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
Dense (Layer)	A layer where each neuron is connected to every input neuron
Fully-Connected (Layer)	Same as *Dense*
Q	The significant of a fix point integer
m	the exponent 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] which 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 2.3 for details how the network is implemented in Software. An excellent overview in deep learning can be found in [Schmidhuber, 2015] and also in [Goodfellow et al., 2016]. 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 has 28-by-28 pixel.

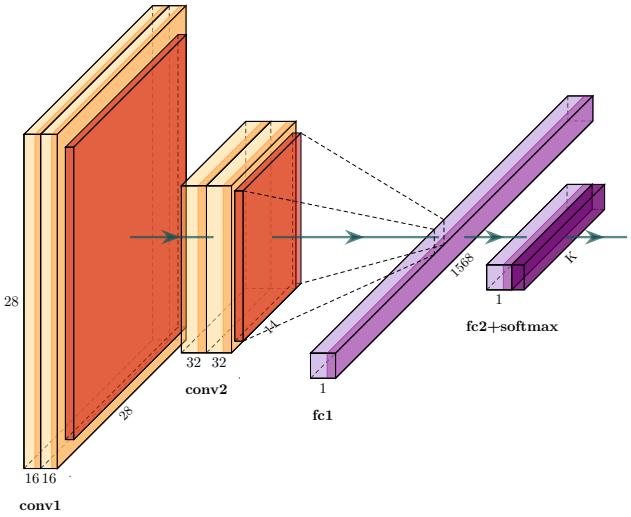


Figure 1: The Eggnet structure

2.2 Hardware Concept

Figure 2 shows the Concept of implementing an FPGA-based hardware accelerator for handwritten digit recognition. A Zedboard is used as hardware platform, which includes a Zynq-7000 FPGA and provides

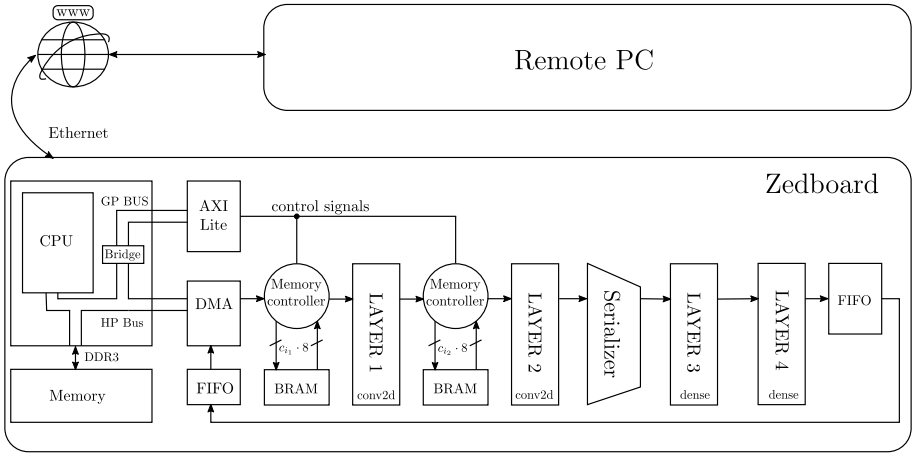


Figure 2: Top-Level concept

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.

2.3 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 quantization. The layers of the neural network are depicted in Figure 1. The network was trained on the *MNIST* dataset which consists of 60.000 images of handwritten digits. Those where split up in 50.000 images used for training and 10.000 used for evaluation. The network maps mathematically an 28×28 input image $x \in \mathbb{R}^{28,28}$ to an output of vector of probabilities $y \in \mathbb{R}^{10}$ where each value corresponds how likely the input image belongs to that class. The layers of the network are listed in Table 1.

Layer	Type	Data Size (output)	Purpose
1	Conv01	$[28, 28, 16]$	Extract features ($[3, 3]$ kernels)
2	ReLU	$[28, 28, 16]$	Introduce nonlinearity
3	Pool01	$[14, 14, 16]$	Introduce nonlinearity, reduce dimensions
4	Conv02	$[14, 14, 32]$	Extract features ($[3, 3]$ kernels)
5	ReLU	$[14, 14, 32]$	Introduce nonlinearity
6	Pool02	$[7, 7, 32]$	Introduce nonlinearity, reduce dimensions
7	Dense01	$[32]$	Combine features
8	ReLU	$[32]$	Introduce nonlinearity
9	Dense02	$[10]$	Combine features
10	Softmax	$[10]$	Normalize output

Table 1: Eggnet Layers

Further 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

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

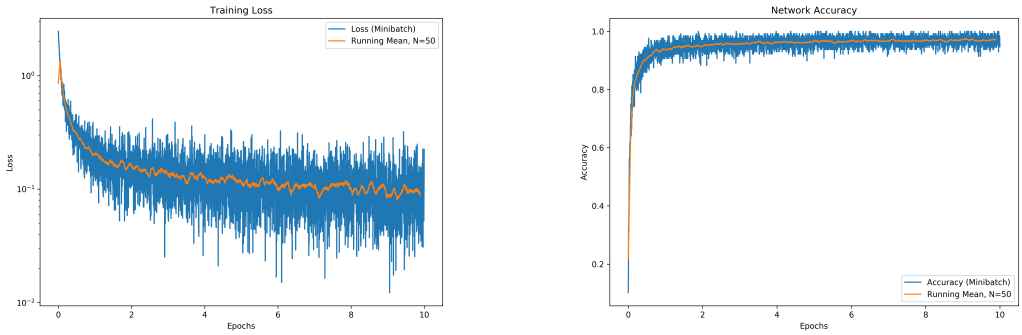
This loss function is standard for classification problems where each sample belongs to exactly one class. The ADAM algorithm can be adjusted by parameters, which control the speed of convergence. The recommended values, listed in Table 2, by [Kingma and Ba, 2014] were used.

Parameter	Value
α	0.001
β_1	0.9
β_2	0.999

Table 2: Network Training Parameters

A useful guide for implementing convolutions can be found in [Dumoulin and Visin, 2016]. The training of the network yielded very high accuracy rates that are typical for the MNIST dataset, which is an easy challenge for machine learning. 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.

Both, PyTorch and Tensorflow/Keras are Python libraries but most neural network (NN) operations are not implemented in Python directly because of performance reasons. This makes debugging more difficult which is why we reimplemented the operations we needed in Python. This was done using NumPy and SWIG, which will be discussed more extensively in Section 3.2, and enabled us to verify the correct ordering of parameters, weights and activations.



(a) Training Loss

(b) Training Accuracy

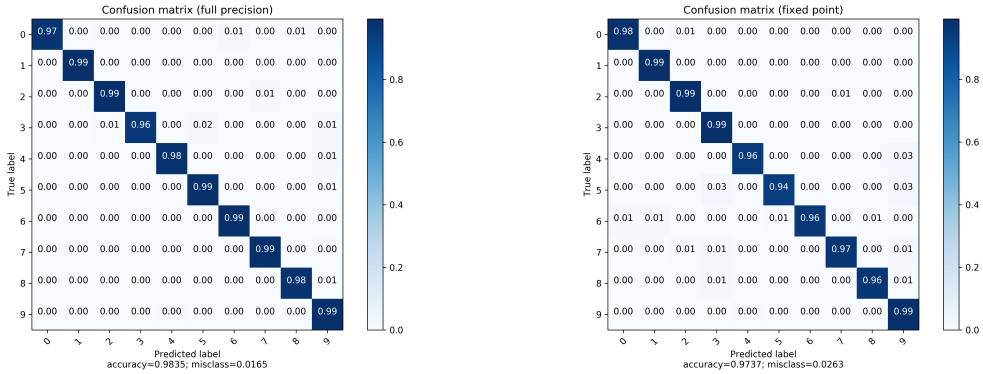
Figure 3: Network loss and accuracy over the training iterations. The blue lines show spikes which occur because of the randomly selected mini batches. The orange line shows the smoothed version over 50 periods

2.4 Quantization

The network is trained and created using 32 bit 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



(a) Floating Point

(b) Quantized Values

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

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 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 [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 analysing the weights of our 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 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 4 bit 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 3.

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 [Lee et al., 2017]. 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" [Zhu and Gupta, 2017]. This was not explicitly performed but is implicitly done by low bit quantization.

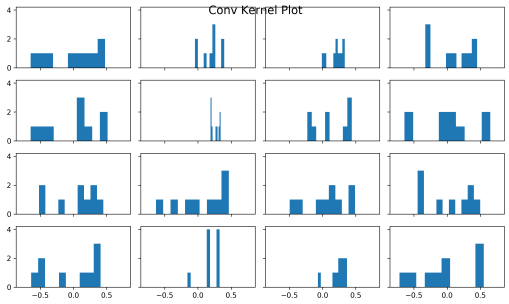
Network Part	$\lvert Q\rvert$	m	$\pm$	v (real value range)
Input	8	8	$+$	$[0, 1]$
L1: Weights	4	2	$\pm$	$[-2, 2]$
L1: Intermediates	12	10	$\pm$	$[-2, 2]$
L1: Accumulated	16	10	$\pm$	
L1 $\rightarrow$ L2	Rshift by $10-2$ and clip values in range $[0, 15]$			
L2: Input	4	2	$+$	$[-2, 2]$
L2: Weights	4	5	$\pm$	$[-0.5, 0.5]$
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 3: Quantization parameters for the 4 bit 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.

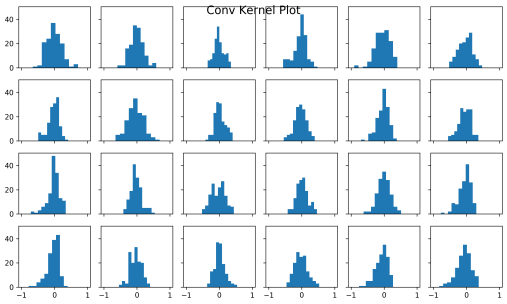
3 Software

To actually test the FPGA implementation of the neural network with data an effective software interface which handles the processing is needed. It should ease the sending of single or multiple images to the FPGA, control and handle the execution of it and collect the estimated results. This software should also take take care that the input data is processed in right order and arriving input images should be effectively queued to prevent errors from evaluating multiple images at once. Further the software should also handle any necessary high level tasks which are needed, e.g. the support of common image datatypes and their conversion to a suitable representation.

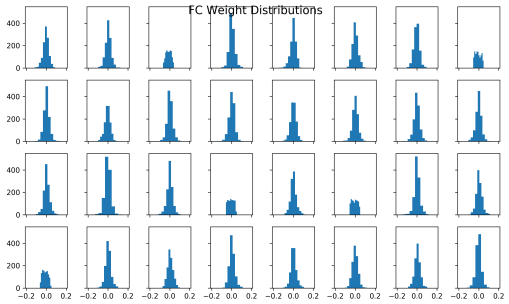
Another important target is that the network should have an interface to Python because it emerged to



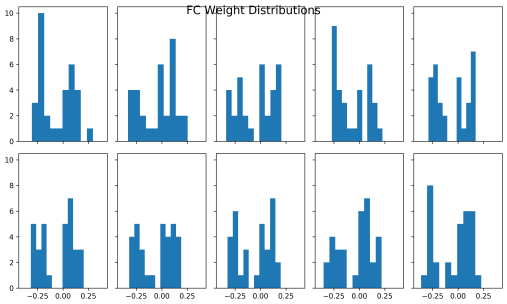
(a) Convolutional Layer 1



(b) Convolutional Layer 2

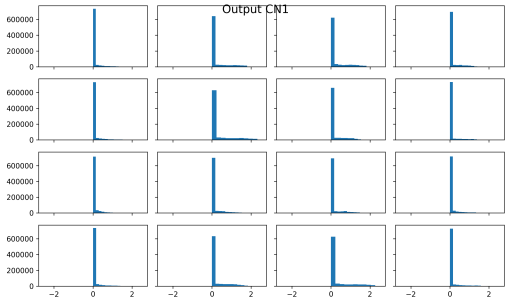


(c) Fully Connected Layer 1

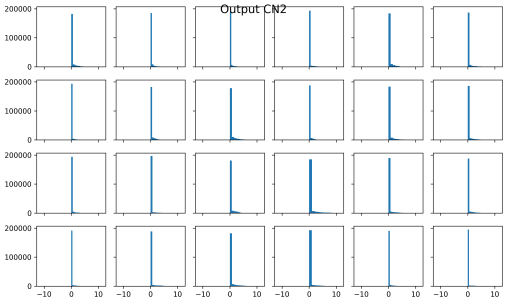


(d) Fully Connected Layer 2

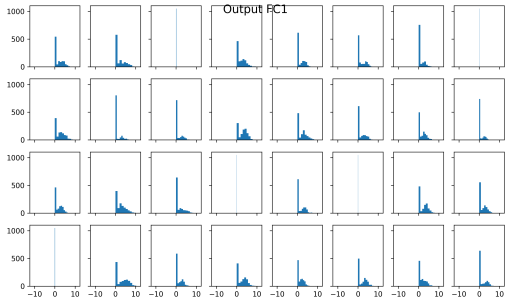
Figure 5: Distribution of the network weights for the different layers. It can be seen that the weights are distributed close to zero.



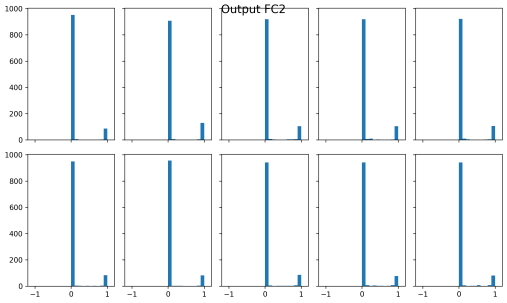
(a) Convolutional Layer 1



(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. The occurring minimum and maximum values were used to define the quantization bit width.

a de-facto standard in the Neural Network, Machine Learning Scientific community [Virtanen et al., 2020]. All the major development in Machine Learning is done in Python and because the Zedboard is able to run Linux and Python an implementation is possible. In the long term it would also be beneficial to control the network via Python and embed it in an existing Machine Learning Framework like Keras [Gulli and Pal, 2017] or Torch [Paszke et al., 2019]. This would enable an high level usage of FPGAs for neural network hardware acceleration.

Besides from the machine learning (ML) tasks the sparse resources of the Zedboard are conserved as much as possible. Therefore no graphical user interface (GUI) is implemented and communication happens exclusively via textual interfaces, e.g. Serial Connections or Ethernet. Transferring data of these textual interfaces is tedious which why also webapp is implemented as a compromise of ease of use and resource consumption.

The overall goals of the software layer therefore can be summarized as:

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

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 directly uses the Linux driver application programmable interface (API) and and 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. It is build on top of the Xilinx FPGA driver.
An important task that needs to handled by the driver is the synchronization of the FPGA and Linux. We use the POSIX standard *pthread* library which provides routines for mutual exclusion to ensure that the FPGA is not accessed from multiple threads at the same time. Further the driver communicates with the low-level Xilinx driver using the function `ioctl()` which blocks the thread until the releasing hardware interrupt occurs. In our case the blocking of the thread is acceptable because multithreading is handled on a higher abstraction level. Copying of the data from user space to the FPGA is done via the C function[1].
Control and debug data is transferred via AXI-lite bus using UIO-device driver. It maps the corresponding AXI-lite bus address to a virtual memory address. This address can be used to read or write from the AXI-lite device.

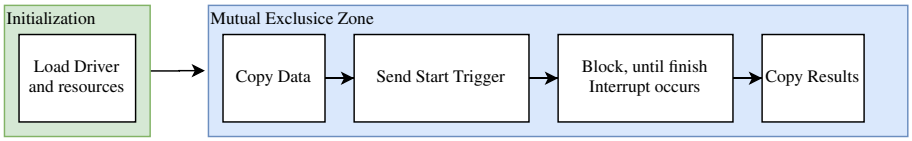


Figure 7: The network driver

Requirements of the User Layer Driver Software:

- Communication with the kernel space drivers
- 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
- Ability to debug the network, e.g. receive data from the intermediate layers

[1]Note: Still the driver need to check that not multiple threads access it at the same time.

3.2 Python Driver Interface

The Python driver interface is build on top of the field programmable gate array (FPGA) driver and should ease the usage of the driver functions for the python interface. A very common library and part of the scientific python package is *NumPy* [Virtanen et al., 2020]. It's basic element is the *NumPy Array* [Walt et al., 2011] which is a very useful abstraction and interface to high dimensional data used in ML contexts and a basic building block in most numeric scientific python programs. The Python driver should therefore receive NumPy arrays on the Python side and translate them for the C driver interface. This translation process can be automated by tools like e.g. *SWIG* [Beazley, 2003]. The (simplified) build process is depicted in Figure 8 which on its basis requires two files: First, a header file which contains the C function declearations that should be wrapped, which is the public interface of our driver and second, a special *SWIG* interface files, which contains further rules how the code in the header file should be wrapped. It is often beneficial to create the function declarations in the header files with Python and SWIG in mind. SWIG translates the function declarations to the Python domain and creates a `.py` file with convenience function definitions and a `.c` file which handles the communication between the C library and (C-)Python. The compilation of this file requires linking against platform specific libraries of the driver, the C library and Python. The whole process is done most conveniently via the Python Setuptools.

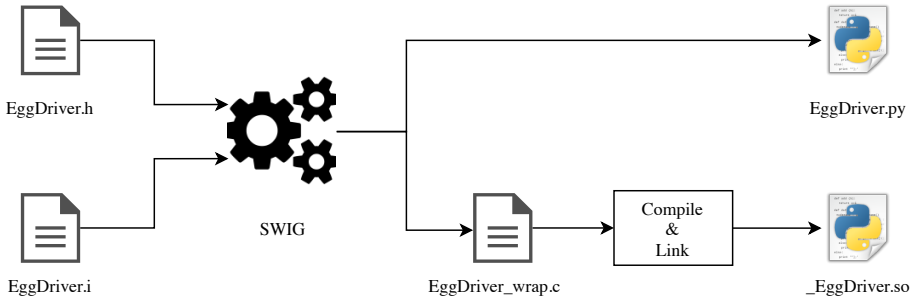


Figure 8: Build process of the Python Driver Interface (simplified)

3.3 Web Interface

The Webapp should provide an easy yet practical 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* [Pallets, 2020]. This enabled us the quick interfacing between web components and our python code and also made the communication of multiple threads easier.

The network speed can be tested and benchmarked to the CPU version (using floats or integers), see the Screenshot in Figure 9. Here different (sub-)datasets can be chosen as well as different batch sizes. It is also possible to upload a single image in various formats which is then resized to 28×28 pixels and converted to a NumPy array, which is ultimately transferred to the FPGA. Additionally the Webapp automatically queries system status information and displays them using interactive graphs. Most of the functionality is implemented using JavaScript and supporting libraries, e.g. *Chart.js* and *Vue.js*. This is very resource sparse because most of the work is done in the client web browser and only the API queries as well as the NN must be processed by the Zedboard.

Initially planned but *not* implemented was a possibility of updating of the bitstream via the Webapp. This would have enabled to quickly benchmark various architectures against each other as long as the all conform to common driver interface.

3.4 Build Software and Setup

Additionally to the software needed for operating the network various tools are needed to actually *build* the network itself. 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.

Requirements of the Training Software:

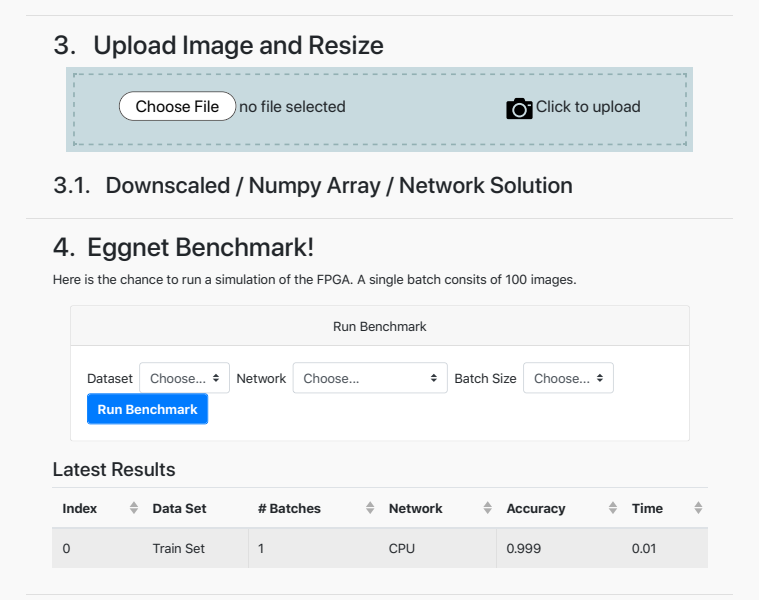


Figure 9: Screenshot of the Webapp's benchmark interface

- 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 Training Software:

- Sends image data to Zedboard
- Receives results from Zedboard
- Create a figure of accuracy and performance
- Optional: Send bitstream to hardware for partial reconfiguration

3.4.1 Interface to Zedboard

As an 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.

4 Hardware

Neural Networks consist of distributed memory and computational elements which is in stark contrast to common von-Neumann architectures. Implementing NN in von-Neumann architectural style is therefore not practical and the overall efficiency can be increased if the circuits resemble the structure of the network. This can be seen in the top level schematic in Figure 2.

In general the implementation of the network followed the same structure as in Table 1. Directly implementing an extra layer for ReLU is not practical and it is therefore implicitly implemented in the preceding layer as part of the quantization clipping (see Section 2.4). Also in contrast to the evaluation in software the hardware layer do not receive the whole data at once, instead it is streamed through the network in small chunks (with the exception of the Fully-Connected layer, see Section 4.6). Further to control the streaming of data through the network, the common interface for each layer is the AXI interface. This is exemplified in Table 4.

Parameter Name	VHDL Type	Direction	Description
s_Clk_i	std_logic	Input	Clock input
s_n_Res_i.	std_logic	Input	Reset input (active low)
s_Valid_i	std_logic	Input	Flag, '1' if the input is valid
s_Ready_o	std_logic	Output	Flag, signals to the previous layer that it is ready to receive data
s_Last_i	std_logic	Input	Flag, signals end of a data 'packet', i.e. image line, 10-byte output

Table 4: Neural Network Layer AXI Slave-Interface for VHDL Entities

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 which can be used for debug purposes. Each memory controller is assigned a unique address via generics in VHDL. 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. In Table 5 is shown how the 32 bit are separated.

	Bit Address	Comment
BRAM	23 downto 0	Address of the block RAM
32 bit vector	27 downto 24	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	31 downto 28	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.

Table 5: AXI Lite component Address

4.3 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 is parameterized with its own set of values. Those values are implemented as constants in VHDL, which has the advantages of greater optimization options by the compiler as well as reduced data movement. Therefore template entities are created for the convolution channel- and layer entities where the weights are then populated via a Python script.

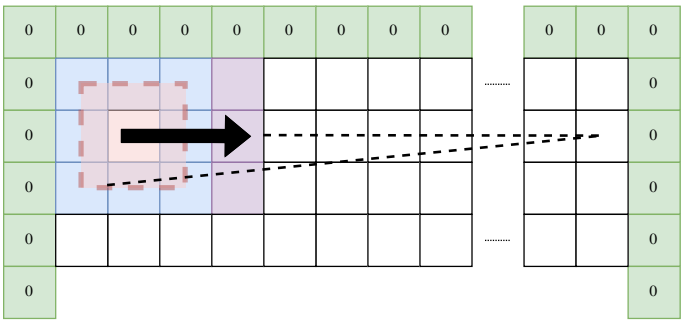


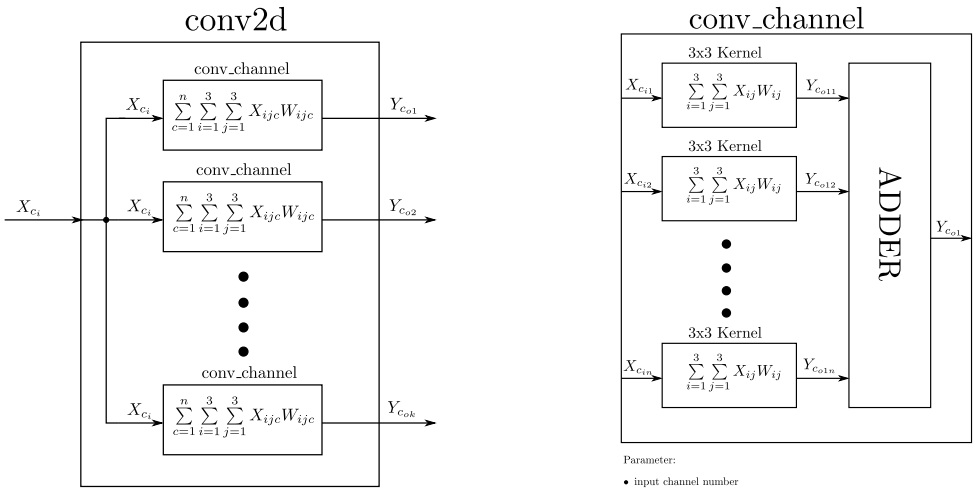
Figure 10: Convolution Operation. The convolutional kernel (red-dashed-square) is shifted over the input data. The currently processed data is shown in blue and the only three pixels (violett) need to be loaded for the next convolution because of data reuse. Note that for the convolution operation the image must be padded (in our case by zeros) to sustain image dimensions. Depth channels omitted for illustration purposes.

The principal operation can be shown in Figure 12. Here the convolutional kernel slides over the input data from left to right, top to bottom. It can be seen that most of the 6 of 9 input pixels (of each input channel) can be reused. This is done via shift registers and because all weights of the kernel remain unchanged for the operation this results in minimum data movement. Note that for the convolutional operations the input data must be padded in order to sustain the image dimensions. If a new line occurs, two cycles are required to load a new 3x3 kernel instead of one as an additional 3x1 vector needs to be loaded from memory.

In Figure 11a the block diagram of the top-level conv2d module is shown. It consists of k conv_channel modules to realise k output channels. All conv_channel modules recieve the same input vector X_{c_i}. The internal structure of a conv_channel module is shown in Figure 11b. 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.

4.3.1 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.



(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 11: Block diagram of the Convolutional Layer

- 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

Parameter	VHDL Datatype	Type
BIT_WIDTH_IN	integer	Generic
BIT_WIDTH_OUT	integer	Generic
INPUT_CHANNELS	integer	Generic
OUTPUT_CHANNELS	integer	Generic

Convolution 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

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. The weights are specified in a single generic constant array in row-major notation.

4.3.3 Interface

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

4.3.4 Parameter

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

4.4 Max Pooling Layer

After the convolutional layer a pooling layer is placed. This reduces the dimensionality of the data and introduces local non-linearity to the network, enabling it to achieve better function approximation. In our case the layer performs a max operation among a 2×2 image patch. This means that a 2-by-2 image patch is mapped to 1 output pixel and it's value is equivalent to the maximum value of the input image patch. Because of the processing of the convolutional layer (see Figure 12) a whole input image input row must be buffered before any processing can happen. Because the pooling layer requires a 2-by-2 input window it is limited by the output of the convolutional layer. To increase throughput the convolutional layer could be parallelized by doing two convolutions at once at the cost of increased area size. Alternatively the pooling operation can be serialized by only operating on half the output channels of the convolutional layer at a time.

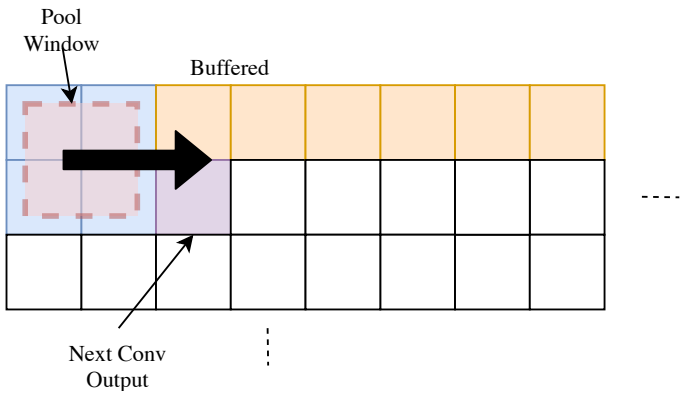


Figure 12: Pooling Operation. The pool window (red-dashed-square) applied to current image patch (blue) and shifted over the input data. The first input row needs to be buffered (orange) before processing can happen. Depth channels omitted for illustration purposes.

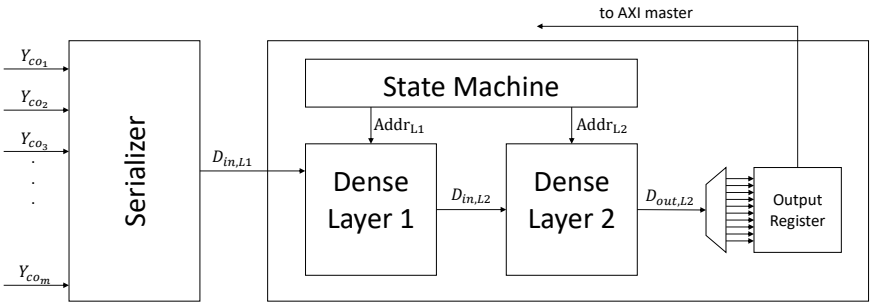


Figure 13: Diagram of the combined, fully connected NN.

4.5 Fully-Connected Neural Network

4.5.1 Operation

The fully-connected neural network is shown in Figure 13. 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 is a 10-value vector which represent the confidence that the input image showed a specific numeral.

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

4.5.2 Interface

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

4.5.3 Parameter

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

4.6 Dense Layer

4.6.1 Operation

Figure 14 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 ROM one by one. Each of the input neurons is multiplied by appropriate weight for each of the output neurons. The products are then fed into accumulators, which calculate a sum of all products of all neurons with the inputs. When all of the incoming neurons are processed, the calculation is finished and a Finished_o signal 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.

4.6.2 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.

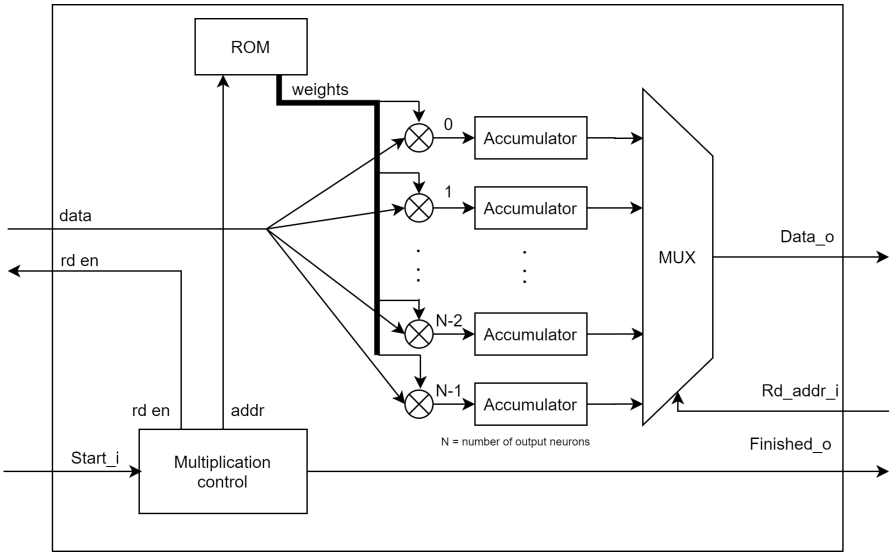


Figure 14: Dense layer diagram.

4.6.3 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.

4.6.4 Parameters

VECTOR_WIDTH : integer Bit width of input data.

INPUT_COUNT : integer Number of input neurons

OUTPUT_COUNT : integer 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 Appendix

5.1 Network Operations

Convolutional Layer

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: [Dumoulin and Visin, 2016]

Pooling Layer

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

For a vector $x \in \mathbb{R}^n$ the softmax function is defined as

$$f_{\text{Softmax}} : f(x) = \frac{\exp x_i}{\sum_i^n \exp x_i} \tag{7}$$

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 6: 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 [Walt et al., 2011], SWIG [Beazley, 2003], Keras [Gulli and Pal, 2017] and Torch [Paszke et al., 2019]

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`

List of Figures

List of Tables

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.

[Beazley, 2003] Beazley, D. M. (2003). Automated scientific software scripting with swig. *Future Generation Computer Systems*, 19(5):599–609.

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

[Goodfellow et al., 2016] Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

[Gulli and Pal, 2017] Gulli, A. and Pal, S. (2017). *Deep learning with Keras*. Packt Publishing Ltd.

[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.

[Lee et al., 2017] Lee, E. H., Miyashita, D., Chai, E., Murmann, B., and Wong, S. S. (2017). Lognet: Energy-efficient neural networks using logarithmic computation. In *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 5900–5904.

[Pallets, 2020] Pallets (2020). Flask. [Online; accessed 9. Apr. 2020].

[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.

[Virtanen et al., 2020] Virtanen, P., Gommers, R., Oliphant, T. E., Haberland, M., Reddy, T., Cournapeau, D., Burovski, E., Peterson, P., Weckesser, W., Bright, J., van der Walt, S. J., Brett, M., Wilson, J., Jarrod Millman, K., Mayorov, N., Nelson, A. R. J., Jones, E., Kern, R., Larson, E., Carey, C., Polat, İ., Feng, Y., Moore, E. W., Vand erPlas, J., Laxalde, D., Perktold, J., Cimrman, R., Henriksen, I., Quintero, E. A., Harris, C. R., Archibald, A. M., Ribeiro, A. H., Pedregosa, F., van Mulbregt, P., and Contributors, S. . . (2020). SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272.

[Walt et al., 2011] Walt, S. v. d., Colbert, S. C., and Varoquaux, G. (2011). The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30.

[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*.

[Zhu and Gupta, 2017] Zhu, M. and Gupta, S. (2017). To prune, or not to prune: exploring the efficacy of pruning for model compression. *arXiv preprint arXiv:1710.01878*.