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

Contents

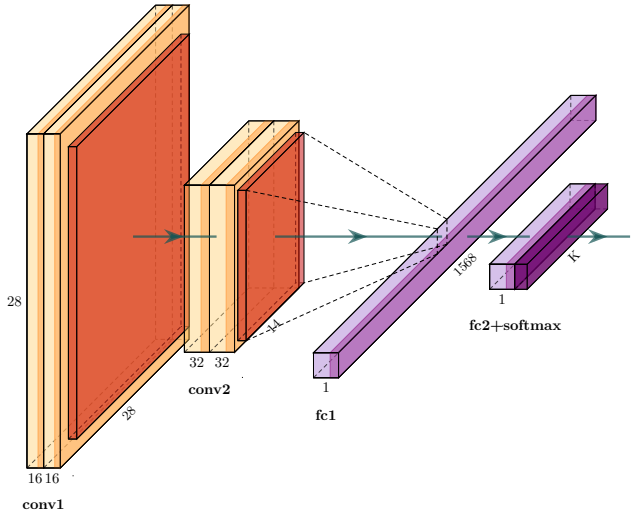


Figure 1: The Eggnet structure

1 Introduction

Notation

Weights	the parameter of the neural network
Activations	the input and output values of the layers
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 [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 ??. 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.

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.
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

Maybe add an additional Input Layer which is responsible to communicate with the DMA and converts the data from 32 bit to 8 bit and sends it to the memory controller

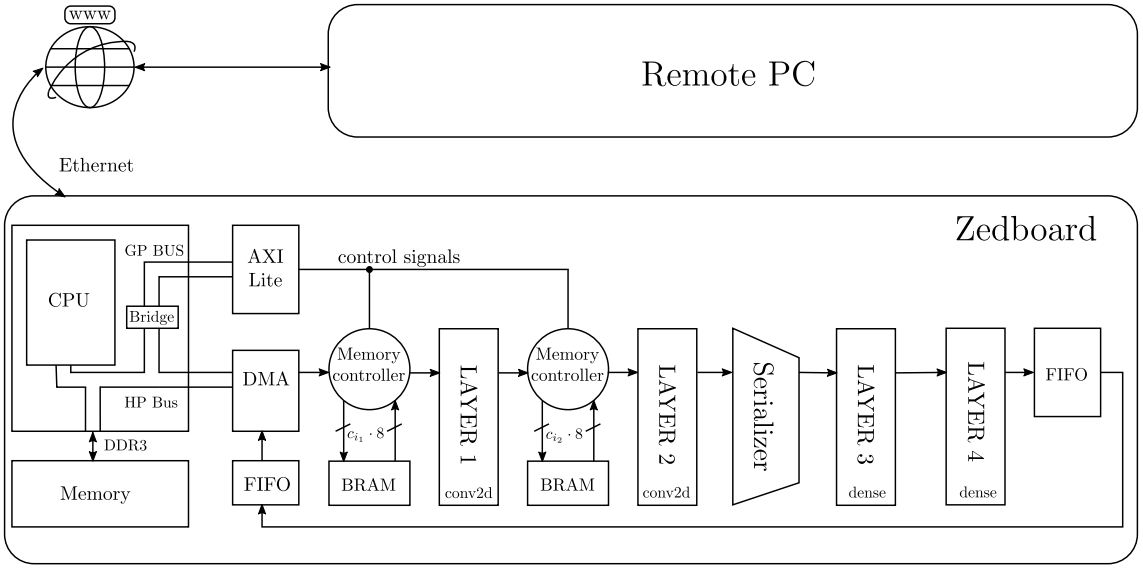


Figure 2: Top-Level concept

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.

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 network can be seen in Figure ??. 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

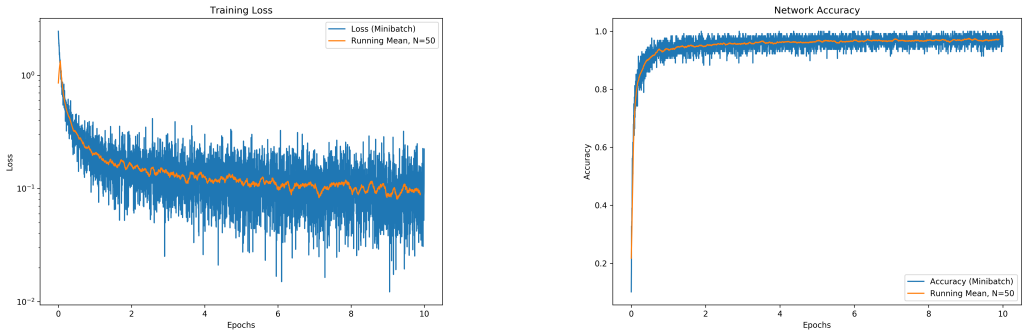
$$J = -y \log(h) + (1 - y) \log(1 - h) \tag{1}$$

For controlling the ADAM algorithm the recommended values, listed in Table 1, by [Kingma and Ba, 2014] was used.

Table 1: Network Training Parameters

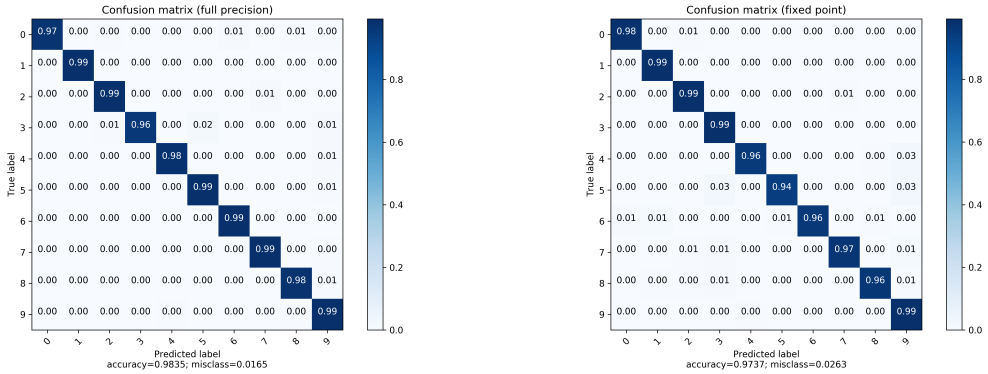
Parameter	Value
α	0.001
β_1	0.9
β_2	0.999

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 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



(a) Floating Point (b) Quantized Values

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

3.2 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

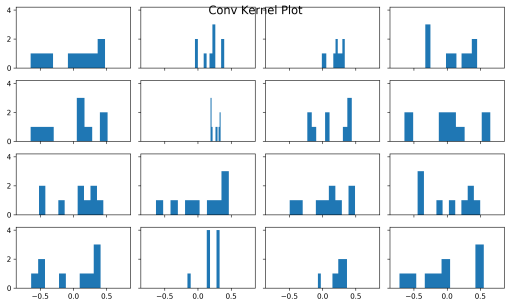
$$v \approx Q \cdot 2^{-m} \tag{2}$$

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.

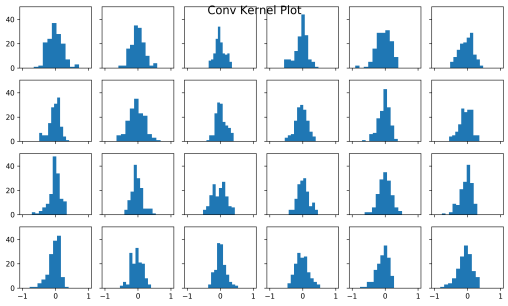
This is a cost-accuracy trade-off where higher bit widths would improve accuracy as well as increase the amount of hardware resources needed. In [Wu et al., 2018] different strategies of choosing bit widhts for Q and m are compared and they observed three main configurations, which are (from simple to 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

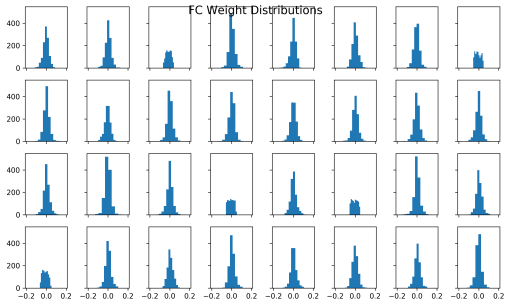
In the third configuration the authors could reduce the bit widths the most without sacrificing accuracy this increases the complexity in transferring the weights from layer to layer because the additional shift operations are necessary in order to adjust for the different values of m. In [Wu et al., 2018] the authors also deduced from their experiments that the accuracy of the weights can be reduced the most, followed by the activations. By analyzing 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.



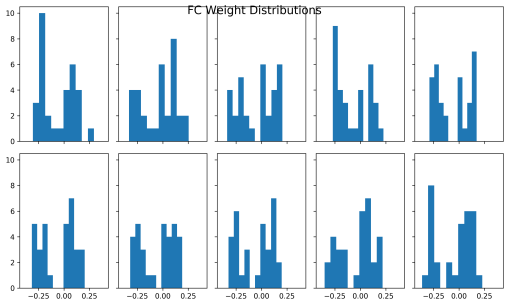
(a) Convolutional Layer 1



(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

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

$$v = Q \cdot 2^{-m} \tag{3}$$

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

$$v_1 v_2 = \text{right-shift}\left(Q_1 Q_2 \cdot 2^{-(m+m)}; m\right) \tag{4}$$

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 Host software

The remote software is either implemented on a PC or on a server. 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

Ethernet is used for the communication of the remote host system and the embedded Linux which is running on the Zedboard. The embedded Linux distribution running on the board should automatically receive an IP address when connected to a network. When in doubt the address can be found out with the `ifconfig` command. The software has a client-server model with the embedded system acting as a server and the host as a client. Once running, the server software is listening for new outside connections. Different types of data need to be transmitted:

- The 28x28 input images showing digits between 0 and 9 is transferred from host to Zedboard.
- The probability of resulting numbers between 0 and 9 is transmitted from Zedboard to host.
- control and status signals in both directions
- Optional: Bitstream file for dynamically update the bitstream at the Zedboard

3.3.2 Notes

On Windows host systems, *Network Discovery* needs to be enabled and in some cases a Firewall exception for the used ports needs to be set for a connection to be established.

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.

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.

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

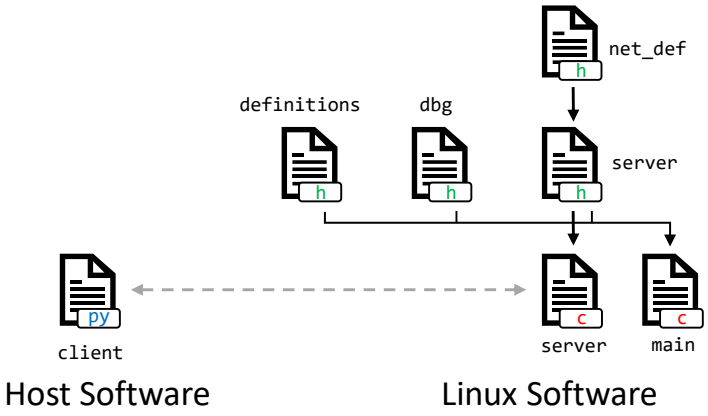


Figure 6: File tree for the software

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.
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

3.5.1 File Tree of User Layer Driver Software

Would be nice if we have something similar as in 3.4.5

4 Hardware

5 Appendix

5.1 Network Operations

Convolutional Operations

The output of an convolutional layer is defined by

$$z(i,j) = (f * g)(i,j) = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} f(m,n) g(m-i, n-j) \tag{5}$$

It is explained in more detail here: [Dumoulin and Visin, 2016]

Fully Connected Layer

The output of an fully connected layer is defined by

$$z = xW + b \tag{6}$$

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

Rectified Linear Unit (ReLU)

$$f(x) = \begin{cases} x & \text{if} \quad x > 0 \\ 0 & \text{else} \end{cases} \tag{7}$$

Softmax

5.2 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:

$$\frac{\partial \mathbf{z}}{\partial \mathbf{x}} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}} \frac{\partial \mathbf{z}}{\partial \mathbf{y}} \tag{8}$$

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

5.3 Fix-Point Arithmetic

Multiplication of two fix point values yields

$$v_1 v_2 = \text{right-shift}\left(Q_1 Q_2 \cdot 2^{-(m+m)}; m\right) \tag{9}$$

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

Addition of two fix point values

$$v_1 + v_2 = (Q_1 + Q_2) \cdot 2^{-m} \tag{10}$$

5.4 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 [Swi, 2020], 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

5.5 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`

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.

[Wu et al., 2018] Wu, S., Li, G., Chen, F., and Shi, L. (2018). Training and inference with integers in deep neural networks. *arXiv preprint arXiv:1802.04680*.

List of Figures

List of Tables

Todo list