Yellow Of The Egg
Lukas Baischer
Benjamin Kulnik
Add your names ...

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.

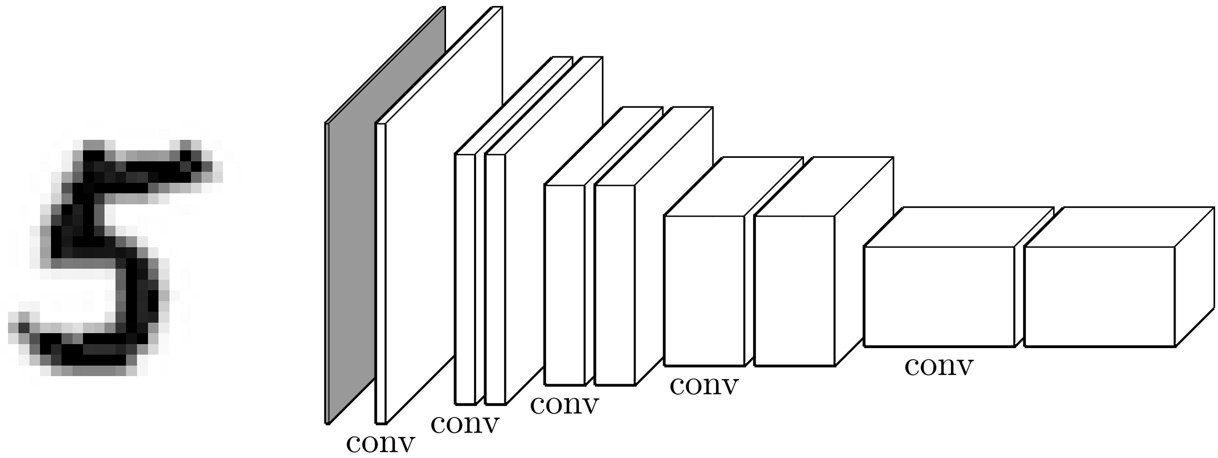

Figure 1: Example CNN.

2.2 Hardware Concept

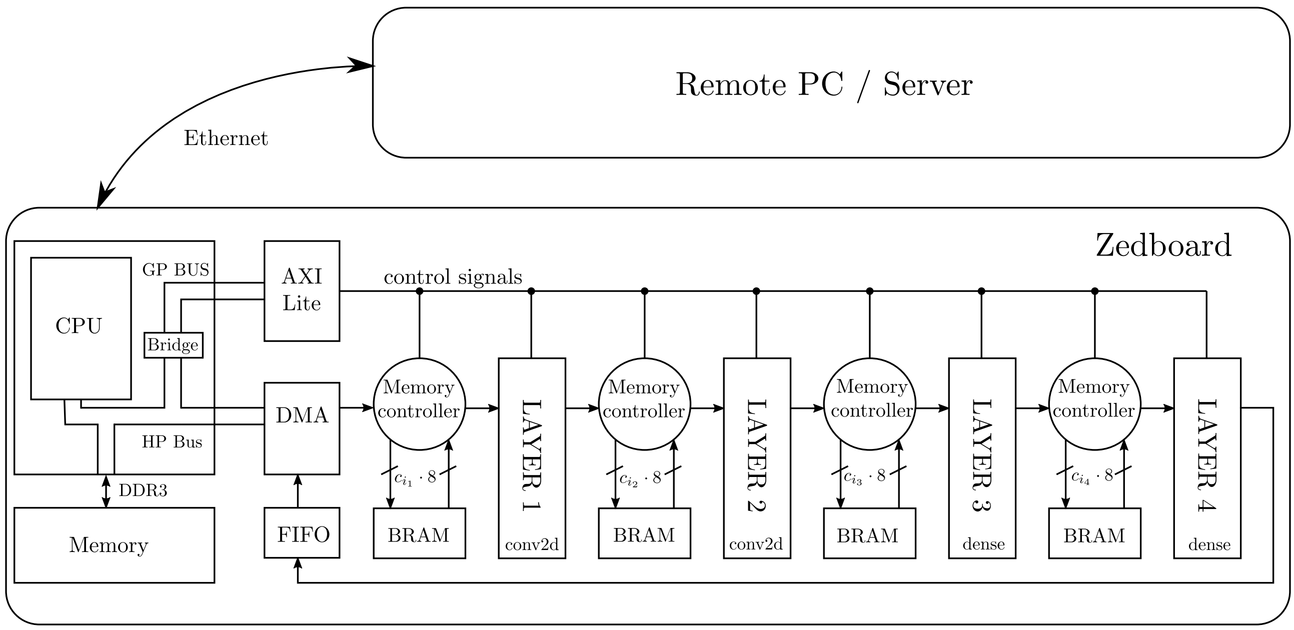

Figure 2: Top-Level concept

Maybe ad an additional Input Layer which is responsib

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 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 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

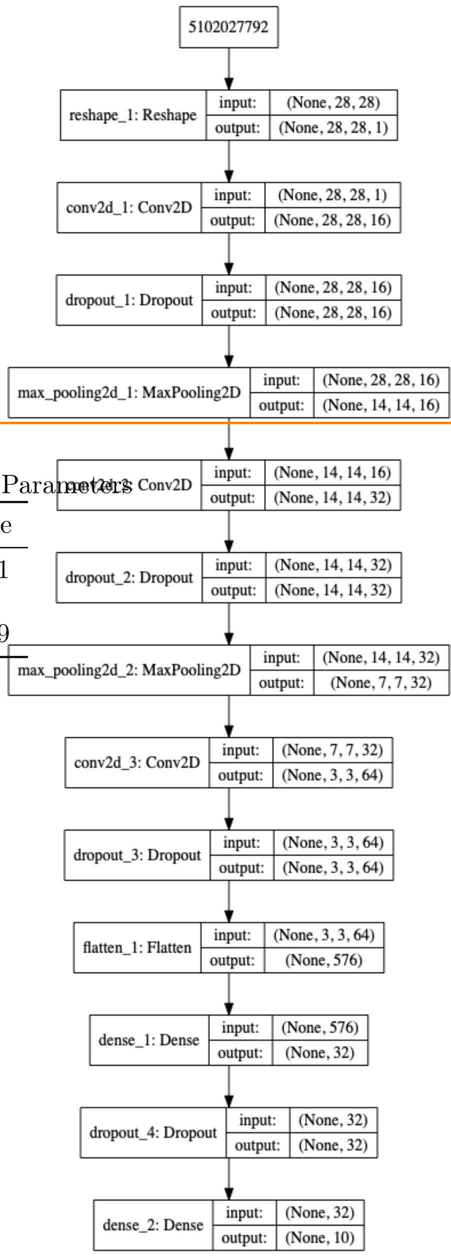

Figure 3: Network Layers

Force table to be centred in text

Add qunatization details

3.2 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 hyperparameter 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

3.2.1 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:

Who is the host and client now?

- 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

3.2.2 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.

3.3 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.
Optional feature: Update Bitstream file using /dev/xdevcfg
Requirements of the ARM Top-Level Software:

Add more information and specify th requirements

- Receive image data
- 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

3.3.1 Interface to remote PC

See Section 3.2.1.

3.3.2 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

Add more informa- tion and

3.3.3 File Tree of ARM Top-Level software

Update this sec- tion. Do we still us the C cod or do we plan to im plement everything in python

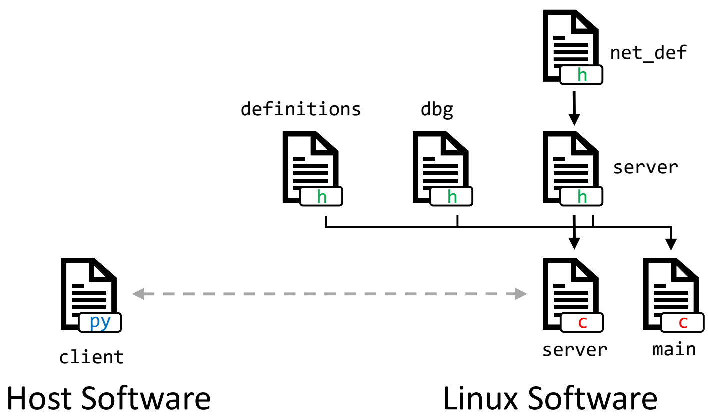

Figure 4: File tree for the software

- `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.

3.4 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

3.4.1 File Tree of User Layer Driver Software

Would be nice if we have some thing sim- ilar as in 3.3.3

4 Hardware

4.1 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 dis cussed las time in th memory controller because in this case the layer don't have to know anything about the data it gets

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)

| 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 |

4.1.2 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!

4.2 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.

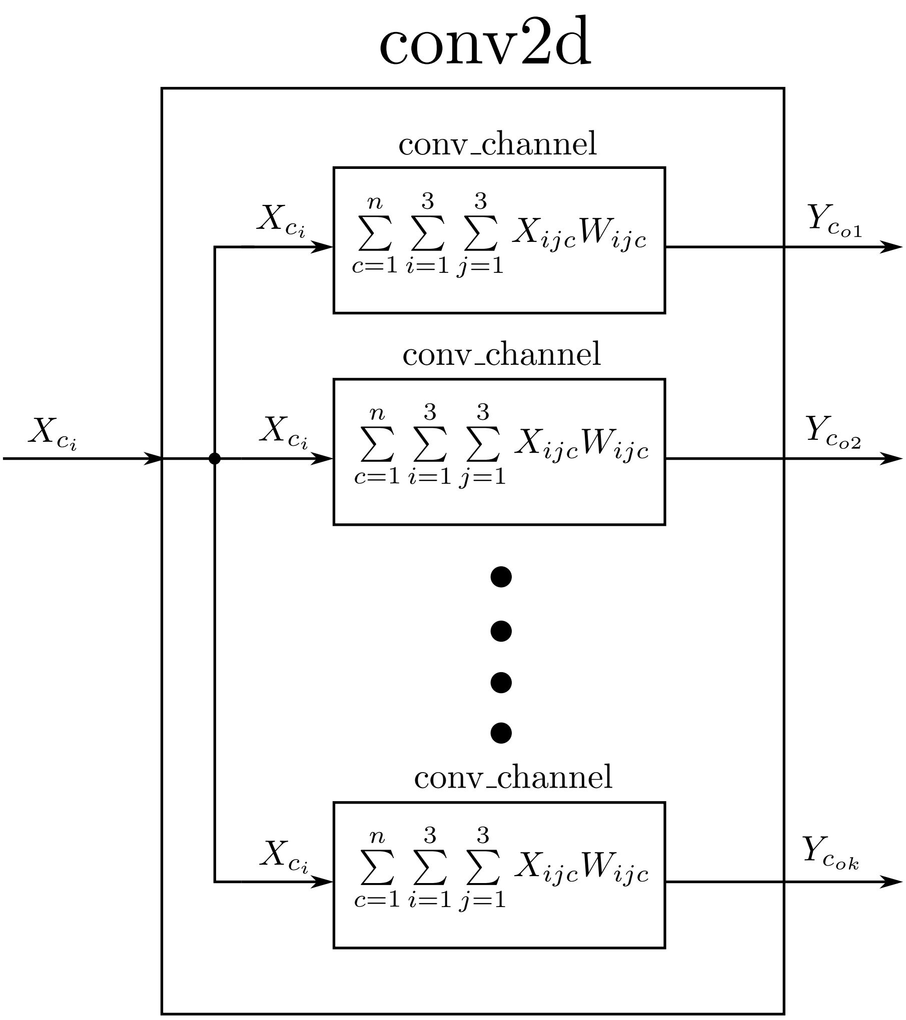

Figure 5: Conv2d block diagram. For each output channel a conv_channel module is used. k indicates the number of output channels.

4.3 conv2d

Figure 5 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} which consists of $n \cdot 3 \times 1$ vector, in which n is the number of input channels.

4.3.1 Interface

define in-put outpu interface

4.3.2 Parameter

- INPUT_CHANNEL_NUMBER : integer
- OUTPUT_CHANNEL_NUMBER : integer
- MATRIX_WIDTH: integer
- MATRIX_HEIGTH: integer

4.4 conv_channel

Figure 6 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.

4.4.1 Interface

define in-put outpu interface

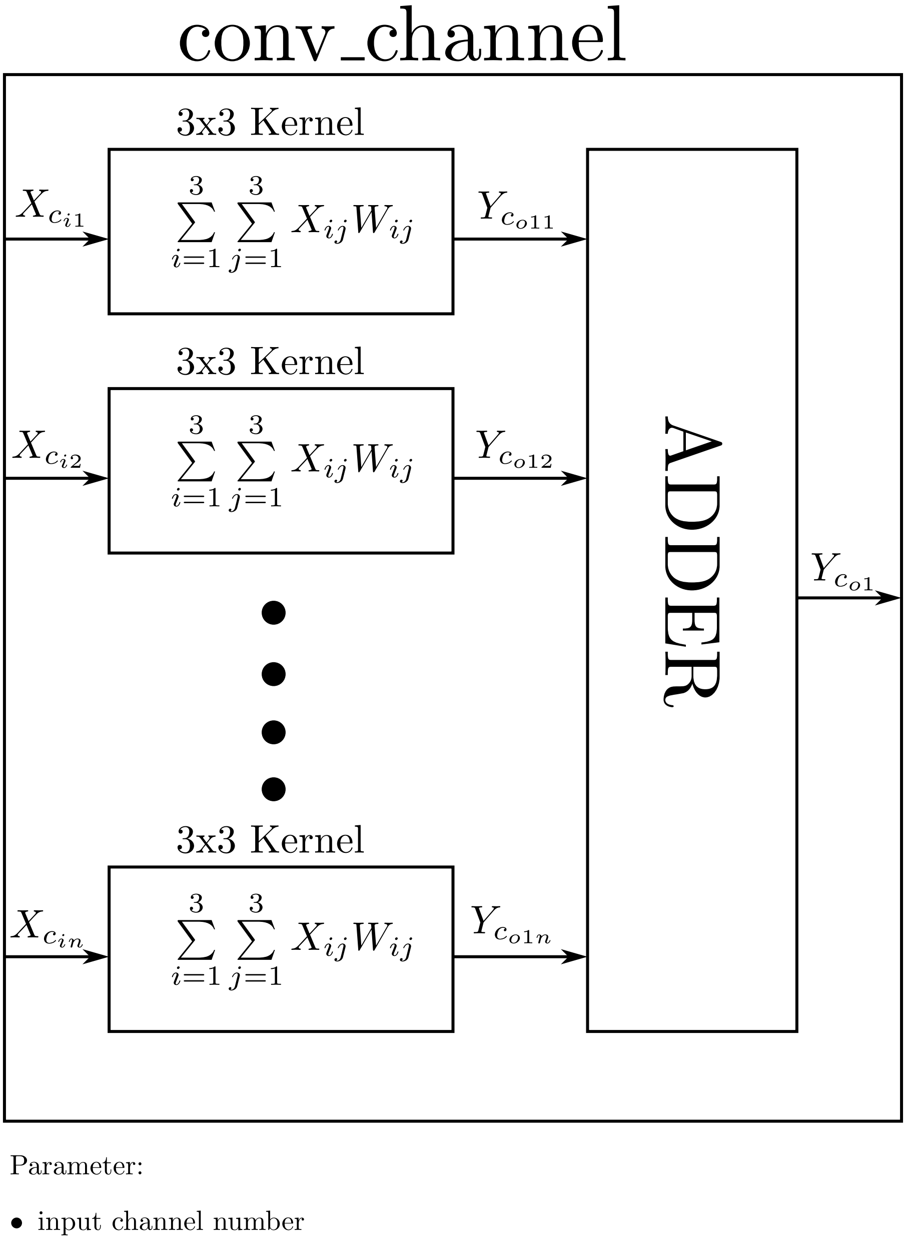

Figure 6: conv_channel block diagram. For each input channel a kernel_3x3 module is used. n indicates the number of input channels.

4.4.2 Parameter

- INPUT_CHANNEL_NUMBER : integer
- OUTPUT_CHANNEL_NUMBER : integer
- MATRIX_WIDTH: integer
- MATRIX_HEIGTH: integer

Appendix

Network Operations

Convolutional Operations

The output of an convolutional layer is defined by

$$z(i,j) = (f * g)(i,j) = \sum_{m=-\infty}^{\infty} \sum_{n=-\infty}^{\infty} f(m,n)g(m-i, n-j) \tag{2}$$

Fully Connected Layer

The output of an fully connected layer is defined by

$$z = xW + b \tag{3}$$

where $x \in \mathbb{R}^{b,m}$, $W \in \mathbb{R}^{m,n}$ and $b \in \mathbb{R}^{n}$. In short, this is the

Rectified Linear Unit (ReLU)

$$f(x) = \begin{cases} x & \text{if} \quad x > 0 \\ 0 & \text{else} \end{cases} \tag{4}$$

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:

$$\frac{\partial \mathbf{z}}{\partial \mathbf{x}} = \frac{\partial \mathbf{y}}{\partial \mathbf{x}} \frac{\partial \mathbf{z}}{\partial \mathbf{y}} \tag{5}$$

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 2: Useful derivatives equations

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)

4.5 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

[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.

[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.