Heriot-Watt University

Masters Thesis

# Performance Metrics for Approximate Deep Learning on Programmable Hardware

*Author:*
Denis Pascal Bacchus

*Student Number: H00303843*

*Supervisor:*
Dr. Robert Stewart

*Second Reader:*
Dr. Ekaterina Komendantskaya

*A thesis submitted in fulfilment of the requirements for the degree of MSc. in Data Science*

*in the*

School of Mathematical and Computer Sciences

August 2019

# *Abstract*

Neural networks are increasingly used in image recognition, autonomous systems, language processing. The computational requirement for training and deploying neural networks has increased a lot over the last decade because for neural networks these application domains have grown from single hidden layer topologies with several hundred weights to deep models reaching one hundred hidden layers with millions of weights. This prohibits the use of embedded system accelerators for executing "out of the box" full precision neural networks because they have limited hardware resources for storage and computation. This is unfortunate because embedded and programmable hardware, e.g Field Programmable Gate Arrays (FPGA) are capable of performing computations extremely quickly, at a low energy cost. For neural networks, this means a high-speed classification speed, e.g classifying thousands of images each second.

Several methods exist to compress neural networks to reduce both the memory size and computation time. Those methods generally approximate a neural network by pruning the number of parameters used. Another approximation approach is the quantisation of weights and activation functions to simplify networks. The result is neural networks with smaller spacial complexity, and low precision arithmetic operations that are cheaper to compute which improves runtime speed, at the cost of classification accuracy.

Using the FINN Python framework, this dissertation measures the effects of quantisation of neural networks targeting FPGAs, against two performance metrics: the amount of hardware resources required (space) and classification accuracy (algorithmic performance).

The outcomes of this dissertation are timely with industry rapidly shifting their deep learning implementation efforts, moving AI towards Edge computing, i.e. away from centralised GPUs and towards remote smart sensors.

The empirical experiment results in this dissertation identify sweet spots in the accuracy and space trade-offs for a neural network with three hidden layers that performs digit recognition using the MNIST dataset. The dissertation's chosen workflow is representative of similar frameworks, e.g. Google's Tensorflow Lite and Intel's Distiller framework, and will raise the awareness of deep learning performance metrics when targeting resource-constrained devices, with users and developers of these industry led emerging frameworks.

# Chapter 1

# Introduction

Deep learning is used by scientists and companies alike, to solve problems involving complex analysis such as image classification or speech recognition. Deep learning technologies such as Googles's Tensorflow [Abadi et al., 2015] for neural network construction and training are generally computationally expensive. The results of some deep learning algorithms need to be computed in real time to be useful e.g. stock market trading and autonomous vehicles, and others at a low energy cost e.g. remote computer vision on smart sensors and drones, where access to power is scarce.

GPUs are often used to train and test neural networks. This is becoming prohibitively expensive for energy use. The energy required by GPUs for state-of-the-art deep learning AI has carbon footprint estimated to be around five times the lifetime emissions of an average car [Lu, 2019]. Instead, an alternative is to train neural networks on GPUs, then deploy them to special purposes hardware accelerators, e.g. FPGAs to reduce energy costs and to increase classification speed. This introduces a new problem, in that the model trained on a GPU has a memory footprint and high precision computational requirement that are both far too large for FPGAs

The goal of this dissertation is to better understand the effect of compressing neural networks targeting FPGAs using quantisation, through the use of a Python framework called FINN [Umuroglu et al., 2016]. We train neural networks with different quantisation configurations, and measure their classification accuracy performance and their hardware resource required for a mid range FPGA. Neural network performance is often assessed on its classification accuracy (algorithmic performance). The focus of this dissertation is targeting resource constrained FPGAs, so it also measures hardware resource requirements (space), and compares these two metrics.

FINN allows the user to train its neural networks through the Python's libraries Theano and Lasagne. Currently, three different datasets are directly available for training as shown in Figure 1.1:

- **MNIST:** Handwritten digit classified with a DNN. [LeCun and Cortes, 2010]
- **CIFAR-10:** Collection of images used for computer vision and classified with a CNN. [Krizhevsky, 2009]
- **GTSRB:** German Traffic Sign Recognition Benchmark classified with a CNN. [Stallkamp et al., 2012]

(A) MNIST [LeCun and Cortes, 2010] (B) CIFAR-10 [Krizhevsky, 2009] (C) GTSRB [Stallkamp et al., 2012]

FIGURE 1.1: Datasets already implemented in the framework

The last two datasets are more complex in contrast to the first one as the number and size of the images is far greater. Thus, the dissertation focuses on the MNIST dataset, in the first time, as training and testing time is significantly lower.

Classification accuracy is often considered *the* performance metric of neural networks. The *robustness* of a neural network to malicious adversarial attacks has until recently rarely been considered a performance metric. However, with the increased reliance on neural networks for safety-critical tasks like autonomous driving, in future, this will have to be a first-class concern and this is attracting growing research activity. As future work, formal verification of the robustness and fragility of quantised neural networks would be interesting, and indeed essential to investigate.

This project is part of a wider research group within the Lab for AI and Verification (LAIV), composed of other MSc students and PhD students, that focuses more on the verification methods of neural networks. This report consists of a literature review of previous work done in this area of approximate deep learning and the framework to be used in this project. Then Section 3 consists of a detailed explanation of the whole FINN framework and quantisation processes. Section 4 is dedicated to the evaluation of the experiments and a discussion of the results.

# Chapter 2

# Literature Review

This chapter explains the technologies used in this dissertation. Section 2.1 presents the current advances in deep learning with the application of neural networks. Section 2.2 describes the approximation methods used in this dissertation, namely the quantification of neural networks on FPGAs and some other compression techniques. Section 2.3 focuses on FPGA hardware and neural network frameworks. Section 2.4 presents an explanation of verification techniques.

## 2.1 Deep Learning and its Application

### 2.1.1 Types of Neural Networks

The three most classic and widespread architectures currently used in the deep learning field are DNN (Deep Neural Network or Deep feed-forward Neural Network), RNN (Recurrent neural network) and CNN (Convolutional neural network) as described in [Goodfellow et al., 2016].
A neural network is defined with just a few elements: the different layers, the topology between each layer, the weights associated with those connections and finally the activation function for each node. A neural network consists of three types of layers:

1. The input layer which receives the input data (e.g. pixels of an image).
2. The hidden layers that compute intermediate features.
3. The output layer that corresponds to the classification of the neural net.

A neural network is trained (i.e. the weights are changed) with a dataset and then tested on another one. DNNs are trained through backpropagation where we adjust the weights according to how well the weights are tuned, for the known input and output.

When a DNN includes some connections between a layer and its previous layer via feedback connections then this network is called recurrent. These connections enable the feedback of updated weight values and activation function outputs for a given input when processing future input stimulus. This is used when the classification needs to keep some of the previous information to successfully classify the next input.

The spatial counterpart is CNN. The functions used do not necessarily apply to each individual input but rather to a subset of the inputs. Figure 2.1 describes the architecture of those three neural networks. The pooling operation applies to a group of pixels and not to a specific pixel. The strength of the topology is that it brings invariance to the classification even with different geometric rotation or translation [LeCun and Bengio, 1998].

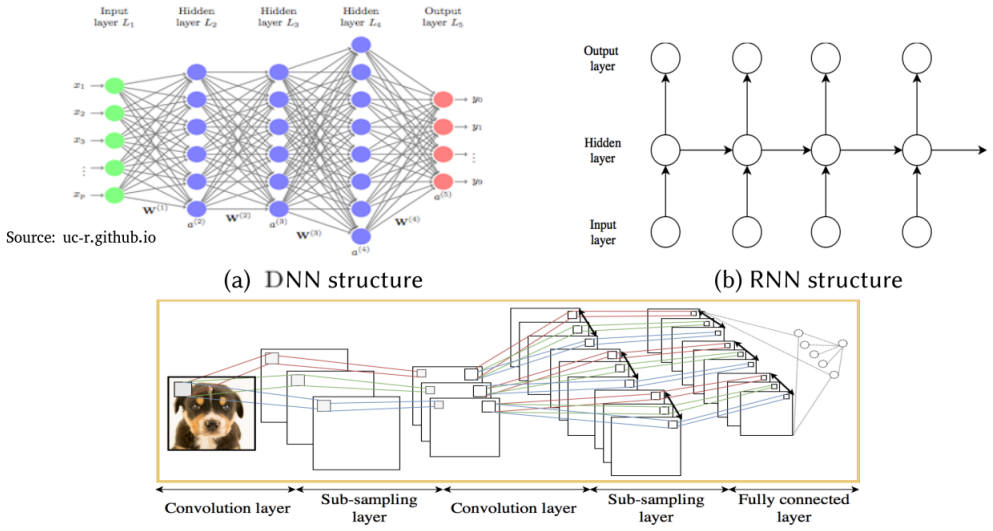



FIGURE 2.1: DNN, CNN and RNN architecture [Pouyanfar et al., 2018]

Other architectures exist but there are more at a research stage. Those architectures include DGN (Deep Generative networks) which are built around probabilistic method and Bayesian inferences [Goodfellow et al., 2016].

### 2.1.2 Applications

The three fields where deep learning is widely used are NLP (Natural language processing), visual data processing and audio processing as described by [Pouyanfar et al., 2018]. Of course, other areas are also transformed by deep learning as shown in Figure 2.2. NLP is composed of different application such as machine translation, sentiment analysis or summarization. Such problems require a knowledge of what has been said or some context around a word or sentence and therefore are often using RNNs or similar topologies where the network keep track of other words. We have something similar in image processing where CNNs are often the main base model. The problem gets tougher when trying to work on video processing or audio processing as we have to deal with multiple dimensions. We have seen the emergence of a fusion between the existing model to create a 3D CNN network that studies two streams of information: spatial stream (RGB frame) and temporal stream (optical flow) [Pouyanfar et al., 2018].

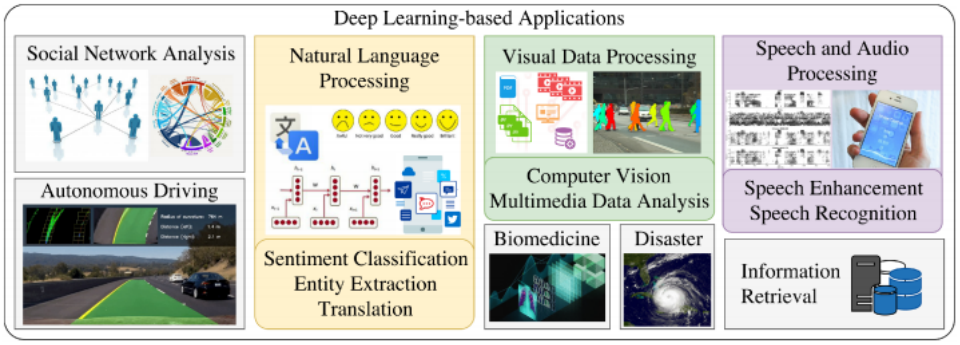



FIGURE 2.2: Some of the popular deep learning applications [Pouyanfar et al., 2018]

The applications are as diverse as the architectures, and new models regularly emerge with their own specificities. They all have in common the resolution of problems that are a priori not trivial because they are easily solved by humans but difficult to formalise.

### 2.1.3 Neural Network Performance Metrics

We have seen that neural networks differ from each other in term of topology and application but then we need to find the right metric to evaluate them as different functions are calling for different needs. The following list comes from research done by [Wang et al., 2019] and [Huang et al., 2016] as well as personal reflexion around influential parameters.

- **Accuracy:** This is either with respect to the training or testing dataset, but for scalability purposes and to obtain more general results this dissertation focus on the testing accuracy.

- **Throughput:** This represents the efficiency of the network with the number of classification per unit of time.
- **Latency:** The time needed to obtain a classification.
- **Energy:** The energy consumed by the network during runtime.
- **Compression ratio:** The weight storage of the approximate network compared to the original network.
- **Training Time:** The time needed to obtain a fully operational network throughout an offline process.
- **Online Learning:** How well the network is suited to train dynamically. In this configuration, the network will be used while training.
- **Robustness:** The robustness of the network when confronted with perturbations.

However, each application applies a compromise between all these parameters. To make the example more meaningful, Figure 2.3 presents two very different hypothetical scenarios: the real-time use of obstacle detection on an autonomous car and an NLP translation application. We cannot have a network in a car that will be slow to answer, sometimes inaccurate and that could be easily fooled by random pertubations as it would mean the death of pedestrians. But the training time is not an issue as it will be trained in a dedicated computer once a while to update the neural network. We can see the exact opposite for a common NLP application where failure is not severe but the system has to be quick for the user not be annoyed. The result is fast training time and also an online training time so that the network will improve while being used.

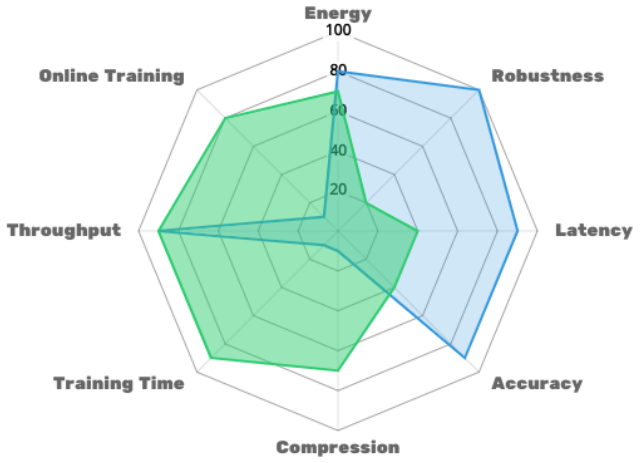



FIGURE 2.3: Radar Chart for autonomous car (Blue) and a NLP application (Green)

## 2.2 Approximate Deep Learning Approaches

Neural networks have proven their effectiveness in solving complex problems, especially since the recent sharp increase in computing power of computers. But as the raw power is limited on embedded systems, the need to reduce the consumption of applications, which are already very demanding in terms of computing power and memory usage, is becoming more important. Approximate deep learning is one method that focuses on the complexity of a network (both space and time complexity) to increase performances. Those methods generally apply on customed hardware as we can more easily tune the resources for a specific need. A complete study has been done by [Wang et al., 2019] for the ACM (Association for Computing Machinery) in this field.

### 2.2.1 Quantisation

Quantisation is a method that reduces the size and computation time of neural networks by shifting values from 32-bit floating point number (almost continuous values) to less than 8-bit values (discrete values). In a network all sorts of values can be quantised:

- Weight between neurons.
- Activation function of neurons.

Each parameter can be reduced to a specific precision to better adapt to the problem.

#### 2.2.1.1 Binarisation

Binarisation is a special case of quantisation where only a single bit is used and therefore offer only two values for weights and/or activation functions. According to [Courbariaux and Bengio, 2016] the most simple binarisation function is the deterministic one: the Sign function. Each value will either be 1 or -1 respect to whether it is positive or negative. It then works similarly to the standard floating-point network with forward and back propagation to update the weights. Other models involved stochastic approximation where the threshold is not symmetric between all values of the function. It can simulate in a more accurate way the non-linearity of a function.

The strength of those methods is mostly a huge improvement in computation time as it replaces the standard operation of integers with bit-wise operations, also it reduces the energy consumption (less complex computation) and the memory storage. Of course, this approximation leads to a higher error rate classification compared to the standard quantisation but the error is still bounded near the floating-point results.

#### 2.2.1.2 Fixed Point Representation

Quantisation is the action to reduce the number of bits needed to code an integer. Usually, integers are coded with 32 or 64-bit and the goal here is to reduce to less than 8-bit. Both weights and activation functions can be quantised to reduce effectively the time and spatial complexity of the network. The work done by [Hubara et al., 2016] shows a great improvement in the accuracy for quantised neural networks compared to their binarised counterpart. With only 4-bit they achieved a similar result as 32-bit networks. The process to correctly quantise the parameters is to round a value to the nearest threshold according to the number of value we have left.
In the following formula $x$ is the input value and $minV$, $maxV$ are the minimum and maximum scale range respectively.

$$LinearQuant(x, bitwidth) = Clip(round(\frac{x}{bitwidth}) * bitwidth, minV, maxV)$$

Figure 2.4 underlines the aspect that we can calibrate the quantisation parameters (minimum value and maximum value) depending on the function used, the values we will have and the section of the function that is interesting to adjust.

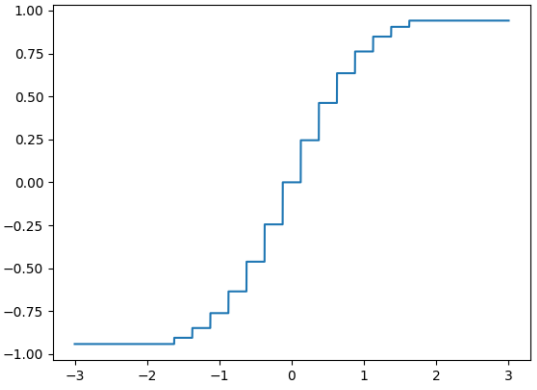



FIGURE 2.4: Quantisation of the hyperbolic tangent function over 4-bit

#### 2.2.1.3 Logarithmmic Quantisation

The former quantisation implies an even distribution of all the values in the range $[min; max]$ but the authors suggest that the range of values taken by the parameters is more important than the precision. The following express how to construct a logarithmic quantisation where AP2(x) is the approximate power of 2 of $x$ [Hubara et al., 2016].

$$LogQuant(x, bitwidth) = Clip(AP2(x), minV, maxV),$$

The idea is that it is better to have a low precision but a high range as it will increase the number of very different behaviour and thus a larger number of possibilities will be available for each weight and activation function. To use the previous example of the hyperbolic tangent, it is better to have 4 values far from each other which represents 4 choices of values: very low value for insignificant weights, high value for essential weights and two values close to 0 to differentiate what is rather good and bad. With 4 values that have close gaps, the difference between each of the values becomes small and it is difficult to differentiate gaps.

### 2.2.2 Weight Reduction

Quantisation was looking at how to reduce the weight and the computation time of each parameter. Here with the method of weight reduction, we want to keep a high bit precision while dismissing unimportant parameters in the network. It will, therefore, reduce the workload and the total memory usage of the network.

#### 2.2.2.1 Pruning

The purpose of pruning is to keep only the most useful connections between nodes so that the overall size will be much lower but the accuracy will still be correct. The general workflow as described by [Han et al., 2015] is divided into three steps:

1 Train the network with all connections.

2 Dismiss unimportant connections that have the lowest weight (absolute value).

3 Retrain the network to better calibrate the remaining connections.

Figure 2.5 describes how typical networks will look like.

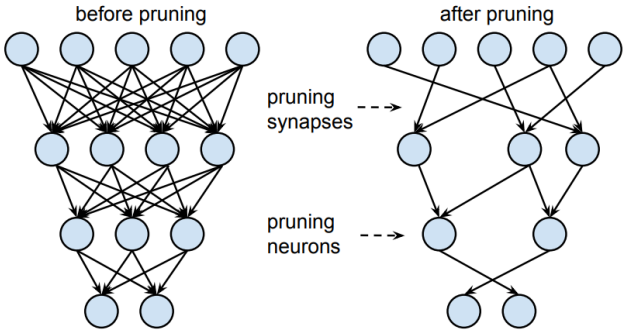



FIGURE 2.5: Synapses and neurons before and after pruning [Han et al., 2015]

It is crucial to retrain the network otherwise the accuracy will drop down. Due to a reduced size both the memory usage and the computational time are lower. They obtain compression ratio around 10 will keeping a similar accuracy for ImageNet and AlexNet.

#### 2.2.2.2 Weight Sharing

The idea initially proposed by [Cheng et al., 2015] is to pack groups of weights together given they have similar values. It works with Gaussian processes and will cluster all similar group of weights together to give the mean and a standard deviation of the centroid. Their experiments showed a 160 compression ration for MNIST and a loss of accuracy of only 0.1 points of percentage.

#### 2.2.2.3 Structural Matrices

A matrix is a good structure to represent and store all the weights of a neural network as we can divide the layers and neurons into respectively columns and rows. The author of [Cheng et al., 2015] tried to use circulant matrices to more efficiently store all the weights. A circulant matrix is defined as followed:

$$\begin{bmatrix} w_0 & w_{n-1} & w_{n-2} & \dots & w_1 \\ w_1 & w_0 & w_{n-1} & \dots & w_2 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ w_{n-1} & w_{n-2} & w_{n-3} & \dots & w_0 \end{bmatrix}$$

The weights in all columns and rows are the same, except that they are shifted version of the first column. The idea is that we do not need to have $n^2$ weights (n is the dimension of the matrix and of the problem) to describe everything as most of them are quite similar.

The benefits of this matrix are that firstly it will store all the weights in just a column and not an actual matrix, so we reduce the spatial complexity from $O(n^2)$ to $O(n)$. Secondly, It reduces the time complexity from $O(n^2)$ to $O(n \log n)$ because we can use FFT (Fast Fourier Transform) on such a structure (one-dimensional array). The gain is important, mainly on memory consumption and we have to deal with only a few points of percentages of loss rate.

#### 2.2.2.4 Knowledge Distillsation

We often use similar neural networks for related applications. For example, if we already have a network trained to recognise cats then we could try to apply it to classify feline with a little extra training. All the idea is that the problem is similar thus the parameters are probably not so different. In the process of knowledge distillation, as described by [Hinton et al., 2015], the objective is to train a complete model and then extract the most useful connections of it to obtain the smallest net. In a multiple classification problem, we will try to keep only the connection that fire the good result for a given input and output. We do not actually care about the smallest probability as they will just ake the model complex and heavier.

The method works well when trying to transfer simple and highly regularised dataset such as MNIST but big neural networks are almost impossible to distilled with their technique. Eventually, the authors consider the use of smaller very specialised neural networks that will then be applied to medium networks.

### 2.2.3 Input-Dependant Computation

The dropout method is used when trying to reduce the number of neurons and connections by periodically removing some of them and see how the error evolved. However as described by [Bengio et al., 2015] this is a "form of "unconditional" computation, in which the computation paths are data-independent" (page 2). The authors of this article propose an approach where the computation in the following layers is dependent on the input. The suggestion is that in a classification problem where outputs can be very different (clothes for instance) we can dynamically remove irrelevant neurons. For example, if we discover soon enough that the cloth seems to be a T-shirt or related to it, then we could remove neurons responsible for activation of shoes and jeans.

This technique relies on reinforcement learning and will lead to faster calculation as well as better training as the number of links will be lower and thus the backpropagation of the gradient descent will be more effective.

### 2.2.4 Discussion and Comparison

It is not possible to straightforwardly compare these surveyed compression approaches because of the huge diversity of approximation methods, hardware and performance metric used. The results are highly dependent on the hardware used. Also, many studies combine several of those methods to attain better performances [Wang et al.,

2018, 2017]. The difference in metrics can be due to the objectives of each respective studies.

## 2.3 Quantised Neural Networks on FPGAs

### 2.3.1 Hardware Implementations of Approximate Methods

All of the approximate neural networks techniques mentioned above could work on regular CPUs or even GPUs to get a better computation time as neural networks application tend to be computationally expensive. In this dissertation, we favour FPGAs over regular hardware since they offer much lower energy consumption for similar peak performance as shown in Table 2.1. The comparison was made with hardware datasheets from Intel's CPUs, NVIDA's GPUs and Xilinx's FPGAs. The main advantages of FPGAs are that they combine really low power consumption with more computational power than regular CPUs.

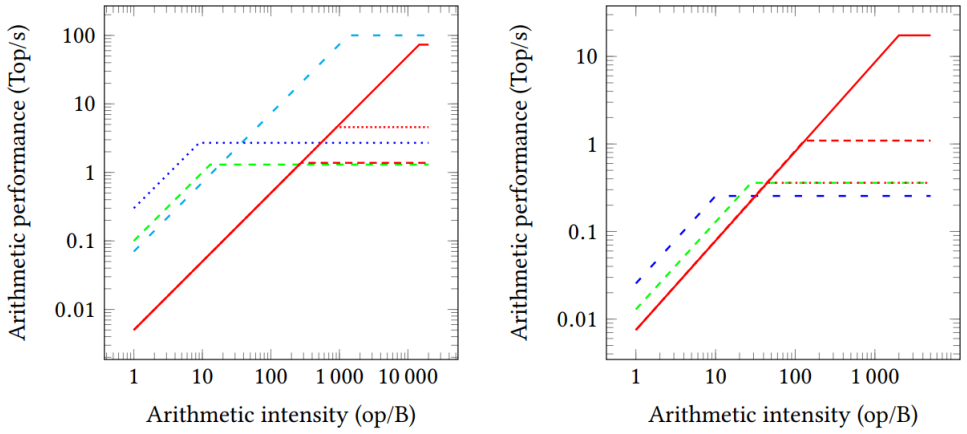
TABLE 2.1: Performance comparision between CPUS, GPUs and FPGAs

| | CPU | GPU | FPGA |
|---|---|---|---|
| Size Threads | 10/100 | >1000 | >10000 |
| Peak Performance | ∼50 GFlops | ∼10 TFlops | ∼500 GFlops |
| Energy Consumption | ∼100 W | ∼200 W | ∼20 W |

But to increase the throughput there is not only parallelism available but also workload and memory reduction [Wang et al., 2019]. The authors explain their point of view with the following roofline approximation model (Figure 2.6). Arithmetic intensity is the number of arithmetic operations performed per byte of memory and arithmetic performance is the number of operation per unit of time.

Custom hardware such as FPGAs usually achieve more than an order of magnitude in performance compared to standard hardware. This is especially true for embedded systems and really interesting as embedded systems cannot possess huge CPUs nor GPUs due to physical space and energy limitation, and often require a custom tailored made hardware that perfectly fit the application purposes.

What is worth noticing is that "due to their specialised support for floating-point arithmetic operations, GPUs can deliver the highest arithmetic performance for FP32 DNN inference. When moving from floating-point to lower-precision fixed-point data representations, however, custom hardware design flexibility facilitates the trading off of precision for increased performance" [ [Wang et al., 2019] page 5]. Thus given that




(a) Datacentre-scale platforms: 18-core Intel Haswell CPU (- - -), Nvidia Tesla K80 GPU (·····), Google TPU ASIC (- - ) and Xilinx Kintex UltraScale KU115 FPGA with 16-bit (----), eight-bit (·····) and one-bit (——) fixed-point weights [66, 141].

(b) Embedded-scale platforms: Nvidia Jetson TX1 GPU (- - ), TI Keystone II DSP (- - -) and Xilinx Zynq ZC706 FPGA with 16-bit (·····), eight-bit (----) and one-bit (——) fixed-point weights [58].

FIGURE 2.6: Comparison of roofline models of datacentre and embedded scale DNN inference platforms [Wang et al., 2019]

quantised networks have similar accuracy than floating point and FPGAs become really attractive. They benefit from both a theoretical speed up due to a new computing method and a speed up from a change of hardware.

Overall what makes custom hardware and FPGAs really good at approximate neural networks is their flexibility, low power consumption and high performance on low bit integers.

### 2.3.2 FPGAs Architecture

FPGA stands for Field Programmable Gate Array and consists essentially of a large array of ressources, as shown on Figure 2.7, that is completly reconfigurable. The blocks available to map an application are [Krishna and Roy, 2017]:

- Logic Blocks: Those include NAND gates, multiplexors, LUTs (lookup tables).
- Routing Blocks: They allow logic blocks to be connected.
- Input/Output: They are connected to the logic blocks with routing blocks and enable communication with the environment.
- Memory: Of multiple sorts (BRAM, SRAM...) and store the data needed by the logic functions.

To construct an FPGA application consist of mapping elements into the array with a hardware description language (VHDL, Verilog) to produce the circuit. The distance between connected logic blocks has an impact on the resulting performance as the achievable clock cycle of the overall system will be determined by the longest distance between two connections in the array. The throughput is determined by the latency of the hardware design and the clock frequency. It is crucial to reduce the design to increase the performance.

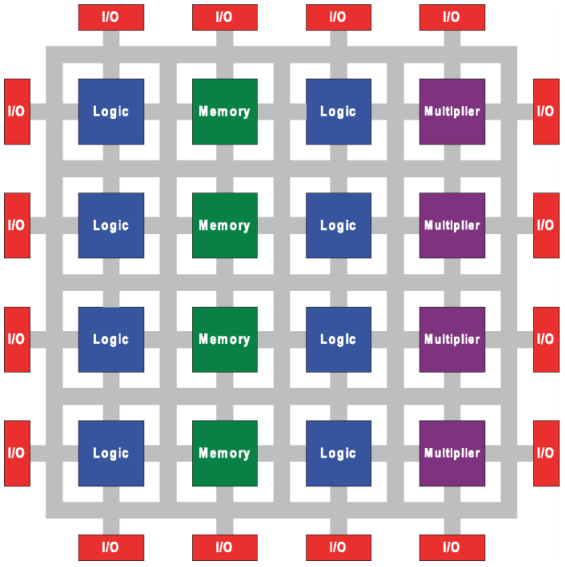



FIGURE 2.7: Basic FPGA structure [Kuon et al., 2008]

### 2.3.3 Existing Frameworks

In order to effectively use neural networks on FPGAs, several frameworks already exist. They provide support to application that are often lacking on FPGAs.

FPGAConvNet [Venieris and Bouganis, 2017] and Caffeinated FPGA [DiCecco et al., 2016] are both frameworks that implement small to medium size CNN models for FPGAs. The former use non-quantised network whereas the latter is an extension of the already existing Caffe framework but for Binarised CNN on FPGAs. They are specialised in CNN mostly because that it the favoured neural network when working with computer vision and this is one of the most used applications used in embedded systems. Other more generalised frameworks exist such as ReBNet [Ghasemzadeh et al., 2017] or FINN [Umuroglu et al., 2016]. They propose complete workflows from training the quantised neural network to the actual transformation of the model into VHDL or Verilog to implement it on actual hardware.

For the rest of the project, FINN was chosen as it was not limited to CNN, enable both binarisation and quantisation and finally because the framework and the repository associated has been updated since the first paper. Unfortunately, ReBNet is lacking support and is restricted to binarised networks. Another framework is worth mentioning for further work, Distiller for Pytorch developed by Intel [Zmora et al., 2018]. It does not provide an implementation for FPGAs but is gaining popularity and frequent updates. This framework works with many different compression techniques (pruning, quantisation etc) and is an interesting tool to study compression on regular hardware.

### 2.3.4 FINN

FINN is a Python library that takes pre-trained, already binarised/quantised NNs, and generates efficient hardware for that quantised neural network. FINN is updated relatively frequently and posses some extensions such as FINN-L [Rybalkin et al., 2018] specialised in LSTMNN (Long-Short Term Memory Neural Networks) which are a kind of recurrent networks. The workflow of the FINN framework is described in Figure 2.8. The first FINN framework could only work with BNN [Umuroglu et al., 2016] but Xilinx, the authors of FINN, released a few years later a framework that work with arbitrary bitwidth for both weights and activation functions [Blott et al., 2018].

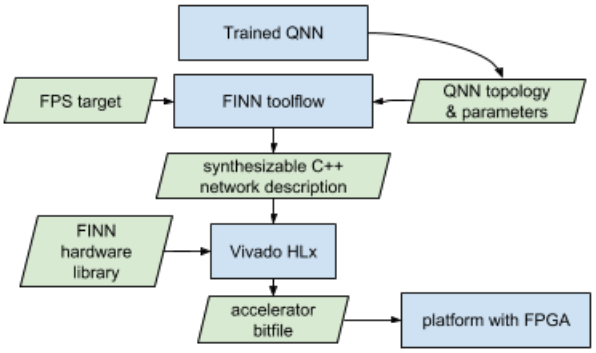



FIGURE 2.8: Generating a quantised neural network on FPGA [Umuroglu et al., 2016]

FINN can work with different target devices (PYNQZ1-Z2, Ultra96) and neural network framework (Theano, Caffe, tensorflow) but to use the fully quantised network, the framework is restricted to Theano. We have to train a network and FINN will then work on an intermediate internal representation of it with some optimisation. Also, the user can tune some of the parameters of the FPGA such as the number of parallelism that will be used while implementing the network on the board. After those architectural choices, we can use the backend of FINN to actually synthesize some synthesisable C++ code for QNNs that Vivado HLS (Xilinx synthesis HDL design software) can use to generate the proper HDL code that design the network on the FPGA.

A common misconception about quantised neural networks is that we could just quantise all the parameters after the neural network has been trained with floating point number. This is fine when we are reducing the precision from 32-bit floating point number to 8-bit or even less but we have to expect high error loss. To ensure the result we get is consistent we have to directly train a quantised neural network. This is a method called quantisation-aware training [Abadi et al., 2015] and employed by Tensorflow with a framework which has a dedicated version for 8-bit networks: Tensorflow Lite. For futher experiments, we train a new network for each parameter between 1-bit and 8-bit for both weights and activation functions.

FINN also provide, after synthesising the code, a clock accurate layer-by-layer cost model. It calculates how many clock cycles are required for each layer so we can find the layer which is the performance bottleneck. The user can react to these cost model results by assigning more hardware to the bottleneck layer, to reduce the number of clock cycles it requires. The user can do this by factorising the input feature map, and applying more processing elements to the layer. We can then, as shown in Figure 2.9, determinate the characteristics of the approximate network without even having to test it with the actual board.

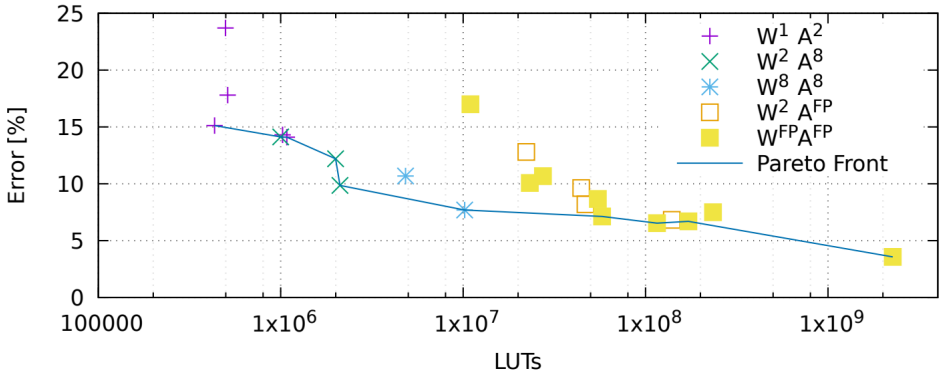



FIGURE 2.9: Accuracy vs. Hardware Cost with Different Precisions for ImageNet [Blott et al., 2018]

## 2.4 Verification of Neural Networks

Robustness is one axis of many metrics to evaluate the performance of a neural net. It has to ensure that the network is robust given certain assumption. We talk about adversarial examples when two similar images (even indistinguishable) for humans are perceived differently by the computer. Those adversarial examples cause misclassfication and give a completely different result while being the same for humans. This topic is quite recent and has grown importance as networks are being used in fields where security is crucial: autonomous driving is one of many examples. This is even more relevant when verifying QNNs of arbitrary precision is an open area and is required to assess the robustness of new approximate methods for neural networks.

### 2.4.1 Floating Point Networks

A branch of the current research is around the use of SMT solver (Satisfiability Modulo Theories). The work done by [Huang et al., 2016] uses linear equation and then solve them with the Z3 solver. They search layer by layer the manipulation of the original image that would fire a different classification by being too far from a safe distance. Another framework called DeepPoly [Singh et al., 2019] has a similar objective that is to say to find adversarial examples. But the method is different as the authors write that the current branch of SMT solvers can only work with small to medium network and that it can be an issue for large CNNs. Their system works with intervals and rectified activation function so that they can handle a larger search space.

### 2.4.2 Quantised Networks

The previous work was aimed at floating point network and not quantised one. The work in [Cheng et al., 2017] focuses exclusively on BNNs as their popularity is growing. Similarly to the work of [Huang et al., 2016], the objective is to transform the problem into an SAT (Satisfiability) problem. All the ideas rely on the use of a combinational mitre which is then transferred to the SAT solver. A combinational mitre is a circuit that has multiple inputs (all the weights and inputs) but only one boolean output. We have to replace all the connections between one layer to the other with a new domain where the connections are much simpler.

## 2.5 Summary

The influence of approximate neural networks is all the more important today as the power required to run them increases faster than the computational power of custom hardware accelerators. The number of different methods for approximate deep learning is important even if the field is quite young. Most of the work relies on quantisation and it is not uncommon to see a mixture of different algorithms as they are independent of one another.

The rest of the project will focus on the FINN framework [Umuroglu et al., 2016] and its quantisation implementation that directly come from the work of [Courbariaux and Bengio, 2016] and [Hubara et al., 2016].

# Chapter 3

# FINN Framework

This chapter presents in detail the workflow and specificities of the FINN framework. Section 3.1 describes the overall workflow and how performances will be measured. Section 3.2 explains the neural network topology used during the project as well as the training environment. Section 3.3 explains the quantisation mechanism used in FINN. Section 3.4 explains the tuning of parallel resources before generating synthesisable code.

## 3.1 Workflow

FINN supports the quantisation of weights and activation functions as well as an optimization of parallel resources used during synthesis. The experiments will assess very low precision neural networks, i.e. reducing precision from 32 bits to 1-8 bits to fit within the resource constraints of FPGAs. The workflow between these components is in figure 3.1. Each experiment will work through the same following process:

**1) Prepare NN:** Import training/testing data in numpy arrays as well as a model in Theano.

**2) Train QNN:** Set the precision for both parameters, chose the training time and training parameters. Once the training process with Theano has finished, we obtain a list of numpy arrays which correspond to the quantised trained weights in each layer. This step will give the error rate for training and testing data.

**3) Generate binary weight file:** In order to load them into the hardware design, numpy arrays need to be converted from real floating-point values into binary values and packed into bin files. Parallel resources can be changed at this time to further tune the performance.

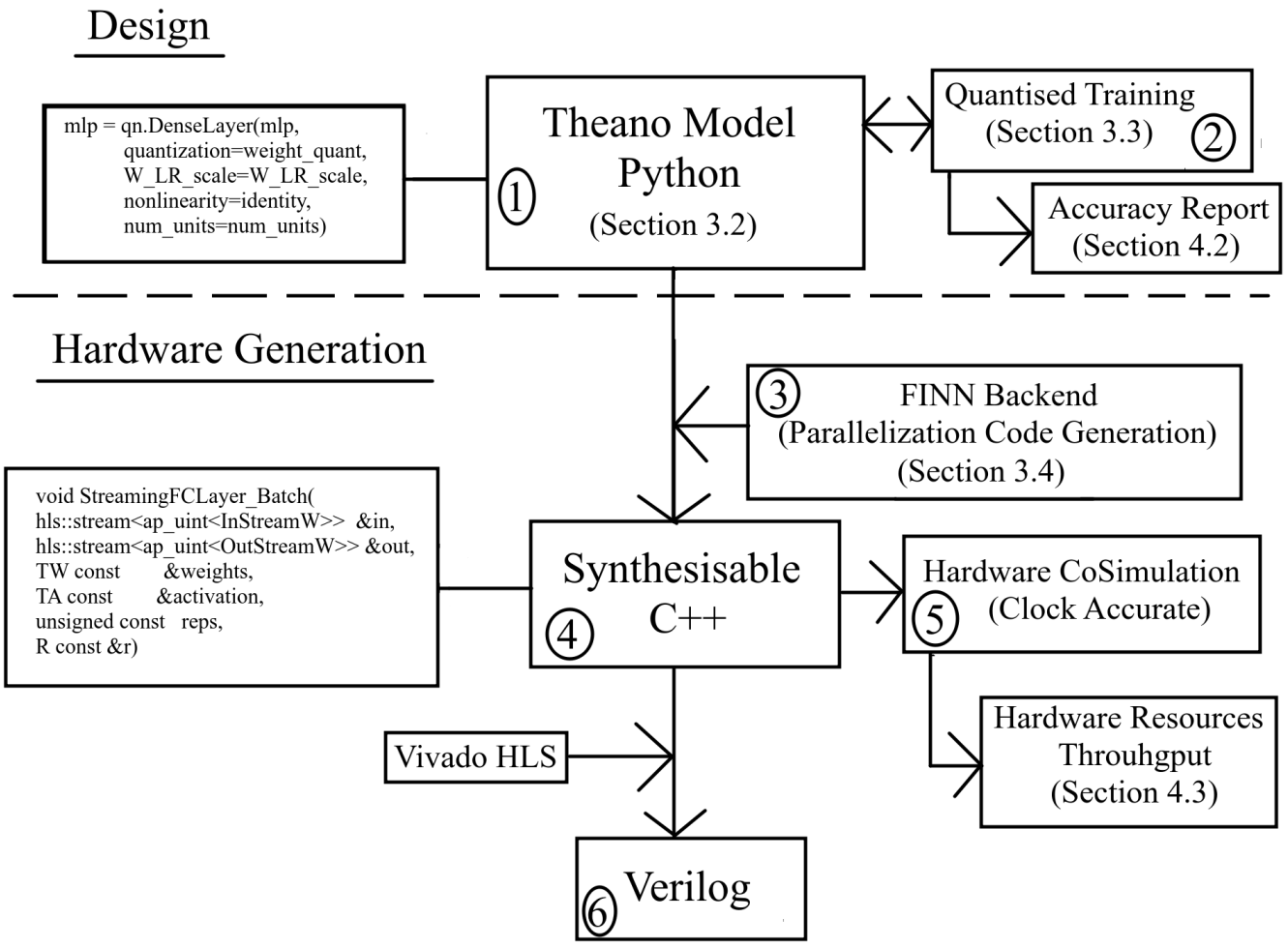



FIGURE 3.1: FINN workflow with the example of a multilayer perceptron

**4) Create configuration files:** We need to fully define the config header file (pre-generated during the weights generation) as well as create the top-level file that instantiates all the layers and the configuration-specific software code that loads the dataset, initializes memories and launches the accelerator.

**5) Synthesize hardware:** We launch HLS synthesis, CoSimulation (use simulated hardware in a software) and the overlay generation for a given precision and network. The output will be an estimatation of the resources used, the Vivado project and the bitstream. This step will give all the hardware related estimation and the throughput of the neural network after a clock accurate simulation.

**6) Generate Verilog:** The last step would be to use Vivado HLS to generate Verilog code to implement the neural network on FPGA.

## 3.2 Neural Network Topology

Figure 3.2 represents the model of the neural network used for all further experiments: a multilayer perceptron. The input layer consists of 784 neurons that represent the 784 pixels from MNIST dataset (28*28 grayscale image). The output is represented by 10 neurons that are the 10 possible classifications for digits.

Between these two, we have three fully connected hidden layers that have all 1024 neurons and each of them is running the same activation function, the hyperbolic tangent. The complete network is described in code Listing 3.1.

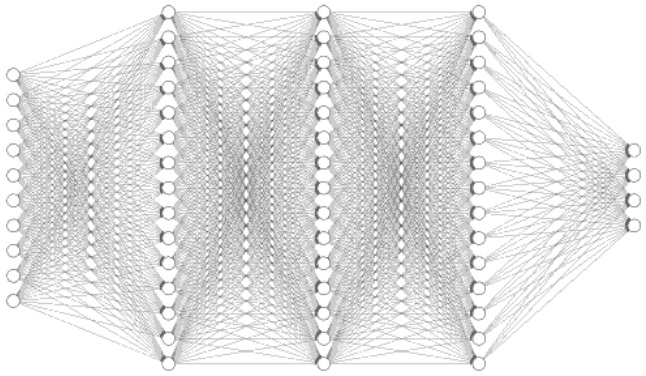


FIGURE 3.2: Neural network topology [LeNail, 2019]

Dropout layers are used during training only as a mean of regularization. A dropout layer will randomly remove at each training batch, some connections between two layers. It helps to avoid overfitting and obtain a model that can generalize on new data.
Batch normalization is a stochastic operation that generally improves the speed and performance of a network. Lasagne's normalization is based on the following:

$$y = \frac{x - \mu}{\sqrt{\sigma^2 + \epsilon}} * \gamma + \beta$$

$\mu$ Mean of the current batch input $x$.

$\sigma$ Variance of the current batch input $x$.

$\epsilon$ Small variable to avoid numerical discontinuty.

$\gamma$ Average statistic computed during training time.

$\beta$ Average statistic computed during training time.

The value stored after training are $\mu, \sigma, \gamma, \beta$. They are used during testing and by the deployed neural network. They are transferred in C++ and hardware implementation with the use of a threshold based on the value after training [Umuroglu et al., 2016]. The other learning parameters used are in Table 3.1.

```
1  mlp = InputLayer(shape=(None, 1, 28, 28),
2          input_var=input)
3
4  mlp = DropoutLayer(mlp,
5          p=dropout_in)
6
7
8  for k in range(n_hidden_layers):
9      mlp = qn.DenseLayer(mlp,
10         quantization=weight_quant,
11         W_LR_scale=W_LR_scale,
12         nonlinearity=identity
13         num_units=1024)
14
15     mlp = NonlinearityLayer(mlp,
16     nonlinearity=activation)
17
18     mlp = BatchNormLayer(mlp,
19         epsilon=epsilon,
20         alpha=alpha)
21
22     mlp = DropoutLayer(mlp,
23     p=dropout_hidden)
24
25
26 mlp = qn.DenseLayer(mlp,
27         quantization=weight_quant,
28         W_LR_scale,
29         nonlinearity=identity,
30         num_units=num_outputs)
31
32 mlp = BatchNormLayer(mlp,
33         epsilon,
34         alpha)
```

**Input size fits a 28*28 pixels image**

**Reduce overfitting during training with dropout regularization**

**Enhanced DenseLayer function from Lasagne that uses quantised weight**

**The activation function used is a custom hyperbolic tangent that works with quantised values**

**Improves speed and stability with normalized inputs**

**Regularization of the hidden layers**

**Output layer**

**Alpha is the exponential moving average factor and epsilon is used to avoid numerical problems while working with square roots**

Listing 3.1: Neural Network Topology

TABLE 3.1: Learning parameters

| Parameters in MLP | | | |
|---|---|---|---|
| Batch Size | 100 | Epochs | 100 |
| Alpha | 0,15 | Epsilon | 10-4 |
| Dropout Input Layer | 0.2 | Dropout Hidden Layer | 0.5 |
| Weight Initialization | [Glorot and Bengio, 2010] | Learning Rate Decay | $(LRE - LRS)^{\frac{1}{NbEpoch}}$ |
| Learning Rate Start (LRS) | $3 * 10^{-3}$ | Learning Rate End (LRE) | $3 * 10^{-7}$ |

The training is done using 50000 images from the MNIST dataset to adjust weights and biases on the neural network. A validation dataset of 10000 images is then used to minimize overfitting. The purpose of this dataset is to be sure that when accuracy is increasing for the training set, there is also an increase in the validation set. If the accuracy were to decrease (or stagnate) for the validation dataset then the neural network would be doing overfitting over the training dataset and the training should stop. Finally, accuracy is measured over a testing dataset.

FINN's backend then converts the model (numpy arrays) in binary weight file and a C++ interpretation. This is then used with Vivado HLS to run C synthesis, CoSimulation and finally generate Verilog. This workflow is described in the general workflow in Figure 3.1.

## 3.3 Quantisation Algorithms

Quantisation restricts real numbers to a range of fixed values based on a predefined quantisation scheme to fix the number of possibilities. This section gives the quantisation operation used for activation function and weights.

### 3.3.1 Binarisation

Binarisation is a fairly simple method. Here the choice is made to transform a value into the scale factor which is 1. So, the two possibilities are -1 and 1 for weight.
The general binarisation process is equation (3.1) and corresponds to Listing 3.2.

$$f(x) = Switch(\lfloor (Sigmoid(x)) \rfloor, -1, 1) \tag{3.1}$$

The value is normalised between [0;1] with the use of a sigmoid function. The function is strictly increasing on $\mathbb{R}$ so the order is kept after shifting. The value is then rounded to be either 0 or 1 and finally, the *switch* condition gives either -1 or 1. The *switch* condition is similar to an IfElse condition but it is based on a tensor condition and not a boolean condition.

```
1  class QuantizationBinary(object):
2      def __init__(self, scale=1.0):
3          self.scale = scale
4          self.min = -scale
5          self.max = scale
6
7      def quantizeWeights(self, X):
8          Xa = hard_sigmoid(X / self.scale)
9          Xb = T.round(Xa)
10         return T.switch(Xb,self.scale,-self.scale)
```

**The choice will either -1 or 1 for binarisation**

**The range is converted to [0;1] and then to -1 OR 1**

Listing 3.2: Weigth Binarisation in FINN

The values during training are still of type *float* but are quantised to a fixed number of possibilities. The actual storage on reduced memory is done during the hardware implementation using FINN backend.

An approximate sigmoid function is used to shift the range of values before binarisation. But the activation function is the hyperbolic function. Figure 3.3 shows those functions as well as the real shape.

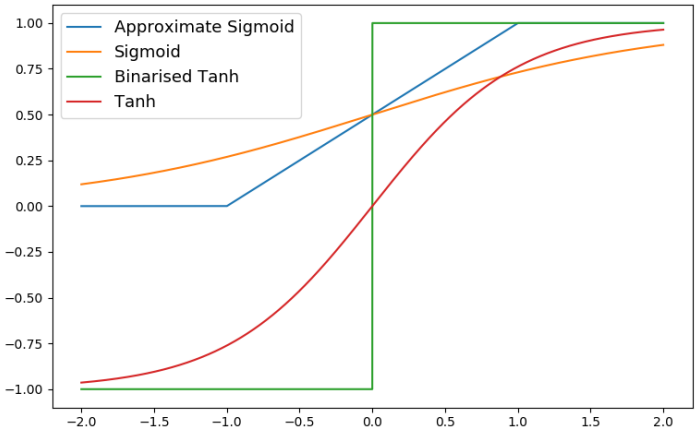



Figure 3.3: Usual activation functions and their binarised counterparts

Round3 on line 5 and the functions used in Listing 3.3 all come from [Courbariaux and Bengio, 2016]. They approximate standard activation functions in deep learning. They are all strictly increasing function and differentiable.

```
1 def hard_sigmoid(x):
2     return T.clip((x+1.)/2.,0,1)
3
4 def binary_tanh_unit(x):
5     return 2.*round3(hard_sigmoid(x))-1
6
7 class QuantizationBinary(object):
8     def quantize(self, X):
9         return binary_tanh_unit(X / self.scale)*self.scale
```

**A call to this function will return a binarised value based on the binarised scheme**

LISTING 3.3: Activation Function Binarisation in FINN

### 3.3.2 Quantisation

For quantisation, we discretise the range of the problem so that values in that range will be rounded to a close neighbour. The general quantised scheme used by FINN is described in equation (3.2).

$$f(x) = \frac{\lfloor (Clip(x, min, max) + shift) * scale \rfloor}{scale} - shift \tag{3.2}$$

$$max = 2^{Wordlength - Fraclength - 1} - 2^{-Fractlength}; min = -max$$

$$shift = 2^{Wordlength - Fraclength - 1} - 2^{-Fractlength}$$

$$scale = \frac{2^{Wordlength-1} - 1}{2^{Wordlength - Fraclength - 1} - 2^{-Fractlength}}$$

- *Wordlength*: The number of quantised value in the new range.
- *Fraclength*: The precision of the fractional part of the rounded values.

Both *Wordlength* and *Fraclength* work together as they define the number and precision of values inside the set of qunatised number. *Fractlength* defines the minimal step between two consecutive value and *Wordlength* defines the number of total step in the range.

In all neural network, *Fraclength* is always set to *Wordlength - 2*, so we can operate further optimisation for the quantised computation. We will indicate *Wordlength* with $n \in [2; 8]$, it indicates the quantisation precision. The general quantisation formula becomes for $x \in [min; max]$ :

$$f(x) = \frac{\lfloor x2^n + 2^{n-1} - 1 \rfloor}{2^{n-2}} - 2 + \frac{1}{2^{n-2}} \tag{3.3}$$

The other components in equation (3.2) can be simplified:

$$max = 2 - \frac{1}{2^{n-2}}; min = -2 + \frac{1}{2^{n-2}}$$

$$shift = max = 2 - \frac{1}{2^{n-2}}$$

$$scale = \frac{2^n - 2}{2 * (2 - \frac{1}{2^{n-2}})} = 2^{n-2}$$

Table 3.2: Example of quantised values

| Value | Precision | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| 0.136 | 1 | 0 | 0 | 0.25 | 0.125 | 0.125 | 0.125 | 0.140625 |
| 0.357 | 1 | 0 | 0.5 | 0.25 | 0.375 | 0.375 | 0.34375 | 0.359375 |
| 0.639 | 1 | 1 | 0.5 | 0.75 | 0.625 | 0.625 | 0.625 | 0.640625 |
| 1.135 | 1 | 1 | 1 | 1.25 | 1.125 | 1.125 | 1.125 | 1.140625 |
| 2 | 1 | 1 | 1.5 | 1.75 | 1.875 | 1.9375 | 1.96875 | 1.984375 |
| 314 | 1 | 1 | 1.5 | 1.75 | 1.875 | 1.9375 | 1.96875 | 1.984375 |

The quantised values, as shown in Table 3.2 are always set between -2 and 2 with $2^n - 1$ values in this interval. The values presented are all strictly positive but the quantisation range is symmetric. The step between each quantised value is $Fractlength = \frac{1}{2^{n-2}}$. When $n$ increase, the number of quantised value increase and we can obtain values close to the upper and lower bound of the interval. Figure 3.4 shows the mapping of the full range of real value in the limited range of quantised value.

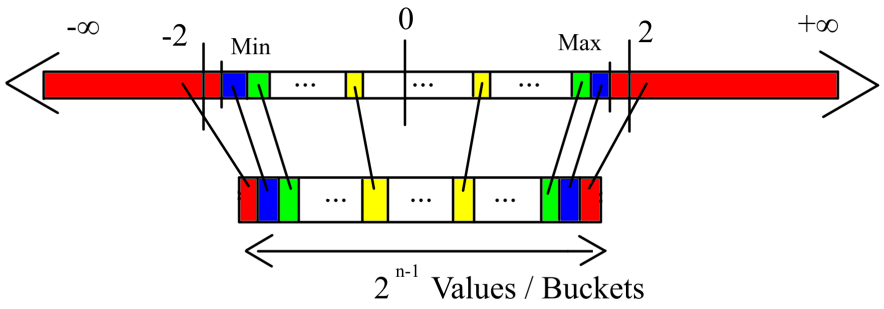



FIGURE 3.4: Mapping of quantised values

The range is optimised for this problem because the only activation function used is the hyperbolic tangent $tanh(x) = \frac{e^x - e^{-x}}{e^x - + e^{-x}}$. It has two asymptotes that goes towards -1 and 1, and we have $tanh(2) = 0.964$. So, the saturation plateau of the activation function is almost attained and the majority of the function is being quantised.

A final optimisation can be done when $x \notin [min; max]$

if x $\leq$ min :

$$\begin{aligned} f(x) &= \frac{\lfloor (-shift + shift) * scale \rfloor}{scale} - shift \\ &= -shift = min \\ &= -2 + \frac{1}{2^{n-2}} \end{aligned}$$

if x $\geq$ max :

$$\begin{aligned} f(x) &= \frac{\lfloor 2^n - 2 \rfloor}{2^{n-2}} - 2 + \frac{1}{2^{n-2}} \\ &= \frac{2^n - 2}{2^{n-2}} - 2 + \frac{1}{2^{n-2}} \\ &= \frac{2^n - 2^{n-1} - 1}{2^{n-2}} \\ &= 2 - \frac{1}{2^{n-2}} = max \end{aligned}$$

The quantisation process is shown in Listing 3.4. During run time, for each value, the minimal and maximal range must be found based on the precision given. After that, we need to find the values that will do the mapping of our input to the new range. The last step is the actual computation of the new quantised value. The overall result is that the input is rounded to the closest quantised value in the new range.

```
1  class QuantizationFixed(object):
2      def __init__(self, wordlength, fraclength):
3          self.wordlength = wordlength
4          self.fraclength = fraclength  <---------  In all layers fractlength is set
5          self.set_min_max()                        up to the value wordlength - 2
6          self.set_scale_shift()
7
8      def set_min_max(self):
9          min_val = - (2.**(self.wordlength - self.fraclength - 1))
10         max_val = - min_val - 2.**-self.fraclength
11         self.min = min_val
12         self.max = max_val  <----  Find the new range for quantised values
13                                                    Find the shift
14     def set_scale_shift(self):                     and scale value
15         self.scale = (2.**self.wordlength - 2) / <----  associated with
16                         (self.max - self.min)      this quantisation
17         self.shift = -self.min
18
19     def quantizeWeights(self, X):  <--  For further details, see equation (3.2)
20         return T.round( (T.clip(X, min, max) + shift)*scale)
21                                 / scale - shift
```

Listing 3.4: Weigth Quantisation in FINN

Figure 3.5 shows the shape of tanh for different quantisation configuration and Listing 3.5 shows how this function is being constructed in FINN.

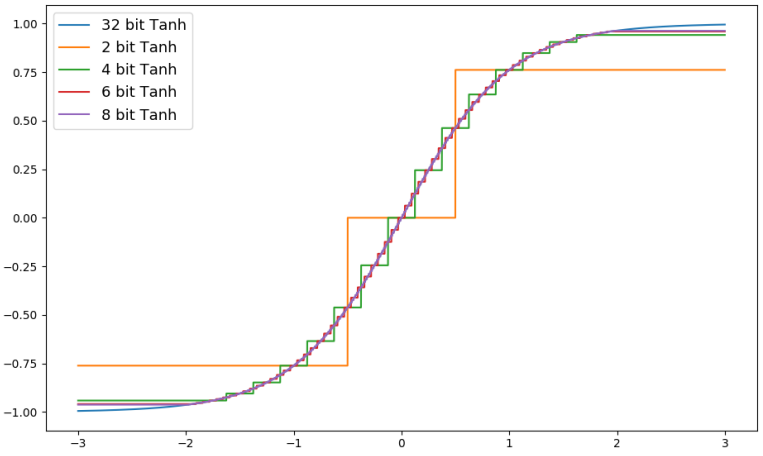



FIGURE 3.5: Hyperbolic tangent with different quantisation configuration

The quantisation of activation functions work similarly to weight quantisation. The only fixed point function used is the hyperbolic tangent even though other usual function can easily be created.

```
1  class QuantizationFixed(object):
2      def quantize(self, X):
3          return round3((T.clip(X, min, max) + shift)*scale)
4                        / scale - shift
5
6  class FixedHardTanH(object):  <---------
7      def __init__(self, quantization):
8          self.quantization = quantization
9
10     def __call__(self, x):  <----
11         y = T.clip(x, -1, 1)
12         return self.quantization.quantize(y)
```

**Create a quantisation object that has the same property explained in Listing 3.4**

**This function returns a quantised value based on the previous quantisation scheme**

LISTING 3.5: Activation Function Quantisation in FINN

## 3.4 Parallelising Quantised Neural Networks

Before transforming the trained neural network into synthesizable C++ (with Finnthesizer.py), we can specify the number of parallel resources used for each layer of the network: PE (Processing Element) and SIMD (Single Instruction Multiple Data).
The amount of SIMD for each layer is based on the weight precision of the network and is limited to 64 for hardware reasons. A valid value must satisfy one condition: 64 divided by the layer's precision has to be a power of 2. So 8 would work for every precision from 1 to 8 bit but low precision enables the user to choose highly parallelizable architecture. PEs are more flexible and therefore be used to obtain a uniform latency for each layer. The estimated latency is calculated with the following formula:

$$Latency = \frac{Ops}{2 * SIMD * PE}$$

Ops Number of operations between the two layers, calculated with the number of neurons of two consecutive layers.

SIMD SIMD value for this layer.

PE PE value for this layer.

The latency is given for each layer which gives the user information about which layer will be a bottleneck during runtime.

# Chapter 4

# Evaluation

This chapter presents the methodology, results and analysis of experiments that measure the trade-off between accuracy and hardware resources for different neural network quantisation configurations. Sections 4.1 the setup for the experiments. Section 4.2 evaluates absolute accuracy performance across all the experiments. Section 4.3 evaluates space performance of each neural network. Section 4.4 compares the *relative* performance across six performance metrics for the 64 quantisation configurations and Section 4.5 discusses the results.

## 4.1 Measurements Platform

### 4.1.1 Host Device and Software Components

All experiments were done using Ubuntu 18.04 with an Intel i5-7200U and 8 Gb of RAM. The whole process takes around seven hours for a single quantisation configuration.
Some other library specifications are needed to properly run the FINN front-end to quantised a neural network:

- FINN, a framework for generating high performance FPGA hardware implementations of neural networks
- Vivado v2018.3, a software suite produced by Xilinx for synthesis and analysis of hardware designs
- Python 2.7.15 with Numpy 1.15 and Scipy 0.19.1
- Theano 0.9.0 for neural network implementation and training

- Lasagne 0.2.dev1
- Pylearn

### 4.1.2 Target Device

FINN provides us with information on hardware resource estimates and the actual implementation for a target device. The duration of the experiments does not allow to provide exhaustive tests on the two targets available in FINN (pynqZ1-Z2 and ultra96) so the choice was made to work only with the pynqZ XC7Z020 as it takes less time to synthesize hardware designs for it. The board specifications are shown in table 4.1.

TABLE 4.1: Feature Summary for Pynq Zynq XC7Z020

Pynq Zynq XC7Z020 Specification

| | | | |
|---|---|---|---|
| **Processor Core** | Dual-core ARM Cortex-A9 MPCore™ with CoreSight™ | **Programmable Logic Cells** | 85K |
| **Maximum Frequency** | 766 MHz | **Look-Up Tables (LUTs)** | 53,200 |
| **L1 Cache** | 32 KB Instruction<br>32 KB data per processor | **Flip-Flops (FFs)** | 106,400 |
| **On-Chip Memory** | 256 KB | **Block RAM** | 4.9 Mb |

Of the 64 quantised neural networks, only four actually fit on this FPGA. The other 60 do not fit within the resource constraints of this FPGA, validating the need for aggressive compression approaches such as quantisation, on relatively small boards.

### 4.1.3 Performance Metrics

The following sections measure the algorithmic accuracy (Section 4.2) and hardware resource requirements (Section 4.3) of the quantised neural networks. We take the classification speed of 356633 28*28 images per second, for the MLP neural network topology being evaluated, from the literature [Doe, 2019].

Whilst this result is for 2 bit activation function precision and 1 bit weight precision, Xilinx state that for the same parallelisation configuration this classification speed is the same for all quantised configuration [Gambardella, 2019]. The latency at different precisions for one image is going to be different (less latency for 1 bit precision), but the throughput for multiple images will be the same for all precisions because of dataflow pipelining. We therefore assume a classification speed of 356633 images per second for all 64 quantised configuration.

## 4.2 Absolute Accuracy Performance

Each colour in all following heat maps represents the measurement of one performance metric of 64 neural networks with a quantised weight W-X and quantised activation function A-Y with $X, Y \in [1; 8]$. Accuracy is measured at an interval of 10 epochs, starting at 10 up to 100. Using 1-3 bits weights has a noticeable effect on accuracy, i.e. between 3.9%-4.7% dropping down to below 3.7% using 4 bits or more. Training further with 40-100 epochs shifts the noticeable accuracy boundary to just 1 bit weight. The quantisation of activation functions has a steady impact on accuracy, i.e. higher precision activation function result in better accuracy, however, this change is not as dramatic as changes to weight precision.

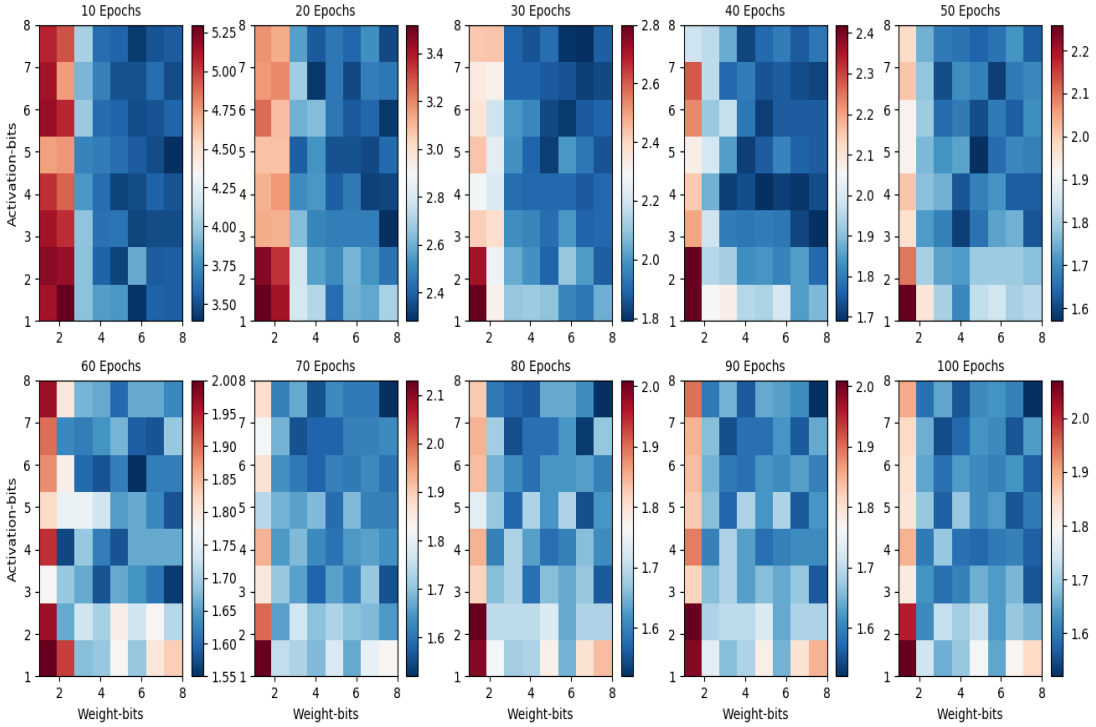



FIGURE 4.1: Error rate based on quantisation precision during training, measured in epochs

Figure 4.2 shows previous heatmaps as surfaces for several timestamps. We see that as training time last the overall surface flatten which means the absolute difference between the best and worst quantisation configuration greatly diminish. Also, we observe a major gap between low precision configuration (1 or 2 bit) and the rest.

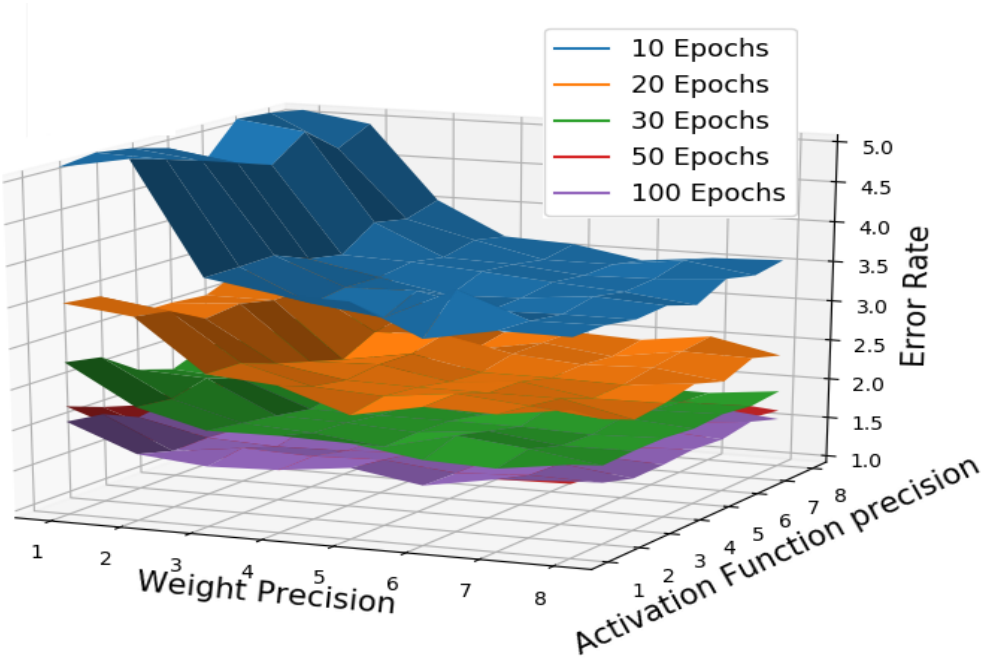



FIGURE 4.2: Accuracy plot as a Surface

Figure 4.3 shows the diagonal of the previous surfaces which means only configurations where both weight and activation function have the same precision are represented, i.e. W1A1, W2A2, etc, since users may want to quantise the whole network to one single precision for convenience. The accuracy steadily increases over time even though it is hardly perceptible after 50 epochs. We must be close to an upper limit and any further training would result in either overfitting or a tiny increase.

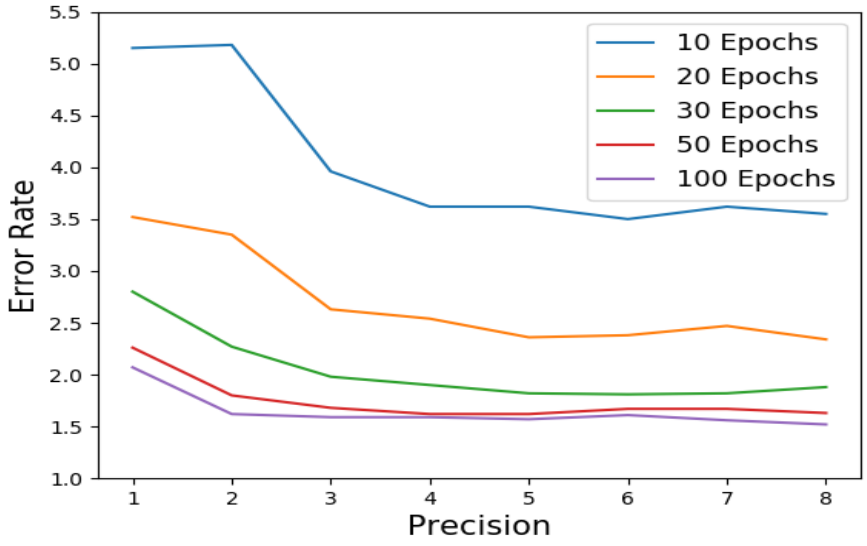



FIGURE 4.3: Accuracy of the Diagonal

## 4.3 Absolute Space Performance

The following results are based on a cosimulation (simulated hardware in software) and some parallel resources have been allocated for each layer. To keep the results consistent and to compare only the influence of quantisation, each configuration has the same SIMD and PE value, shown in Table 4.2.

TABLE 4.2: Parallel resources used in each configuration

| | Input layer | Hidden layer 1 | Hidden layer 2 | Hidden layer 3 |
|---|---|---|---|---|
| SIMD Value | 8 | 8 | 8 | 8 |
| PE Value | 32 | 64 | 32 | 16 |

As explained in Section 3.4, a SIMD value of 8 is the highest possible value for all quantisation configuration. The PE values are chosen to obtain a uniform latency for each layer in the network.

### 4.3.1 LUT Resources

LUTs are used as processing units and mimic logic gates, this is the basic block in an FPGA board.
The overall heat map of Figure 4.4 is close to having a linear behaviour according to both weight and activation function precision. The results are slightly skewed toward the activation function precision as changes are faster according to this parameter. Also, we observe an expected outcome with the top-right and bottom-left corner, the former consumes the most resources and the latter uses the least resources.

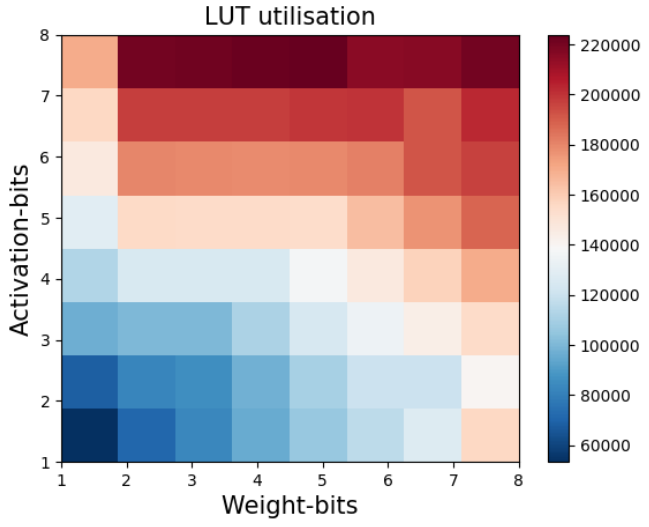



FIGURE 4.4: LUT used for each neural network

### 4.3.2 FF Resources

FFs are used as a form of storage units as they can store state information over a clock cycle. They are also part of the basic blocks of an FPGA.
Figure 4.5 heat map is close to having a linear behaviour according to activation function precision but is rather uniform with weight precision. We observe the same outcome as seen in Section 4.3.1 with LUTs for extreme values. We will find this behaviour between the two extremes of precision where the top-right corner consumes the most and the bottom-left the least, in all space-related metric but the heat maps will have a different evolution.

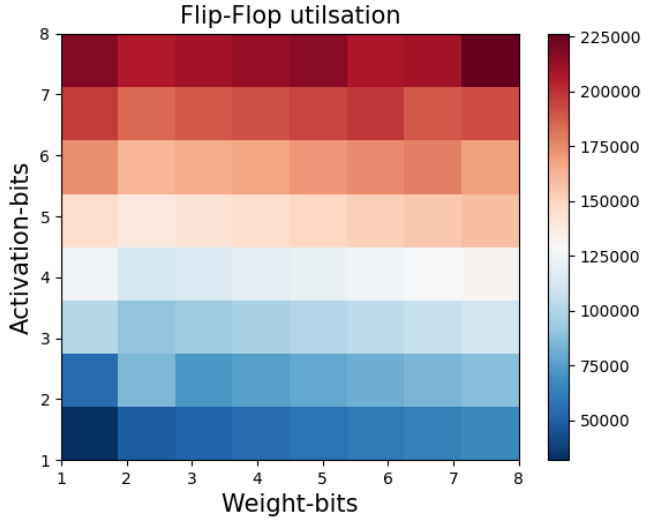



FIGURE 4.5: FF used for each neural network

### 4.3.3 BRAM Resources

Even though FFs and LUTs can store small amounts of data, BRAMs have far greater storage capacity and are used by hardware synthesis tools for arrays and large data structures.
Values in Figure 4.6 are rather uniform column-wise with changes inferior to 20% between the two extremes of the same column. Therefore BRAM consumption depends almost exclusively on weight precision. The increase in BRAM use is linear to the precision of weights, whilst activation function precision does not affect BRAM use. Its evolution is linear regarding this parameter.

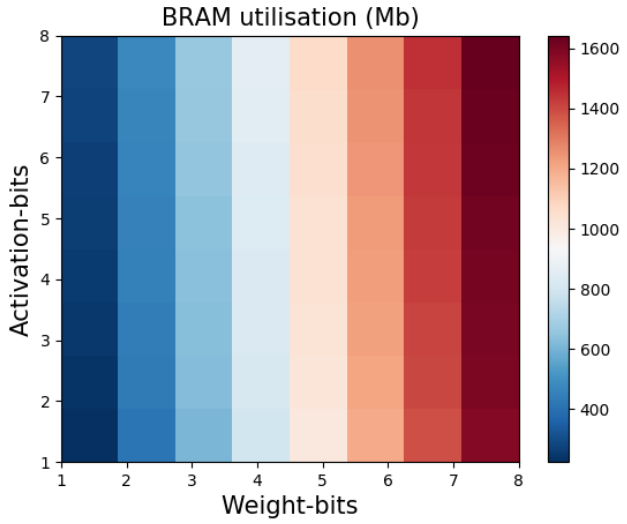



FIGURE 4.6: BRAM used for each neural network

### 4.3.4 Hardware Design Complexity

The hardware design complexity refers to the size of two VHDL files ( *procsysBlackBox-Jamsimnetlist.v* and *procsysBlackBoxJamsimnetlist.vhdl*) within the Vivado project created for each experiment. Netlist files describe the connectivity of hardware components and give further information on how widespread and complex hardware connections will be. These files reflect how changes in precision impacts the complexity of the generated hardware circuit.

Compared to all other measures, we obtain here linearity with respect to the two variables. The heat map is slightly skewed toward the weight precision but it is a minor effect that can be observed.

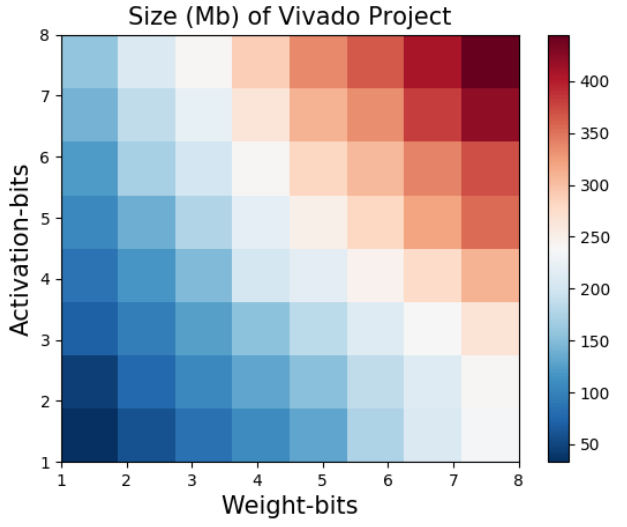



FIGURE 4.7: Hardware complexity for each neural network

## 4.4 Relative Quantisation Performance

### 4.4.1 Relative Performance

Sections 4.2 to 4.3 present absolute performance values. This section highlights the *relative* trade offs and sweet spots in the 64 quantisation configuration from those sections.

Each metric will define one branch in a radar chart. For each metric, the scale has been normalised and inverted between 0 and 1 so that 0 equals the worst performance across all experiments and 1 represents the best overall performance. For each branch the closer the value is to 1, the better its performance is relative to the other 63 quantisation configurations. This means that for hardware resources a higher value means low hardware use which is indeed better. The following list explain the scale of each metrics used in plots of Section 4.4.2.

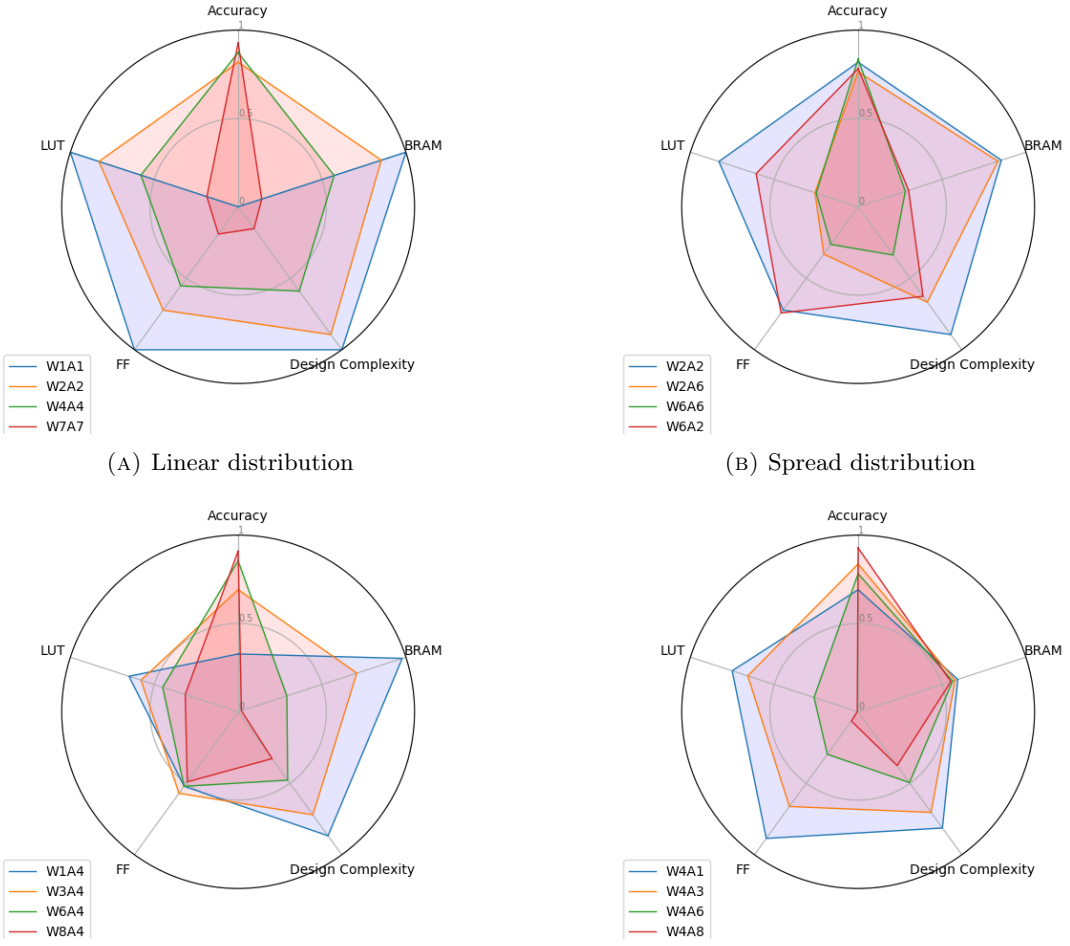
- **Accuracy** 1.52% (0) up to 2.07% (1).
- **Design Complexity Size** 32.9Mb (0) up to 445Mb (1).
- **FF** 31954 (0) up to 226282 (1).
- **BRAM** 224 (0) up to 1643 (1).
- **LUT** 53336 (0) up to 223910 (1).

### 4.4.2 Radar Charts

The four examples in Figure 4.8 represents different configuration.

- Linear distribution (Fig 4.8.a) takes four values across the diagonal: W1-A1, W2-A2, W4-A4 and W7-A7.
- Spread distribution (Fig 4.8.b) take four values that are in the four different corners of each heat map, W2-A2, W2-A6, W6-A6 and W2-A6.
- Weight oriented distribution (Fig 4.8.c) is linear to the weight parameter for a constant activation function precision: W1-A4, W3-A4, W6-A4 and W8-A4.
- Activation oriented distribution (Fig 4.8.d) is linear to the activation function parameter for a constant weight precision: W4-A1, W4-A3, W4-A6 and W4-A8.

For some quantisation configuration, we clearly see a correlation between the configurations and a performance metric, e.g. BRAM use for weights. For other metrics, such as accuracy, the pattern is less clear because for these metrics the scale before normalisation is not very wide and thus all values seem to be stack around one point in the radar chart.




(A) Linear distribution

(B) Spread distribution

(C) Weight oriented distribution

(D) Activation oriented distribution

FIGURE 4.8: Relative radar charts for several configuration

## 4.5 Evaluation Discussion

### 4.5.1 Analysis

The accuracy and hardware resource (space) metric results give a clear insight into the impact of weight quantisation and activation function quantisation. Activation function precision greatly affects processing hardware elements (FF, LUT). Weight precision mostly affects BRAM use and both parameters equally affect the complexity if the

generated hardware. Increasing training to more than 40 epochs allows weights to be quantised to 2 bits representations without significant impact on accuracy versus 8 bits representation.
A general criteria for the trade-off across the performance metrics are:

- The amount of hardware resources is mostly affected by activation functions.
- Memory mainly depends on weight precision and greatly increase with it.
- Accuracy is highest with higher precision, i.e. least aggressive quantisation. The biggest improvement step in accuracy is switching from 1 to 2 bits weight precision.
- The precision of a neural network affects the complexity of the hardware design generated by FINN, i.e. a higher precision neural network requires more complex interconnected hardware designs.

The sweet spot in the quantisation design space seems to be around W3 A3 where the maximum accuracy for all the 64 quantisation configuration is almost achieved whilst minimising hardware resource use.

### 4.5.2 Limitations

The neural network used to classify the MNIST are quite simple and provide good accuracy for all configurations. A more complex network could lead to more disparate results and larger variation across configurations and a shift in where the sweet spots lie. That could be done with the CIFAR10 dataset, but the training time would increase dramatically as the number of weights and neurons of the neural network increases.

We must also question the fragility of the results and question if they are only valid for this specific experiment platform, i.e. using FINN, the MNIST model, Theano and Vivado, and the versions of these softwares that were used.

A valid test would be to compare the results obtained with different model description (Keras, Tensorflow...) and also with different implementations of the same methods. However, this comparison could not be done for neural networks implemented on FPGAs as very few frameworks are able to generate hardware description code for FPGAs from high level neural network descriptions. We could still compare quantised neural networks between them on general metrics using different frameworks.

# Chapter 5

# Conclusion

## 5.1 Summary

Methods for compressing neural networks are needed to address the needs from industry to extend the deployment of deep learning algorithms to embedded processors, e.g. for autonomous vehicles and drones. Such methods not only reduce the hardware resources used but also the overall energy consumption of the system as less workload and smaller chips diminish the energy needed for computation.

The field of compression techniques is large and possesses a diversity of approaches that are often intertwined. This dissertation only focuses on the quantisation of weights and activations functions to evaluate the performance of one kind of neural network on FPGAs. Using the FINN Python framework, this dissertation finds a compromise between hardware resources required and classification accuracy.

The empirical results in Chapter 4 show that quantised networks achieve similar accuracy while vastly reducing the number of resources needed. FINN framework also provides an easy solution to implement an existing network on FPGAs and then assist in the deployment of custom hardware to embedded applications. The loss of accuracy with aggressive quantisation is low compared to the impact it has on other metrics and as such the best trade-off to optimise both resources used and performance achieved would be to aim for a 2 or 3-bit quantisation for the neural network with three hidden layers that this dissertation evaluates.
Those results raise awareness on compression methods and sweet spots that could be achieved in approximate neural networks.

## 5.2 Further Work

This dissertation could be extended with further experiments to explore new metrics and enhance the knowledge of deep learning performance when targeting resource-constrained devices.

This dissertation focused only on the influence of weight and activation function quantisation on the performance metrics. But another key aspect of FINN is performance tuning. Further experiments could explore the impact for a fixed precision of the tuning of parallel resources used. The goal would be to evaluate how those resources could change the outcome of the performance metrics. It would be useful to measure the performance for different precision while adjusting the parallel resources to obtain the same overall hardware utilisation on the FPGA board.

An important metric was set aside during this project even though it is crucial in many applications. This is the efficiency of the network or energy consumption. It would be measured in classification per Joule. This is a determining metric for embedded application as one of the limiting factors with low hardware available is the energy needed to power all the system. To that end, we would need to implement on FPGA board the neural network to conduct the same experiment on the hardware device, rather than just clock accurate simulation.

Finally, there is a need to explore the effect of applying quantisation to a neural network in a heterogeneous way, i.e. ending with a neural network with each layer, weight and activation function with its own finely tuned precision. Indeed, the FINN frameworks supports this flexibility, the scope of this dissertation was to apply quantisation homogeneously. The goal would be to empirically find sweet spots where we could aggressively quantise the most negligible layers while keeping a higher precision where it matters the most.

# Bibliography


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.

Bengio, E., Bacon, P., Pineau, J., and Precup, D. (2015). Conditional computation in neural networks for faster models. *CoRR*, abs/1511.06297.

Blott, M., Preußer, T. B., Fraser, N. J., Gambardella, G., O'Brien, K., and Umuroglu, Y. (2018). FINN-R: an end-to-end deep-learning framework for fast exploration of quantized neural networks. *CoRR*, abs/1809.04570.

Cheng, C., Nührenberg, G., and Ruess, H. (2017). Verification of binarized neural networks. *CoRR*, abs/1710.03107.

Cheng, Y., Yu, F. X., Feris, R. S., Kumar, S., Choudhary, A. N., and Chang, S. (2015). Fast neural networks with circulant projections. *CoRR*, abs/1502.03436.

Courbariaux, M. and Bengio, Y. (2016). Binarynet: Training deep neural networks with weights and activations constrained to +1 or -1. *CoRR*, abs/1602.02830.

DiCecco, R., Lacey, G., Vasiljevic, J., Chow, P., Taylor, G. W., and Areibi, S. (2016). Caffeinated fpgas: FPGA framework for convolutional neural networks. *CoRR*, abs/1609.09671.

Doe, C. (2019). Lfc qnn inference. git commit `https://github.com/Xilinx/BNN-PYNQ/commit/10b2ab55a2920f0e9e9ea3cafb5ea4a55a95f615#diff-c4016f96c5a0a373ddd69f66ed8aea1c`.

Gambardella, G. (2019). Personal communication on throughput management in finn (email conversation).

Ghasemzadeh, M., Samragh, M., and Koushanfar, F. (2017). Resbinnet: Residual binary neural network. *CoRR*, abs/1711.01243.

Glorot, X. and Bengio, Y. (2010). Understanding the difficulty of training deep feedforward neural networks. In *In Proceedings of the International Conference on Artificial Intelligence and Statistics (AISTATS'10). Society for Artificial Intelligence and Statistics*.

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

Han, S., Pool, J., Tran, J., and Dally, W. J. (2015). Learning both weights and connections for efficient neural networks. *CoRR*, abs/1506.02626.

Hinton, G., Vinyals, O., and Dean, J. (2015). Distilling the knowledge in a neural network. In *NIPS Deep Learning and Representation Learning Workshop*.

Huang, X., Kwiatkowska, M., Wang, S., and Wu, M. (2016). Safety verification of deep neural networks. *CoRR*, abs/1610.06940.

Hubara, I., Courbariaux, M., Soudry, D., El-Yaniv, R., and Bengio, Y. (2016). Quantized neural networks: Training neural networks with low precision weights and activations. *CoRR*, abs/1609.07061.

Krishna, G. and Roy, S. (2017). *Fundamentals of FPGA Architecture*.

Krizhevsky, A. (2009). Learning multiple layers of features from tiny images. Technical report.

Kuon, I., Tessier, R., and Rose, J. (2008). Fpga architecture: Survey and challenges. *Foundations and Trends® in Electronic Design Automation*, 2(2):135–253.

LeCun, Y. and Bengio, Y. (1998). The handbook of brain theory and neural networks. chapter Convolutional Networks for Images, Speech, and Time Series, pages 255–258. MIT Press, Cambridge, MA, USA.

LeCun, Y. and Cortes, C. (2010). MNIST handwritten digit database.

LeNail (2019). Nn-svg: Publication-ready neural network architecture schematics. *Journal of Open Source Software*, 4(33), 747.

Lu, D. (2019). Creating an ai can be five times worse for the planet than a car. New Scientist.

Pouyanfar, S., Sadiq, S., Yan, Y., Tian, H., Tao, Y., Reyes, M. P., Shyu, M.-L., Chen, S.-C., and Iyengar, S. S. (2018). A survey on deep learning: Algorithms, techniques, and applications. *ACM Comput. Surv.*, 51(5):92:1–92:36.

Rybalkin, V., Pappalardo, A., Ghaffar, M. M., Gambardella, G., Wehn, N., and Blott, M. (2018). FINN-L: library extensions and design trade-off analysis for variable precision LSTM networks on fpgas. *CoRR*, abs/1807.04093.

Singh, G., Gehr, T., Püschel, M., and Vechev, M. (2019). An abstract domain for certifying neural networks. *Proc. ACM Program. Lang.*, 3(POPL):41:1–41:30.

Stallkamp, J., Schlipsing, M., Salmen, J., and Igel, C. (2012). Man vs. computer: Benchmarking machine learning algorithms for traffic sign recognition. *Neural Networks*, (0):–.

Umuroglu, Y., Fraser, N. J., Gambardella, G., Blott, M., Leong, P. H. W., Jahre, M., and Vissers, K. A. (2016). FINN: A framework for fast, scalable binarized neural network inference. *CoRR*, abs/1612.07119.

Venieris, S. I. and Bouganis, C. (2017). fpgaconvnet: A toolflow for mapping diverse convolutional neural networks on embedded fpgas. *CoRR*, abs/1711.08740.

Wang, E., Davis, J. J., Zhao, R., Ng, H., Niu, X., Luk, W., Cheung, P. Y. K., and Constantinides, G. A. (2019). Deep neural network approximation for custom hardware: Where we've been, where we're going. *CoRR*, abs/1901.06955.

Wang, S., Li, Z., Ding, C., Yuan, B., Wang, Y., Qiu, Q., and Liang, Y. (2018). C-LSTM: enabling efficient LSTM using structured compression techniques on fpgas. *CoRR*, abs/1803.06305.

Wang, Z., Lin, J., and Wang, Z. (2017). Accelerating recurrent neural networks: A memory-efficient approach. *IEEE transactions on very large scale integration (VLSI) systems*, 25(10):2763–2775.

Zmora, N., Jacob, G., Zlotnik, L., Elharar, B., and Novik, G. (2018). Neural network distiller.