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

SoC Design Laboratoy

384.157, Winter Term 2019

MNIST-FPGA
Specification

1 Introduction

2 Concept

2.1 Neural Network

For the neural network we base the architecture of our network on the well known *LeNet* architecture from [LeCun et al., 1998] is chosen due to its simplicity and ease to implement. Additionally the performance is improved by using modern, established techniques like batch normalization [Ioffe and Szegedy, 2015] and dropout [Srivastava et al., 2014] layers. The training of network is done using PyTorch [Paszke et al., 2019] 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 3.1 for details how the network is implemented in Software. An excellent overview in deep learning can be found in [Schmidhuber, 2015]. To train and test the network we chose the MNIST dataset [LeCun, 1998]. It consists of 50.000 training images and 10.000 test images of handwritten digits, where each is 28-by-28 pixel.

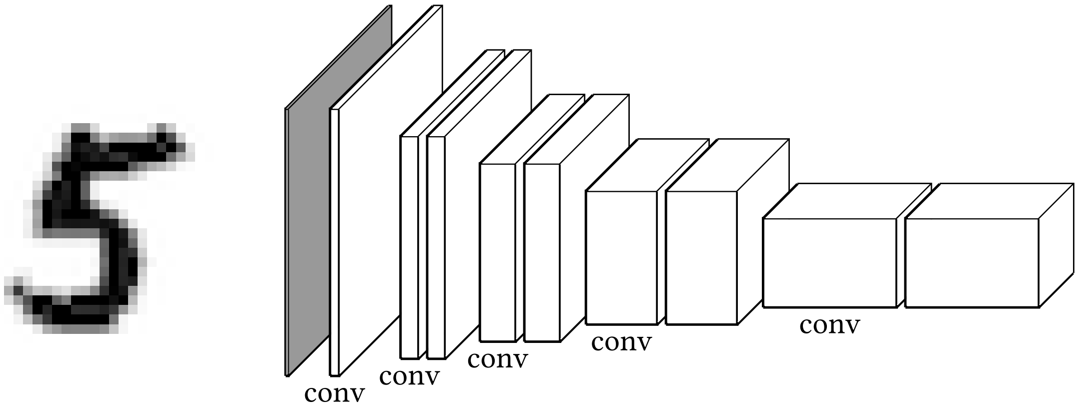


Figure 1: Example CNN.

2.2 Hardware Concept

Figure 2 shows the Concept of implementing an FPGA-based hardware accelerator for handwritten digit recognition. A Zedboard is used as hardware platform, which includes a Zynq-7000 FPGA and provides various interfaces. It is also shown that the design can be remote controlled via a server which is running on the Zedboard.

A hardware accelerator for 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.

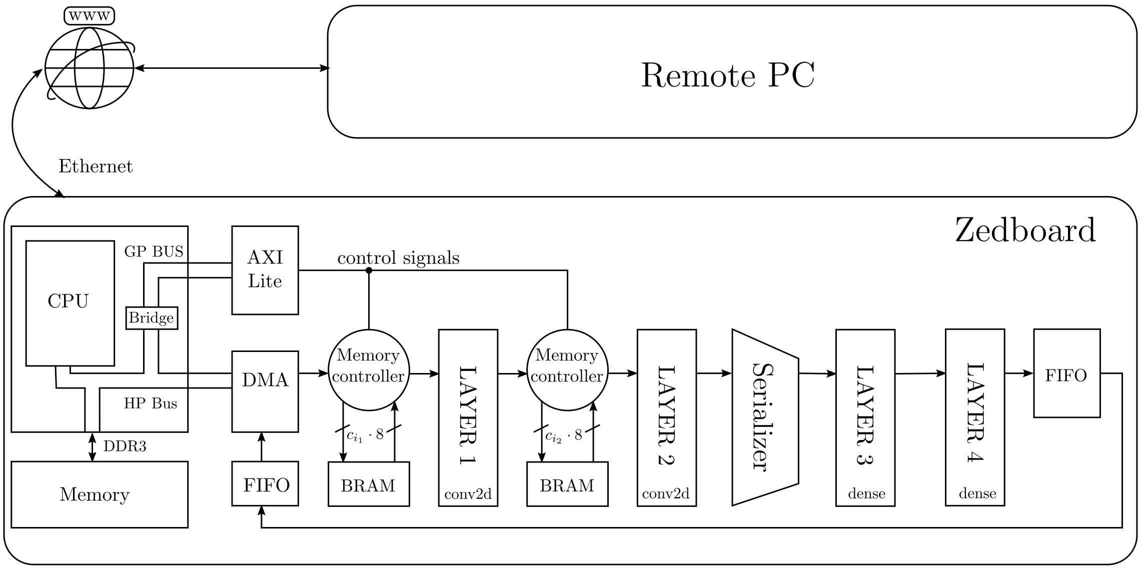


Figure 2: Top-Level concept

3 Software

3.1 Neural Network Design and Training

The network was implemented in PyTorch [Paszke et al., 2019] as well as Tensorflow [Abadi et al., 2015]. 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 neural network can be seen in Figure 3. For training of the network the *ADAM* optimization algorithm [Kingma and Ba, 2014] 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 [Kingma and Ba, 2014] 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 [Dumoulin and Visin, 2016]. 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 5 respectively in Figure 4. The final output of the network over the training is evaluated in Figure 6 for real values and in Figure 7 for fake quantized values.

3.2 Quantization

The network was reduced to fixed point integers defined by the equation

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

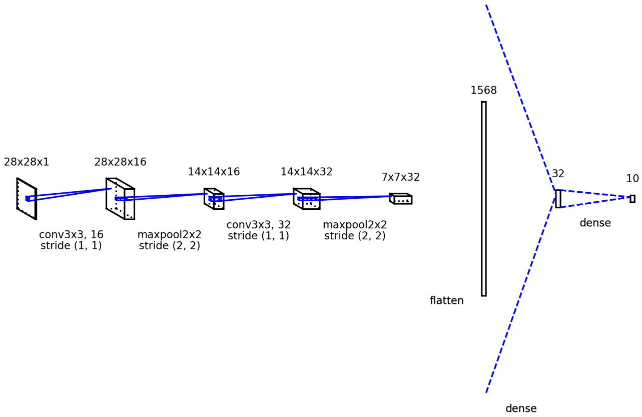


Figure 3: Network overview

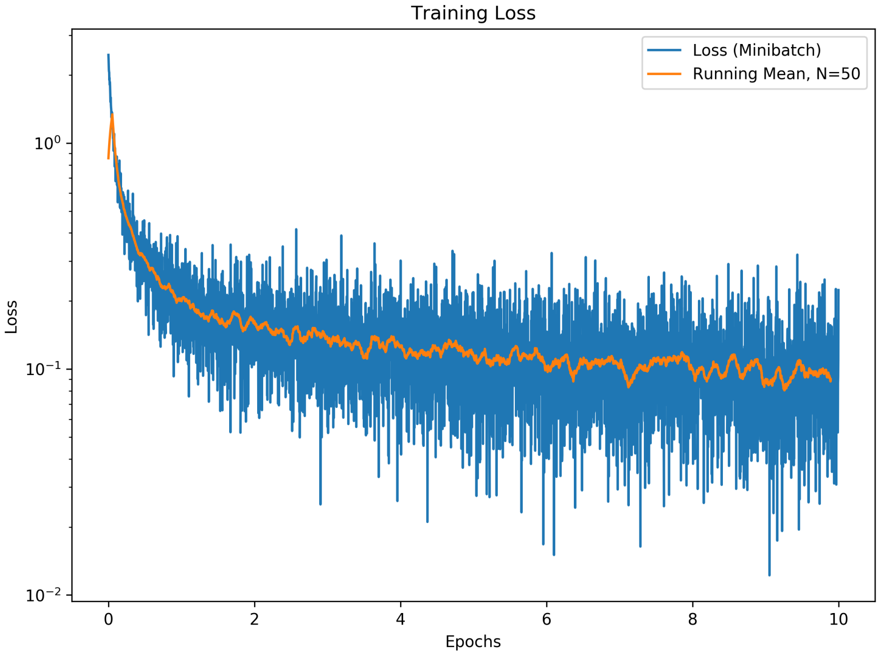


Figure 4: Training Loss

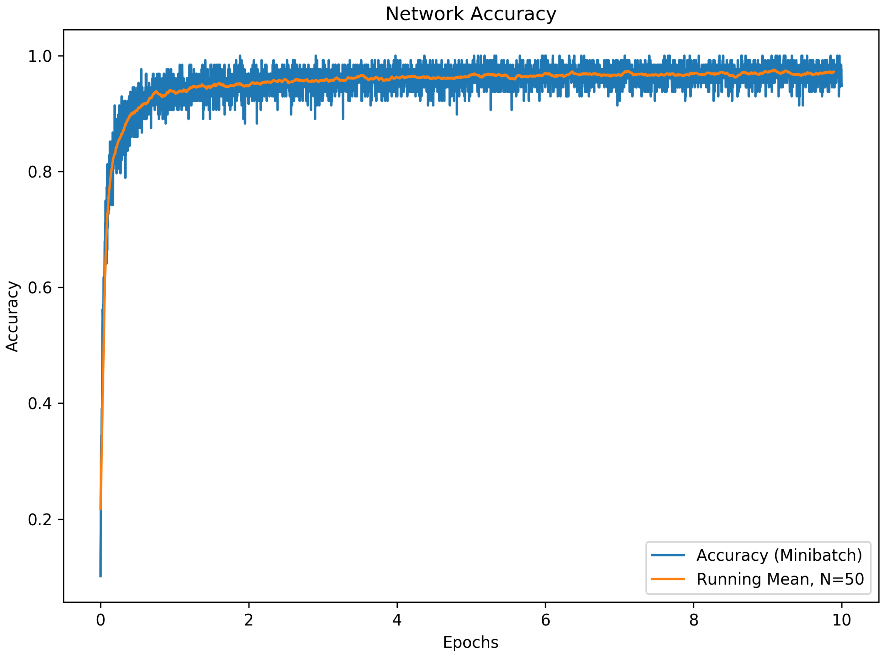


Figure 5: Training Accuracy

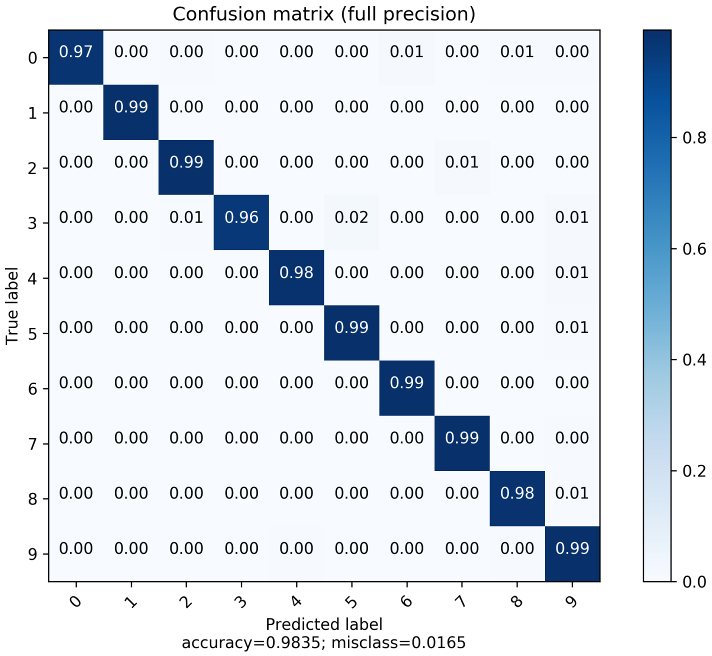
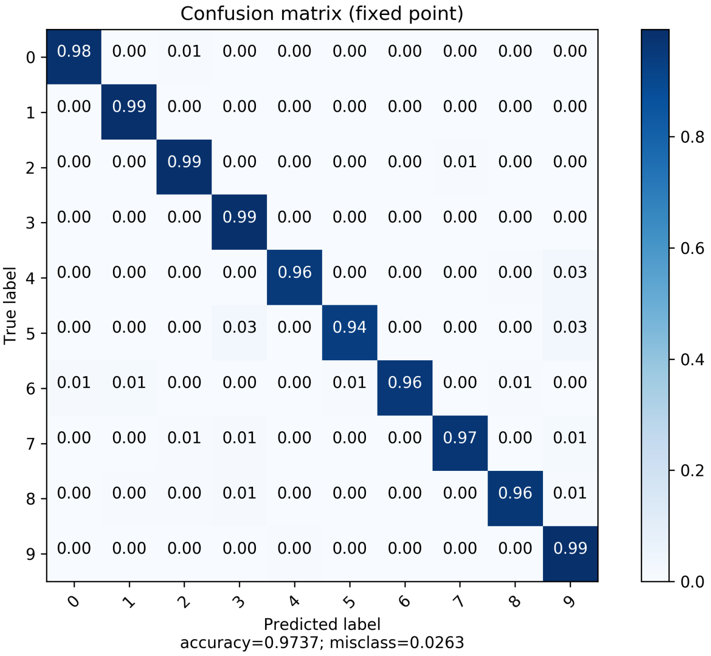


Figure 6: Confusion Matrix (real)



Confusion matrix (fixed point)

True label \ Predicted label	0	1	2	3	4	5	6	7	8	9
0	0.98	0.00	0.01	0.00	0.00	0.00	0.00	0.00	0.00	0.00
1	0.00	0.99	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00
2	0.00	0.00	0.99	0.00	0.00	0.00	0.00	0.01	0.00	0.00
3	0.00	0.00	0.00	0.99	0.00	0.00	0.00	0.00	0.00	0.00
4	0.00	0.00	0.00	0.00	0.96	0.00	0.00	0.00	0.00	0.03
5	0.00	0.00	0.00	0.03	0.00	0.94	0.00	0.00	0.00	0.03
6	0.01	0.01	0.00	0.00	0.00	0.01	0.96	0.00	0.01	0.00
7	0.00	0.00	0.01	0.01	0.00	0.00	0.00	0.97	0.00	0.01
8	0.00	0.00	0.00	0.01	0.00	0.00	0.00	0.00	0.96	0.01
9	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.00	0.99

Figure 7: Confusion Matrix with quantized values

where both Q and m are integers and v is the real value representation of the fixed point number. For our network implementation we chose a vairable fraction length for each layer and a total bitlength for Q of 8. This quantization can further be adjusted for each convolutional channel separately but optimization in this way were beyond the scope of this project. The multiplication of two fixed point integers is trivial, because the product for two real values v_1 and v_2 is

$$v_1 v_2 = \text{right-shift}\left(Q_1 Q_2 \cdot 2^{-(m+m)}; m\right) \tag{3}$$

In our tests a quantization with 8 bit fixedpoint integers and even 4 bit integers showed only a small drop in accuracy of around 1%.

Add Histogram

3.3 Remote software

It is used for performing the training of the network and for generating a FPGA-bitstream based on the computed weights. Additionally the remote software is used to send the image data to the Zedboard and receive the results of the network for each image.

Therefore the Host software can be separated in two parts:

- Trainings software
- Communication software

Requirements of the Trainings Software:

- Training of the network considering bit resolution of implemented hardware
- Create VHDL code based on the network hyper-parameter and on the computed weights
- Create a bitstream with the generated VHDL code

Requirements of the Trainings Software:

- Sends image data to Zedboard
- Receives results from Zedboard

- Create a figure of accuracy and performance
- Optional: Send bitstream to hardware which updates the bitstream

3.3.1 Interface to Zedboard

As interface to the zedboard a flask application is running on the zedboard which allows to upload a new image of a digit, test the hardware accelerate using the MNIST-dataset, gives useful information and allows to update the FPGA bitstream.

- The uploaded images are resized to 28 × 28 automatically
- The MNIST-dataset can be used
- Status information about the inference time and CPU utilization are shown
- Bitstream file for update the hardware accelerator can be uploaded.

3.4 ARM Top-Level Software

The ARM top-level software receives the image data from a remote device and sends the results back to this device. Control of the hardware.

3.4.1 Requirements

Add more information and specify the requirements

Requirements of the ARM Top-Level Software:

- Receive image data
- Also use image data set already stored on device
- send results to remote PC
- Send and receive control signals from remote PC
- Send image data to driver user layer and receive results from driver user layer
- Send and receive status and control signals to driver user layer
- Run at start-up

3.4.2 Dynamic Updating of the Bitstream

Optional feature: Update Bitstream file using `/dev/xdevcfg`.

Update: For newer versions it looks like `/dev/xdevcfg` doesn't exist anymore. The problem is discussed here [1] and a potential solution can be found here. [2]

3.4.3 Interface to remote PC

See Section 3.3.1.

3.4.4 Interface to kernel layer

Python wrapper are used for the interface between the top level software which is programmed in python and the hardware drivers which are programmed in C. For usability a high level interface to the underlying C wrapper is made. This header interface can then be wrapped to multiple target languages using [Swi, 2020]. In our case this was done for Python.

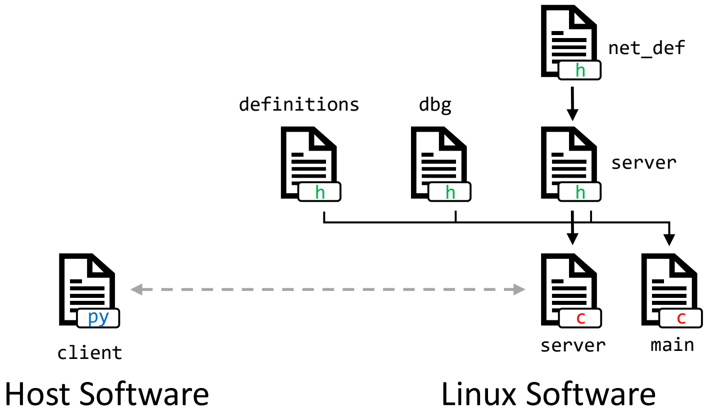


Figure 8: File tree for the software

3.4.5 File Tree of ARM Top-Level software

Update this section. Do we still use the C code or do we plan to implement everything in python?

- `net_def.h` Contains definitions for networking, e.g. ports used.
- `dbg.h` Contains debugging macros for logging and error handling.
- `definitions.h` Contains information about the neural network, e.g. the number and type of Convolutional Neural Network (CNN) stages, layers in the fully connected network, input size and so on.
- `server.{c,h}` Handles the connection with the host software.
- `main.c` Contains the `main()` function with the main program loop that transmits and manages data to the hardware and from the host system.
- `client.py` Handles the connection with the client software.

3.5 User Layer Driver Software

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.
The ARM top-level software sees the network as a class in python which has methods to load a 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

3.5.1 File Tree of User Layer Driver Software

Would be nice if we have something similar as in 3.4.5

[1] `https://forum.digilentinc.com/topic/18194-dynamically-load-bitstream-on-petalinux/`
[2] `https://github.com/Digilent/zynq-dynamic-tools`

4 Hardware

4.1 Memory Controller

The task of the memory controller is to provide valid data for the NN-layers in every cycle. It receives data from the previous layer of from the DMA and stores it in a Block-RAM. If a full image or feature map is received. The memory controller starts sending the data to the next layer. Receiving and sending data is performed in parallel, which therefore doesn't lead to additional delay time. In order to read the 3×3 kernel in every clock cycle a linebuffer is used to store 2 lines. This enables the memory controller to provide a 3×1 vector in every cycle. To get a 3×3 kernel additionally a shiftregister is used.

4.1.1 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)

4.1.2 Parameter

- BRAM_ADDR_WIDTH: integer: Address width of the connected Block-RAM
- DATA_WIDTH: integer: Data width of activations
- IN_CHANNEL_NUMBER: integer: Number of input channels
- LAYER_HEIGTH: integer: Height of input matrix
- LAYER_WIDTH: integer: WIDTH of input matrix
- AXI4_STREAM_INPUT: integer: 1 for input layer else 0
- MEM_CTRL_ADDR: integer: Layer number of memory controller (used for AXI-lite interface)
- C_S_AXIS_TDATA_WIDTH: integer: AXI-stream data width (required for input layer)
- C_S00_AXI_DATA_WIDTH: integer: AXI-lite data width

4.2 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.

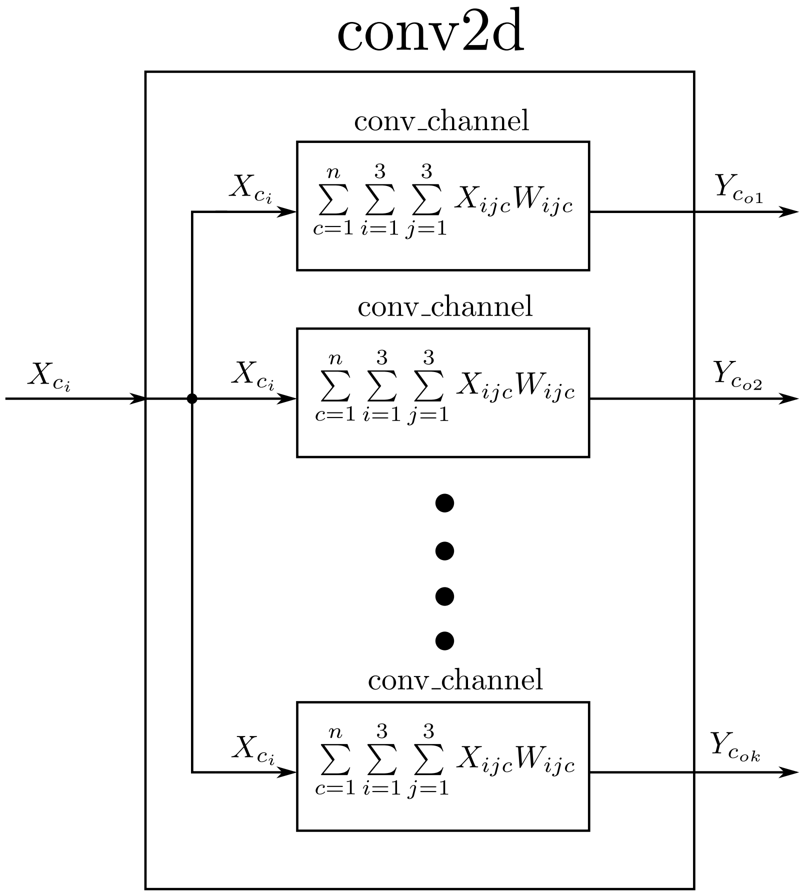

conv2d
conv_channel
X_{c_i}
$\sum_{c=1}^{n}\sum_{i=1}^{3}\sum_{j=1}^{3} X_{ijc}W_{ijc}$
$Y_{c_{o1}}$
conv_channel
X_{c_i}
$\sum_{c=1}^{n}\sum_{i=1}^{3}\sum_{j=1}^{3} X_{ijc}W_{ijc}$
$Y_{c_{o2}}$
X_{c_i}
conv_channel
X_{c_i}
$\sum_{c=1}^{n}\sum_{i=1}^{3}\sum_{j=1}^{3} X_{ijc}W_{ijc}$
$Y_{c_{ok}}$

Figure 9: Conv2d block diagram. For each output channel a conv_channel module is used. k indicates the number of output channels.

4.3 conv2d

Figure 9 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.

4.3.1 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.

4.3.2 Parameter

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

4.4 conv_channel

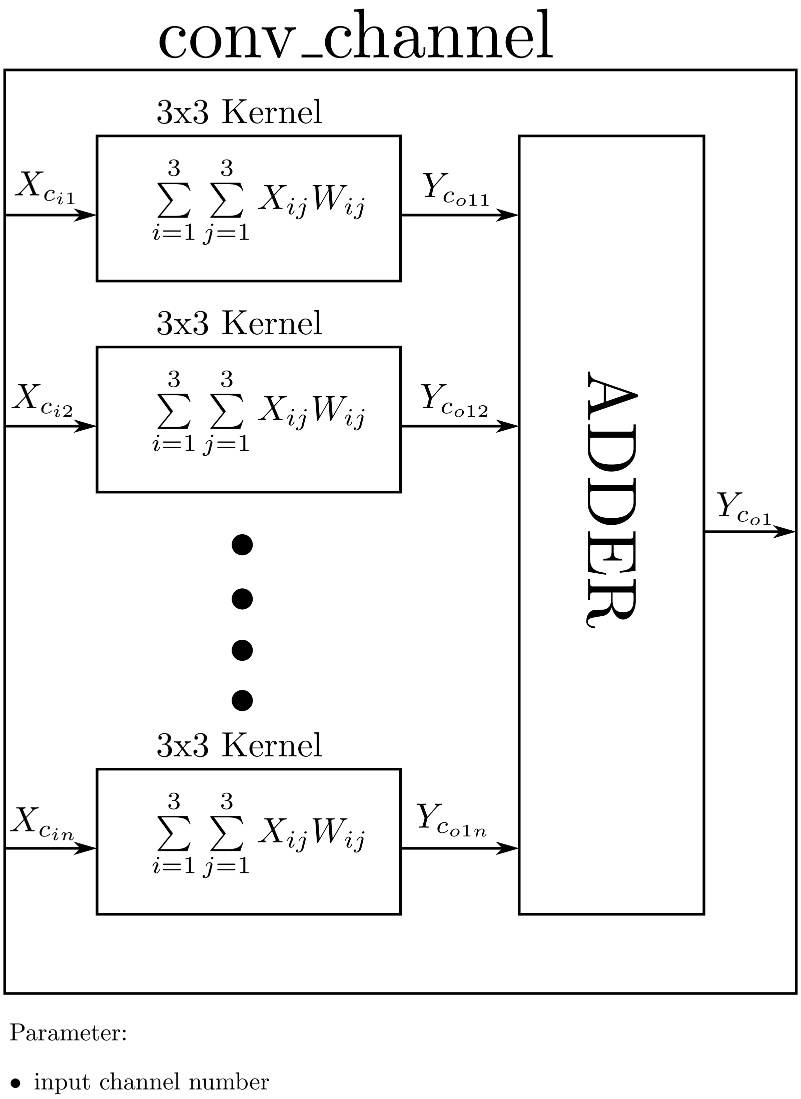


Figure 10: conv_channel block diagram. For each input channel a kernel_3x3 module is used. n indicates the number of input channels.

Figure 10 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, same as conv2d.
- Output interface connected to the pooling layer, which is a value of length BIT_WIDTH_OUT.

4.4.2 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

4.5 kernel-3x3

This modules performs a multiplication of 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, after which a ReLu step is performed where outputs above 255 are clipped to 255 and outputs below 0 are clipped to 0.

4.5.1 Interface

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

4.5.2 Parameter

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

4.6 NN

4.6.1 Operation

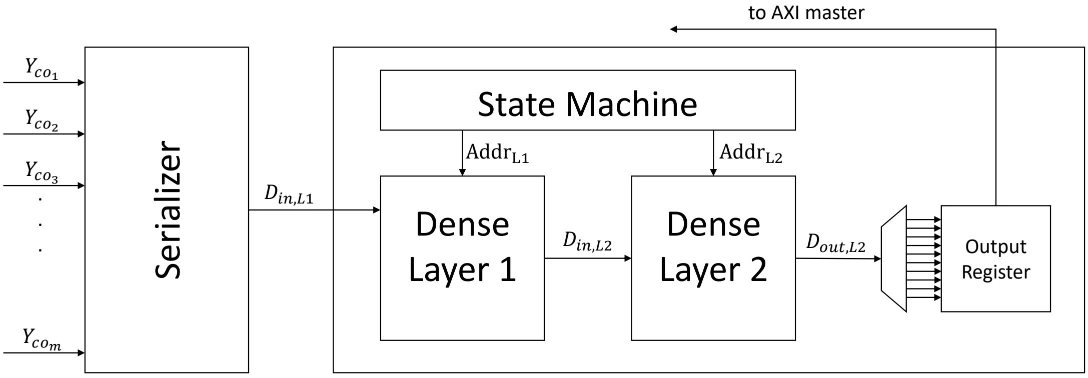


Figure 11: Diagram of the combined, fully connected NN.

The fully-connected neural network is shown in figure 11. 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.

4.6.2 Interface

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

4.6.3 Parameter

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

4.7 Dense Layer

4.7.1 Operation

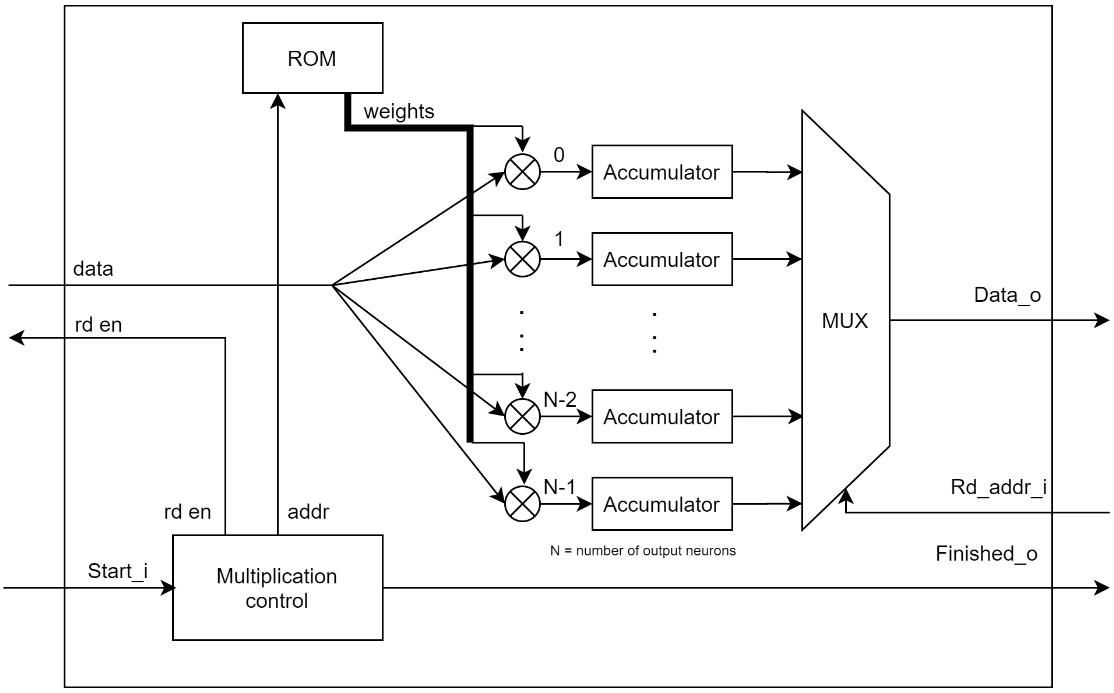


Figure 12: Dense layer diagram.

(Schematic is on figure 12.) 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.

4.7.2 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.

4.7.3 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.

4.7.4 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.

Appendix

Network Operations

Convolutional Operations

The output of an convolutional layer is defined by

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

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

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:

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

Source Code

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

Add cite to github using Zenodo (must be done by Anton)

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

Table 2: Useful derivatives equations

4.8 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`

References

[Swi, 2020] (2020). SWIG. [Online; accessed 9. Mar. 2020].

[Abadi et al., 2015] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

[Dumoulin and Visin, 2016] Dumoulin, V. and Visin, F. (2016). A guide to convolution arithmetic for deep learning.

[Ioffe and Szegedy, 2015] Ioffe, S. and Szegedy, C. (2015). Batch normalization: Accelerating deep network training by reducing internal covariate shift. *arXiv preprint arXiv:1502.03167*.

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

[LeCun, 1998] LeCun, Y. (1998). The mnist database of handwritten digits. *http://yann.lecun.com/exdb/mnist/*.

[LeCun et al., 1998] LeCun, Y., Bottou, L., Bengio, Y., Haffner, P., et al. (1998). Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324.

[Paszke et al., 2019] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., and Chintala, S. (2019). Pytorch: An imperative style, high-performance deep learning library. In Wallach, H., Larochelle, H., Beygelzimer, A., d'Alché-Buc, F., Fox, E., and Garnett, R., editors, *Advances in Neural Information Processing Systems 32*, pages 8024–8035. Curran Associates, Inc.

[Schmidhuber, 2015] Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural Networks*, 61:85 – 117.

[Srivastava et al., 2014] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I., and Salakhutdinov, R. (2014). Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research*, 15(1):1929–1958.