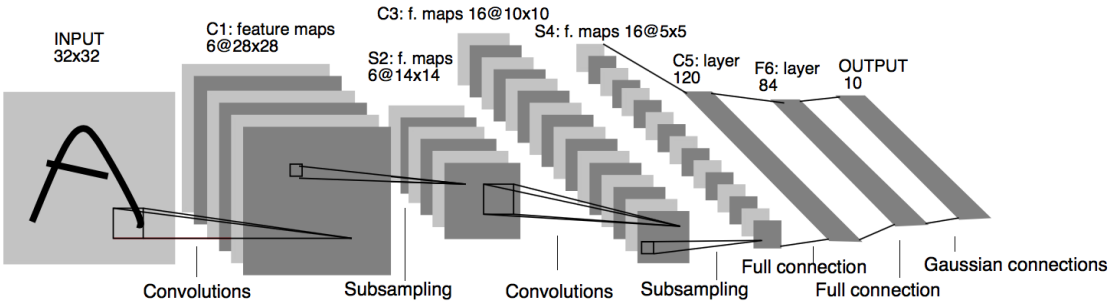

Figure 2.8: An example of CNN Architecture (LeNet-5)

Convolution Layer

The role of this layer is to enhance the similarity by scanning the pre-trained filters which have 2-dimensional space feature. Filters usually have 3x3 or 5x5 window size, and its pre-trained features are obtained by the supervised learning beforehand. The output is obtained from the convolutional operation, product-sum operation of filters and input. This output is called a feature map, and it is passed into the next layers input.

Pooling Layer

Pooling has a role of contraction for extracted features from the Convolution operation. Contraction approach is applied by using a pooling window, which has a specific region. Typical pooling operations are called average pooling or max pooling, which extracts the only average or max values in the pooling window. This function can reduce the dimensionality of the feature map and enhance the robustness of the extracted features. Pooling operations are usually applied after the Convolution operation, and that processing enables the network to extract the high-level features.

Fully-connected Layer

Extracted features form the Convolution, and Pooling layers are put into this layer for classification. This is the completely same structure with MLP. At the output layer, Softmax function is usually used for the multi-class classification task.

2.5 Learning

To achieve high performance, neural network must learn. In contrast to the previous methods such as human-based feature definition, The network can learn patterns only from data. Human only need to set the classification goal and prepare the data which is necessary to achieve the goal. The data is labeled so that network can recognize what is correct or not. Learning is done by providing the data and make the network classify and evaluate its result. Network can know the loss which is obtained by evaluating from the difference between computed value by network and actual correct label. By changing the network parameters so that the loss is reduced, the model can improve its classification accuracy. This process is repeated until its improvement reaches to a certain result.

2.5.1 Learning Data

Data is essential for learning, and this is very important factor. Neural Network can automatically learn certain patterns from a given data to classify things. For example of object recognition, images are given as learning data, and the network learns features

of the object in the images. To achieve high accuracy such as human-level, the designer should prepare enough amount of the data which contains wide variety of features. If data is insufficient or certain biased patterns are alive, it cannot achieve human-level robustness and accuracy.

The goal of learning is to achieve generalization performance. And data is classified to 3 parts, train, test, and validation according to its purpose. Train data is used for learning the model, change the parameters so that it can achieve higher performance. Test data is used for evaluating the generalization performance of the model. The model must be evaluated by unknown data, not train data, for achieving generalization performance. Validation data is used for verifying the validity of its learning. This can be done by comparing the difference on accuracy between the test and validation data. We can detect wrong learning by monitoring difference between test and validation data accuracy. Network can achieve high accuracy by collecting enough amount of data and dividing it properly to give validity on its learning.

Chapter 3

Farm Monitoring System

This chapter introduces the proposed farm monitoring system with a system overview, shown in Figure 3.1 This system composed of one monitoring camera, FPGA that stores recognition system by CNN to detect invasion of target animal, and one speaker to scare away target animal by sounding. We combine this recognition system and a monitoring camera and set this system on the observed farm. Position estimation of the target animal and resizing image is conducted as preprocess before the classification. Position estimation in the captured image from the camera is done by selective-search algorithm. After the position estimation, the suspected region of the image data is resized and put into CNN for identifying the invasion of target animal.

CNN classify the class of the input image from the monitoring camera, and it commands the countermeasure to the speaker to scare away the target animal.

Those are the expected process of this proposed farm monitoring system. However, at this time, we could only implement the CNN part. And we evaluated the performance of the CNN implementation.

3.1 Classification Target Animal

We set swine as the classification target animal in this research. Currently, the primary source of beast damage on a farm situation are Swine, Deer, and Monkeys, especially the damage from Swine is serious. [17] Our approach is to classify the target, whether its swine or scene. We collected the images for learning from some open database, ImageNet [18] and Flickr [19]. Figure 3.2 shows a sample of learning data.

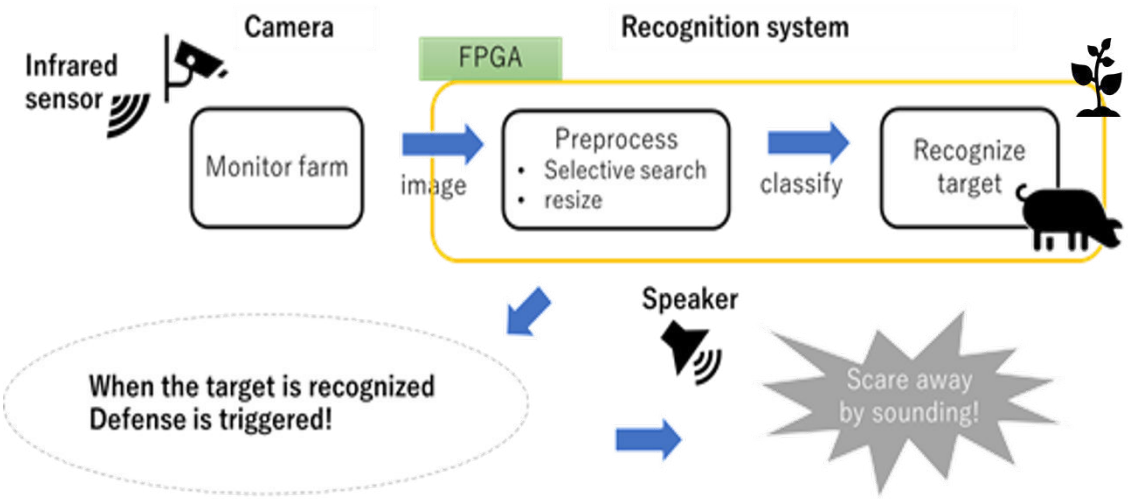

Figure 3.1: System Overview: Proposed Farm Monitoring System.

Figure 3.2: Examples of Dataset

3.2 Object Detection

The monitoring camera is driven by the detection of the objects movement. The infrared sensor is attached to the monitoring camera, and it starts the camera to monitor the farm and send the captured image to the system. The detector that gets the input image from the monitoring camera estimates the object position by using selective search algorithm. [20] Then it sends the suspected region to the classifier to recognize the target. All of those processes are in FPGA and embedded to the monitoring camera.

3.3 About Recognition System

In this chapter, we introduce our model of the recognition system. In this time, our software implementation was done with Python [21]19] and Chainer [22]20] framework. Table 3.1 shows an implementation platform.

Table 3.1: Learning System Environment

Name	Details
OS	Ubuntu 16.04.3 LTS
CPU	Intel(R) Core(TM)i7-4770CPU@ 3.40GHz
GPU	GeForce GTX 1060
Language	Python (ver:2.7.13)
Framework	Chainer (ver: 6.0.0)

3.4 Network Structure

We introduce our implemented network structure. Table **??** shows the network structure. This network composed of three series of simple Convolution and Pooling layers besides 3 layer Fully Connection layers. Input image size is 60x60 RGB image. All Convolution filters are 5x5 size and stride is 0 with no padding. All Pooling is done by 2x2 window size and stride is 2. Fully Connection layer has only one hidden layer, and the neuron number is only 2 in the output layer. We didnt use activation function because this classification task doesnt need the complicated non-linear boundary for its classification task.

Table 3.2: Network Structure

CNN Layer	filterHxW	stride	output ch	output HxW
data	-	-	3	60x60
conv1	5x5	1	4	56x56
pool1	2x2	2	4	28x28
conv2	5x5	1	4	24x24
pool2	2x2	2	4	12x12
conv3	5x5	1	8	8x8
pool3	2x2	2	8	4x4
FC Layer	input	output	-	-
linear1	128	80	-	-
linear2	80	2	-	-

3.5 Learning Dataset

In this research, we collected 1,370 images for learning from ImageNet and Flickr. Table 3.3 shows the distribution of the original data.

Table 3.3: Data Distribution

Class	Train	Test	Valid
Pig	411	137	137
Other	411	137	137

Since the collected images were not enough for high accuracy, we augmented the data number by applying some image conversions. And it is well known that shot of image data easily to cause overfitting problem in its learning process. Applying image conversions like noising, moving, and rotating are used to increase the amount of learning data. [23] We expected this approach can improve the classification accuracy and robustness of object movement. Applied conversion in this research is the following. Move the object position with -4 to +4 in both vertically and horizontally, and flipped horizontally. We randomly combined those two conversions and applied this augmentation to training data. We increased 2 times of original number, and now we have 1,644 images for train data.

3.6 Network Performance

We measured our model of recognition accuracy and network dimensionality. The training was conducted with 300 iterations with 128 batch size. Adam [24] is used as an optimization method. And current recognition accuracy is 98.18 %. Figure 3.3 shows the learning curve.

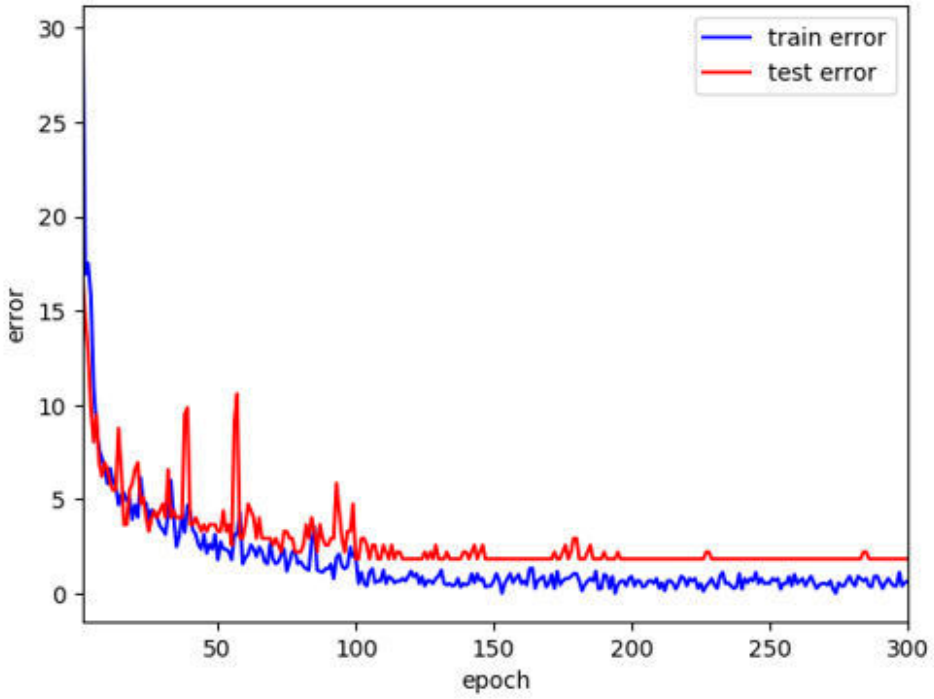

Figure 3.3: System Overview: Proposed Farm Monitoring System.

We calculated the network dimensionality [25], shown in Table 3.4, that indicates the complexity of the network. Equation (3.1) estimates the number of weights.

$$|W| = \sum_{i=1}^{\#layers} (Filter\_ch_L \times FilterWxH_L + 1) \times Filter_{num_L} \times byte\_num \quad (3.1)$$

And the number of arithmetic operations is calculated by Equation (3.2).

$$FLOPS_L = filter\_ch_L \times filter\_num_L \times filterW_L \times filterH_L \times outputW_L \times outputH_L \times 2 \quad (3.2)$$

Table 3.4: Network Dimensionality

Data bytes (Conv)	8	Data bytes (FC)	8			
Conv Layer	filter HxW	stride	output ch	output HxW	Qty of params	Qty of computation
data	-	-	3	60	0	0
conv1	5	1	4	56	2,400	1,881,600
pool1	2	2	4	28	0	12,544
conv2	5	1	4	24	3,200	460,800
pool2	2	2	4	12	0	2,304
conv3	5	1	8	8	6,400	102,400
pool3	2	2	8	4	0	512
total(Conv)	-	-	-	-	12,000	2,460,160
FC Layer	input	output	-	-	Qty of params	Qty of computation
linear1	128	80	-	-	81,920	20,480
linear2	80	2	-	-	1,296	320
total(FC)	-	-	-	-	83,216	20,800
total(Conv+FC)	-	-	-	-	95,216	2,480,960

Chapter 4

Implementation Details

In this thesis, we implemented our CNN design in FPGA (Field Programmable Gate Array) by High-Level Synthesis [26] that can generate hardware description from software description. FPGA is an integrated circuit that can reconfigure any logic circuits by defining with software programs. Users can design a problem specific circuit that is well suited to operate particular executions. With this advantage, we can improve computation speed and reduce power consumption compared with the CPU based system. Furthermore, the device can reconfigure the circuit design as necessary. Therefore the convenience of hardware design was much improved than a conventional hardware design. In this research, we did hardware implementation on Convolution and Pooling parts of CNN.

4.1 Development Environment

For this hardware design, we chose the Xilinx SDSoC [27] platform, that offers an HLS based development environment of FPGA. Zynq [28] series is a FPGA device family that provides users with an SoC platform where we design FPGA and CPU codesign system. We chose ZCU102 [29] for our target device that contains Processing System (PS) and Programmable Logic (PL). PS is a 64-bit quad core ARM Cortex A-53 processor that can reach clock speeds of up to 1 GHz, and PL is a ZU-9EG programmable logic module. Direct Memory Access (DMA) ports support efficient data transferring. In this system, PS reads the input data and parameter files of Convolution filters and Fully-Connection layers neurons that is already trained by the Python implementation.

Figure 4.1 shows ZCU102, and Table 4.1 shows the specification of ZCU102.

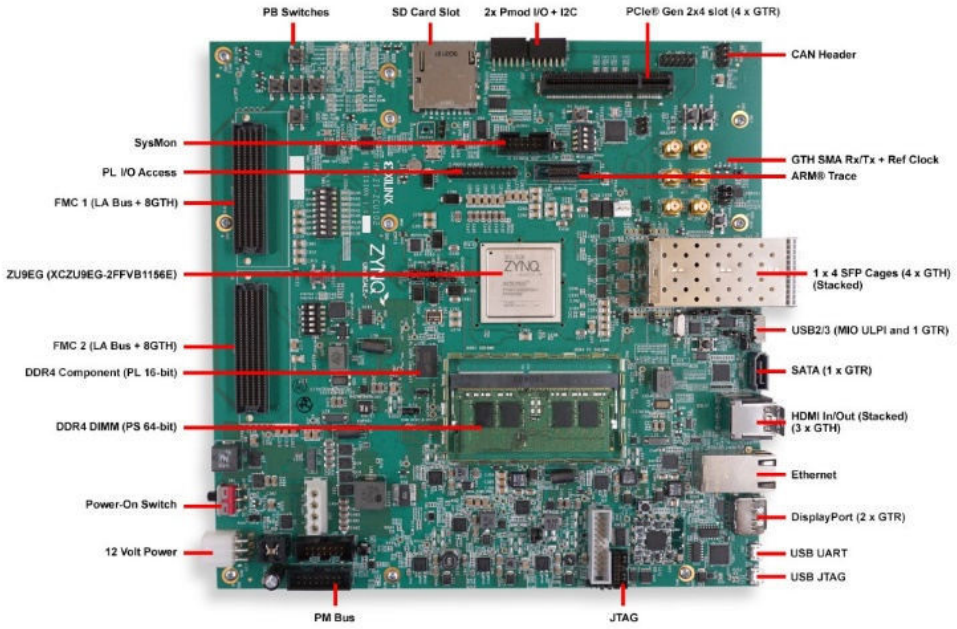

Figure 4.1: Target FPGA Board: ZCU102

Table 4.1: Resouce Avaibility (ZCU102)

Name	Available
BRAM	32 Mbits
LUTs	274,080
Flip Flop	548,160
DSP48E	2,520

4.2 Design Architecture

In this thesis, we implemented a CNN accelerator on a Xilinxs Zynq platform using SDSoC. Figure 4.2 shows the target platform. In our implementation, we distributed computation of CNN to PS and PL. In PS, ARM processor reads image data and weight parameters for convolution filters and fully-connection perceptron. In our implementation, we didnt use bias value in both convolution and fully-connection layers. Also, we didnt apply zero-padding, which adds 0 pixels to input data so that its image size is not

reduced by convolution operations. Image data is sent to on-chip buffers in PL through interconnection called Accelerator Coherency Port (ACP). In PL, 3-layers of convolution modules executes convolution and sab-sampling process. After feature extraction of convolution operation, max-pooling operation is executed as sab-sampling process. After both convolution and sab-sampling process, the result data is sent back to host RAM in PS through ACP interconnection.

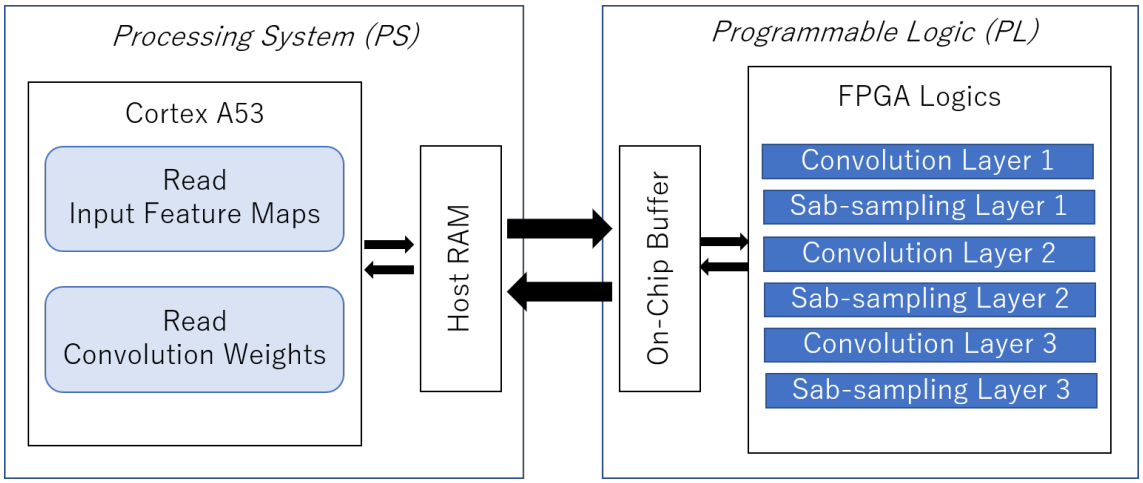

Figure 4.2: Design Architecture

4.3 Design Optimization

In this research, we used two main methods to optimize hardware implementation. One is pipeline design [30] that is generally used to improve the throughput of the computation system by operating Convolution operation in parallel. Another method is replacing the floating-point execution unit with fixed-point [31] execution units.

4.4 Pipeline Design

As we mentioned above, hardware design enables the system to execute the operations concurrently. Even though the clock frequency of FPGA is not as much high as CPU, we can design the specific execution unit that can compute in parallel. It is ideal to implement convolution operation unit in parallel so that it can execute entire output feature map at one time. However, preparing large resources for entire convolution

computation leads to a higher power consumption. It is not preferable for embedded system. Therefore, we adopted the way of calculating only one output feature map in parallel. This means that computation in filter resolution level is executed at one time, while computation in filter depth level is computed in pipeline process. Pipeline design is the way to improve the throughput of whole computation. To implement this module, we used UNROLL and PIPELINE pragma which Vivado HLS platform offers. Figure 4.3 shows a pseudo code of our implementation. With UNROLL pragma, SDSoC generates unrolled loop program which is used to make parallel instantiations for parallel execution. This pragma creates multiple hardware logic blocks corresponding to the loop iteration counter. PIPELINE pragma is used for improving throughput by executing loop iterations in a pipelined way. With PIPELINE pragma, each iteration is divided into small stages and a pipeline logic that meets the timing constraints is generated.

```
//Load input feature map, convolution kernel to buffer
for(b=0; b<B; b++){// Input image
 for(o=0; o<O; o++){// Output feature map depth
  for(ir=0,ic=0;ir<IR,ic<IL;ir++,ic++){// Input feature map position
   sum = 0;
   for(kd = 0; kd < KD; ++kd){// Kernel depth
    #pragma HLS PIPELINE
    for(kr = 0; kr < KR; ++kr){// Kernel row
     #pragma HLS UNROLL
     for(kc = 0; kc < KC; ++kc){// Kernel column
      #pragma HLS UNROLL
      sum += input_data[b][kd][ir+kr][ic+kc] * kernel[o][kd][kr][kc];
     }
    }
   }
   output[b][o][ir][ic] = Activation(sum + bias[o]);
}}}
```

Figure 4.3: Pseudo Code of Designed CNN accelerator

4.5 Fixed-Point Operation

In theory, a fixed-point execution unit can compute arithmetic operations with fewer clock cycles compared with floating-point execution units. It is reported that inference of deep learning model doesnt need high precision like 64 bits. [32] We successfully reduced the bit width from the original 64 bits to 16 bits without no classification accuracy degrade and 8 bits with slight accuracy degrade. And it could dynamically reduce the use of hardware resources. In this research, we used the Xilinx Vivado HLS library, that is called ap_fixed [33] in our fixed-point implementation. Here we introduce how we implemented fixed-point calculation operation unit.

4.5.1 Fixed-Point Calculation Implementation

As an importance in implementing fixed-point calculation operation unit, we need to devise the way to avoid a large rounding errors. Since fixed-point calculation unit does not have an exact position of a decimal point. Designers must care about how to control its position so that mantissa and exponent parts are correctly treated in calculation. In addition or subtraction operation, the decimal point position of input values must be set with the same position before calculation. In this time, bit position may become different because of decimal position adjustment. Fractional part will be directly calculated while integer part will be filled with zero value or most significant bit (MSB) according to data type, unsigned or signed. The result will become larger bit length compared with the original input value. In multiplication operation, the position of decimal point is not cared and calculated as it is. The bit length of result value and the position of decimal point will changes with according to input values. In most fixed-point operation, the position of decimal point is cared by designer on each operation. The bit length of the intermediate result register is constant while the position of a decimal point moves with arithmetic operations. Therefore, the execution unit needs to round-off some fractional part of bits so that the result value can fit into the length of intermediate register. It is obvious that the rounding error accumulates with arithmetic operations in fixed-point implementation. We must care for this problem, because CNN contains large amount of product-sum operation in its computation. In this thesis, we adopted the way to regulate the value range so that its decimal point position does not change and leads to a large

rounding error.

Value Range Regulation

In theory, if [1,1] and [1,1], [1,1]. We regulated the computation rage of both input and parameter values by dividing with the probable absolute maximum value. By regulating the computation value range, only 1 bit for inter part, we can fix the position of decimal point on each arithmetic operations. Since we know already the parameter values, we can estimate the probable maximum values by using dataset. We used our training dataset and input them to our CNN, then we observed the both minimum or maximum values. Table 4.2 shows the observed absolute values in our design. Dividing parameter values by absolute maximum values, the value range is limited to range [-1:1]. However, there is some risks of overflow, since the absolute maximum value is calculated by dataset, we scaled up the absolute maximum values with 15%. And our design processes overflow to round off to 0 if the overflow is detected. This method worked well on our experiments, and we show its results on chapter 5.

Table 4.2: Absolute maximum values on each layers

con1	2.9133616
con2	2.53624909
con3	3.797857
linear 1	3.1460109
linear 2	17.70926402

Chapter 5

Evaluation Result and Analysis

In this paper, we proposed the farm monitoring system and implemented a classification system in CNN architecture on an embedded SoC platform. Same with the implementation, we used ZCU102 for this evaluation. We measured the performance in terms of computation speed and power consumption by comparing both software and hardware implementation. We used 1,000 images from our dataset for this measurement. Table 5.1 shows the specification of the implementation. SW_{PS} is the metrics which only use PS side for computation. HW_O is a metrics of PS / PL codesing with hardware optimizations.

Table 5.1: Specification of the Proposed Implementation

Metrics	SW_{PS}	HW_O
Core	Cortex A53	Cortex A53 / FPGA
Clock (GHz)	1.334	1.334 (PS) / 0.1 (FPGA)
Num of Core	4	4 (PS)

5.1 Performance and Energy Consumption

Table 5.2 shows the performance comparison results of execution time and power consumption between software implementation (SW_{PS}) and optimized hardware implementations that include three sub-metrics of 64 bits floating-point precision (HWfloat64), 16 bits fixed-point (HWfixed16), and 8 bits fixed-point (HWfixed8). Power consumption and energy dissipation were measured by Vivado power analysis tool. HW_O shows

better performance both on computation speed and energy consumption wit no or slight accuracy degrade. HWfixed16 achieved 3.67 times faster computation speed improvement compared with SW_{PS}. Though the power consumption of HWfixed16 increased because of additional hardware use, its energy consumption was saved 3.19 times lower than SW_{PS}. HWfixed8 achieve performance improvement on computational speed and energy consumption with slight accuracy degrade. HWfixed8 achieved 4.66 times faster computation speed improvement and 4.06 times lower energy consumption. The accuracy of HWfixed8 is 94.53%, 3.65% degrade. Further quantization requires us to complimental approach for maintaing accuracy.

Table 5.2: Performance Comparison

Metrics	SW_{PS}	HWfloat64	HWfixed16	HWfixed8
Accuracy (%)	98.18	98.18	98.18	94.53
Total Time (s)	43.76	26.49	11.98	9.81
Convolution Time (s)	43.36	26.09	7.39	7.37
Power (W)	3.49	4.21	4.01	4.00
Energy (J)	151.92	97.42	47.60	37.4

5.2 Hardware Utilization

In our implementation, we applied two optimization methods. Pipeline design on one feature map computation in parallel needed a large amount of resource usage. And fixed-point implementation required a large amount of DSP usage for its execution unit. Table 5.3 shows the resource utilization of PL design.

Table 5.3: PL resource utilization

Metrics	DSP (%)	BRAM (%)	FF (%)	LUT (%)
HWfloat64	242 (9)	270 (14)	33285 (6)	31764 (11)
HWfixed16	233 (9)	50 (2)	15906 (2)	24931 (9)
HWfixed8	230 (9)	34 (1)	15187 (2)	24511 (8)

Chapter 6

Conclusion and Future Work

In this thesis, we could successfully developed a power efficient object recognition system by applying hardware optimizations. This system uses a Convolution Neural Network which can recognize the target animals invasion. We optimized the design for embedded system in terms of computation speed and energy efficiency. The evaluation results showed the efficiency of our design by comparison between software and hardware implementation. The two main optimizations applied in this research were pipeline design on feature map computation and reduction of unnecessary bit width used on filter computation. Computation speed and energy consumption in 16 bit fixed-point implementation were improved 3.67 times and 3.19 times compared with software implementation respectively. Computation speed and energy consumption in 8 bit fixed-point implementation were improved 4.66 times and 4.06 times compared with software implementation respectively. As a future work, we are going to design a farm monitoring system, incorporating object detection system, camera interfaces. We will conduct experiments on the platform to explore how much computation speed and recognition accuracy are necessary for successful protection.

References

[1] Y. O. The H. Vu and A. B. Abdallah, "Comprehensive analytical model and k-means based multicast routing algorithm and architecture for 3d-noc of spiking neurons," *ACM Journal on Emerging Technologies in Computing Systems, Special Issue on Hardware and Algorithms for Learning On-a-chip for Energy-Constrained On-Chip Machine Learning*, 2019.

[2] O. M. I. The H. Vu and A. B. Abdallah, "Fault-tolerant spike routing algorithm and architecture for three dimensional noc-based neuromorphic systems," *IEEE Access*, 2019.

[3] Y. O. The H. Vu and A. B. Abdallah, "Analytical performance assessment and high-throughput low-latency spike routing algorithm for spiking neural network systems," *The Journal of Supercomputing*, no. doi:10.1007/s11227-019-02792-y, pp. pp 1–31, 2019.

[4] A. B. A. The H. Vu, "Low-latency k-means based multicast routing algorithm and architecture for three dimensional spiking neuromorphic chips," *Big Data and Smart Computing (BigComp) 2019 IEEE International Conference*, pp. pp. 1–8, Feb. 2019.

[5] Y. M. The H. Vu and A. B. Abdallah, "Graceful fault-tolerant on-chip spike routing algorithm for mesh-based spiking neural networks," *Proceedings of the International Conference on Intelligent Autonomous Systems (ICoIAS2019)*, Feb. 2019.

[6] T. H. Vu and A. B. Abdallah, "A low-latency tree-based multicast spike routing for scalable multicore neuromorphic chips," *ACM 5th International Conference of Computing for Engineering and Sciences*, July 2019.

[7] Y. O. A. B. A. The H. Vu, Ryunosuke Murakami, "Efficient optimization and hardware acceleration of cnns towards the design of a scalable neuro-inspired architecture in hardware," *IEEE International Conference on Big Data and Smart Computing (BigComp)*, 2018.

[8] Adaptive systems laboratory. [Online]. Available: https://adaptive.u-aizu.ac.jp/

[9] M. T. Hagan, H. B. Demuth, M. H. Beale, and O. De Jesús, *Neural network design*. Pws Pub. Boston, 1996, vol. 20.

[10] S. Herculano-Houzel, "The human brain in numbers: a linearly scaled-up primate brain," *Frontiers in human neuroscience*, vol. 3, p. 31, 2009.

[11] J. Donahue, Y. Jia, O. Vinyals, J. Hoffman, N. Zhang, E. Tzeng, and T. Darrell, "Decaf: A deep convolutional activation feature for generic visual recognition," in *International conference on machine learning*, 2014, pp. 647–655.

[12] Wikipedia, "Artificial neural network — Wikipedia, the free encyclopedia," http://en.wikipedia.org/w/index.php?title=Artificial%20neural%20network&oldid=901207463, 2019, [Online; accessed 18-June-2019].

[13] J. Zhu, S. Liao, Z. Lei, and S. Z. Li, "Multi-label convolutional neural network based pedestrian attribute classification," *Image and Vision Computing*, vol. 58, pp. 224–229, 2017.

[14] Y. LeCun, Y. Bengio, and G. Hinton, "Deep learning," *nature*, vol. 521, no. 7553, p. 436, 2015.

[15] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *Advances in neural information processing systems*, 2012, pp. 1097–1105.

[16] R. Al-Jawfi, "Handwriting arabic character recognition lenet using neural network." *Int. Arab J. Inf. Technol.*, vol. 6, no. 3, pp. 304–309, 2009.

[17] F. Ministry of Agriculture and Fisheries. (2016) Crop damage by wild birds and animals. [Online]. Available: http://www.maff.go.jp/j/nousin/saigai/pdf/pdf/report_all_2.pdf

[18] J. Deng, W. Dong, R. Socher, L.-J. Li, K. Li, and L. Fei-Fei, "ImageNet: A Large-Scale Hierarchical Image Database," in *CVPR09*, 2009.

[19] flickr. [Online]. Available: https://www.flickr.com/

[20] J. R. Uijlings, K. E. Van De Sande, T. Gevers, and A. W. Smeulders, "Selective search for object recognition," *International journal of computer vision*, vol. 104, no. 2, pp. 154–171, 2013.

[21] Python. [Online]. Available: https://www.python.org/

[22] Chainer. [Online]. Available: https://chainer.org/

[23] P. Y. Simard, D. Steinkraus, J. C. Platt *et al.*, "Best practices for convolutional neural networks applied to visual document analysis." in *Icdar*, vol. 3, no. 2003, 2003.

[24] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[25] M. Motamedi, P. Gysel, V. Akella, and S. Ghiasi, "Design space exploration of fpga-based deep convolutional neural networks," in *2016 21st Asia and South Pacific Design Automation Conference (ASP-DAC)*. IEEE, 2016, pp. 575–580.

[26] Vivado high-level synthesis. [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[27] Sdsoc development environment. [Online]. Available: https://www.xilinx.com/products/design-tools/software-zone/sdsoc.html

[28] Zynq-7000 xilinx. [Online]. Available: https://japan.xilinx.com/products/boards-and-kits/device-family/nav-zynq-7000.html

[29] Zynq ultrascale+ mpsoc zcu102 evaluation kit. [Online]. Available: https://japan.xilinx.com/support/documentation-navigation/design-hubs/dh0048-zcu102-evaluation-kit-hub.html

[30] S.-S. Park, K.-B. Park, and K.-S. Chung, "Implementation of a cnn accelerator on an embedded soc platform using sdsoc," in *Proceedings of the 2nd International Conference on Digital Signal Processing*. ACM, 2018, pp. 161–165.

[31] R. A. Solovyev, A. A. Kalinin, A. G. Kustov, D. V. Telpukhov, and V. S. Ruhlov, "Fpga implementation of convolutional neural networks with fixed-point calculations," *arXiv preprint arXiv:1808.09945*, 2018.

[32] N. Wang, J. Choi, D. Brand, C.-Y. Chen, and K. Gopalakrishnan, "Training deep neural networks with 8-bit floating point numbers," in *Advances in neural information processing systems*, 2018, pp. 7675–7684.

[33] Vivado design suite user guide. [Online]. Available: https://www.xilinx.com/support/documentation/sw_manuals/xilinx2018_1/ug910-vivado-getting-started.pdf

[34] S. Ren, K. He, R. Girshick, and J. Sun, "Faster r-cnn: Towards real-time object detection with region proposal networks," in *Advances in neural information processing systems*, 2015, pp. 91–99.

[35] G. K. Venayagamoorthy, V. Moonasar, and K. Sandrasegaran, "Voice recognition using neural networks," in *Proceedings of the 1998 South African Symposium on Communications and Signal Processing-COMSIG'98 (Cat. No. 98EX214)*. IEEE, 1998, pp. 29–32.

[36] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio, "Learning phrase representations using rnn encoder-decoder for statistical machine translation," *arXiv preprint arXiv:1406.1078*, 2014.

[37] J. Deng, A. Berg, S. Satheesh, H. Su, A. Khosla, and L. Fei-Fei, "Ilsvrc-2012, 2012," *URL http://www. image-net. org/challenges/LSVRC*, 2012.

[38] S. Mittal, "A survey of fpga-based accelerators for convolutional neural networks," *Neural computing and applications*, pp. 1–31, 2018.

Appendix

cnn.py

```
1  #include <stdio.h>
2  #include <stdint.h>
3  #include <string.h>
4  #include <stdlib.h>
5  #include <math.h>
6  #include "cnn.hpp"
7  #include "common.hpp"
8
9  /*
10    Convolution()
11  */
12 int Convolution0
13 (
14   myFixed *filter,
15   int filter_size,
16   myFixed *input_data,
17   int input_width,
18   int input_height,
19   int input_depth,
20   myFixed *conv_out
21 )
22 {
23   int x, y;
24   int conv_width  = input_width  - 2 * (filter_size / 2);
25   int conv_height = input_height - 2 * (filter_size / 2);
26   int d;
27   fullBit sum[3];
28   int offset_i, offset_f;
29
30   for(y = 0; y < conv_height; ++y){
31     for(x = 0; x < conv_width; ++x){
32
33       for(d = 0; d < input_depth; ++d){
34         #pragma HLS PIPELINE
35         offset_i = (input_width * input_height * d);
36         offset_f = (filter_size * filter_size * d);
37      sum[d]=input_data[offset_i+((y+0)*input_width)+(x+0)]*
           filter[offset_f+(0*filter_size)+0]
38            +input_data[offset_i+((y+0)*input_width)+(x+1)]*
              filter[offset_f+(0*filter_size)+1]
39            +input_data[offset_i+((y+0)*input_width)+(x+2)]*
              filter[offset_f+(0*filter_size)+2]
40            +input_data[offset_i+((y+0)*input_width)+(x+3)]*
              filter[offset_f+(0*filter_size)+3]
41            +input_data[offset_i+((y+0)*input_width)+(x+4)]*
              filter[offset_f+(0*filter_size)+4]
```

```
42              +input_data[offset_i+((y+1)*input_width)+(x+0)]*
                   filter[offset_f+(1*filter_size)+0]
43              +input_data[offset_i+((y+1)*input_width)+(x+1)]*
                   filter[offset_f+(1*filter_size)+1]
44              +input_data[offset_i+((y+1)*input_width)+(x+2)]*
                   filter[offset_f+(1*filter_size)+2]
45              +input_data[offset_i+((y+1)*input_width)+(x+3)]*
                   filter[offset_f+(1*filter_size)+3]
46              +input_data[offset_i+((y+1)*input_width)+(x+4)]*
                   filter[offset_f+(1*filter_size)+4]
47              +input_data[offset_i+((y+2)*input_width)+(x+0)]*
                   filter[offset_f+(2*filter_size)+0]
48              +input_data[offset_i+((y+2)*input_width)+(x+1)]*
                   filter[offset_f+(2*filter_size)+1]
49              +input_data[offset_i+((y+2)*input_width)+(x+2)]*
                   filter[offset_f+(2*filter_size)+2]
50              +input_data[offset_i+((y+2)*input_width)+(x+3)]*
                   filter[offset_f+(2*filter_size)+3]
51              +input_data[offset_i+((y+2)*input_width)+(x+4)]*
                   filter[offset_f+(2*filter_size)+4]
52              +input_data[offset_i+((y+3)*input_width)+(x+0)]*
                   filter[offset_f+(3*filter_size)+0]
53              +input_data[offset_i+((y+3)*input_width)+(x+1)]*
                   filter[offset_f+(3*filter_size)+1]
54              +input_data[offset_i+((y+3)*input_width)+(x+2)]*
                   filter[offset_f+(3*filter_size)+2]
55              +input_data[offset_i+((y+3)*input_width)+(x+3)]*
                   filter[offset_f+(3*filter_size)+3]
56              +input_data[offset_i+((y+3)*input_width)+(x+4)]*
                   filter[offset_f+(3*filter_size)+4]
57              +input_data[offset_i+((y+4)*input_width)+(x+0)]*
                   filter[offset_f+(4*filter_size)+0]
58              +input_data[offset_i+((y+4)*input_width)+(x+1)]*
                   filter[offset_f+(4*filter_size)+1]
59              +input_data[offset_i+((y+4)*input_width)+(x+2)]*
                   filter[offset_f+(4*filter_size)+2]
60              +input_data[offset_i+((y+4)*input_width)+(x+3)]*
                   filter[offset_f+(4*filter_size)+3]
61              +input_data[offset_i+((y+4)*input_width)+(x+4)]*
                   filter[offset_f+(4*filter_size)+4];
62          }
63          conv_out[y * conv_width + x] = sum[0]+sum[1]+sum[2];
64        }
65    }
66
67    return 0;
68  }
69
70  int Convolution1
71  (
72    myFixed *filter,
73    int filter_size,
74    myFixed *input_data,
75    int input_width,
76    int input_height,
77    int input_depth,
78    myFixed *conv_out
79  )
80  {
81    int x, y;
82    int conv_width  = input_width  - 2 * (filter_size / 2);
83    int conv_height = input_height - 2 * (filter_size / 2);
```

```
84    int d;
85    fullBit sum[4];
86    int offset_i, offset_f;
87
88    for(y = 0; y < conv_height; ++y){
89      for(x = 0; x < conv_width; ++x){
90
91
92         for(d = 0; d < input_depth; ++d){
93           #pragma HLS PIPELINE
94           offset_i = (input_width * input_height * d);
95           offset_f = (filter_size * filter_size * d);
96
97        sum[d]=input_data[offset_i+((y+0)*input_width)+(x+0)]*
             filter[offset_f+(0*filter_size)+0]
98              +input_data[offset_i+((y+0)*input_width)+(x+1)]*
                 filter[offset_f+(0*filter_size)+1]
99              +input_data[offset_i+((y+0)*input_width)+(x+2)]*
                 filter[offset_f+(0*filter_size)+2]
100             +input_data[offset_i+((y+0)*input_width)+(x+3)]*
                 filter[offset_f+(0*filter_size)+3]
101             +input_data[offset_i+((y+0)*input_width)+(x+4)]*
                 filter[offset_f+(0*filter_size)+4]
102             +input_data[offset_i+((y+1)*input_width)+(x+0)]*
                 filter[offset_f+(1*filter_size)+0]
103             +input_data[offset_i+((y+1)*input_width)+(x+1)]*
                 filter[offset_f+(1*filter_size)+1]
104             +input_data[offset_i+((y+1)*input_width)+(x+2)]*
                 filter[offset_f+(1*filter_size)+2]
105             +input_data[offset_i+((y+1)*input_width)+(x+3)]*
                 filter[offset_f+(1*filter_size)+3]
106             +input_data[offset_i+((y+1)*input_width)+(x+4)]*
                 filter[offset_f+(1*filter_size)+4]
107             +input_data[offset_i+((y+2)*input_width)+(x+0)]*
                 filter[offset_f+(2*filter_size)+0]
108             +input_data[offset_i+((y+2)*input_width)+(x+1)]*
                 filter[offset_f+(2*filter_size)+1]
109             +input_data[offset_i+((y+2)*input_width)+(x+2)]*
                 filter[offset_f+(2*filter_size)+2]
110             +input_data[offset_i+((y+2)*input_width)+(x+3)]*
                 filter[offset_f+(2*filter_size)+3]
111             +input_data[offset_i+((y+2)*input_width)+(x+4)]*
                 filter[offset_f+(2*filter_size)+4]
112             +input_data[offset_i+((y+3)*input_width)+(x+0)]*
                 filter[offset_f+(3*filter_size)+0]
113             +input_data[offset_i+((y+3)*input_width)+(x+1)]*
                 filter[offset_f+(3*filter_size)+1]
114             +input_data[offset_i+((y+3)*input_width)+(x+2)]*
                 filter[offset_f+(3*filter_size)+2]
115             +input_data[offset_i+((y+3)*input_width)+(x+3)]*
                 filter[offset_f+(3*filter_size)+3]
116             +input_data[offset_i+((y+3)*input_width)+(x+4)]*
                 filter[offset_f+(3*filter_size)+4]
117             +input_data[offset_i+((y+4)*input_width)+(x+0)]*
                 filter[offset_f+(4*filter_size)+0]
118             +input_data[offset_i+((y+4)*input_width)+(x+1)]*
                 filter[offset_f+(4*filter_size)+1]
119             +input_data[offset_i+((y+4)*input_width)+(x+2)]*
                 filter[offset_f+(4*filter_size)+2]
120             +input_data[offset_i+((y+4)*input_width)+(x+3)]*
                 filter[offset_f+(4*filter_size)+3]
```

```
121             +input_data[offset_i+((y+4)*input_width)+(x+4)]*
                   filter[offset_f+(4*filter_size)+4];
122         }
123         conv_out[y * conv_width + x] = sum[0]+sum[1]+sum[2]+sum
               [3];
124       }
125     }
126
127     return 0;
128   }
129
130   /*
131      Pooling()
132    */
133   void Pooling(
134     myFixed *conv_out,
135     int conv_width,
136     int conv_height,
137     myFixed *pool_out,
138     int pool_size,
139     int pool_stride
140   )
141   {
142     int x = 0;
143     int y = 0;
144     int n = 0;
145     int pool_width  = conv_width / pool_stride;
146     int pool_height = conv_height / pool_stride;
147     myFixed max_tmp[2];
148
149     for(y = 0; y < pool_height; ++y){
150       for(x = 0; x < pool_width; ++x){
151
152         for(n = 0; n < pool_size; ++n){
153           if(conv_out[(y * conv_width * pool_stride) + (n *
                 conv_width ) + (x * pool_stride)] >
154             conv_out[(y * conv_width * pool_stride) + (n *
                   conv_width ) + (x * pool_stride) + 1])
155               max_tmp[n] = conv_out[(y * conv_width *
                     pool_stride) + (n * conv_width) +(x *
                     pool_stride)];
156           else max_tmp[n] = conv_out[(y * conv_width *
                 pool_stride) + (n * conv_width) + (x * pool_stride
                 + 1)];
157         }
158
159         if(max_tmp[0] > max_tmp[1]) pool_out[(y * pool_width) +
                 x] = max_tmp[0];
160         else pool_out[(y * pool_width) + x] = max_tmp[1];
161
162       }
163     }
164   }
165
166   /*
167      CNNLayer()
168    */
169   int CNNLayer0(
170     myFixed  *filter,
171     int filter_num,
172     int filter_size,
173     myFixed  *input_data,
```

```
174    int input_width,
175    int input_height,
176    int input_depth,
177    myFixed  *conv_out,
178    int conv_width,
179    int conv_height,
180    myFixed  *pool_out,
181    int pool_width,
182    int pool_height,
183    int pool_size,
184    int pool_stride
185  )
186  {
187
188    int i;
189    int offset;
190
191    for(i = 0; i < filter_num; ++i){
192
193
194      Convolution0(
195        &filter[filter_size*filter_size*input_depth*i],
196        filter_size,
197        input_data,
198        input_width,
199        input_height,
200        input_depth,
201        conv_out
202      );
203
204      offset = pool_width * pool_height * i;
205
206      Pooling(
207        conv_out,
208        conv_width,
209        conv_height,
210        &pool_out[offset],
211        pool_size,
212        pool_stride
213      );
214    }
215    return 0;
216  }
217
218   int CNNLayer1(
219    myFixed  *filter,
220    int filter_num,
221    int filter_size,
222    myFixed  *input_data,
223    int input_width,
224    int input_height,
225    int input_depth,
226    myFixed  *conv_out,
227    int conv_width,
228    int conv_height,
229    myFixed  *pool_out,
230    int pool_width,
231    int pool_height,
232    int pool_size,
233    int pool_stride
234  )
235  {
```

```
236   int i;
237   int offset;
238
239   for(i = 0; i < filter_num; ++i){
240     Convolution1(
241       &filter[filter_size*filter_size*input_depth*i],
242       filter_size,
243       input_data,
244       input_width,
245       input_height,
246       input_depth,
247       conv_out
248     );
249     offset = pool_width * pool_height * i;
250     Pooling(
251       conv_out,
252       conv_width,
253       conv_height,
254       &pool_out[offset],
255       pool_size,
256       pool_stride
257     );
258   }
259   return 0;
260 }
261
262 /*
263    InitFilter()
264     Load filter paramter.
265 */
266 void InitFilter_1(
267   myFixed  *filter,
268   int filter_size,
269   int filter_channel,
270   int filter_num
271 )
272 {
273   FILE *fp;
274   if((fp=fopen("/mnt/params/weight_conv1.dat", "r+")) != NULL){
275     fread(filter, sizeof(myFixed) * filter_size * filter_size *
          filter_channel * filter_num, 1, fp);
276   }
277   fclose(fp);
278 }
279
280 void InitFilter_2(
281   myFixed *filter,
282   int filter_size,
283   int filter_channel,
284   int filter_num
285 )
286 {
287   FILE *fp;
288   if((fp=fopen("/mnt/params/weight_conv2.dat", "r+")) != NULL){
289     fread(filter, sizeof(myFixed) * filter_size * filter_size *
          filter_channel * filter_num, 1, fp);
290   }
291   fclose(fp);
292
293 }
294
295 void InitFilter_3(
```

```
296   myFixed *filter,
297   int filter_size,
298   int filter_channel,
299   int filter_num
300 )
301 {
302   FILE *fp;
303   if((fp=fopen("/mnt/params/weight_conv3.dat", "r+")) != NULL){
304     fread(filter, sizeof(myFixed) * filter_size * filter_size *
          filter_channel * filter_num, 1, fp);
305   }
306   fclose(fp);
307
308 }
309
310 #pragma SDS data access_pattern(filter0:SEQUENTIAL)
311 #pragma SDS data access_pattern(input_data0:SEQUENTIAL)
312 #pragma SDS data access_pattern(conv_out0:SEQUENTIAL)
313 #pragma SDS data access_pattern(pool_out0:SEQUENTIAL)
314 #pragma SDS data zero_copy(filter0[0:5*5*3*4-1])
315 #pragma SDS data zero_copy(input_data0[0:60*60*3-1])
316 #pragma SDS data zero_copy(conv_out0[0:56*56*4-1])
317 #pragma SDS data zero_copy(pool_out0[0:28*28*4-1])
318 #pragma SDS data mem_attribute(filter0:PHYSICAL_CONTIGUOUS)
319 #pragma SDS data mem_attribute(input_data0:PHYSICAL_CONTIGUOUS)
320 #pragma SDS data mem_attribute(conv_out0:PHYSICAL_CONTIGUOUS)
321 #pragma SDS data mem_attribute(pool_out0:PHYSICAL_CONTIGUOUS)
322
323 #pragma SDS data access_pattern(filter1:SEQUENTIAL)
324 #pragma SDS data access_pattern(input_data1:SEQUENTIAL)
325 #pragma SDS data access_pattern(conv_out1:SEQUENTIAL)
326 #pragma SDS data access_pattern(pool_out1:SEQUENTIAL)
327 #pragma SDS data zero_copy(filter1[0:5*5*4*4-1])
328 #pragma SDS data zero_copy(input_data1[0:28*28*4-1])
329 #pragma SDS data zero_copy(conv_out1[0:24*24*4-1])
330 #pragma SDS data zero_copy(pool_out1[0:12*12*4-1])
331 #pragma SDS data mem_attribute(filter1:PHYSICAL_CONTIGUOUS)
332 #pragma SDS data mem_attribute(input_data1:PHYSICAL_CONTIGUOUS)
333 #pragma SDS data mem_attribute(conv_out1:PHYSICAL_CONTIGUOUS)
334 #pragma SDS data mem_attribute(pool_out1:PHYSICAL_CONTIGUOUS)
335
336 #pragma SDS data access_pattern(filter2:SEQUENTIAL)
337 #pragma SDS data access_pattern(input_data2:SEQUENTIAL)
338 #pragma SDS data access_pattern(conv_out2:SEQUENTIAL)
339 #pragma SDS data access_pattern(pool_out2:SEQUENTIAL)
340 #pragma SDS data zero_copy(filter2[0:5*5*4*8-1])
341 #pragma SDS data zero_copy(input_data2[0:12*12*4-1])
342 #pragma SDS data zero_copy(conv_out2[0:8*8*8-1])
343 #pragma SDS data zero_copy(pool_out2[0:4*4*8-1])
344 #pragma SDS data mem_attribute(filter2:PHYSICAL_CONTIGUOUS)
345 #pragma SDS data mem_attribute(input_data2:PHYSICAL_CONTIGUOUS)
346 #pragma SDS data mem_attribute(conv_out2:PHYSICAL_CONTIGUOUS)
347 #pragma SDS data mem_attribute(pool_out2:PHYSICAL_CONTIGUOUS)
348
349 void execCNN(
350   myFixed *filter0,
351   int filter_num0,     int filter_size0,    myFixed *input_data0
      ,
352   int input_width0,    int input_height0,   int input_depth0,
353   myFixed *conv_out0,   int conv_width0,     int conv_height0,
354   myFixed *pool_out0,   int pool_width0,     int pool_height0,
355   int pool_size0, int pool_stride0,
```

```
356
357   myFixed *filter1,
358   int filter_num1,     int filter_size1,    myFixed *input_data1
        ,
359   int input_width1,    int input_height1,   int input_depth1,
360   myFixed *conv_out1,   int conv_width1,     int conv_height1,
361   myFixed *pool_out1,   int pool_width1,     int pool_height1,
362   int pool_size1, int pool_stride1,
363
364   myFixed *filter2,
365   int filter_num2,     int filter_size2,    myFixed *input_data2
        ,
366   int input_width2,    int input_height2,   int input_depth2,
367   myFixed *conv_out2,   int conv_width2,     int conv_height2,
368   myFixed *pool_out2,   int pool_width2,     int pool_height2,
369   int pool_size2, int pool_stride2
370
371 )
372 {
373   myFixed out;
374
375   myFixed buffer00[5*5*3*4]; myFixed buffer01[5*5*4*4]; myFixed
        buffer02[5*5*4*8];
376   myFixed buffer1[60*60*3];
377   myFixed buffer20[56*56*4]; myFixed buffer21[24*24*4]; myFixed
        buffer22[8*8*8];
378   myFixed buffer30[28*28*4]; myFixed buffer31[12*12*4]; myFixed
        buffer32[4*4*8];
379
380   #pragma HLS interface ap_memory port=buffer00
381   #pragma HLS interface ap_memory port=buffer01
382   #pragma HLS interface ap_memory port=buffer02
383   #pragma HLS interface ap_memory port=buffer1
384   memcpy(buffer00, filter0, sizeof(myFixed)*filter_size0*
        filter_size0*input_depth0*filter_num0);
385   memcpy(buffer01, filter1, sizeof(myFixed)*filter_size1*
        filter_size1*input_depth1*filter_num1);
386   memcpy(buffer02, filter2, sizeof(myFixed)*filter_size2*
        filter_size2*input_depth2*filter_num2);
387   memcpy(buffer1, input_data0, sizeof(myFixed)*input_width0*
        input_height0*input_depth0);
388   CNNLayer0(
389   buffer00,
390   filter_num0,
391   filter_size0,
392   buffer1,
393   input_width0,
394   input_height0,
395   input_depth0,
396   buffer20,
397   conv_width0,
398   conv_height0,
399   buffer30,
400   pool_width0,
401   pool_height0,
402   pool_size0,
403   pool_stride0
404   );
405
406   CNNLayer1(
407   buffer01,
408   filter_num1,
```

```
409    filter_size1,
410    buffer30,
411    input_width1,
412    input_height1,
413    input_depth1,
414    buffer21,
415    conv_width1,
416    conv_height1,
417    buffer31,
418    pool_width1,
419    pool_height1,
420    pool_size1,
421    pool_stride1
422    );
423
424    CNNLayer1(
425    buffer02,
426    filter_num2,
427    filter_size2,
428    buffer31,
429    input_width2,
430    input_height2,
431    input_depth2,
432    buffer22,
433    conv_width2,
434    conv_height2,
435    buffer32,
436    pool_width2,
437    pool_height2,
438    pool_size2,
439    pool_stride2
440    );
441
442    #pragma HLS interface ap_memory port=buffer32
443    memcpy(pool_out2, buffer32, sizeof(myFixed)*pool_width2*
          pool_height2*filter_num2);
444  }
445
446  #if 1
447  /*
448     Forward()
449        E       E       E  */
450
451  int Forward(
452    myFixed  *input_data,
453    int input_num,
454    fcFixed *weight_hidden,
455    fcFixed *weight_out,
456    int hidden_num,
457    int out_num
458  )
459  {
460    int i = 0;
461    int j = 0;
462    fcFixed hidden_out[80];
463    fcFullBit sum;
464    fcFullBit out[2];
465    int label;
466
467    for(i = 0; i < hidden_num; ++i){
468        sum = 0;
469      for(j = 0; j < input_num; ++j){
```

```
470         sum += (fcFixed)input_data[j] * weight_hidden[i * (
                input_num) + j];
471       }
472       hidden_out[i] = (sum);
473     }
474     for(i = 0; i < out_num; ++i){
475       out[i] = 0;
476       for(j = 0; j < hidden_num; ++j){
477       out[i] += hidden_out[j] * weight_out[i * (hidden_num) + j];
478       }
479     }
480 #if 0
481     printf("Show␣hidden_out.\n");
482     show_fc(hidden_out, 1, hidden_num);
483     printf("\nout[0]:␣%.8lf,␣out[1]:␣%.8lf\n", (double)out[0], (
          double)out[1]);
484 #else
485     printf("out[0]:␣%.8lf,␣out[1]:␣%.8lf,", (double)out[0], (
          double)out[1]);
486 #endif
487     label = (out[0]>out[1]) ? 0 : 1;
488     return label;
489 }
490 #endif
```