University of Manchester
School of Computer Science
Project Report 2017

Hardware Acceleration of Deep Neural Networks using Tensorflow

Author: Oliver Kugel

Supervisor: Dr. Dirk Koch

Abstract

Hardware Acceleration of Deep Neural Networks using Tensorflow

Author: Oliver Kugel

With interest in machine learning increasing every year, focus should not only be layed on inventing and improving algorithms but also on the choice of hardware these algorithms run on. The aim of this project therefore is to investigate hardware-acceleration of Artifcial Neural Networks, one of the most common and successful models in the field of machine learning. The Neural Network to be operated on is Inception-v3, a model developed by Google as part of their open-source machine learning library *Tensorflow*. Certain operations of this model can be mapped to hardware, thereby utilizing a more *taylored* hardware module for calculations rather than processing on a general-purpose processor. The implementation and the results of testing such modules are presented. Manipulating a trained Neural Network to make it accessible by native hardware is a main topic of the project. Results shall be presented regarding a system that allows a Neural Network to be arbitrarily *split*, with parts of it running on software and others in hardware simulations.

Supervisor: Dr. Dirk Koch

Acknowledgements

I would like to thank my supervisor Dr. Dirk Koch for his support and advice concerning this project. Equally, I would like to thank PhD student Malte Vesper for the great patience and practical help he has given me during our meetings.
I also want to thank my parents.

Contents

1 Context **3**
- 1.1 What is an Artificial Neural Network? . . . 3
- 1.2 Tensorflow and Inception-v3 . . . 3
- 1.3 Idea of hardware-accelerating the ANN . . . 5
- 1.4 Idea of serializing the ANN . . . 5

2 Development and Implementation **7**
- 2.1 Computation Cell Design . . . 7
- 2.2 Dataflow Scheduling . . . 7
- 2.3 Inception-v3 Graph: Reduction and Quantization . . . 9
- 2.4 Inception-v3 Graph: Splitting and Data Extraction . . . 10
- 2.5 Hardware-Software Interfacing . . . 11
- 2.6 Convolution Module Implementation . . . 12
- 2.7 Applied Methodologies . . . 13

3 Evaluation and Testing **14**
- 3.1 Graph Analyser Script . . . 14
- 3.2 Verilog Testbenches . . . 14
- 3.3 Simulating Erroneous Input Data . . . 15
- 3.4 Accuracy and Speed Comparison . . . 15

4 Reflection and Conclusion **18**
- 4.1 Planning and Scheduling the Project . . . 18
- 4.2 Understanding Tensorflow's Codebase . . . 18
- 4.3 What went well, what didn't? . . . 19

References **21**

A Example of operation **23**
- A.1 Graph-Execution Script . . . 23
  - A.1.1 Input . . . 23
  - A.1.2 Output . . . 23

Chapter 1

Context

1.1 What is an Artificial Neural Network?

An Artificial Neural Network (ANN) constitutes a computational model that emerged from the field of machine learning in the 1940s [3]. As the name suggests, the model consists of a network of interlinked artificial neural nodes. "Artificial Neural" in the sense that they vaguely mimic the kind of biological nodes of the human brain. A "neural node" in an ANN is best described as a tiny computation cell that performs some calculation on its input data. Mostly this calculation is a simple dot-product calculation. Each node in an ANN has a set of weights associated with it. These weights (w) are multiplied with their corresponding input data values (i) and then summed together: $dotproduct = w_0 i_0 + w_1 i_1 + ... + w_n i_n$.
The purpose of the weights is to later be able to tune the network by adjusting them. All nodes of an ANN are grouped into layers. The first layer is called the *input layer*, the last one the *output layer* and the ones inbetween are so-called *hidden layers*. Nodes linked together make up the actual network. Figure 1.1 shows an ANN with two hidden layers. As shown in the figure, only the input layer receives its values from outside the network. The nodes in all subsequent layers simply use the outputs (i.e. the dotproducts) of their preceding layer as their inputs. There are two typical phases to an ANN: the *training phase* and the *testing phase*. Since the weights in an ANN are initially set to random values, there needs to be a mechanism for setting those weights to values that maximise the ANNs performance, e.g. to achieve the highest accuracy for image classification tasks. This is what happens in the training phase. The ANN is fed with input data (e.g. images) and then checked against an available "ground truth". Based on this, the network's error is then backtracked to the individual nodes. After that, the weights of the affected nodes are adjusted to compensate the error and to achieve a better performance the next time around. Once the ANN is trained, it can then be used in the *testing phase* (or "classifying phase") where no more ground truth is available [6]. In the example of image classification, an ANN in its testing phase uses its set parameters (weights) to classify unlabelled images that it has never seen before [3].

1.2 Tensorflow and Inception-v3

Because of the rising interest in machine learning techniques, expressed from both industry and academia [7], and the difficulty of correctly implementing the variety of machine learning algorithms that had been discovered, there was a general desire in the early 2000s for a unified

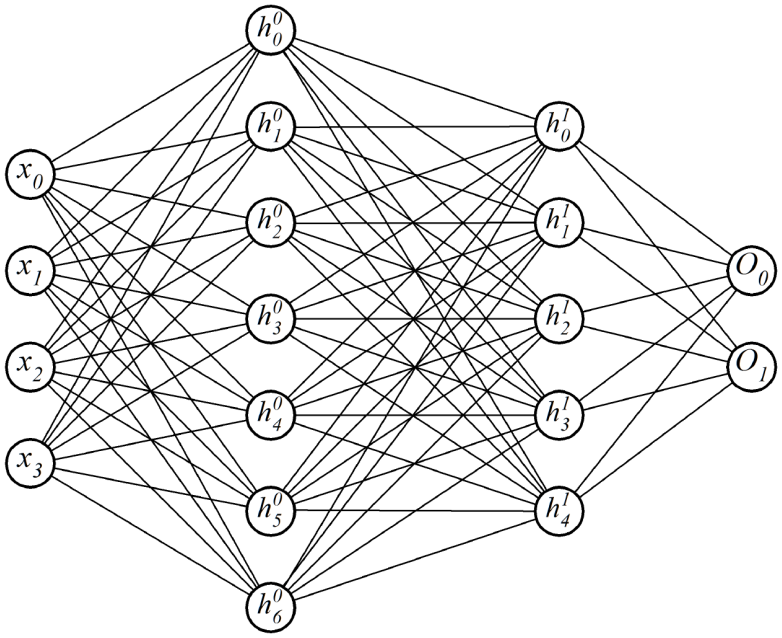

Figure 1.1: Artificial Neural Network with two hidden layers, from [4].

open-source machine learning library. The obvious advantage of this is being able to quickly make use of machine learning techniques for a software system without having to spent much time and resources on set up and implementation [16]. Because of this demand, Google in November 2015 released **TensorFlow**, an open-source software library for machine learning that provides code, APIs, documentation and tutorials for the most common machine learning algorithms.
One particularly useful tool that comes with installing TensorFlow is a complete, trained, convolutional neural network model named **Inception-v3**. The model is trained on a large image dataset and is capable of classifying images into 1000 different classes with a top-5 error rate of just 3.46% [10]. Figure 1.2 shows the general structure of this Convolutional Neural Network (CNN). It is important to note the difference between this neural network and the one shown before in Figure 1.1. As opposed to a conventional ANN, a CNN uses an operation called convolution to analyse input data such as images. Convolution can be thought of as a window of specific values that slides over the image from the top-left to the bottom-right corner. Depending on the values the convolution window is populated with, different classification tasks and accuracies can be achieved. This operation is equivalent to the dot-product calculation with weights that was described in section 1.1. In addition to this operation, Inception-v3 comprises operations for finding averages and max values in a window, for concatenation of results, as well as an operation that serves as an activation function (Softmax).

The images that trained the Inception-v3 network and that were also used as classification examples for this project came from ImageNet [8], a large online dataset of formatted images very commonly used in academia and industry to train and compare Neural Networks.

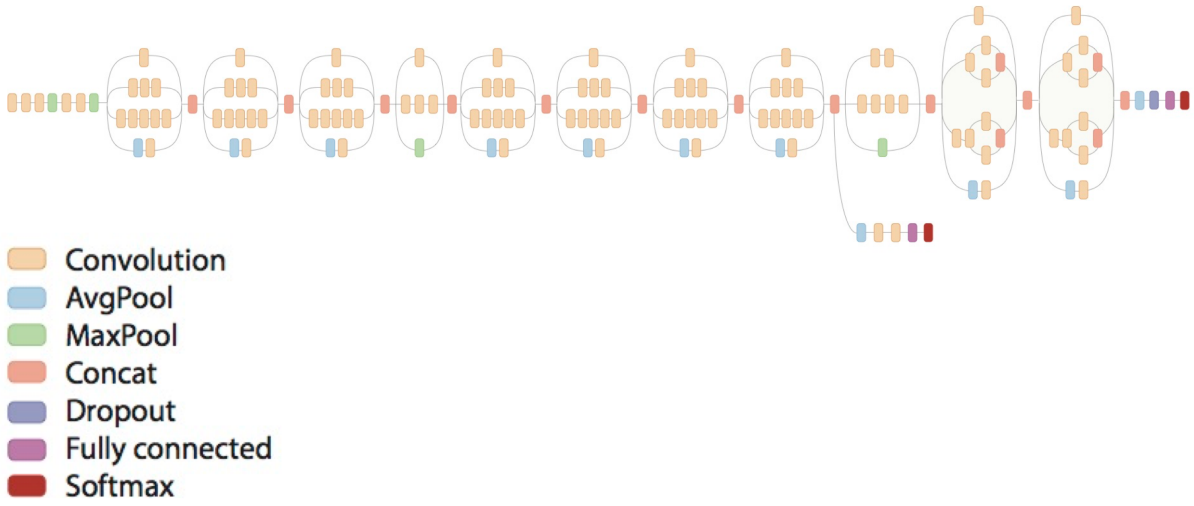

Figure 1.2: Convolutional Neural Network Model: Inception-v3, from [11]

1.3 Idea of hardware-accelerating the ANN

As the output of the graph analyser script (described in section 3.1) shows, the calculation graph of the original Inception-v3 model contains 23885411 weights. Roughly 24 Million integers make up around $(24 * 10^6 * 32)/8$ bytes $= 96 * 10^6$ bytes = 96MB of memory storage. However, it will be shown that this number can further be reduced by applying techniques of reduction and quantization, as discussed in section 2.3. With applying quantization, we use 8 bits for weights instead of 32. This brings it down to $24 * 10^6 = 24$MB of memory storage. After reduction, this number becomes roughly 20MB. Per chip, the FPGA possesses around 5MB. This means that parts of the ANN can be processed on the FPGA with all weights kept on-chip. With plugging several FPGAs together, using the interface architecture JetStream [13], even the entire ANN can be mapped onto FPGAs. In addition to this fact, it was shown that an ANN, as well as a CNN, are based on very simple operations (mainly dot product calculations) that take place numerous times after another and in parallel. These simple, number-crunching-heavy and parallelizable operations are ideal for an FPGA, and actually less ideal for the kind of general purpose processors that ANNs normally run on. GPUs would do reasonably well but the ability to chain things make FPGAs a more ideal solution. For these reasons, it is a sensible idea to map a Neural Network, or even just parts of it, onto an FPGA. The implementation of this idea shall be discussed in chapter 2.

1.4 Idea of serializing the ANN

Implementing an ANN such as the one in Figure 1.1 on an FPGA is problematic. The difficulty lies in a phenomenon called *high fan-out*. Looking at cell x_0 in Figure 1.1, we see that this value needs to be forwarded to 7 different cells in the subsequent layer. In fact, each cell in the network, apart from the ones in the output layer, needs to forward its result to each other cell in the layer after it. With a network this small, the problem might not be obvious but in a real-life ANN with more than 10 layers and hundreds of cells, the problem of having to distribute results to thousand other places becomes significant. Another issue is that cells that

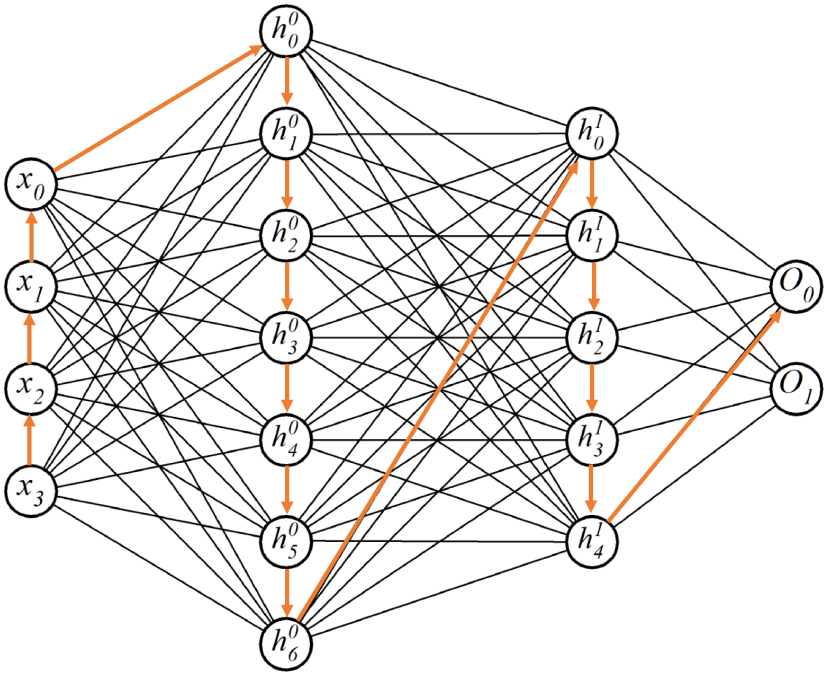

Figure 1.3: Computational Flow in Serialized Neural Network, from [4] (modified).

conceptually lie right beside each other might end up physically located at two opposite ends of the chip. However, this problem of high fan-out can be resolved with a simple yet powerful idea: serializing the ANN [4]. Figure 1.3 shows the computational flow in a serialized neural network, details of which shall be discussed in section 2.2.

Chapter 2

Development and Implementation

2.1 Computation Cell Design

The first step to implementing an ANN such as the one in Figure 1.1 in hardware is to think about the design of the computation cells that the ANN consists of. Figure 2.1 shows the RTL design of such a cell. The key to understanding the design is the observation that each cell can either forward a result (which is the dot-product of its inputs and its weights) or propagate some input-data coming from previous cells. This is the reason for having the two registers **result_reg** and **streamOut.data** controlled by a multiplexer. More obvious is the usage of a multiply-accumulate unit, which is the component that performs the actual dot-product calculation. Moreover, it needs a number of control signals to control the dataflow and computation scheduling (see section 2.2 for details on this). With the approach of allowing cells to forward results as well as input-data, we implement serialization of the ANN. Each cell now only has to forward its result (or input-data) to its successor rather than to every single cell in the next layer. This solves the problem of high fan-out. By plugging several of these cells together, we then get a computation pipeline equivalent to the one in the original ANN, but with the added flexibility of being able to split the pipeline at any stage. This is useful, for example for distributing the ANN onto several FPGAs, each only processing a piece of the serialized ANN, but with the combined memory of multiple FPGAs in order to hold all necessary weights on-chip. FPGA-to-FPGA communication could be done using PCIe links. Figure 2.2 shows a visual representation of this idea.

2.2 Dataflow Scheduling

The idea of serializing the ANN is beneficial. However, this approach leads to a number of challenges, most important of which being the problem of dataflow scheduling. In other words: how to get input-data and results that are computed and forwarded in every clock cycle into the right places without ever overriding data. To illustrate a solution to this problem, Figure 2.3 shows an example ANN with just one hidden layer and Figure 2.4 a table of timeslots showing how data would flow through it. The grey x-symbol in figure 2.4 means *unknown data*. Green means the data item in this result-register at this point in time is a complete dot-product (i.e. a "result"). Orange means the data item is a result, but one that lies in the cell's input-data-register and that needs to be forwarded, not treated as regular input-data. Regular input-data for the use in dot-products are the white boxes. To give an example: The **17** that ends up in

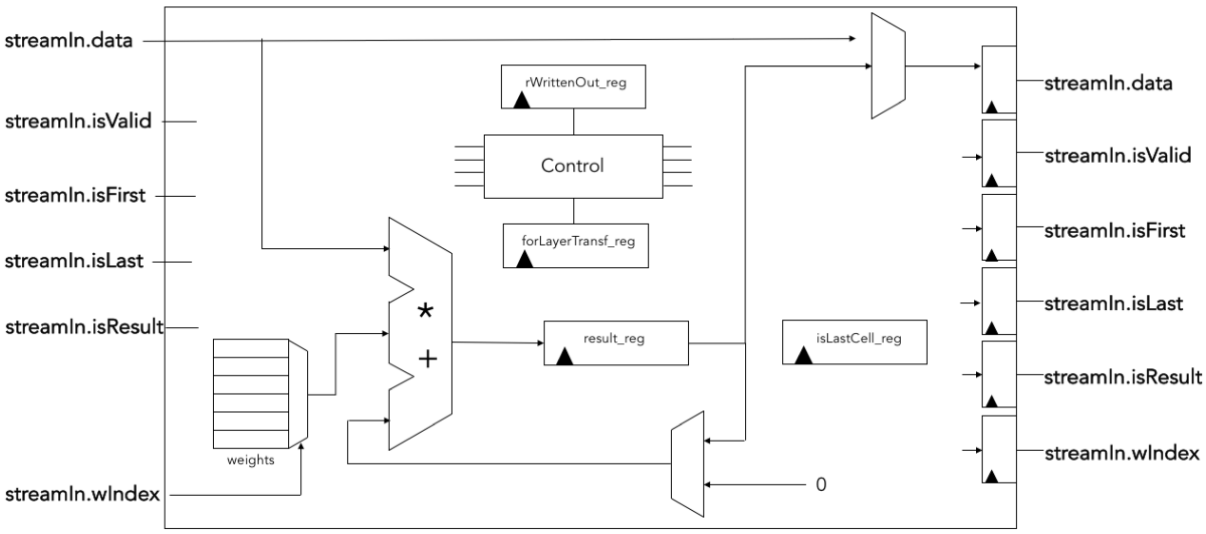

Figure 2.1: RTL Design of an ANN's Computation Cell

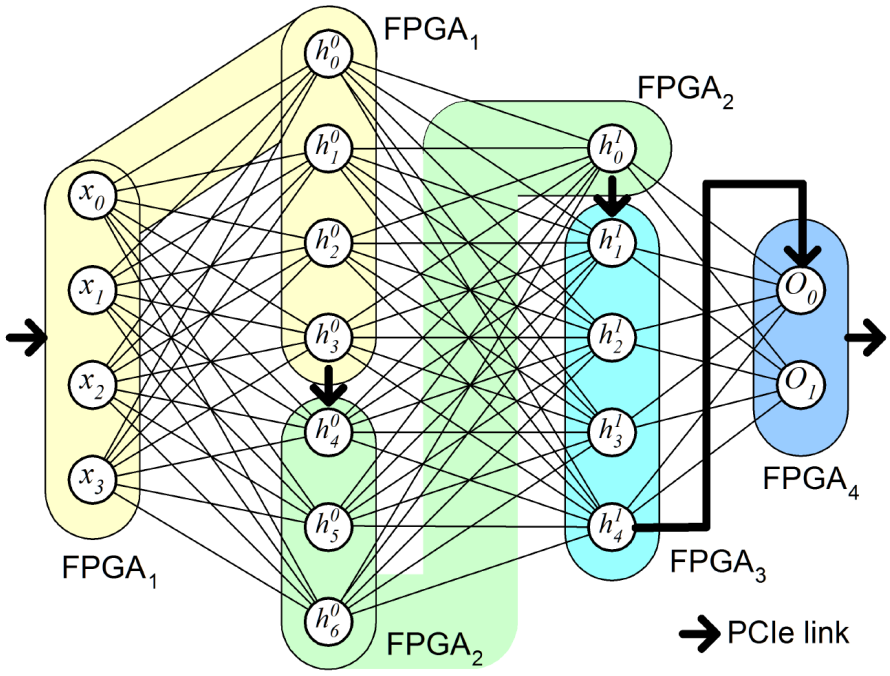

Figure 2.2: Computation Pipeline using multiple FPGAs, from [4].

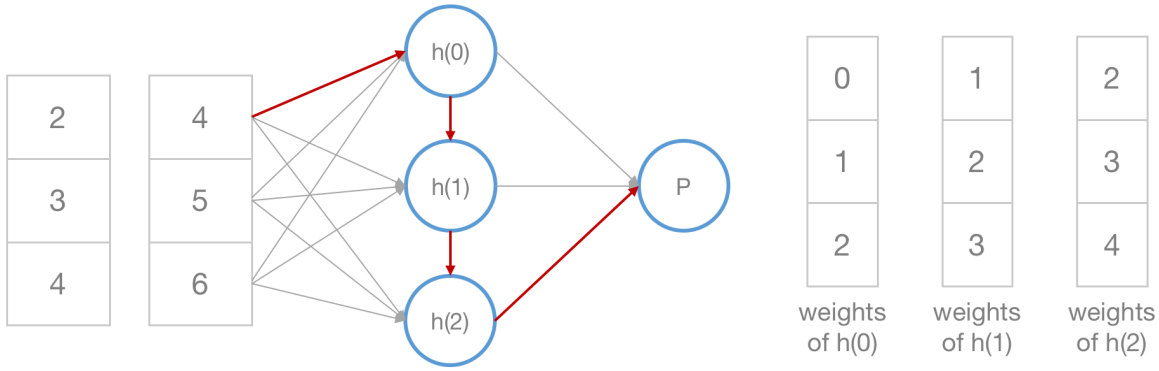

Figure 2.3: Example ANN with Input-Data and Weights

	T	t(0)	t(1)	t(2)	t(3)	t(4)	t(5)	t(6)	t(7)	t(8)	t(9)	t(10)	t(11)	t(12)	t(13)
H0	streamIn.data	4	5	6	x		2	3	4	x		...	...	...	x
	result_reg	x	0	5	17	17	17	0	3	11	11	11	...	...	...
H1	streamIn.data	x	4	5	6	17	x	2	3	4	11	x	...	...	...
	result_reg	x	x	4	14	32	32	32	2	8	20	20	20	...	...
H2	streamIn.data	x	x	4	5	6	17	32	2	3	4	11	20	...	...
	result_reg	x	x	x	8	23	47	47	47	4	13	29	29	29	...
P0	streamIn.data	x	x	x	x	x	x	17	32	47	x	x	11	20	29
	result_reg														

Figure 2.4: Dataflow Scheduling for Example ANN

P0's data-register at timeslot t(6) results from H0's input data **4**, **5** and **6** and its weights **0**, **1** and **2** since $4*0+5*1+6*2=0+5+12=17$. This data-item is available at t(3) and is then forwarded to cell H1 where it arrives at t(4). At t(5), it arrives at H2 and eventually gets to P0 (in the subsequent layer) at t(6).

With this kind of scheduling, no data is ever overwritten. What allows this to work is the use of delay slots. As shown in Figure 2.4, after one data-vector was fed to H0, the first item of the next data-vector has to wait for 2 timeslots. Generally, data has to wait for **input vector size - 1** slots. During these timeslots, no computation takes place in some cells, which can be criticised as "not keeping the pipe full". With an ANN this small, the problem is barely noticeable, but with bigger ANNs, this approach can become problematic.

2.3 Inception-v3 Graph: Reduction and Quantization

As mentioned in Section 1.2, the ANN that is described by the Inception-v3 Graph (Figure 1.2) is actually a Convolutional Neural Network (CNN). Just like an ANN, a CNN also has a training and a testing phase. Because of that, the full graph of the Inception-v3 network includes plenty of nodes that are only needed during the training phase. When mapping this CNN onto hardware, we are not interested in these nodes anymore. Therefore, it is possible to simply remove those unnecessary nodes from the graph [9]. This is useful, firstly because it reduces the total number of total weights needed to be kept on-chip and secondly because it speeds up the computation pipeline.

On top of that, there is another technique that can be applied to the graph in order to make it smaller. The smaller we can make the graph without losing functionality, the easier and faster it gets for the hardware implementation. This second technique is called *quantization*. Looking at all given weights in the graph, it can be noted that they are all integers within a certain numerical range. However, research has shown that for the representation of weights in a Neural Network, 8 bits are often enough [14]. Because of this finding, it is sensible to run the Inception-v3 graph through a script that maps all of the contained weights (32-bit integers) to 8-bit values. The advantage of this is self-evident: the storage that weights occupy in the hardware implementation can be reduced by a quarter, which is a significant gain [15]. To which extent quantization influences the Neural Network's accuracy and performance shall be examined in the evaluation chapter, in section 3.4.

2.4 Inception-v3 Graph: Splitting and Data Extraction

Since the goal of this project is to map (parts of) a neural network onto an FPGA, there needs to be a way to extract data from the calculation graph. Ideally, there should be a way of taking the provided network (described by the Inception-v3 graph), running it through a script and splitting it at any arbitrary node to view its intermediate results. These results could then be extracted, written to a file, and provided as inputs to a native hardware module. For example: Looking at the graph (Figure 1.2), we could run it in its native environment (on a conventional general purpose processor) *until* the fourth (green) node, then export the data (the "intermediate results") at this stage, use them as inputs to a hardware convolution module (implemented in SystemVerilog, see 2.6), run the following two (orange) nodes on the hardware module, then take the results from there and reinject them into the graph, which then runs to completion until the last node.

Writing a script that performs exactly this and that takes care of always presenting data in the right format (to make it compatible to both the software environment and the hardware modules) was a significant challenge. Since this script constitutes a major part of the project, there shall be a brief description of a crucial code snippet to illustrate how the desired functionality was implemented.

```
1  std::vector<Tensor> outputs;
2  std::vector<Tensor> filterTensors;
3  ------------------------------------------------------------------------------
4  string_flags.at(1) = GRAPH_FILE_ORI;
5  string_flags.at(5) = "DecodeJpeg/contents";
6  string_flags.at(6) = "Mul";
7  outputs = runPartOfGraph(string_flags, int_flags, "tensor0.data", outputs);
8  ------------------------------------------------------------------------------
9  string_flags.at(1) = GRAPH_FILE_QTR;
10 string_flags.at(5) = "Mul";
11 string_flags.at(6) = "conv";
12 outputs = runPartOfGraph(string_flags, int_flags, "tensor1.data", outputs);
13 ------------------------------------------------------------------------------
14 string_flags.at(5) = "conv";
15 string_flags.at(6) = "conv/conv2d_params";
16 filterTensors = runPartOfGraph(string_flags, int_flags,
```

```
17                                  "filter.data", outputs);
18 ------------------------------------------------------------------------------
19 // conv ------> conv_1  is processed through a hardware module
20 string_flags.at(5) = "conv";
21 string_flags.at(6) = "conv_1";
22 outputs = getHardwareResults(string_flags, int_flags,
23                               "hardware_out.data", outputs);
24 ------------------------------------------------------------------------------
25 string_flags.at(5) = "conv_1";
26 string_flags.at(6) = "softmax";
27 outputs = runPartOfGraph(string_flags, int_flags, "tensor2.data", outputs);
```

This code snippet, together with the two functions **runPartOfGraph()** and **getHardwareResults()** implements **splitting** of the graph as well as **data extraction**.

- Lines **4** and **9** define which graph is being manipulated, where ORI stands for "original graph" and QTR means "quantized, then reduced graph".
- This graph file is then used in function calls to **runPartOfGraph()** and **getHardwareResults()**, together with two arguments describing start-node and stop-node, e.g. **Mul** and **conv**.
- The graph is then run from start-node to stop-node and the intermediate results of the graph immediately after the stop-node are returned.
- The results come in the shape of a vector of tensors, where "tensor" simply means a *multidimensional array of data*.
- Line **16** retrieves the *filter* of the convolution operation that was performed on the same line. This filter (**filterTensors**) as well as the output-results (**outputs**) will be used by the convolution hardware module.
- Lines **20** to **23** assume that the hardware module has correctly done its job in performing the convolution operation. As described in detail in section 2.6, this hardware module will take input-data and a filter, compute dot-products and write the results to a file called **hardware_out.data**.
- Finally, lines **25** to **27** compute the rest of the graph, stopping after the graph's final node, which is the activation function **softmax**.

For an example run of the script, showing **input**, and **output**, please refer to the appendix in chapter A.

2.5 Hardware-Software Interfacing

Being able to split the graph that describes our Neural Network as well as extract intermediate results from it now raises the question of how to design the interface between software and hardware. How does the data get from being a tensor in a software process to a stream of bytes that a hardware module can work with? The following code snippet shows how this is achieved.

```
1  string OUTPUT_TENSOR_DATA;
2  OUTPUT_TENSOR_DATA = bin2hex(outputs.data()[0].tensor_data().ToString());
3
4  ofstream tensorDataFile;
5  tensorDataFile.open (output_file);
6  tensorDataFile << OUTPUT_TENSOR_DATA; // export tensor data to file
7  tensorDataFile.close();
8
9  cout << "Written data from output tensor to file " << output_file << endl;
```

Line **2** does the job. The tensor data we are looking for is hidden inside the construct **outputs.data()[0].tensor_data()**, and it comes in binary format. After applying a little helper function to it that converts binary to hexadecimal, we get our data in the desired format. The only thing left to do is to write this data, which is returned as a long string of hexa numbers, to a file. This is what happens in lines **4** to **7**.

Inside the hardware module (described in section 2.6), a similar thing happens. After having calculated all the desired dotproducts from the given inputs and weights (filters), they too are written to a file (**hardware_out.data**). This file is later read from a software process to continue the calculation of the entire graph (see code line **22** in section 2.4). Of course, in order to convert from the hardware-compatible data format to the software-compatible one, there needs to be another stage of data reformatting that translates from hex-numbers to tensors.

2.6 Convolution Module Implementation

Figure 2.5 shows a schematic of the hardware module that implements convolution.

As already mentioned, convolution can be best thought of as a window (the "filter") sliding across an image, thereby manipulating it or performing calculations on it in some way. However, from an implementation point of view, obviously neither the filter nor the image is actually moving. What actually happens is that an image is read by the hardware module pixel by pixel, treating the image as a one-dimensional array of pixels. These pixels are read into a shift-register from where they flow into a combination of shift-registers and buffers (top-right part of figure 2.5, will be called "construct"). The crucial idea here is that the width of the construct is exactly the same as the image width. In other words:
number of registers in one line of the construct = number of pixels in one line of the image

Because of this equality, with pixels moving one register further in every clock cycle, the pixels will always be in exactly those places **where they would be if we were sliding a window across the image.** In Figure 2.5, since there are 3 rows of 3 registers in the construct ("filter-registers"), the window-dimension being processed would be 3x3.

Now, in order to compute a dot-product of the **pixels in the filter-registers** with the **filter**, there needs to be a **multiplier** as well as an **adder**. These are the green and yellow boxes in

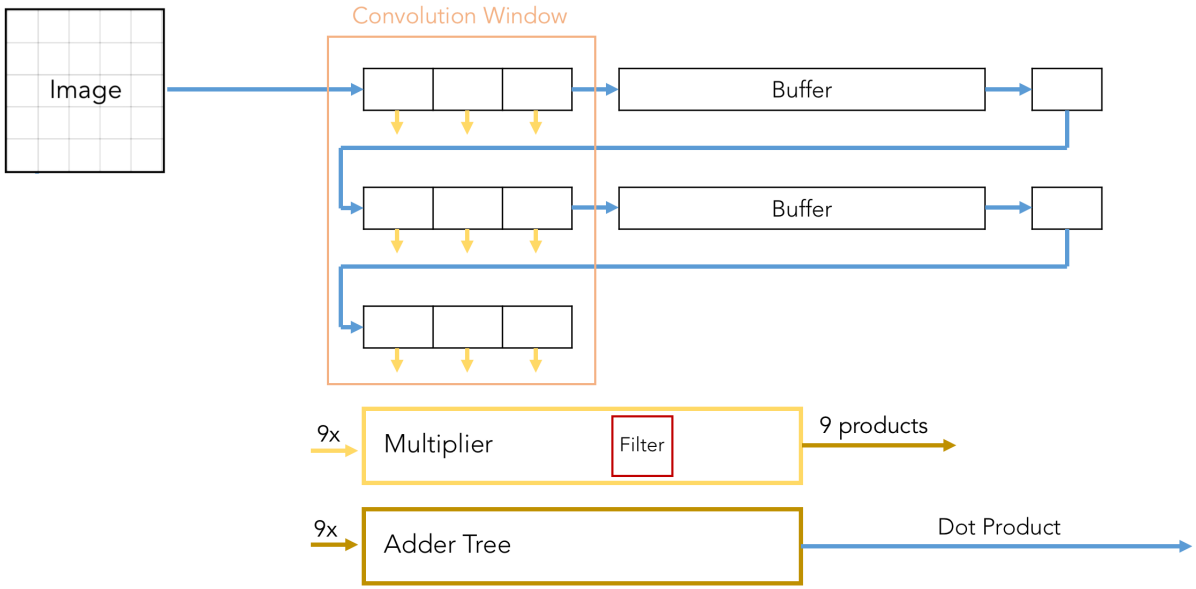

Figure 2.5: Schematic of Hardware Convolution Module

figure 2.5. The filters are located inside the multiplier. Here is where the products of filter-registers and filters are computed. Those products are then summed together in the adder. The output of the adder gives us the final dot-product.

2.7 Applied Methodologies

During the course of the project, software development methodologies of several kinds have been taken into account.

- All C++ and Python code that was written, was developed in an **iterative** and **incremental** manner. This means rather than writing a whole script and debugging it at the end, code was compiled and debugged frequently after every addition or modification (*incremental*). Functionality was also implemented in the context of iterations, i.e. time-spans (normally 2-weeks) were defined for the implementation of certain functionalities, for example for the development of the **avgPool** and **maxPool** hardware modules (*iterative*).

- The agile practice of "frequent feedback and customer collaboration" [1] was realized through meetings and discussions with supervisors, at least weekly. These meetings involved design discussions, code reviewing and iteration planning.

- No Verilog code was written that wasn't directly tested by a testbench and a waveform file. Details on these testbenches can be found in section 3.2.

- Scenarios where invalid or erroneous data is provided to the hardware module have been covered. The module doesn't break down but knows how to deal with such data. The working of this is described in section 3.3.

Chapter 3

Evaluation and Testing

3.1 Graph Analyser Script

The graph describing Inception-v3 in Figure 1.2 is actually describing the CNN at quite a high abstraction level. A lot of the details concerning the involved operations are hidden. Apart from that, with solely the information from the graph, no statement can be made about the details of a node (e.g. which operations and suboperations it comprises), the involved data-types, the number of weights/constants, the range of the parameters, the number of tensors manipulated by each node, etc.

Because of that, there was a need for a script that could analyse and evaluate the Inception-v3 graph (or any tensorflow graph in general) *in detail*. For the project, this script was already provided by one of the supervisors. However, it was adapted and modified by the student to suit particular needs.

The reason why detailed information about the graph is necessary, is the fact that the approach taken involved splitting the graph and extracting data from it (see Section 2.4). This can only be done with a full picture of every step of the graph, e.g. to know the exact tensor-dimensions and data-types that occur at a particular node. Another reason for the usefulness of this script is the knowledge about the number of weights in a graph, especially after the graph was manipulated (see Section 2.3). This for example showed that **reduction** indeed had an impact on the total number of weights in the graph.

3.2 Verilog Testbenches

The best way to evaluate verilog code is to write testbenches for each module. This allows simulation of data flowing through the module. Testbenches can generate waveform-files that allow the programmer to analyse every single signal (wire, register, etc.) in a module.

Figure 3.1 shows the waveform for the convolution module from Section 2.6. Without going into the details, a one-sentence explanation of this waveform shall suffice: Data flows into the module (see **currentData**), moves through a number of shift-registers and buffers (see **shiftreg** and **buffer**), gets multiplied with a convolution window (see **filter**) and finally those products get summed together (see **addReducer**). The orange circle depicts the current 3x3 square of image values, the pink circle highlights the products (image value * filter value) and finally the blue circle shows the final dot-product, i.e. the sum of those products. In this case the value is

408 because $7*3+8*2+9*1+4*15+5*14+6*13+1*27+2*26+3*25=408$.

What unit and integration tests are to software development, testbenches are to hardware development. They ensure a module has the right behaviour and show the programmer when something was broken after a modification [2]. Because of that, testbenches were written for all verilog modules.

3.3 Simulating Erroneous Input Data

In addition to ensuring that signals always have the right behaviour, one needs to deal with the scenario of the module receiving erroneous or unexpected input data. The yellow circle in Figure 3.1 shows such a case. The convolution module receives as its input the value **X** which means **unknown**. This was deliberately introduced in the testbench of the convolution module. As the waveform shows, the undefined value is not forwarded to the following registers and doesn't find its way to the multiplier or adder. In other words: unexpected or undefined values are ignored by the module. The reason for this is that undefined values later lead to wrong dot-products which eventually will have a negative impact on the entire performance of the ANN.

Dealing with wrong (user) input obviously doesn't only apply to verilog code but also to the C++ and Python scripts that were developed. Here is a code snippet showing how this problem was dealt with in a C++ script.

```
1  const bool parse_result = tensorflow::Flags::Parse(&argc, argv, flag_list);
2  if (!parse_result) {
3  LOG(ERROR) << usage;
4  return -1;
5  }
6
7  std::vector<string> string_flags = {image, graph, labels,
8                                      input_layer, output_layer,
9                                      from_node, to_node, root_dir};
10 std::vector<int> int_flags = {input_width, input_height,
11                               input_mean, input_std};
```

In this case, the helper-function **Parse()** does the job of checking whether all provided arguments are valid.

3.4 Accuracy and Speed Comparison

One crucial aspect of this project was to compare accuracy and speed of different ANN implementations. These include several modifications of the original Inception-v3 graph, namely

- Original graph (ORI)
- Quantized graph (QUA)
- Reduced graph (RED)
- Quantized, then reduced graph (QTR)
- Reduced, then quantized graph (RTQ)

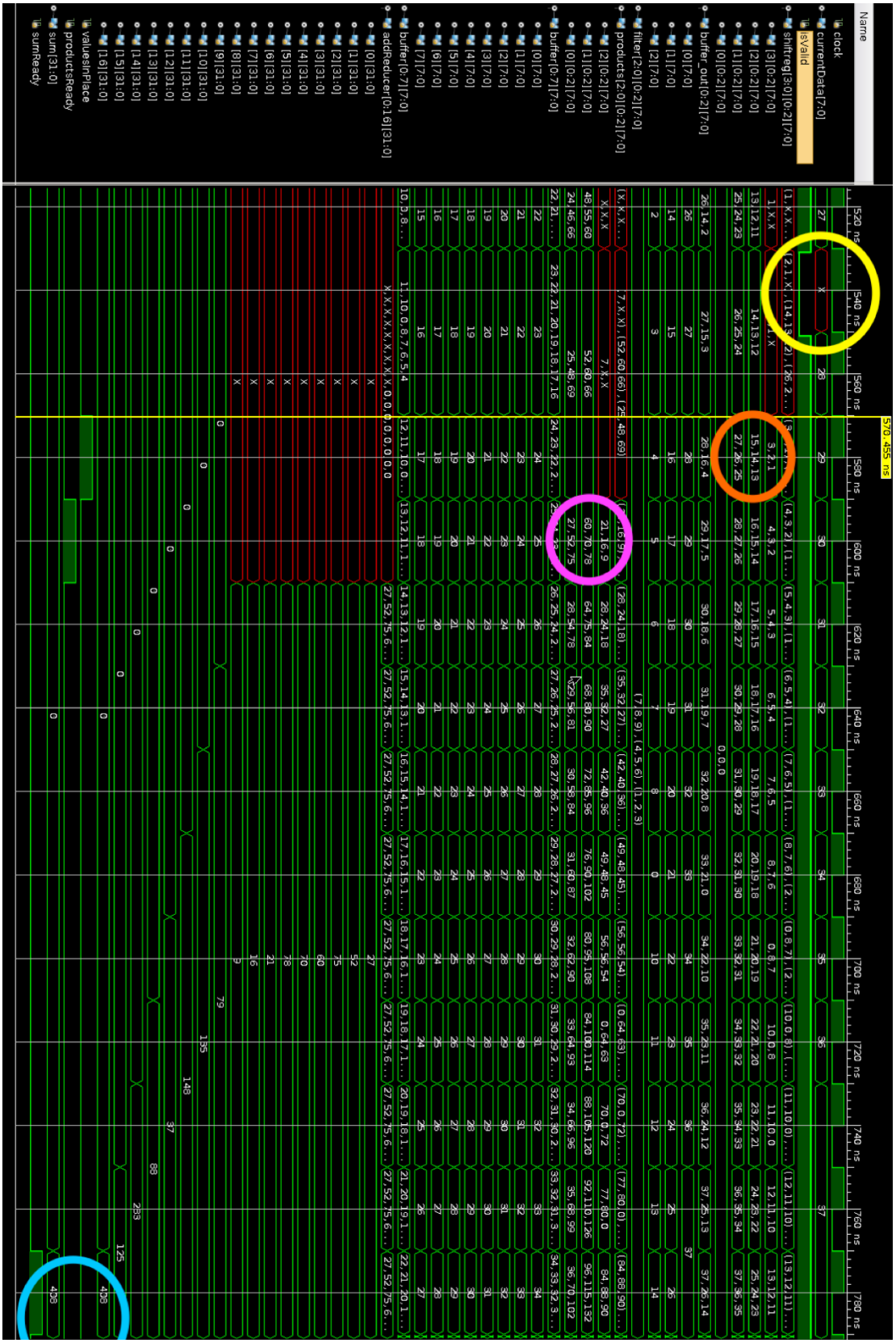

Figure 3.1: Waveform of Convolution Module

Graph	Top-1 Accuracy	Computation Time
ORI	89.1%	2.263s
QUA	79.3%	1.816s
RED	88.1%	2.059s
QTR	71.2%	1.763s
RTQ	69.3%	1.761s

Table 3.1: Comparison of Variations of the Inception-v3 Graph

Table 3.1 compares classification accuracies as well as computation speed of the five graphs listed above, without hardware interfacing.

As the table shows, the original graph performs best in terms of accuracy. With quantizing the graph (i.e. using 8-bit instead of 32-bit weights), accuracy drops by around 10% but the computation becomes faster by around 20%. Reducing the graph (i.e. removing unnecessary nodes) has little impact on accuracy, due to the fact that the removed nodes are only needed in the training phase of the ANN. Computation time stays more or less the same. With applying both quantization and reduction, we get a drop in accuracy of more than 10% and a gain in computation speed by around have a second. Interesting to note is the difference between QTR and RTQ, namely that QTR gives 2% more accuracy than RTQ with the same computation time.

In addition to that, comparisons were made between

- Running the graph just in software
- Running one or more convolution operations on hardware
- Running convolution as well as other operations on hardware

When performing hardware-software interfacing (see section 2.5), we obviously have the speed disadvantage of having to write graph data to a file, reading from that file, then writing results back to a file and again reading from that file. However, it is interesting that accuracy doesn't suffer much from this interfacing. In multiple test runs, the difference between purely classifiying in software and a hardware-software combination was in the range of 3% to 5%. This is mainly due to implementation details of the ANN in software that are hard to mimic in hardware. In terms of speed, the problem isn't that severe because once a large part of the graph is run on hardware, the overall speed performance will improve. This is because the hardware is *taylored* to the operations the graph comprises, and therefore computes those faster than a general purpose processor.

Chapter 4

Reflection and Conclusion

4.1 Planning and Scheduling the Project

Planning for the project began in September 2016. In a gantt chart from the first weeks of the project period, I came up with several periods for the duration of the project. These periods lasted between 3 and 8 weeks. The first period involved learning about the main languages and technologies, namely Verilog, SystemVerilog, Python, C++ and Neural Networks as well as doing research on topics like hardware-acceleration, convolutional neural networks, etc. This period also included a tutorial on FPGA programming, a tutorial on Verilog, online courses on machine learning (especially on Neural Networks) [5] as well as a lot of Python and C++ self-teaching.

The second period involved me getting my head around Tensorflow and its massive codebase. A considerable number of hours was spent on setting up and installing Tensorflow to make it work smoothly on both the lab machines and my private laptop. (see section 4.2).

The third period was about experimentation. It is here where I developed the first drafts of the computation cell and thought about a way of scheduling the dataflow. A hardware implementation of an activation function (Relu) was also developed in this period.

The fourth period was where most of the actual implementation work was completed. This included writing verilog code for the hardware modules but also writing/modifying several scripts in Python and C++, especially those for analysing, splitting and modifiying the computation graphs. Evaluation and testing was also done in this period, parallel to implementation. Several refinements of the computation cell were also developed. In fact, the design from figure 2.1 was the sixth version of it, mainly because many more control signals turned out to be necessary in the implementation process.

The fifth and final period was about code refinements, debugging, preparing for the project demonstration, doing research for and writing the report as well as recording the project screencast.

4.2 Understanding Tensorflow's Codebase

Since working with Tensorflow's codebase constituted a major part of the project, there shall be a brief discussion of the most basic idea of the Tensorflow architecture.
According to the official website, Tensorflow "is an open source software library for numerical computation using data flow graphs". The idea of the architecture is actually quite simple.

Multi-dimensional arrays, so called *tensors* are moved between nodes. Nodes represent some manipulation of the tensor, i.e. a calculation on it. Since the operations performed in nodes as well as the tensors themselves can take many shapes, Tensorflow is inherently customizable. "Many shapes" in the sense that an operation can range from being a simple addition to a large matrix multiplication, and tensors can hold a single value, an n-dimensional data-array and everything in between.
This said, understanding Tensorflow's codebase then actually becomes a question of understanding how tensors are created, manipulated and moved around. Here is an extract of Tensorflow's documentation on the implementation of convolution that shows well the concept of tensor manipulation.

```
Given an input tensor of shape `[batch, in_height, in_width, in_channels]`
and a filter / kernel tensor of shape
`[filter_height, filter_width, in_channels, out_channels]`, this op
performs the following:

1. Flattens the filter to a 2-D matrix with shape
   `[filter_height * filter_width * in_channels, output_channels]`.
2. Extracts image patches from the input tensor to form a *virtual*
   tensor of shape `[batch, out_height, out_width,
   filter_height * filter_width * in_channels]`.
3. For each patch, right-multiplies the filter matrix and the image patch
   vector.
```

This being the basic concept, Tensorflow then provides a huge API in both Python and C++ for implementing a wide range of functions and algorithms. For the visualisation of Tensorflow graphs, there is an additional tool called Tensorboard that allows the creation of figures such as 1.2.

4.3 What went well, what didn't?

Many things during the course of the project went well. On the other hand, it can't be denied that there were parts to the project that didn't go that well. This section shall outline both, good and bad aspects of the project work.

Went well:

- Getting comfortable with C++ and Python: using Tensorflow's APIs, scripting, testing.
- Setting up Tensorflow, experimenting with its features, dealing with the trouble of Tensorflow's **1.0 release** [12].
- Designing and implementing hardware modules, using them with real data from the CNN
- Exctracting data, bringing it into the right format such as a tensor or a byte-stream.
- Successful interfacing between hard- and software (relates to previous point)
- Achieving a working convolutional neural network that performs image classification. To be precise: taking a pre-trained convolutional neural network, modifying it heavily, running parts of it on hardware and still achieving good classification accuracy.

Didn't go well:

- Getting comfortable with Verilog. Coming from a software and Java-oriented background, it was a challenge to get used to the principles of coding in a hardware description language.
- Project scheduling, evenly distributing workload. Even with a project schedule at hand, it happened that a lot more work was done in the last two months compared to the preceding months.
- Experimenting with multiple FPGAs. That is, implementing the idea described by figure 2.2. This would have been the next step and it would be very interesting to see which gains in efficiency and speed can be achieved by plugging multiple FPGAs together to implement an entire Neural Network.

Overall however, the project was a very rewarding experience during which many useful skills were acquired. Especially the exposure to Tensorflow, to date one of the most popular and widely-used machine learning libraries in existence, turned out to be highly useful as well as enjoyable.

References

[1] K. Beck et al. Principles behind the Agile Manifesto. http://agilemanifesto.org/principles.html, 2001. Accessed: 19/04/2017.

[2] EmbeddedMicro. Writing Test Benches. https://embeddedmicro.com/tutorials/mojo/writing-test-benches, 2015. Accessed: 20/04/2017.

[3] A. Jain and J. Mao. Artificial Neural Network: A Tutorial. Technical report, Michigan State University, 1996.

[4] D. Koch. ARTIFICIAL NEURAL NETWORK CASE STUDY. Technical report, The University of Manchester, 2016.

[5] A. Ng. Machine Learning. https://www.coursera.org/learn/machine-learning, 2016. Accessed: 24/04/2017.

[6] I. Russell. Neural Networks Module. http://uhaweb.hartford.edu/compsci/neural-networks-definition.html, 1996. Accessed: 20/04/2017.

[7] S. Scardapane. What is machine learning, and why you should care. http://2017.datadriveninnovation.org/scardapane-what-is-machine-learning-and-why-you-should-care/, 2017. Accessed: 21/04/2017.

[8] Stanford Vision Lab. ImageNet. http://image-net.org/, 2016. Accessed: 01/05/2017.

[9] TensorFlow. Optimize for Inference. https://github.com/tensorflow/tensorflow/blob/master/tensorflow/python/tools/optimize_for_inference.py, 2015. Accessed: 19/04/2017.

[10] TensorFlow. Image Recognition. https://www.tensorflow.org/tutorials/image_recognition, 2017. Accessed: 21/04/2017.

[11] TensorFlow. Inception in TensorFlow. https://github.com/tensorflow/models/tree/master/inception, 2017. Accessed: 22/04/2017.

[12] TensorFlow. Transitioning to TensorFlow 1.0. https://www.tensorflow.org/install/migration, 2017. Accessed: 21/04/2017.

[13] M. Vesper et al. JetStream: An Open-Source High-Performance PCI Express 3 Streaming Library for FPGA-to-Host and FPGA-to-FPGA Communication. Technical report, The University of Manchester, 2016.

[14] P. Warden. Why are Eight Bits Enough for Deep Neural Networks? https://petewarden.com/2015/05/23/why-are-eight-bits-enough-for-deep-neural-networks/, 2015. Accessed: 18/04/2017.

[15] P. Warden. How to Quantize Neural Networks with TensorFlow. https://petewarden.com/2016/05/03/how-to-quantize-neural-networks-with-tensorflow/, 2016. Accessed: 18/04/2017.

[16] S. Yegulalp. 11 open source tools to make the most of machine learning. http://www.infoworld.com/article/2853707/robotics/11-open-source-tools-machine-learning.html, 2014. Accessed: 21/04/2017.

Appendix A

Example of operation

A.1 Graph-Execution Script

Script that runs Inception-v3 graph, with hardware-software interfacing.

A.1.1 Input

```
>> Take the script \textbf{main.cc} in Scripts/graph_segment (see
   submission files)
>> Take both the original and the QTR graph
>> bazel build tensorflow/examples/graph_segment/...
>> bazel-bin/tensorflow/examples/graph_segment/run_graph_parts
```

A.1.2 Output

```
    Pixel Count:  89401
    Chanpix Count: 268203
    Hexadigit Count: 1072812
    Byte Count:  536406
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    Written data from output tensor to file 'tensor0.data'
    Output Tensor Debug String:
  Tensor<type: float shape: [1,299,299,3] values: [[[-0.4296875 -0.4375
      -0.46875]]]...>
    Size of Output Tensor: 268203
    Dims of Output Tensor: 4
    From-To:    DecodeJpeg/contents -----> Mul
    Graph:
  /Users/oliverkugel/Git/Uniproject/Graphs/tensorflow_inception_graph.pb
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    Hexadigit Count: 1072812
    Byte Count:  536406
~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

    Written data from output tensor to file 'tensor1.data'
    Output Tensor Debug String:
  Tensor<type: float shape: [1,299,299,3] values: [[[-0.4296875 -0.4375
      -0.46875]]]...>
```

```
21      Size of Output Tensor: 268203
22      Dims of Output Tensor: 4
23      From-To:    Mul ------> conv
24      Graph:
25    /Users/oliverkugel/Git/Uniproject/Graphs/quantized_then_reduced_graph.pb
26  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

27      Hexadigit Count: 3456
28      Byte Count:  1728
29  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

30      Written data from output tensor to file 'filter.data'
31      Output Tensor Debug String:
32    Tensor<type: float shape: [3,3,3,32] values: [[[0.012605551 -0.14095107
        -0.24736048]]]...>
33      Size of Output Tensor: 864
34      Dims of Output Tensor: 4
35      From-To:    conv ------> conv/conv2d_params
36      Graph:
37    /Users/oliverkugel/Git/Uniproject/Graphs/tensorflow_inception_graph.pb
38  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

39  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

40      PROCESSED THROUGH HARDWARE MODULE
41      From-To:    conv ------> conv_1
42  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

43      Hexadigit Count: 4032
44      Byte Count:  2016
45  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

46      Written data from output tensor to file 'tensor2.data'
47      Output Tensor Debug String:
48    Tensor<type: float shape: [1,1008] values: [0.00010096782 0.00025016814
        7.98709e-05]...>
49      Size of Output Tensor: 1008
50      Dims of Output Tensor: 2
51      From-To:    conv_1 ------> softmax
52      Graph:
53    /Users/oliverkugel/Git/Uniproject/Graphs/tensorflow_inception_graph.pb
54  ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~

55  I tensorflow/examples/graph_segment/main.cc:228] giant panda (169):
      0.711734
56  I tensorflow/examples/graph_segment/main.cc:228] indri (75): 0.0100605
57  I tensorflow/examples/graph_segment/main.cc:228] lesser panda (7):
      0.00522913
58  I tensorflow/examples/graph_segment/main.cc:228] custard apple (325):
      0.00373578
59  I tensorflow/examples/graph_segment/main.cc:228] earthstar (878):
      0.00344424
```