Yellow Of The Egg
Lukas Baischer
Benjamin Kulnik
Anton Leitner
Stefan Marschner
Miha Cerv

SoC Design Laboratoy

384.157, Winter Term 2019

MNIST-FPGA
Documentation

Contents

1 Introduction

Notation

Weights	the parameter of the neural network
Activations	the input and output values of the layers
Activation Function	Function that is applied at the output of a layer
ReLU	Rectified Linear Unit, defined as $f(x) = \begin{cases} x & \text{if} \quad x > 0 \\ 0 & \text{else} \end{cases}$
Mini-Batch	A small fraction of the input data
Q	The significand of a fix point integer
m	the exponenent of a fix point integer

2 Concept

2.1 Neural Network

For the neural network we base the architecture of our network on the well known *LeNet* architecture from [?] is chosen due to its simplicity and ease to implement. Additionally the performance is improved by using modern, established techniques like batch normalization [?] and dropout [?] layers. The training of network is done using PyTorch [?] on a regular PC and the trained network parameters are then used to create a hardware VHDL model of the network. An overview of the structure can be seen in Figure 1. For verification all neural network operations are checked in separate programmed programs for correctness. See the Section 2.3 for details how the network is implemented in Software. An excellent overview in deep learning can be found in [?] and also in [?]. To train and test the network we chose the MNIST dataset [?]. It consists of 50.000 training images and 10.000 test images of handwritten digits, where each is 28-by-28 pixel.

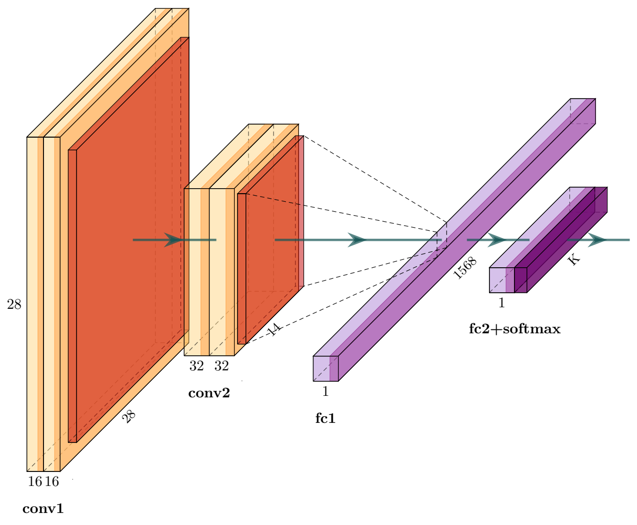


Figure 1: The Eggnet structure

2.2 Hardware Concept

Figure 2 shows the Concept of implementing an FPGA-based hardware accelerator for handwritten digit recognition. It shows that the main components of the concepts are a Zedboard in combination with a remote PC or server. The handwritten digit recognition is performed by the Zedboard while the remote PC is used for training the network, for sending the image data to the Zedboard and for receiving the computed results. The Zedboard includes a Zynq-7000 FPGA and provides various interfaces.

Maybe add an additional Input Layer which is responsible to communicate

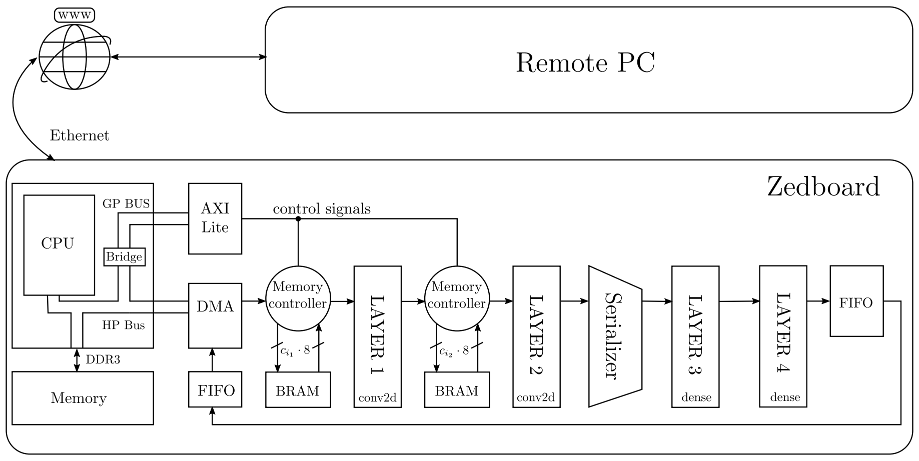


Figure 2: Top-Level concept

The neural network is implemented in the programmable logic part of the Zynq-7000. It is pre-trained using the remote PC, therefore only the inference of the neural network is implemented in hardware.

In order to train the network with the same bit resolution as implemented in the hardware, a software counterpart of the hardware is implemented in a PC using python. Based on the weights calculated by the python script a bitstream for the hardware is generated. This brings the benefit that for the convolutional layer constant multiplier can be used, since the weights of convolutional layer kernels are constant. For the dense layer it is not possible to implement the weights in a constant multiplier because in a dense layer each connection of a neuron requires a different weight, which would result in a huge amount of required constant multipliers. Therefore the weights for the dense layer have to be stored in a ROM inside the FPGA.

2.3 Neural Network Design and Training

The network was implemented in PyTorch [?] as well as Tensorflow [?]. The backend was later exclusively switched to PyTorch (which is also the most common deep learning framework in Science) due to its better support of qunatization. The layers of the network can be seen in Figure ??. For training of the network the *ADAM* optimization algorithm [?] was used to minimize the cross-entropy-loss function which is defined as

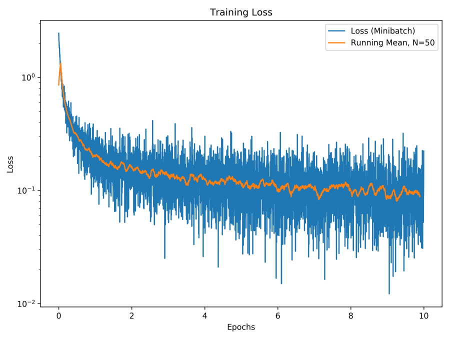
$$J = -y\log(h) + (1-y)\log(1-h) \tag{1}$$

For controlling the ADAM algorithm the recommended values, listed in Table 1, by [?] was used.

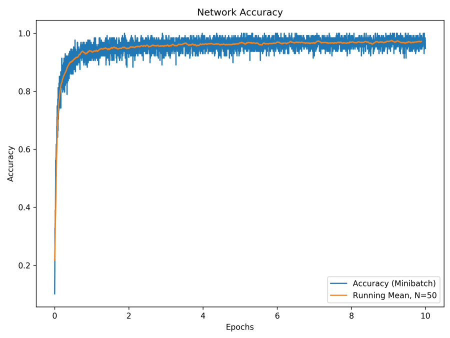
Table 1: Network Training Parameters

Parameter	Value
α	0.001
β_1	0.9
β_2	0.999

A useful guide for implementing convolutions can be found in [?]. The training of the network yielded very high accuracy rates - which is typical for the MNIST dataset, which is for Machine Learning an easy challenge. Even though the network performance could be improved, e.g. by hyperparameter tuning the results were acceptable for our case. The progress of the training in terms of accuracy and loss can be seen in Figure 3b respectively in Figure 3a. The final output of the network over the training is evaluated in Figure 4a for real values and in Figure 4b for fake quantized values.

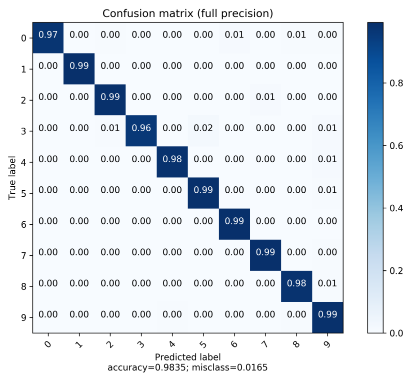


(a) Training Loss

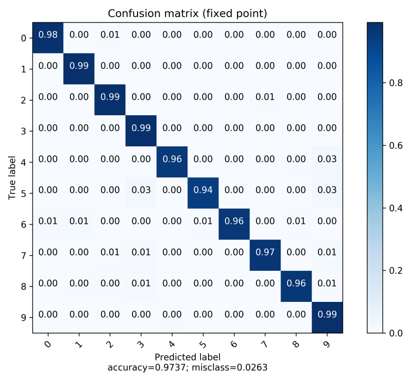


(b) Training Accuracy

Figure 3: Network loss and accuracy over the training iterations. The blue lines show spikes which occur because of the randomly selected mini batches.



(a) Floating Point



(b) Quantized Values

Figure 4: Confusion matrix for the floating point and quantized version of the netowork.

2.4 Quantization

The network is trained and created using 32bit floating point values in Python. Directly porting this all the weights and biases to the FPGA is due to the limited amount of available resources not feasible. The goal is therefore to reduce the amount of required hardware cells by switching from floating point arithmetic to the less expensive integer arithmetic. Then a floating point value v can be approximately represented as

$$v \approx Q \cdot 2^{-m} \tag{2}$$

where Q and m are integers. In our case all input values of the first layer are guaranteed to lie in the interval $[0, 1]$ and all layer weights are known from training. It is therefore possible to precompute the expected range where the output values will be. Depending on this range it is then possible to select a suitable bit width for both Q and m.

This is a cost-accuracy trade-off where higher bit widths would improve accuracy as well as increase the amount of hardware resources needed. In [?] different strategies of choosing bit widths for Q and m are compared and they observed three main configurations, which are (from simple to advanced):

1. Use a (Q, m) configuration for the whole network
2. Use a (Q, m) configuration for each layer
3. Use a (Q, m) configuration for each output channel

In the third configuration the authors could reduce the bit widths the most without sacrificing accuracy this increases the complexity in transferring the weights from layer to layer because the additional shift operations are necessary in order to adjust for the different values of m. In [?] the authors also deduced from their experiments that the accuracy of the weights can be reduced the most, followed by the activations. By analysing the weights of out network (see Figure 5) a per channel quantization is not necessary, because all weights in a Convolutional Layer are equally distributed among the output channels. Another important property that can be noted is the that the weights do have zero mean and most of the values lie very close to zero. Because of the usage of ReLU layer the situation is different for the activations where unsigned integers can be used, the distributions are shown in Figure 6.

Using the distribution histograms we then defined derived the necessary bitwidths for Q and m. In our experiments we were able to reduce them to 8 bit, if we used a single configuration for the whole network and also reducing them down to 4bit if the bitwidth configuration is selected for each layer independently with an accuracy drop from around 98.35 % to 97.37 %. The strategy to the select the values for (Q, m) was

1. Find the value range of the weights and output activations of each layer
2. Select suitable (Q, m) values that most activations fall in that range
3. Calculate the bit widths and exponents of the multiplication operation
4. Add $\lceil \log_2(n) \rceil$ extra bits to account for the accumulation of n values
5. Compare the accumulated exponents and with the exponents of the successive layers input exponents. The difference is the amount of shift required

It is noteworthy that the values for m do not need to be stored in the final network, because those are only used to determine the amount of shifts between the layers. Also the values need to be clipped to their maximum and minimum values. The complete configuration of the network is summarized in Table 2.

Ad 4 and 5: The transition from a layer to the next often changes the exponent m and the available bitwidth. To account for this the values need to accordingly shifted. Also the decreased bitwidth needs clipping to maximum available values for the target bitwidth. This directly alters the behaviour of the network which should be accounted for during training, which is done via a saturated version of ReLU, defined as:

$$\text{ReLU}_{\text{sat}}: \ f(x; p) = \begin{cases} 0 & \text{if} \quad x < 0 \\ p & \text{if} \quad x > p \\ x & \text{else} \end{cases} \tag{3}$$

For our network only linear quantization has been used but also non-linear quantization, e.g. in a $\log_2$ way which is proposed in [?]. Experiments showed that using this technique even further down to 3 bit weights in our case. Another optimization technique that could be explored is the systematically removing of weights (connections) of the network and reduce the amount of operations needed to be performed, a process referred to as ”pruning” [?]. This was not explicitly performed but is implicitly done by low bit quantization.

Network Part	$\|Q\|$	m	$\pm$	v (real value range)
Input	8	8	$+$	$[0,1]$
L1: Weights	4	2	$\pm$	$[-2,2]$
L1: Intermediates	12	10	$\pm$	$[-2,2]$
L1: Accumulated	16	10	$\pm$	
L1 $\rightarrow$ L2	Rshift by $10-2$ and clip values in range $[0,15]$			
L2: Input	4	2	$+$	$[-2,2]$
L2: Weights	4	5	$\pm$	$[-0.5,0.5]$
L2: Intermediates	8	7	$\pm$	$[-1,1]$
L2: Accumulated	16	7	$\pm$	
L2 $\rightarrow$ L3	Rshift by $7-0$ and clip values in range $[0,15]$			
L3: Input	4	0	$+$	$[0,15]$
L3: Weights	4	5	$\pm$	$[-0.5,0.5]$
L3: Intermediates	8	5	$\pm$	$[-7.5,7.5]$
L3: Accumulated	19	5	$\pm$	
L3 $\rightarrow$ L4	Rshift by $5-0$ and clip values in range $[0,15]$			
L4: Input	4	0	$+$	$[0,15]$
L4: Weights	4	5	$\pm$	$[-0.5,0.5]$
L4: Intermediates	8	5	$\pm$	$[-7.5,7.5]$
L4: Accumulated	14	5	$\pm$	$[0,1]$

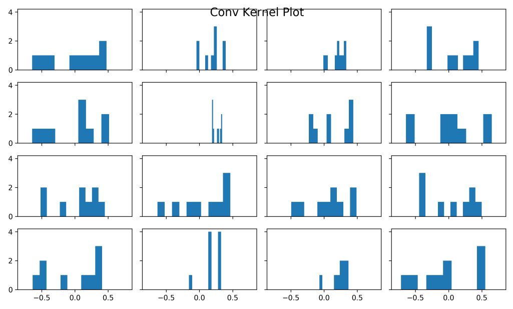
Table 2: Quantization parameters for the 4bit network. The intermediate terms are the values after the multiplication operation and the accumulated term denotes values after summing up of weighted inputs including bias in a channel.

3 Software

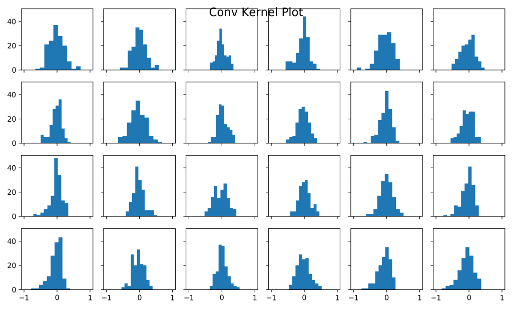
To ease the usage of the network when it is implemented in the FPGA a suitable user driver software is needed. Here it should be possible to send a single or multiple images to the FPGA and collect the estimated results.

Also because the major development in Machine Learning is done in Python and the Zedboard is able to run Linux it would be beneficial to control the network via Python and embed it in an existing Machine Learning Framework like Keras [?] or Torch [?]. Additionally the Zedboard needs additional resources of the FPGA if a graphical user interface should be outputted via the HDMI port. Due to the already sparse available resources the GUI was deactivated and the whole communication was done over Ethernet (or for the initial setup a serial connection).

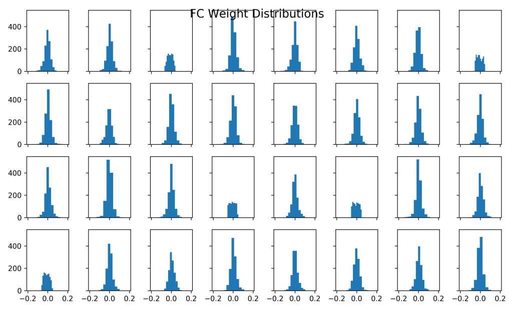
- Write a low level driver in C to communicate with the FPGA
- Provide a Python Interface to the driver
- Provide a Web control interface



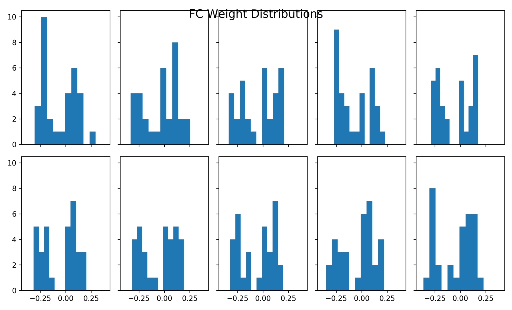
(a) Convolutional Layer 1



(b) Convolutional Layer 2

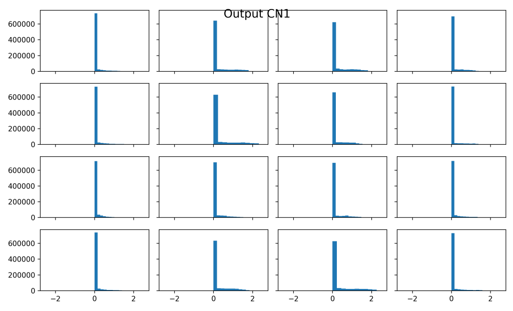


(c) Fully Connected Layer 1

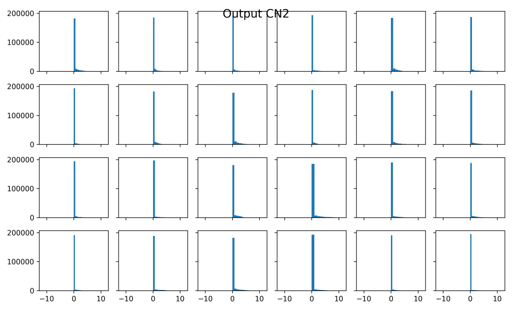


(d) Fully Connected Layer 2

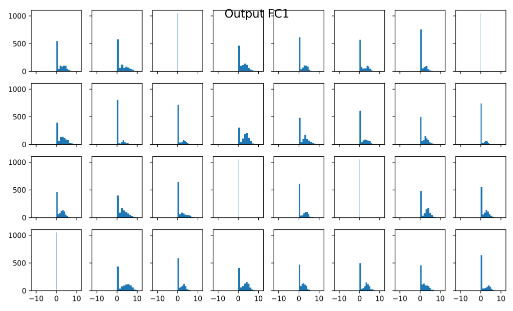
Figure 5: Distribution of the network weights for the different layers



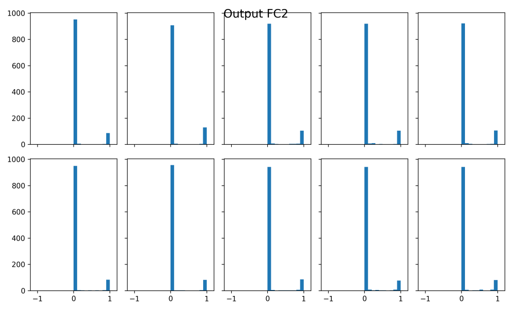
(a) Convolutional Layer 1



(b) Convolutional Layer 2



(c) Fully Connected Layer 1



(d) Fully Connected Layer 2

Figure 6: Distribution of the activations for a randomly selected batch of the input data

3.1 FPGA Driver

The user layer driver software implements an interface between the ARM Top-Level software and the driver for the programmable logic. It is implemented in C. It is supposed to handle the entire communication with the driver so that the hardware is only abstractly visible for the ARM Top-Level software.

For example the ARM top-level software sees the network as a class in python which has a methode_load_new_image data with a numpy array as input and a finish signal as a output. This method should call the user layer driver software which handles the communication between user space and kernel space. In a similar way each IP should be a class in python.

Requirements of the User Layer Driver Software:

- Communication with the kernel space drivers
- Use python wrapper to communicate with ARM Top-Level software
- Easy to use interface from Top-Level
- No knowledge of the hardware should be necessary to use the interface
- Data encapsulation to avoid the Top-Level Software from corrupting the memory

3.2 Python Driver Interface

3.3 Web Interface

The webapp should provide an easy yet good interface to control the network and check its functionality. Because most of the networks software logic is done in python, also the backend of the webapp was done with Python Flask [?]. This enabled us the quick interfacing between web components and our python code and also made the communication of multiple threads easier.

4 Hardware

4.1 Hardware Notation

Generic Layer Interface

Parameter Name	VHDL Type	Description
`s_Clk_i`	`std_logic`	Clock input
`s_n_Res_i`	`std_logic`	Reset input (active low)
`s_Valid_i`	`std_logic`	Flag, if the input is valid
`s_Ready_i`	`std_logic`	the parameter of the neural network
`s_Last_i`	`std_logic`	the parameter of the neural network

Table 3

4.2 Memory Controller

The task of the memory controller is to provide valid data for the NN-layers. It communicates with the Block-Ram. The memory controller is responsible for ensuring that the next layer has valid data at all times. The second task of the memory controller is to save the data of the previous data in a free memory address in the Block-RAM.

Is it better to have the shiftregister, we discussed last time in the memory controller, because in this case the layer don't have to know

4.2.1 Interfaces

- S_LAYER: interface to previous layer
- M_LAYER: interface to next layer

- AXI_lite: interface to AXI lite bus, is used to read BRAM data directly from processor (slow)

signal	direction	type	width	description

- M_LAYER: interface to next layer

signal	direction	type	width	description

- BRAM_PORTA: write interface to BRAM

signal	direction	type	width	description

- BRAM_PORTB: read interface to BRAM

signal	direction	type	width	description

4.2.2 Parameter

- PREVIOUS_LAYER_TYPE boolean: TRUE: conv2d, FALSE: dense
- PREVIOUS_LAYER_WIDTH integer: Row length of input matrix
- PREVIOUS_LAYER_HEIGTH integer: Column length of input matrix
- PREVIOUS_LAYER_CHANNEL integer: Row length of input matrix
- NEXT_LAYER_TYPE boolean: TRUE: conv2d, FALSE: dense
- NEXT_LAYER_WIDTH integer: Row length of input matrix
- NEXT_LAYER_HEIGTH integer: Column length of input matrix
- NEXT_LAYER_CHANNEL integer: Row length of input matrix

use extra parameter for dense or simply use width or height, discuss!

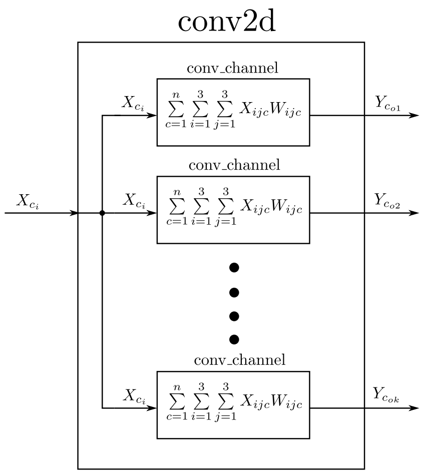
use extra parameter for dense or simply use width or height, discuss!

4.3 AXI lite interface

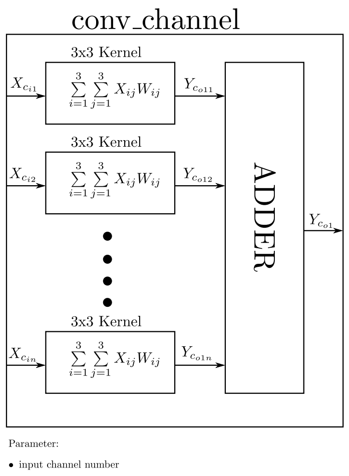
It is used to read the BRAM data directly from the processor. This can be used for debug purposes.
Each memory controller gets an unique address via generics. One 32 bit register of the AXI lite bus is used for all memory controller. If the processor writes all 0 to the register, debugging mode is deactivated. Therefore the memory controller address start with 1 and not with 0. the 32 bit are separated as follows:

- 23 downto 0: BRAM address
- 27 downto 24: 32 bit vector address
- 31 downto 28 : Memory controller address

BRAM address: address of the block ram
32 bit vector address: If the width of one BRAM register is higher than 32 bit, the 32 bit vector address can be used to select the required part of the vector.
Memory controller address: address of the memory controller used in the network starting with 1. If the address of the memory controller is selected debug mode is active.



(a) Conv2d block diagram. For each output channel a conv_channel module is used. k indicates the number of output channels.



(b) conv_channel block diagram. For each input channel a kernel_3x3 module is used. n indicates the number of input channels.

Figure 7

4.4 Convolutional Layer

The convolutional layer - referred to as conv2d - receives an input data tensor X_{ci} of size $[H, W, C_{in}]$ and outputs a data tensor of size $[H, W, C_{out}]$. Here H and W is the height and width of the convolutional kernel. So every input tensor is transferred to each single convolutional computation channel, which is responsible for computing a *single* output channel. Each convolutional channel parameterized for with its own set of values.

Figure 7a shows the block diagram of a conv2d module. It uses k conv_channel modules to realise k output channels. All conv_channel modules get the same input vector X_{c_i}. All conv_channel modules and the two conv2d modules are automatically generated by a Python script.

Figure 7b shows the block diagram of a conv_channel module. It uses n kernel_3x3 modules to realise n input channels. All kernel_3x3 modules get a different input vector $X_{c_{i1}}$ to $X_{c_{in}}$ which are 3×3 input matrices. All kernel outputs are summed up to one final value of length BIT_WIDTH_OUT.

4.4.1 Interface

- Input interface connected to shift register, which consists of a $n \cdot 3 \times 3$ vector of values of length BIT_WIDTH_IN, in which n is the number of input channels.
- Output interface connected to the pooling layer, which is a vector of m values of length BIT_WIDTH_OUT, in which m is the number of output channels.

Both input and output interfaces have ready, last and valid signals to control the flow of data.

4.4.2 Parameter

- BIT_WIDTH_IN : integer
- BIT_WIDTH_OUT : integer
- INPUT_CHANNELS: integer
- OUTPUT_CHANNELS: integer

conv_channel

Interface

- Input interface, same as conv2d.
- Output interface connected to the pooling layer, which is a value of length BIT_WIDTH_OUT.

Parameter

- BIT_WIDTH_IN : integer
- KERNEL_WIDTH_OUT : integer, output bit width of the kernel_3x3 module
- BIT_WIDTH_OUT: integer
- N: integer, number of kernels
- OUTPUT_MSB: integer, defines which of the n=BIT_WIDTH_OUT bits is the most significant bit
- BIAS: integer, currently unused as bias seems to not be very important in the convolutional layers

4.5 conv_channel

Figure ?? shows the block diagram of a conv_channel module. It uses n kernel_3x3 modules to realise n input channels. All kernel_3x3 modules get a different input vector $X_{c_{i1}}$ to $X_{c_{in}}$ which are 3×3 input matrices. All kernel outputs are summed up to one final value of length BIT_WIDTH_OUT.

4.5.1 Interface

- Input interface, same as conv2d.
- Output interface connected to the pooling layer, which is a value of length BIT_WIDTH_OUT.

4.5.2 Parameter

- BIT_WIDTH_IN : integer
- KERNEL_WIDTH_OUT : integer, output bit width of the kernel_3x3 module
- BIT_WIDTH_OUT: integer
- N: integer, number of kernels
- OUTPUT_MSB: integer, defines which of the n=BIT_WIDTH_OUT bits is the most significant bit
- BIAS: integer, currently unused as bias seems to not be very important in the convolutional layers

4.6 kernel-3x3

This modules performs the convolution of a single kernel with a single input image patch. This is a multiplication of - in our case - 9 values of length BIT_WIDTH_IN with their respective weights which are defined in an array that can be set with a generic. The multiplication results are then added up in an adder tree.

4.6.1 Interface

- Input interface, a vector of 9 values of length BIT_WIDTH_IN.
- Output interface, same as conv_channel.

4.6.2 Parameter

- BIT_WIDTH_IN: integer
- BIT_WIDTH_OUT: integer
- WEIGHT: array of 9 integers
- WEIGHT_WIDTH: integer

4.7 NN

4.7.1 Operation

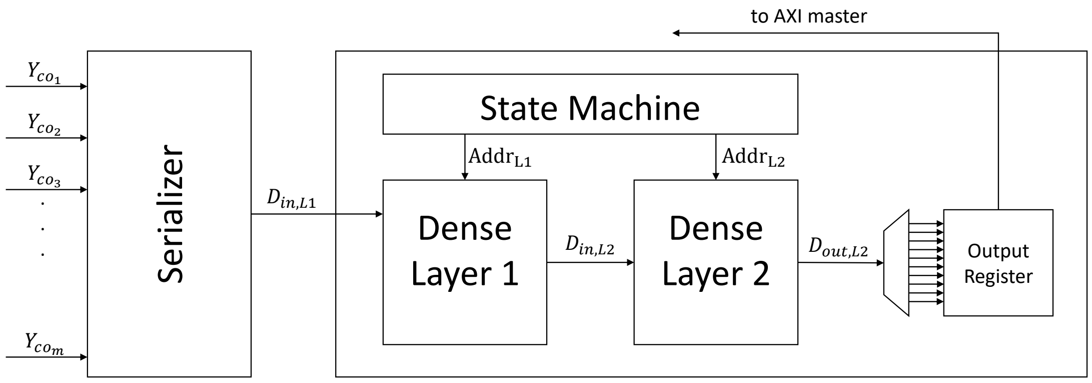


Figure 8: Diagram of the combined, fully connected NN.

The fully-connected neural network is shown in figure 8. It consists of two dense layer instances controlled by a state machine. The output of layer 1 is fed directly into the layer 2. The output of layer 2 are 10 values which represent the confidence that the input image showed a specific number.

The Serializer module is connected to the previous pooling layer. The $m = 32$ output channels need to be converted into a stream of single values of length VECTOR_WIDTH. For this, the previous pooling layer is stalled by keeping the ready signal low while a vector of m values is serialized.

4.7.2 Interface

- Input interface, a stream of values of length VECTOR_WIDTH
- Output interface, a vector of 10 values of length VECTOR_WIDTH

4.7.3 Parameter

- VECTOR_WIDTH: integer
- INPUT_COUNT: integer
- OUTPUT_COUNT: integer

4.8 Dense Layer

4.8.1 Operation

Figure 9 depicts the dense or fully connected layer of the network. This block contains a finite state machine. When the Start_i input port goes high, input neurons are read from an external FIFO one by one. Each of the input neurons is multiplied by appropriate weight for each of the output neurons. These product are then fed to accumulators, which make a sum of all products of all neurons. When all of the incoming neurons are processed, the calculation is finished and a Finished_o output port is raised high to signal that data is available. Result data can be addressed by Rd_addr_i port and read out at the Data_o port.

Number of input neurons, output neurons and data width are generic.

4.8.2 Weights

Weights are stored in a ROM memory. The values are hardcoded at synthesis. The VHDL code reads the weights from a file. File contains the weight values in binary. Each line represents all of the weights for one input neuron. There are as many lines as there are input neurons.

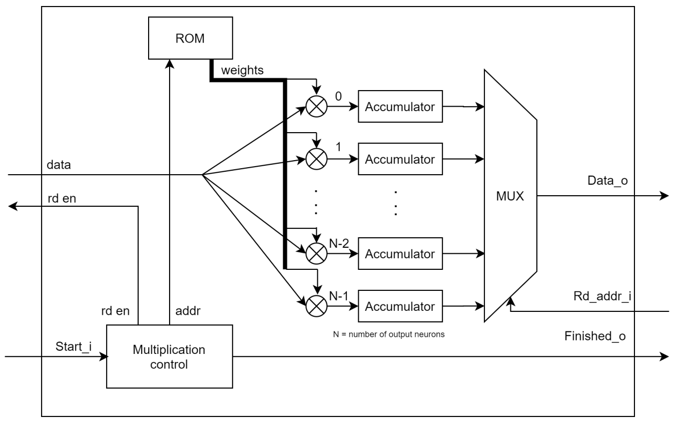


Figure 9: Dense layer diagram.

4.8.3 Bias terms

Bias terms are also loaded from a file. Each output neuron has its own bias term. Each line contains one bias term. Bias term bit width is generic. Bias terms are treated as a signed value.

4.8.4 Parameters

VECTOR_WIDTH : integer	Bit width of input data.
INPUT_COUNT : integer	Number of input neurons
OUTPUT_COUNT : intege	Number of output neurons.
ROM_FILE : string	File, that holds the weight values.
BIAS_WIDTH : integer	Bit width of the bias terms.
BIAS_FILE : string	File, that holds the bias term values.

5 Conclusion

6 Appendix

6.1 Network Operations

Convolutional Operations

The output of an convolutional layer is defined by

$$z(i,j) = (f * g)(i,j) = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} f(m,n) g(m-i, n-j) \tag{4}$$

It is explained in more detail here: [?]

Fully Connected Layer

The output of an fully connected layer is defined by

$$z = xW + b \tag{5}$$

where $x \in \mathbb{R}^{b,m}$, $W \in \mathbb{R}^{m,n}$ and $b \in \mathbb{R}^n$.

Rectified Linear Unit (ReLU)

$$f(x) = \begin{cases} x & \text{if} \quad x > 0 \\ 0 & \text{else} \end{cases} \tag{6}$$

Softmax

6.2 Matrix Calculus

The chain rule for a vectors is similar to the chain rule for scalars. Except the order is important. For $\mathbf{z} = f(\mathbf{y})$ and $\mathbf{y} = g(\mathbf{x})$ the chain rule is:

$$\frac{\partial \mathbf{z}}{\partial \mathbf{x}} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}} \frac{\partial \mathbf{z}}{\partial \mathbf{y}} \tag{7}$$

y	$\frac{\partial}{\partial x} y$
Ax	A^T
$x^T A$	A
$x^T x$	$2x$
$x^T A x$	$Ax + A^T x$

Table 4: Useful derivatives equations

6.3 Fix-Point Arithmetic

Multiplication of two fix point values yields

$$v_1 v_2 = \text{right-shift}\left(Q_1 Q_2 \cdot 2^{-(m+m)}; m\right) \tag{8}$$

Note that for multiplication the exponent m for the values can be different.

Addition of two fix point values

$$v_1 + v_2 = (Q_1 + Q_2) \cdot 2^{-m} \tag{9}$$

6.4 Source Code

All the source code is licensed under the *MIT* Licence and can be found on Github. `https://github.com/marbleton/FPGA_MNIST`

There are also the most up to date build instructions how to compile the project. Those are in short: Dependencies

- Vivado 2017.4
- Python 3.6 including Numpy, SWIG [?], Keras and Torch

1. Train the network using the `net/train_keras.py` script
2. Use the network `net/quant.py` script to quantize the trained network layers
3. Generate the VHDL files

6.5 Other

Other resources which are useful:

How Tensorflow is implementation `https://github.com/dmlc/nnvm-fusion` and `https://github.com/tqchen/tinyflow`

Deep Learning Course from University of Washington `http://dlsys.cs.washington.edu`

List of Figures

List of Tables

Todo list