Energy Efficient Approximations for Neuromorphic Computing

Sankeerth Durvasula

Department of Electrical Engineering
Indian Institute of Technology Madras

This dissertation is submitted for the degree of
Bachelors and Masters in Technology

May 2018

THESIS CERTIFICATE

This is to certify that the thesis titled Energy Efficient Approximations over Neuromorphic Computing, submitted by Sankeerth Durvasula, to the Indian Institute of Technology, Madras, for the award of the degree of Bachelors and Masters of Technology, is a bona fide record of the research work done by him under our supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma

Prof.Kamakoti Veezhinathan
Project Guide
Professor
Dept. of Computer Science
IIT-Madras, 600 036

Prof. Janakiraman Veeraraghavan
Project Co-guide
Assistant Professor
Dept. of Electrical Engineering
IIT-Madras, 600 036

Acknowledgements

I would like to thank, first and foremost Prof. V. Kamakoti and Prof. Vijay Raghunathan whose help and support allowed me to work on my Master's thesis. I would also thank Arnab Raha, whom I had always reached out to for guidance and help with my thesis. Last but not the least, I would thank my friends and family for their support.

Abstract

Artificial Neural Networks are central to some of the most established Deep learning and AI techniques. Many applications of deep learning are error resilient and hence there's value to bringing down the energy consumed by ANNs. Implemented with Software techniques alone, a large percent of energy computations can be traded off for an insignificant quality loss. Here, we present a new approximation strategy for neural networks and demonstrate it's implementation on an FPGA platform.

Table of contents

List of figures

List of tables

Chapter 1

Introduction

1.1 Error Resilient Computing

Many algorithms dealing with data manipulation, such as the simple KNN and KMEANS for image segmentation, JPEG/MPEG encoding and decoding algorithms have a property special to them in that the output of these algorithms do not get affected by any noticeable amount when a small part for the data is corrupted or is slightly inaccurate.
This property can be used to our advantage by artificially introducing these inaccuracies for a fair trade-off of a large number of CPU operations. By compromising the 100% quality of the output of the algorithm, we could save a lot of computation power. Leveraging this idea is at the core of the area of approximate computing.

The field of Approximate computing is a new paradigm that utilizes this property of error resiliency in it's applications by relaxing the number of uncritical computations in the algorithm.

1.1.1 Error curves

There is a question of how much approximation is allowed by the application. Our approximation algorithm should allow for us to use hyperparameters to increase or decrease the output accuracy, while increasing or decreasing power savings respectively. The question of how much depends on the application.

These hyperparameters are known as performance tuning knobs. There can be multiple knobs which adjusts the output accuracy-power tradeoff.
Hoffmann et al. [5] gives a detailed control flow view of how the knobs are adaptively adjusted.

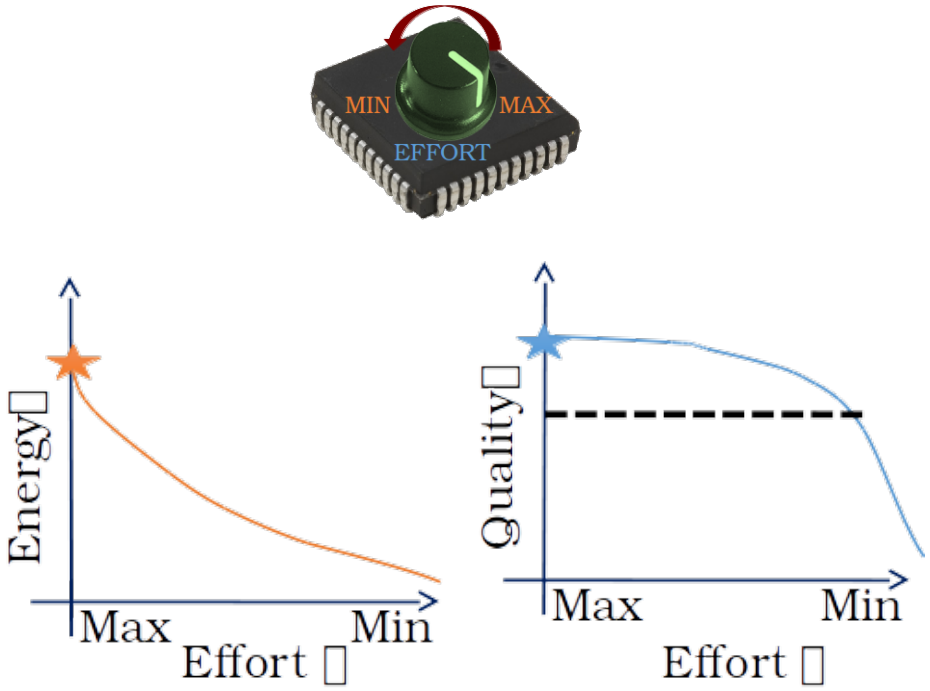

Fig. 1.1 Quality control knob

1.2 Motivation for Neuromorphic Computing

Over the past few years, deep neural networks have witnessed a huge success in the field of artificial intelligence and deep learning. These systems are deployed in a number of real world applications, lie the Google Assistant, Apple's Siri, FaceLock, etc. However, the deep networks are very compute intensive. According to Venkataramani et al,[2] SuperVision [2], a DLN which recently won the Imagenet visual recognition challenge, contains 650,000 neurons and 60 million connections demands compute performance in the order of 2-4 GOPS per classification.

Energy efficiency is also a factor to consider in the case of mobile devices, as mobile devices with limited battery could die out because of the high power consumption.

This raises a demand for a more energy optimized architectures for use in real world devices. The current standard is to run a computation optimized GPU (which are costly) accessed from a cloud server. All the computation occurs remotely and gets interfaced to the terminals.

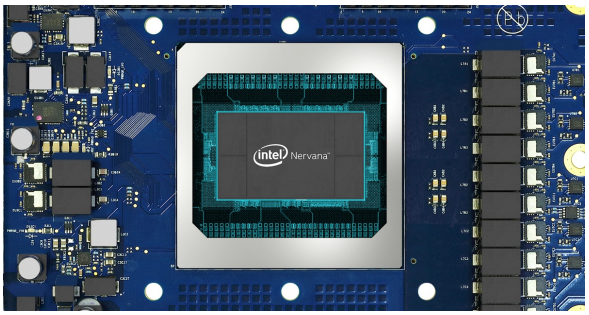

Fig. 1.2 Intel Nervana chip

Neuromorphic Computing is increasingly gaining traction to address this issue. In the October of 2017, Intel introduce it's first Neuromorphic chip, Intel's "Nervana" specifically designed and optimized for use of AI and deep neural networks. the optimizations enabled it to perform faster than the NVIDIA 1080Ti over standard benchmarks.

1.3 Software and Hardware based approaches to approximate computing

Computation resiliencies can be found at all levels. Approximation at the software levels can be classified to be in the domain of systems computer engineering, while hardware approximations come under VLSI and ASIC development.

1.3.1 Software based approximations

These techniques are either algorithmic or operate at the OS level. Full compiler toolchains have been created as a way to approximate error resilient parts of algorithms. Sidiroglou-Douskos et al. [7]

1.3.2 Hardware based approximations

: These approximations deal with hardware architectural modifications which produce inaccurate outputs. For example, approximate Adders: As demonstrated in Gupta et al. [4], The significant bits are evaluated precisely while the lower significant bits (LSBs) are given a more inaccurate but efficient adder design.

All hardware ASICs are generally targeted at one specific component of the hardware accelerator. However, these different systems can be used in conjunction with each other and

full top-down systems such as the one demonstrated by Raha and Raghunathan [6] can also be implemented.

1.4 Objectives of this dissertation

The approximations shown here can be classified as clever ways of computation skipping applicable in neural networks. They are inspired by the works of the AxNN from Venkataramani et al. [2014] and the ApproxNN from Zhang et al. [2015] designs. The AxNN identifies the the error resilient neurons individually and then apply precision scaling on them. It then retrains and precision scales further. ApproxNN on the other hand uses an improved metric to grade neurons for error resilience, and introduces hardware approximations over them. A direct way of avoiding computations in neural networks is to identify the "extraneous" neurons present in the network beforehand, then skip over computing their outputs.

Chapter 2

Approximations

2.1 Need for approximations

Most algorithms dealing with the analytics of big data have the property of being error resilient. Omitting a few operations or replacing them with inaccurate computations would only change the output by a small amount. For practical applications, this could be advantageous.

2.2 Standard Strategies

The approximations that we employ must be such that it is able to fit in with the framework of hardware driver elegantly. Approximations can be done at multiple levels of the chain, either at the software level or at the hardware level.
Several works have developed novel techniques for approximations in the software level. However, in the hardware domain, there are mainly 3 techniques where energy efficiency can be called in for:

- **Supply voltage scaling**: The DRAM gets a rated supply voltage for powering it. The power consumed by the DRAM $\propto SupplyVoltage^2$. A reduction in the supply voltage will decrease the certainity of the stored value being retained.
  The data resilient to errors can be routed to the portion of the DRAM receiving the lesser supply potential, as demonstrated in detail by Chippa et al. [1].

- **Precision scaling**: The DRAM cell expends many data cycles while storing it's data used for processing. If the application of an algorithm shows resiliency to a particular value stored in a DRAM cell, the number of bits used to store data for a less critical value uses up extraneous cycles. this is one area where some corners could be cut.

- **Look-up tables**: The multiplier in the CPU takes up a lot of energy for computation of products. If the products are done over integers and are only few in number, the numbers can be rounded off and the multiplication can be performed by looking at the value stored in the multiplication look-up table.

2.3 Relevant Work

: Here we present some comments on some of the previous works and developments in the area in the context of neural networks leading to more clever approaches to increase efficiency while still maintaining the quality of output.
The approximations shown here can be classified as clever ways of computation skipping applicable in neural networks.

- Du et al. [2] shows how the application of the above techniques over neural networks. Reducing bitwidth over all the storage units alone can give great amount of savings. However, rather than only pruning the components which can be
- The AxNN from Venkataramani et al. [2014] from identifies the the error resilient neurons individually and then apply precision scaling on them. It then retrains and precision scales further.
- ApproxNN from Zhang et al. [2015] on the other hand uses an improved metric to grade neurons for error resilience, and introduces hardware approximations over them.
- Some novel approaches, like the one in Ujiie et al. [8] utilizes the fact that the maxpool layer wastes all but one vector computations as all others are set to 0. It tries to predict which of the inputs is maximum beforehand and saves on the extra left out computations.

In our study, we present a more lightweight approach to the problem. It is largely inspired by the works of the AxNN from Venkataramani et al. [2014] and the ApproxNN from Zhang et al. [2015] designs. In these approaches, the omission of computation is based on the introduction of a new metric called the criticality of the neurons in the neural network.

2.4 Approximations Introduced

Here, we introduce two lightweight novel approaches to pruning away extraneous computations in neural networks as a form of approximation.

2.4.1 Criticality-based approximation

Each layer of the neural network has a number of neurons. These neurons are ranked in the order of their importance in influencing the output. This quantity is termed as the criticality of the neuron. The criticality of neurons in a layer is a scalar number defining how much jitter a neuron's output causes to the overall output, on average. The less critical neurons can be "turned off", or rather be replaced by a bias of a value equal to their mean output over the validation data. This ensures that the output of the neuron is nominal and error to the output is limited to low enough variance to not change the output by much. Leaving out computations for few of the neurons reduces the number of computations significantly as it throws away requirement for the weighted sum computation, especially at populated layers. This is done by thresholding the less critical neurons. This threshold limit gives a fine granular knob to control the degree of approximation. Thresholding on multiple layers gives multiple fine granular knobs.

Criticality

The criticality metric introduced previously is shown below. This is just saying the amount of impact a neuron has on the output. Notation : $z(i,k)$ is the output of the i^{th} neuron in the k^{th} layer.

$$Criticality(Cr_{i,k}) = \delta z_{i,k} \frac{\partial Loss}{\partial z_{i,k}}$$

Computation of the metric The sample variance computation is straightforward: compute sum of squares and sum of outputs of each of the neurons and use them to find variance. The derivative, however requires some effort.

$$Loss(L) = \frac{1}{2}||y - y_{golden}||^2$$

$$\frac{\partial L}{\partial z_{i,k}} = \sum_t (y_t - y_{gold,t}) * \frac{\partial y_t}{\partial z_{i,k}}$$

The derivative term can be found out by back-propagation. Using the loss derivatives of one layer, loss derivative of the neuron outputs of the previous layers can be found.

$$y_t = f(x_{t,K})$$

$$\frac{\partial y_t}{\partial z_{i,k}} = f'(x_{t,K}) * \sum_j w_{K-1,j} * \frac{\partial z_{j,K-1}}{\partial z_{i,k}}$$

For large networks with different layers, the following can be observed:

1) Notice that every pair of connection i, k between two different layers occurs in this computation as in the forward propagation case. Hence the same looping structure can be used to calculate the derivative by backpropagation. It doesn't matter what type of layer it is. The derivatives are then averaged over.
2) for ReLU layers:

$$\frac{\partial y_t}{\partial z_{i,k}} = sgn(x_{t,K}) * \sum_j w_{K-1,j} * \frac{\partial z_{j,K-1}}{\partial z_{i,k}}$$

where sgn is the 0-1 sign function.
3)For maxpool layers, if y is max of the layers:

$$\frac{\partial y_t}{\partial z_{i,k}} = \frac{\partial z_{j,K-1}}{\partial z_{i,k}}$$

otherwise

$$\frac{\partial y_t}{\partial z_{i,k}} = 0$$

4) For LRN layers: Since the alpha is very low, the derivative is taken to be equal to the derivative.(Approximately true).

2.4.2 Input Similarity based approximation

Input similarity filter: In the case of image data input for a network, an image is first windowed individually by the convolutional layers (or fully connected layer in the case of an MLP). In some of the windows, especially in the case of recognition-images, a lot of windows have a constant (less varying) pixel intensities. The convolutional layer filtering over it does a lot of extra multiplications and additions to produce the neuron's output of a constant . If it is known beforehand that a given window contains pixels of less varying intensities, the approximate convolution can be produced in a single multiplication operation. The sum of filter weights over each window have to be precomputed. The variance of the pixel intensities is taken as a parameter that can be thresholded, giving rise to a quality configurable knob. This is a coarse granular knob. Note: The AxNNs proposed have their approximate output

deriving from a smaller bit-width based computation. Here, the approximate output will be a bias unit.

Chapter 3

Experiments and Results

3.1 Experimental Setup

The approximation was tested on four standard Neural networks as benchmarks shown in the table below. The program was set to run on a 3.4GHz Intel Xeon processor. The experiment can be done only on sequential platforms and not on parallel platforms, as we would want speedup to be proportional to the number of computations avoided.

On a software platform, the power consumed it taken to be proportional to the time taken by these algorithms during runtime. This is true when running sequential code on a CPU. The number of computations performed is proportional to the time taken by the algorithm, and hence the power consumed.

The most time dominant operation in the whole of the algorithm are the number of floating point multiplications between every pair of numbers going on.

As we can see, with virtually no loss in quality, selective neuron skipping yields significant savings and speedup.
For the Lenet-5 and CIFARQuick benchmarks, speedup of close to 2X savings can be seen. It is also the case that the bigger networks offer more significant savings, due to the extraneous neurons being more prevalent than on small, under-performing networks.

Table 3.1 Benchmarks considered for testting

Benchmark	Dataset	Peak Accuracy	Application
CIFAR-Quick CNN	CIFAR-10	80.6	Object recognition
LENET-5	MNIST	97.54	Handwriting Recognition
Multi-Layer Perceptron	MNIST	94.53	Handwriting Recognition
CNN over SVHN	SVHN	88.00	House Number Reading
AlexNet	ImageNet-2013	84.1	Image scene Classification

Table 3.2 Benchmarks size

Benchmark	Layers	Number of neurons	Input features
CIFAR-Quick CNN	5	66058	3072
LENET-5	6	8034	1024
Multi-Layer Perceptron	4	1022	784
CNN over SVHN	5	28810	3072
AlexNet	9	1050000	154587

The algorithms picks and identifies the neuron which fall below the threshold of a set criticality metric. These are then replaced simply by a bias unit, while having to avoid the extra computation effort of costly computing matrix-vector multiplication. Hence the more dense layers tend to be the bottleneck and target for the criticality filtering. The neurons omitted are weeded out only from two of the bottleneck layer in the above cases. Each layer is chosen because the thresholds would be different in each layer, leading to multiple quality controlling knobs (one for each layer).

3.2 Quality-Energy graphs obtained

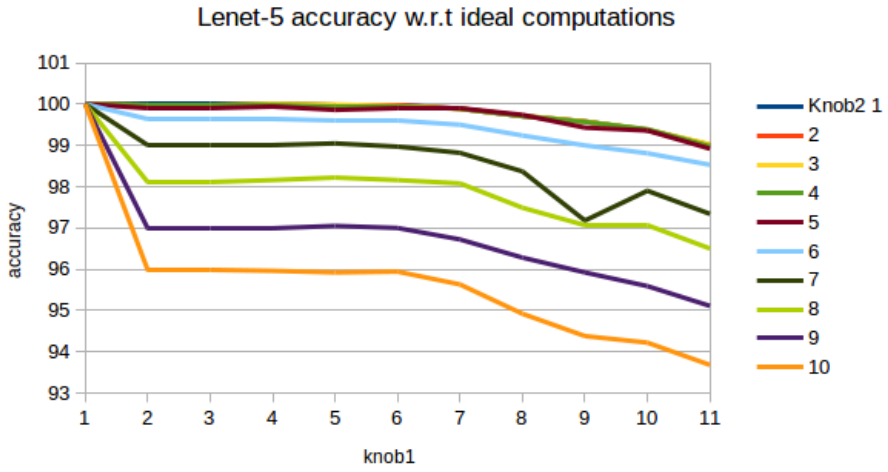

Fig. 3.1 Lenet accuracy compared to ideal

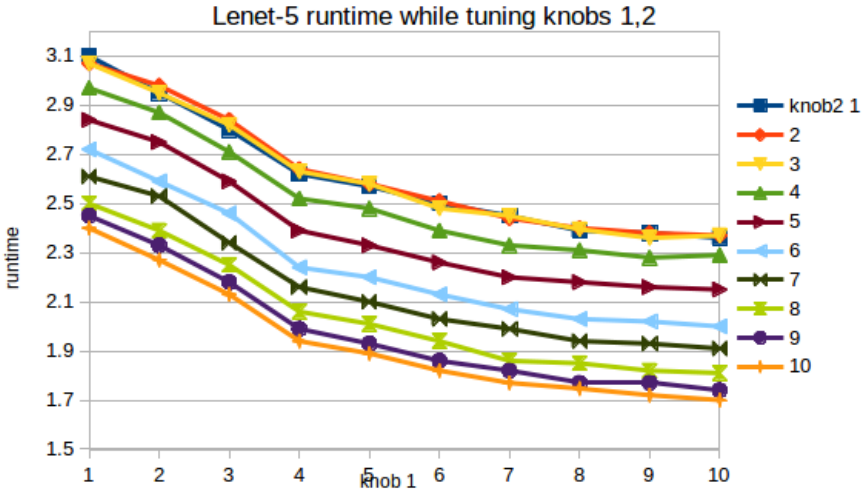

Fig. 3.2 Lenet runtime

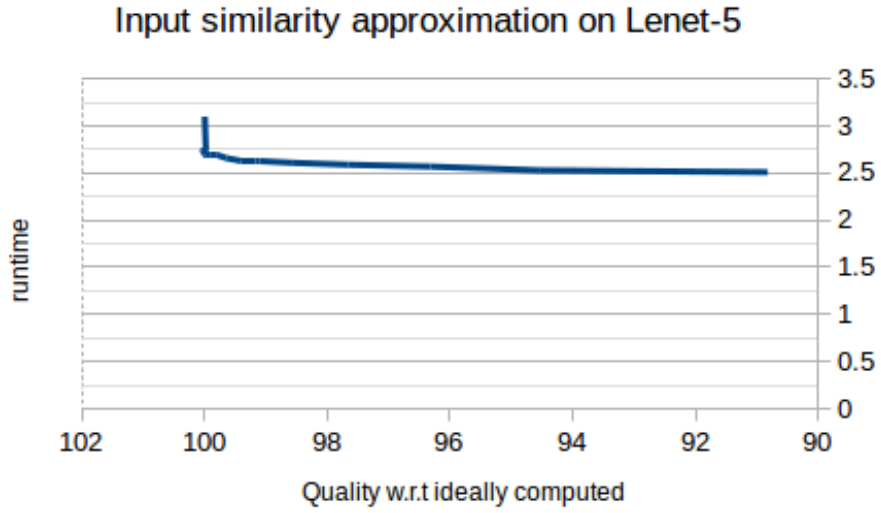

Fig. 3.3 Lenet input similarity

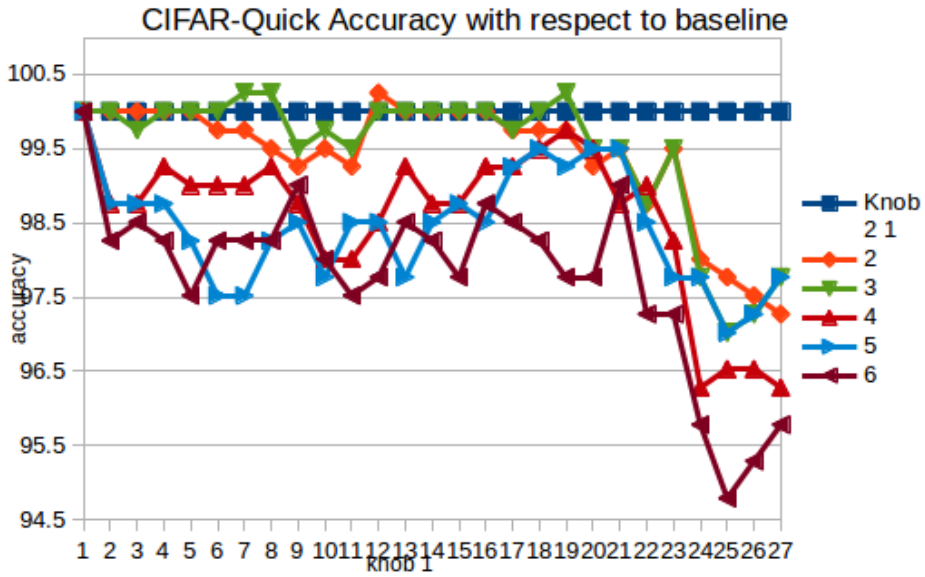

Fig. 3.4 CIFAR10 accuracy compared to ideal

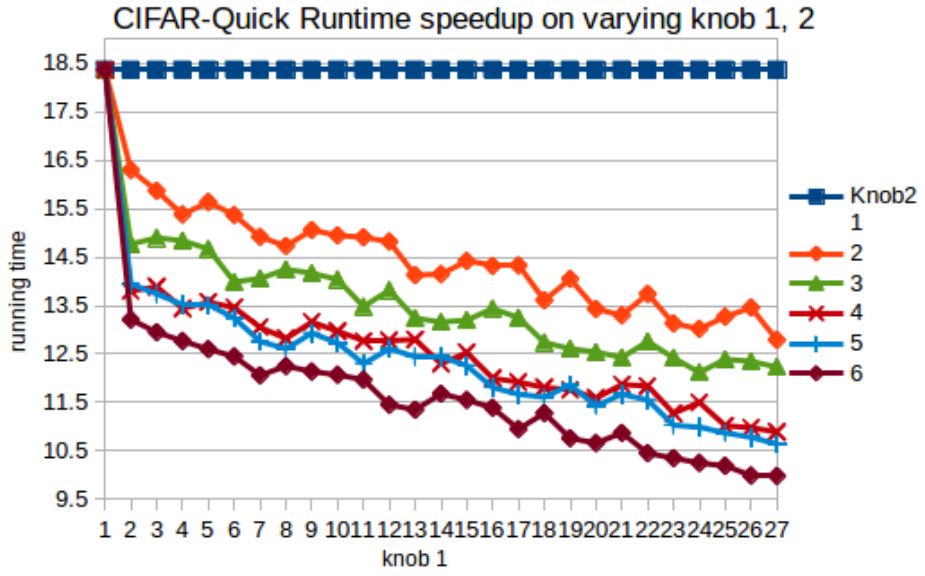

Fig. 3.5 cifar10 runtime

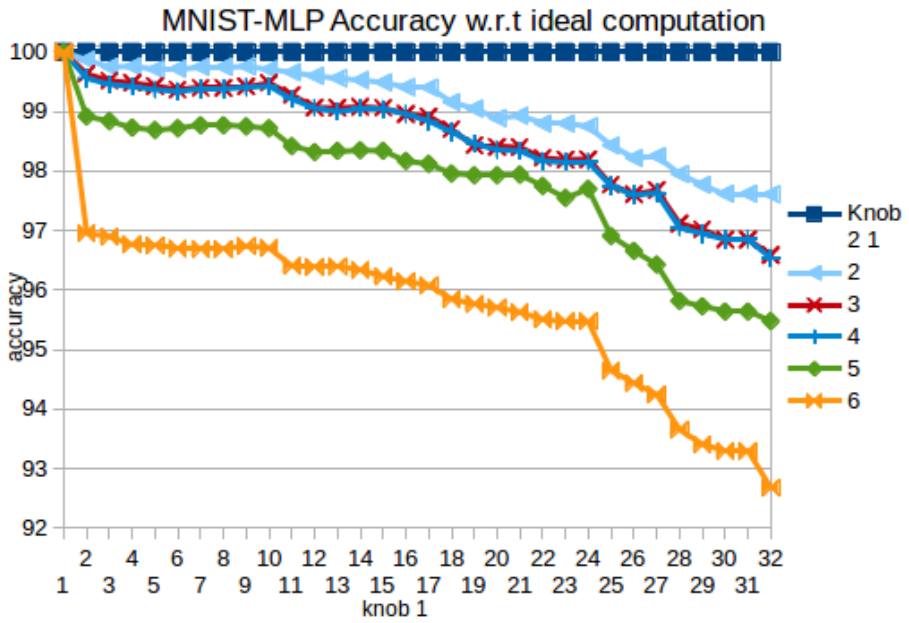

Fig. 3.6 mlp accuracy compared to ideal

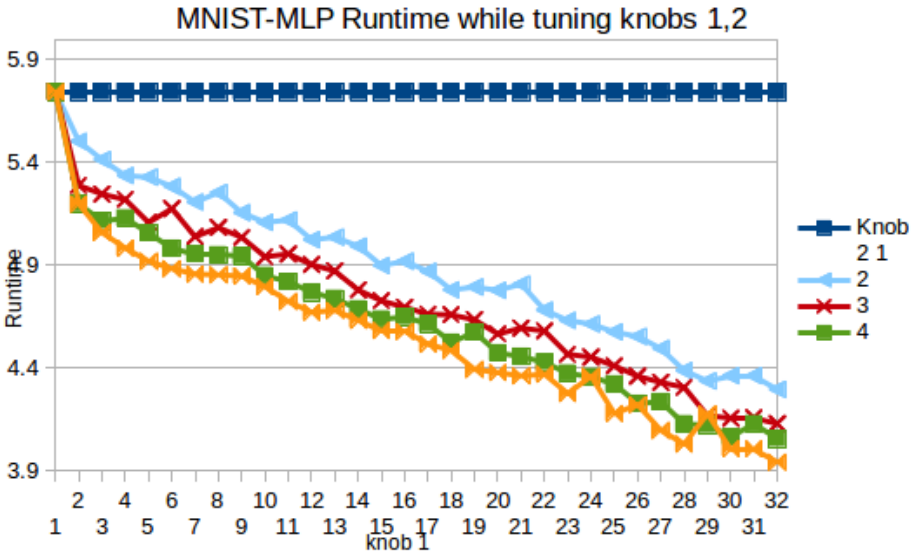

Fig. 3.7 mlp runtime

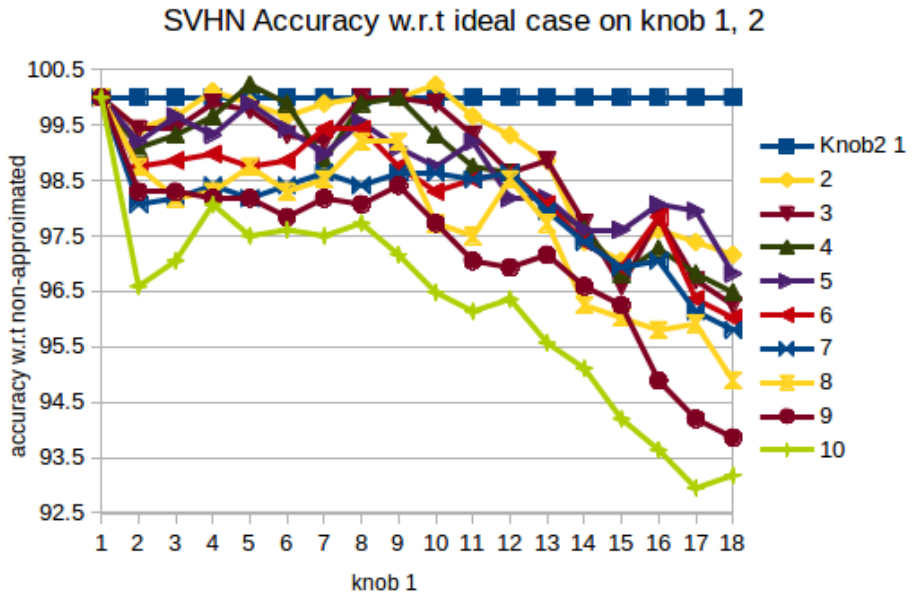

Fig. 3.8 svhn accuracy compared to ideal

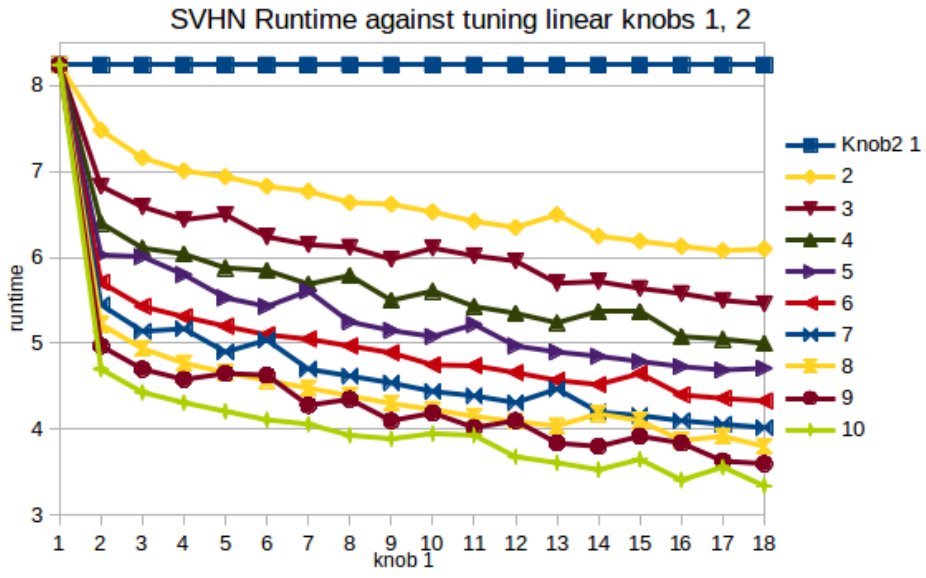

Fig. 3.9 svhn runtime

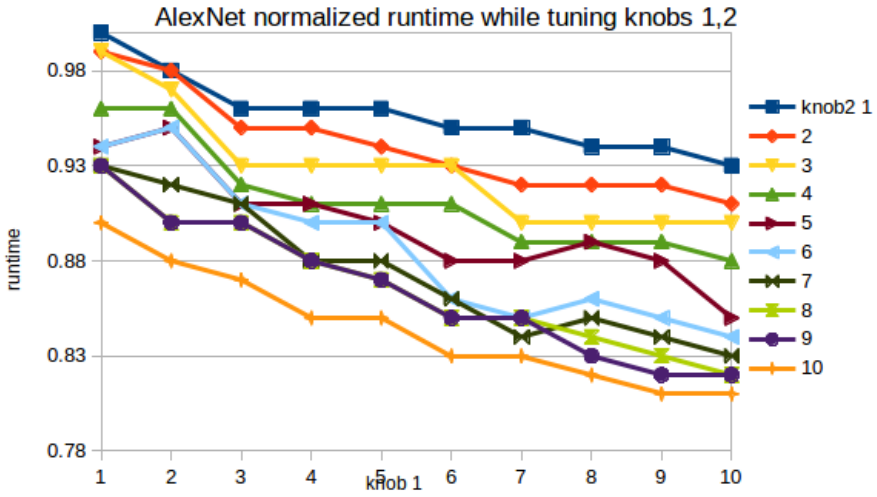

Fig. 3.10 alexNet runtime

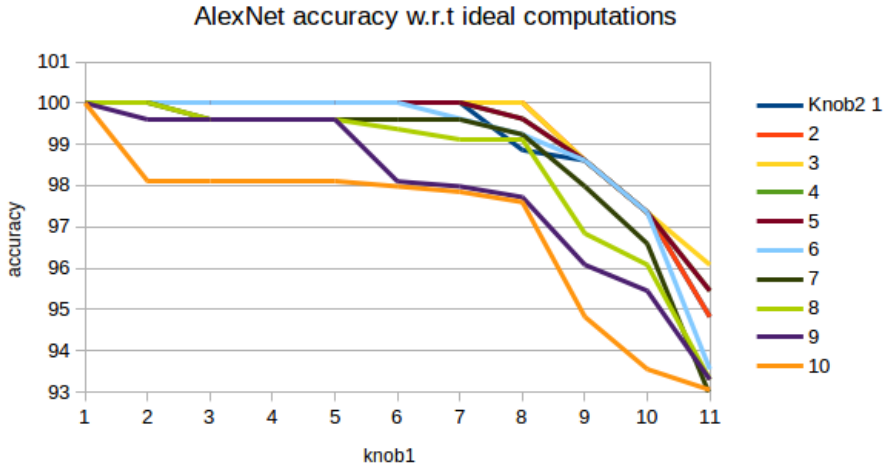

Fig. 3.11 alenNet accuracy compared to ideal

For the case of AlexNet, the accuracy was measured using a GPU with CUDA. However to measure the number of computations skipped, a smaller subset of the test set was considered

and the time was measured and extrapolated. Since the number of computations even after thresholding remains exactly the same for each image, the runtime can be safely assumed to be proportional to the number of computations, hence proportional to the power consumed. For a 2 percent quality degradation when compared to ideally computed case, the best accuracy obtained is as follows:

Table 3.3 Degraded quality to skipped computations

Benchmark	Base runtime	Enhanced runtime	Dataset-size	Approximations
AlexNet	1.0(normalized)	0.84	5000	Criticality

3.3 Degradation vs Speedup

For a 2 percent quality degradation when compared to ideally computed case, the best accuracy obtained shown below. The only overhead involved is to check whether each neuron falls below the threshold or not. Another challenge involved to do this is to identify the ballpark of the variance threshold.

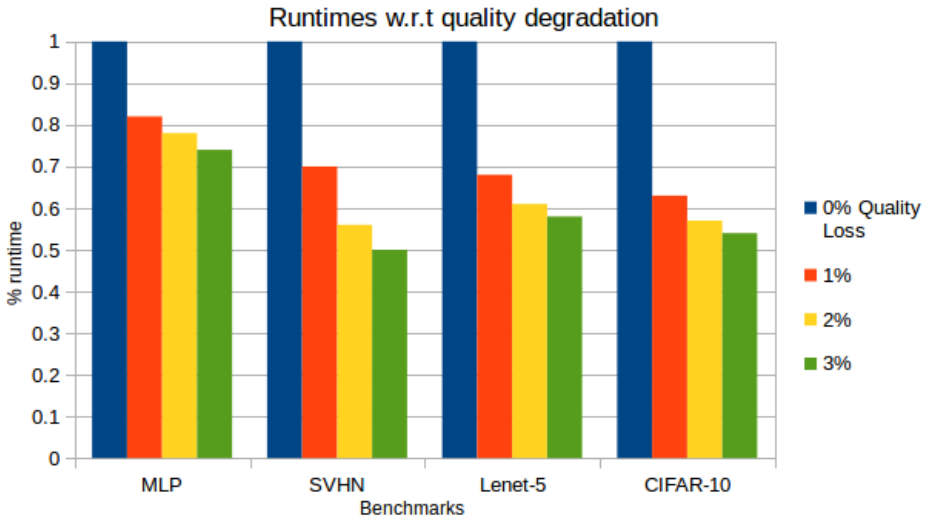

Fig. 3.12 accuracy vs quality loss

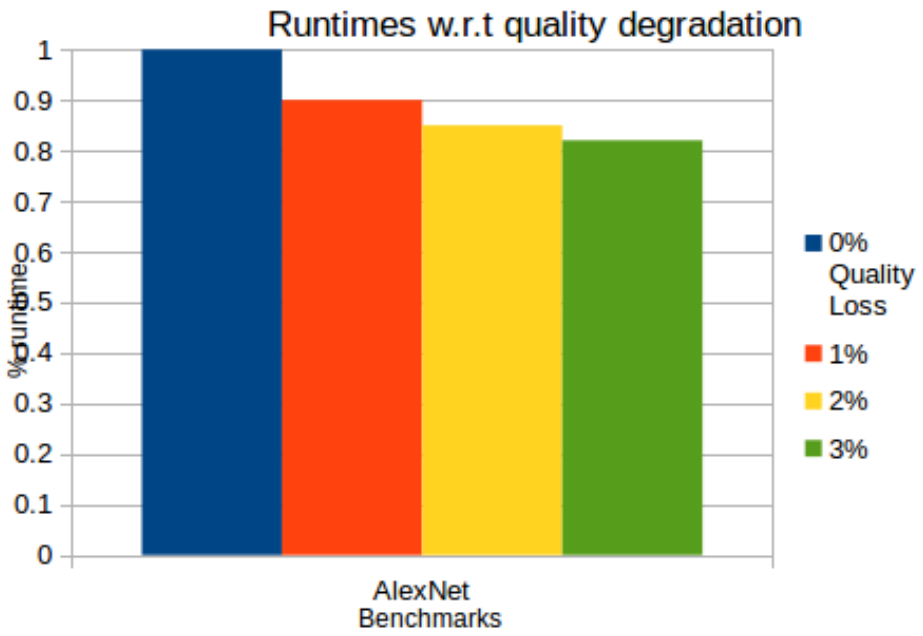

Fig. 3.13 alexnet accuracy vs quality loss

Chapter 4

Implementation on Hardware: FPGA cell

4.1 Deployment in real systems

Many real world chips that utilize neuromorphic computing have a separate hardware driver for accelerating runtime performance. A detailed survey on implementation and deployment of Neural networks has been shown in Implementing this forward pass has been demonstrated in Guo et al. [3].
We adopt the "Quality configurable Neural processing environment" cell model introduced in Venkataramani et al. [9].

4.2 QcNPE cell

Introduced in Venkataramani et al. [9] and Zhang et al. [10], this architecture is implementable on an FPGA. Each NCU evaluator cell performs a dot product of the weight and the input to the neuron and adds them up.
In every step, it produces simultaneously all the outputs of the neurons in a given layer, and stores these outputs in the DRAM.

4.2.1 Compliance to approximation scheme

Each NCU(Neural computing unit) performs one multiplication of weight and input parameter. It takes in control sequences as well. The threshold to omit or perform the multiplications in each of the NCU are precomputed according to our approximation scheme and stored in the

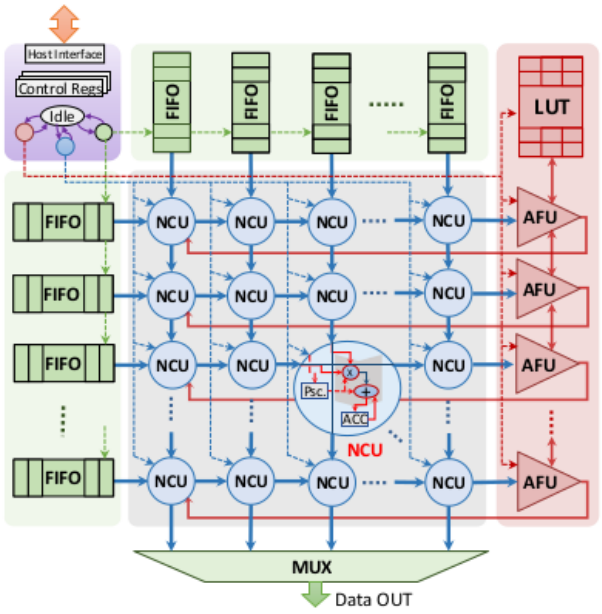

Fig. 4.1 QcNPE cell

FIFO units. These can determine if the NCU has to perform or skip the multiplication. The more multiplications we skip, the more power savings we can get.

[h]

For larger networks like VGG net, the layer with very large number of neurons (> 10000) are sent and computed batchwise in multiple steps.

4.3 Results

Running this model on s standard Xilinx FPGA running on Amazon AFI FPGA instance. The number of multiplications that can be saved here have been noted.

Table 4.1 Multiplications performed in the dominant step

Benchmark	Number of multiplications
CIFAR-Quick CNN	2228224000
LENET-5	2256000
Multi-Layer Perceptron	2662400000
CNN over SVHN	6553600000
AlexNet	720000000

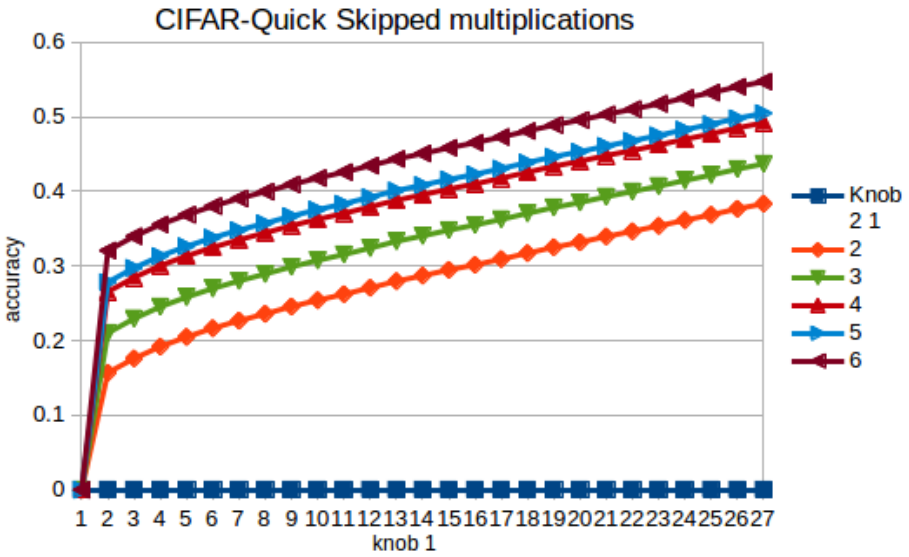

Fig. 4.2 cifar: multiplication skipped

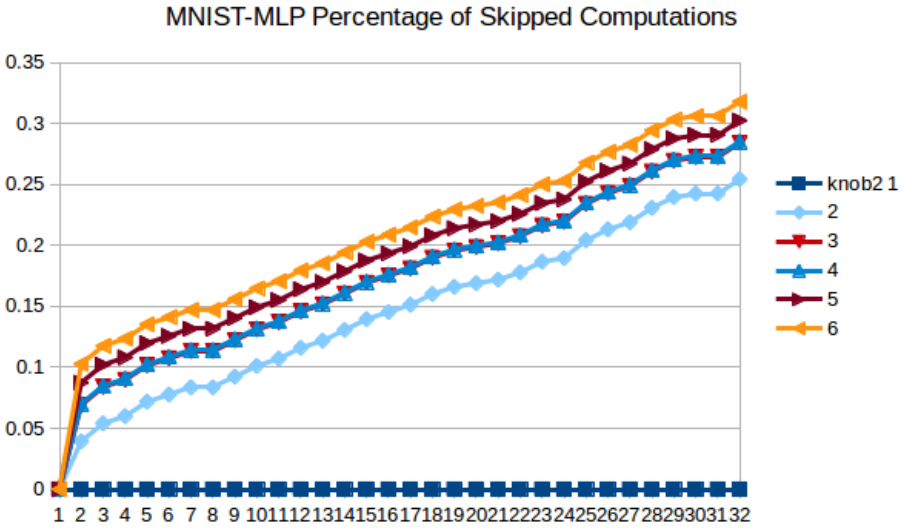

Fig. 4.3 mlp: multiplications skipped

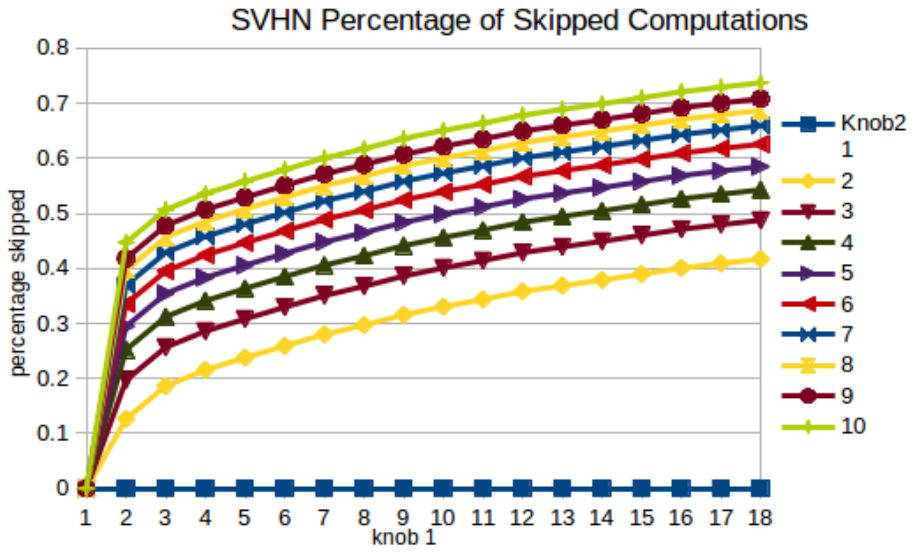

Fig. 4.4 svhn: multiplications skipped

Chapter 5

Conclusion

Significant savings for each of the cases can be observed as above. It is expected that this technique must work on general neural networks as well. It particularly works well over structured data rather than unstructured data because some features learned by neurons on structured data (by CNNs) are local to a region of the image, and it may be potentially so that some are less important areas in the structured data. Base hardware approximations on neural networks are described in Gysel et al. [2016]. Further, simple precision scaling and conventional hardware approximations are presented in Moons et al. [2016]. Other than this, Ujiie et al. [2016] introduces specific approximations on layers of CNNs, such as Lazy-convolution layer, where convolution computations before a maxpooling layer can be avoided.

References

[1] Chippa, V. K., Mohapatra, D., Raghunathan, A., Roy, K., and Chakradhar, S. T. (2010). Scalable effort hardware design: Exploiting algorithmic resilience for energy efficiency. In *Design Automation Conference*, pages 555–560.

[2] Du, Z., Palem, K., Lingamneni, A., Temam, O., Chen, Y., and Wu, C. (2014). Leveraging the error resilience of machine-learning applications for designing highly energy efficient accelerators. In *2014 19th Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 201–206.

[3] Guo, K., Zeng, S., Yu, J., Wang, Y., and Yang, H. (2017). A survey of fpga based neural network accelerator. *CoRR*, abs/1712.08934.

[4] Gupta, V., Mohapatra, D., Park, S. P., Raghunathan, A., and Roy, K. (2011). Impact: Imprecise adders for low-power approximate computing. In *IEEE/ACM International Symposium on Low Power Electronics and Design*, pages 409–414.

[5] Hoffmann, H., Sidiroglou, S., Carbin, M., Misailovic, S., Agarwal, A., and Rinard, M. (2011). Dynamic knobs for responsive power-aware computing. *SIGPLAN Not.*, 46(3):199–212.

[6] Raha, A. and Raghunathan, V. (2017). Synergistic approximation of computation and memory subsystems for error-resilient applications. *IEEE Embedded Systems Letters*, 9(1):21–24.

[7] Sidiroglou-Douskos, S., Misailovic, S., Hoffmann, H., and Rinard, M. (2011). Managing performance vs. accuracy trade-offs with loop perforation. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering*, ESEC/FSE '11, pages 124–134, New York, NY, USA. ACM.

[8] Ujiie, T., Hiromoto, M., and Sato, T. (2016). Approximated prediction strategy for reducing power consumption of convolutional neural network processor. In *2016 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW)*, pages 870–876.

[9] Venkataramani, S., Ranjan, A., Roy, K., and Raghunathan, A. (2014). Axnn: Energy-efficient neuromorphic systems using approximate computing. In *2014 IEEE/ACM International Symposium on Low Power Electronics and Design (ISLPED)*, pages 27–32.

[10] Zhang, Q., Wang, T., Tian, Y., Yuan, F., and Xu, Q. (2015). Approxann: An approximate computing framework for artificial neural network. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 701–706.