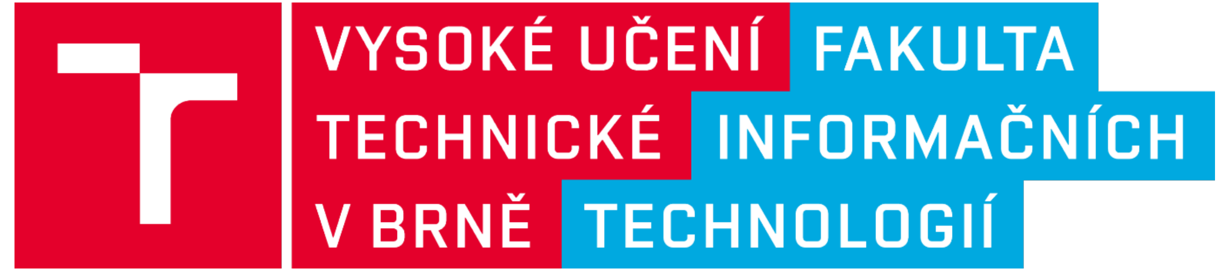

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

EXPLOITING APPROXIMATE ARITHMETIC CIRCUITS IN NEURAL NETWORKS INFERENCE

VYUŽITÍ APROXIMOVANÝCH ARITMETICKÝCH OBVODŮ V NEURONOVÝCH SÍTÍ

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. Tomáš Matula

VEDOUCÍ PRÁCE
SUPERVISOR

RNDr. Milan Češka Ph.D.

BRNO 2019

Brno University of Technology
Faculty of Information Technology

Department of Intelligent Systems (DITS) Academic year 2018/2019

Master's Thesis Specification

Student: **Matula Tomáš, Bc.**
Programme: Information Technology Field of study: Intelligent Systems
Title: **Exploiting Approximate Arithmetic Circuits in Neural Networks Inference**
Category: Artificial Intelligence
Assignment:

1. Study the current methods of approximate circuit design and error metrics allowing to describe the precision of these circuits.
2. Study the modern frameworks for implementation of neural networks (TensorFlow, Theano, Keras, etc.) and the application of approximate circuits in neural networks that aims at reduction of energy consumption and inference acceleration.
3. Extend the selected framework to simulate the integration of approximate circuits into neural networks.
4. Evaluate the effect of approximate circuits on the precision of currently most popular neural networks (e.g. Resnet-50, AlexNet or Inception). Focus on the impact of various error metrics (worst-case absolute error, worst-case relative error, mean absolute error or error rate).
5. Design a metric to compare the performance and energy efficiency of neural networks with both precise and approximate computations. Compare the benefits of approximate circuits with other approaches for neural network optimization (Mobilenet network, networks approximated with the Ristretto framework, etc.)

Recommended literature:

- V. Mrazek, S. S. Sarvar, et al. Design of power-efficient approximate multipliers for approximate artificial neural networks. In Proc. of ICCAD'16. ACM, 2016.
- M. Ceska, J. Matyas, et al. Approximating complex arithmetic circuits with format error guarantees: 32-bit multipliers accomplished. In Proc. of ICCAD'17. IEEE, 2017.
- O. Keszocze, M. Soeken, et al. The complexity of error metrics. In Information Processing Letters, vol. 139. Elsevier, 2018.

Requirements for the semestral defence:

- Items 1 and 2, and parcially item 3.

Detailed formal requirements can be found at http://www.fit.vutbr.cz/info/szz/

Supervisor: **Češka Milan, RNDr., Ph.D.**
Consultant: Matyáš Jiří, Ing., UITS FIT VUT
Head of Department: Hanáček Petr, doc. Dr. Ing.
Beginning of work: November 1, 2018
Submission deadline: May 22, 2019
Approval date: November 24, 2018

Abstrakt

Táto práca sa zaoberá využitím aproximovaných obvodov v neurónových sieťach so zámerom prínosu energetických úspor. K tejto téme už existujú štúdie, avšak väčšina z nich bola príliš špecifická k aplikácii alebo bola demonštrovaná v malom rozsahu. Pre dodatočné preskúmanie možností sme preto skrz netriviálne modifikácie open-source frameworku TensorFlow vytvorili platformu umožňujúcu simulovať používanie approximovaných obvodov na populárnych a robustných neurónových sieťach ako Inception alebo MobileNet. Bodom záujmu bolo nahradenie väčšiny výpočtovo náročných častí konvolučných neurónových sietí, ktorými sú konkrétne operácie násobenia v konvolučnách vrstvách. Experimentálne sme ukázali a porovnávali rozličné varianty a aj napriek tomu, že sme postupovali bez preučenia siete sa nám podarilo získať zaujímavé výsledky. Napríklad pri architektúre Inception v4 sme získali takmer 8% úspor, pričom nedošlo k žiadnemu poklesu presnosti. Táto úspora vie rozhodne nájsť uplatnenie v mobilných zariadeniach alebo pri veľkých neurónových sieťach s enormnými výpočtovými nárokmi.

Abstract

This thesis is concerned with the utilization of approximate circuits in neural networks to provide energy savings. Various studies showing interesting results already exist, but most of them were very application specific or demonstrated on a small scale. To take this further, we created a platform by nontrivial modifications of robust open-source framework Tensorflow allowing us to simulate approximate computing on known state-of-the-art neural networks e.g. Inception or MobileNet. We focused only on replacement of most computationally expensive parts of convolutional neural networks, which are multiplication operations in convolution layers. We experimentally demonstrated and compared various setups and even that we proceeded without relearning, we were able to obtain promising results. For example, with zero accuracy loss on Inception v4 architecture, we gained almost 8% energy savings which could be valuable, especially in low-power devices or in large neural networks with enormous computational demands.

Kľúčové slová

umelá inteligencia, neurónové siete, aproximované obvody, qvantizácia, energetická úspora, TensorFlow, Inception, MobileNet

Keywords

artificial intelligence, neural networks, approximate circuits, quantization, energy savings, TensorFlow, Inception, MobileNet

Exploiting Approximate Arithmetic Circuits in Neural Networks Inference

Declaration

Hereby I declare that this master's thesis was prepared as an original author's work under the supervision of Dr. Milan Češka. The crucial information about approximate circuits and their simulations as well as prepared lookup tables were provided by Ing. Jiří Matyáš. All the relevant information sources, which were used during preparation of this thesis, are properly cited and included in the list of references.

....
Tomáš Matula
May 22, 2019

Acknowledgements

I would like to thank to my supervisor Dr. Milan Češka for his lead and suggestions, Ing. Jiří Matyáš for providing approximated circuits and information about them.

© Tomáš Matula, 2019

Contents

Contents 1
1 Introduction 2
2 Artificial Neural Networks 4
2.1 Neuron 4
2.2 Artificial neural network 7
2.3 Convolutional Neural Networks 9
2.4 Frameworks for neural networks 10
2.5 Architectures 12
2.5.1 ILSVRC 14
2.6 Energy efficiency 15
2.6.1 Quantization 15
3 Approximate computing 17
3.1 Approximate arithmetic circuits 17
3.2 Error metrics for arithmetic circuits 20
3.3 Approximation computing in Neural Networks 21
4 Realization 22
4.1 Selected framework 22
4.2 Approximation scope 23
4.3 Approximation circuits simulation 24
4.4 Framework modifications 25
4.5 Metric design 27
5 Experiments 30
5.1 Experimental setup 30
5.1.1 Approximated multipliers 30
5.1.2 Evaluation strategy 32
5.2 Approximation of all convolutional layers 33
5.3 Approximation of individual layers 38
5.4 Single image inference comparison 41
6 Conclusions and future directions 44
6.1 Comparing to other networks 44
6.2 Future directions 46
Bibliography 48

1 Introduction

Artificial Neural Network is computing system inspired by biological neural network, like human brain [1]. The output of this system is usually not programmed with exact specific rules, but it is rather “learned” from a big amount of example data. These days this field is rapidly gaining popularity, thanks to progress especially in convolutional neural networks. It is no wonder though, since they are able to provide far better results in areas like image or voice recognition, translation, finances or even vehicle control when compared to standard algorithms.

However, these advancements come with the cost of increased complexity and amount of processed data. On the other hand, while rapidly reaching end customers and spreading to mobile platforms, demand for performance optimisations in these computationally expensive solutions increase. Various approaches to deal with limited hardware exist, one of which is use of quantization. Quantization is optimization technique where numbers within processed architecture are mapped from large set of data to a smaller set. In terms of Neural networks, it means representing values as integers instead of floats. Since low level arithmetic integrated circuits working with integers are less computationally expensive, circuits could be smaller and more energy efficient [2].

Researches in field of approximate computing are opening a new horizon for further optimizations – instead of using precise computation we allow small amount of error in result, which in trade could provide additional significant energy and space saving. For example, it is possible to obtain 50% energy saving by introducing just 1% worst-case absolute error in result. Because of great error resiliency of neural networks, this combination of approaches could open new horizons. In fact, recent study [3] already shown outstanding results in small scale with energy savings of more than 90% with accuracy drop of around 3%. However, to our best knowledge all existing solutions were either very application specific, contained nonuniform structure (e.g. low-level modifications of neurons), or were presented on small frameworks (e.g. tiny-cnn) and with small datasets (e.g. MNIST or SVHN).

In this thesis we try to take this a step further by simulating use of approximate arithmetic circuits on well-known robust open source machine learning framework TensorFlow, developed by Google Brain team [4]. We focus on providing evidence of usability of this technique even in large applications. We will do evaluations on commonly available large neural networks as MobileNet or Inception, and carefully compare our results with their baselines.

Contribution

In this thesis, we explored landmark of possibilities for usage of approximated circuits in neural networks. Uniqueness of our approach comes from usage of popular framework TensorFlow and by experimenting with large neural network architectures. This was achieved by creation of platform allowing to simulate approximate computations. We had to deal with:

- Implementation of simulation and nontrivial framework modifications
- Design of metrics allowing to compare results
- High computational demands and duration of experiments
- Experimenting with various approximate circuits on Inception and MobileNet architectures

Our results indicate that this technique could surely find use – even without relearning, we were able to obtain savings of almost 8% with accuracy practically unchanged. This could be valuable, especially in portable or embedded devices as well as in large neural networks with enormous computational demands.

The work is structured as follows: first in chapter 2 we dive into current trend in artificial intelligence which are neural networks, how they emerged, how they work and what are the limits. We especially focus on convolutional neural networks and already show possible frameworks for implementing them. Then we elucidate structure of common architectures as Inception and MobileNet. After that we jump straight to energy efficiency with focus on quantization and transition to approximate computing in chapter 3, where we explain basic principles and methodologies as well as evaluation metrics. We present some existing solutions that combine approximation computing with neural networks and with respect to learned information we start to explore our realization. There, in chapter 4, we first analyse and select the most appropriate framework, define scope of our interests for simulation of approximated circuits and then explain not only how we implemented them but also struggles we have faced. We carefully design an evaluation metric and point to experiments in 5, where we gradually present our results as well as their limitations and possible improvements.

2 Artificial Neural Networks

Artificial neural network is a system greatly inspired by biological neural network found in brain of animals or humans [1]. The main power of this approach is that computer can derive the solution by observing and learning from data opposed to conventional approach where computer is told exactly what and how should be done, trough pre-given algorithms [5].

2.1 Neuron

To understand neural networks and explore how to find use of approximation computing there, we first need to understand the main basic component – a neuron. The biological neuron is a cell that receive, process and transmit information through electrical or chemical signals. Signals are received trough *dendrites*, which are connected to large number of neighbouring neurons. These signals are processed as a summation of weights of each in cell called *soma*. Each time the potential reaches certain threshold there, the pulse is transmitted to *axon*. At its end, axon contains multiple *terminals*, which are connected to dendrites of other neurons. These connections are called *synapses.* See Figure 2.1.

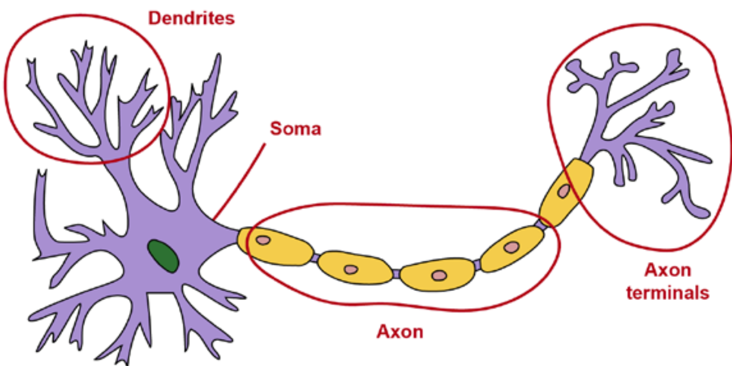

Figure 2.1 Biological neuron – inspired by [6]

In Similarity to biological neuron, an artificial neuron is mathematical function that receives one or more weighted inputs and provide output as result of this function. The basic model developed in 1950s is called perceptron.

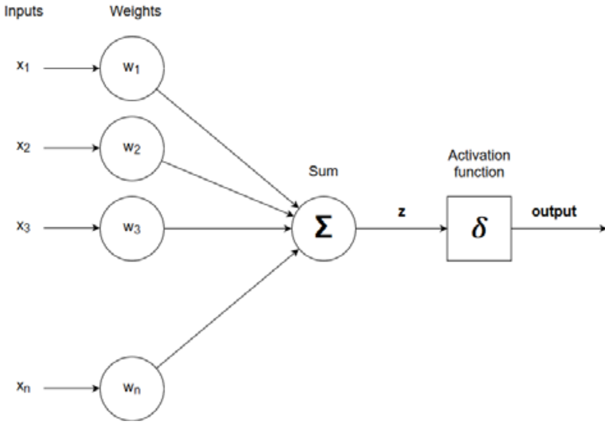

Figure 2.2 Perceptron

As shown in Figure 2.2, perceptron has multiple inputs and each input have corresponding weight. The output of neuron is determined as application of activation function to weighted summation of inputs. In basic perceptron, the activation function is just basic binary step function. If certain threshold or bias (an inverse value of threshold) is reached, output will be activated. This could be represented by Equation 2.1 and Equation 2.2.

$$z = \sum_j w_j x_j$$

Equation 2.1

$$output = \begin{cases} 0 & if\ z \leq treshold \\ 1 & if\ z > treshold \end{cases}$$

Equation 2.2

One can imagine, this could be easily implemented in software or in hardware and yet, as will be explained further, all the power lies in neuron.

Activation functions

The problem with perceptron activation function is, that small change in input could lead to sudden "flip" of output. To make neuron less sensitive to input, in modern neural networks, various activation functions are used, see Table 2.1. Term *bias* in those functions usually represents if the activation curve is shifted higher or lower. Activation function is sometimes also referred to as transfer function.

Name	Equation	Range	Plot
Binary step	$f(x) = \begin{cases} 0 & if\ x \leq 0 \\ 1 & if\ x > 0 \end{cases}$	{0, 1}	
Sigmoid	$f(x) = \frac{1}{1 + e^{-x}}$	(0, 1)	
TanH (Hyperbolic tangent)	$f(x) = tanh(x)$	(-1, 1)	
ReLU (Rectified linear unit)	$f(x) = \begin{cases} 0 & if\ x \leq 0 \\ x & if\ x > 0 \end{cases}$	<0, ∞)	
Leaky ReLU	$f(x) = \begin{cases} 0.01x & if\ x \leq 0 \\ x & if\ x > 0 \end{cases}$	(-∞, ∞)	

Table 2.1 comparison of most common activation functions

2.2 Artificial neural network

One perceptron can classify data into two parts. When we use proper activation function, we can interpret output as probability or as value within certain interval. However, to be able to classify more complex problems, multiple neurons need to be used. We can merge these neurons into multiple layers – some of which are input and output layers, others could be just hidden – see Figure 2.3. The input layer starts with very basic classification and pass output to next layers where more abstract decisions can be made.

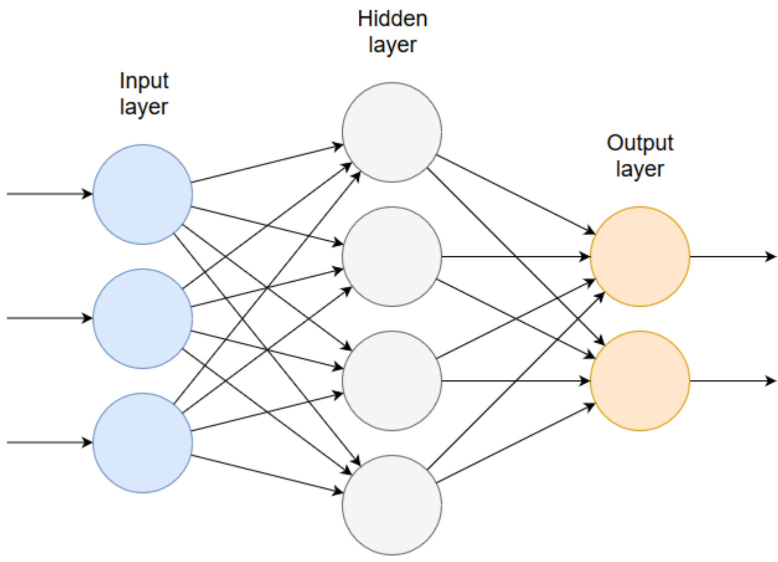

Figure 2.3 multi-layer neural network

Though different architectures exist, most commonly used is *feedforward* neural network with fully connected layers, which means neurons are connected in direction from input to output, with no loops involved. Also, all neurons from one layer are connected to inputs of all neurons in following layer – but it does not mean neuron have multiple outputs. It is just provided to multiple places at once.

Neural networks with a certain level of complexity, where multiple hidden layers are used are called *deep neural networks*.

Inference

Inference is a process of reasoning with a goal to draw a conclusion. In neural networks it is process of passing input data trough network to classify data to certain category usually with certain probability (e.g. 89% cat, 11% dog). Input data are provided to input layer, where each neuron calculates it's value by addition of multiplied weights with input value. After application of activation function data are

passed to next connected neurons. The last output layer will provide output which is within certain interval (determined by activation function).

Learning

The biggest disadvantage in neural network explained so far is that weights and activation function biases need to be predetermined. How could one deduce the correct values to allow for example voice recognition? That would make it no more useful or unique than any other types of data classifications.

However, it turns out that using learning algorithms we can automatically derive and tune those values even without programmer. To do so we use pre-classified external data – learning data. By this, neural networks can learn to solve problems which conventional algorithms would hardly be able to.

Unfortunately, learning does not allow to design neural network itself – even when the number of input and output neurons is determined by application, the number of hidden layers and neurons in each layer needs to be determined by programmer according to difficulty of problem being solved. No exact universal solution exists.

Network is considered as learned when it can approximate correct classification of learning data. To learn the network, it is first randomly initialized with weights and biases. Then input data are passed through to calculate output. This step is called forward propagation – data are propagated in forward direction – from input to output. Basically, it is inference, but since learning data are already pre-classified, we can calculate how well the classification performed. To do that, we use loss function. The learning process is just process of minimalizing this loss function by multiple forward propagations – epochs. Minimalization itself is just internal modification of weights and biases to allow next forward propagation to provide better result. Because this step is going backward, it is called backpropagation. Process is repeated until certain accuracy is reached or after determined number of epochs (to avoid over-fitting).

Over-fitting

Over-fitting or over learning is a problem, when neural network model learns some unwanted regularities from learn data or even memorize all training inputs. Such example could be network identifying cats in the image, but in training data there would be only black cats. This could easily lead to generalisation that all cats need to be black.

2.3 Convolutional Neural Networks

Convolutional neural networks (CNN) are next level of abstraction and they utilize a slightly different architecture. They are gaining popularity thanks to their potential in processing of big amounts of data, like for example in image recognition. While rapidly reaching end customers and spreading to mobile platforms, demand for performance optimisations in these computationally expensive solutions increase. For this reason, we focus our attention on this type of neural networks, and we will explain their details below.

Each layer of CNN is organized in three dimensions, representing width, height and depth. Therefore, neurons in one layer does not connect to all the neurons in second layer, but just into a small part. This grouping allows to increase performance, effectivity and storage space, because compared to fully connected neural network require less connections, therefore less stored weights and less calculations during inference. They are widely used in image classification.

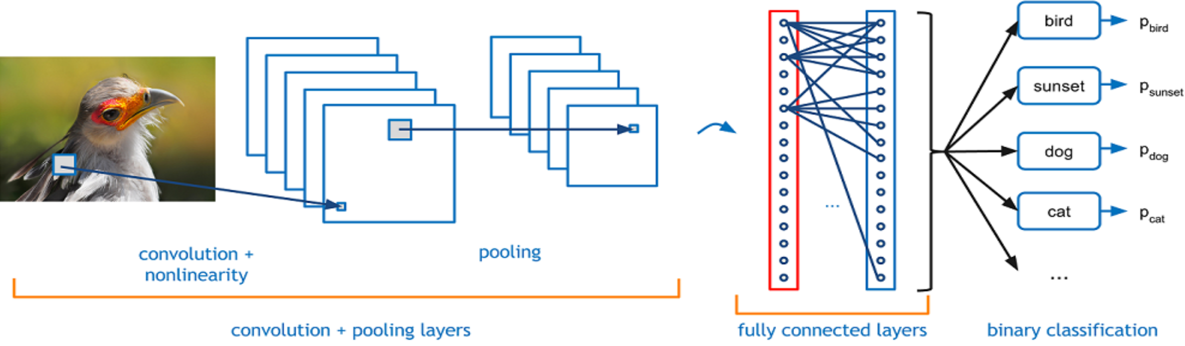

Figure 2.4 scheme of convolutional neural network [7]

As name suggests, convolution is the main operation in CNN. The best idea to describe it is as a window which is sequentially moving through pixel values of input image (or latter just weights), doing matrix multiplication and creating new output called *feature map*. This could be seen in Figure 2.5. The windows are actually called *filter* or *kernel* and it can be in various sizes. Feature maps allows model to learn basic features like geometric shapes (e.g. lines) up to more abstract ones deeper in the network like specific objects (e.g. eye).

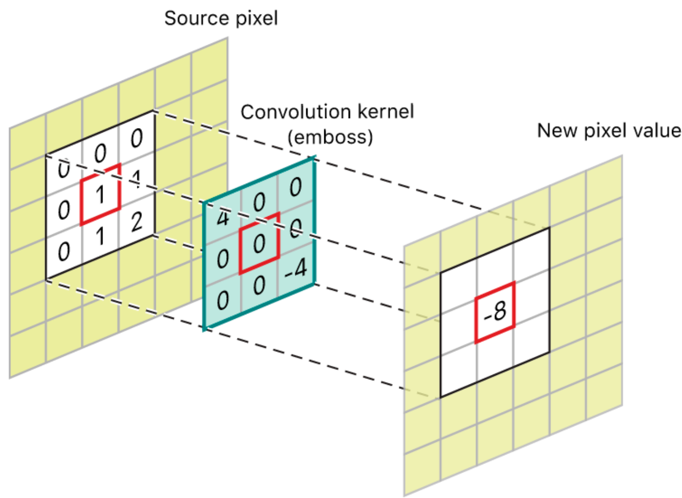

Figure 2.5 Convolution [8]

Limiting factor of convolution is its ability of remembering the position in the input. This could result to different feature maps even for just slightly shifted inputs. One can apply various changes to input like rotation, shifting, cropping or similar, however easier and more effective approach is to use down-sampling. This allows to create so-called summarized version of detected features thus not only reducing size but also lowers risk of over-fitting. This type of layer is called *pooling*.

There are 2 commonly used pooling functions:

- Average pooling
- Max pooling

At the end of the network, there is usually *fully connected* layer stacked, working as a final classifier looking at detected features and predicting output.

2.4 Frameworks for neural networks

In this subchapter, various most popular frameworks for neural networks are going to be explained, so we can make easier decision of ideal choice for our realisation. We require robust widely used solution, that is opened for modifications and could be used with common models of neural networks. For interesting visual comparison see Figure 2.6 at the end of this chapter.

TensorFlow

TensorFlow is an open source software library for high performance numerical computation [4]. It is used for machine learning applications such as neural networks. It was released in 2015 and is

being developed by Google Brain team – research team at Google focusing on deep learning artificial intelligence.

TensorFlow can run across a variety of platforms like CPUs, GPUs or TPUs – a Tensor Processing Unit (application specific integrated circuit developed by Google as AI accelerator for neural network machine learning). This means it can be used from mobiles to desktops, to embedded devices, to specialized workstations, to distributed clusters of servers on the cloud or on-premise. There are also different implementations available, like TensorFlow.js (JavaScript library for use in web browsers or on Node.js run-time environment) or TensorFlow Lite (lightweight solution for mobile and embedded devices).

Theano

Theano is Python library for fast numerical computations developed by LISA group at University of Montreal in Quebec, Canada. Developed in 2007, it was one of first libraries to handle computations required for large neural networks and deep learning. Like TensorFlow, it is also possible to run on CPU or GPU, on mobile or on desktop [9]. Even that Theano was considered as state of the art for deep learning algorithms, unfortunately official support of Theano ceased in 2017.

Keras

Keras is open source neural network library written in Python, or just high-level API [10]. It is capable to run on top of TensorFlow or Theano. For TensorFlow it is an officially recommended way to use it. It allows developers to quickly build neural networks without worrying about low-level aspects like mathematical techniques or used methods [4]. The key advantages are that it has simple and optimized interface for common use cases, it is modular and composable, which means there are configurable building blocks that could be connected. It could be also easily extended to express new ideas for research and is suitable not only for beginners but also for experts.

PyTorch

PyTorch is open source machine learning library for Python. It is primarily developed by Facebook's AI research group. It allows GPU acceleration and runs on various platforms. It was first released in 2016, which makes it one of the youngest libraries in this field. Because of that, the amount of learn materials and examples is much lower.

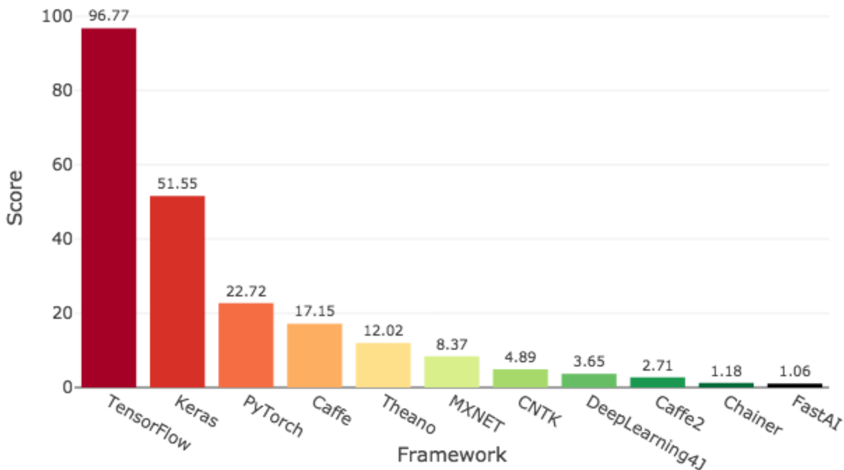

Figure 2.6 Comparison of frameworks by usage, interest and popularity (statistic from September 2018) [11]

2.5 Architectures

Architecture or neural network model is defined way of various layers connected into one working whole. In this thesis, most of experiments are done on one of four architectures created by Google and that are one of the most used solutions in image recognition. Those are Inception v1, Inception v4, MobileNet v1 and MobileNet v2. For both network types applies, that with higher version the higher accuracy is reached, but also more parameters need to be stored and more computations are required for inference. We will introduce interesting details about these networks in this subchapter.

Inception

The main idea between Inception network are so-called *inception modules* [12] which are repeated multiple times forming multi-layer structure. The basic module from Inception v1 could be seen in Figure 2.7.

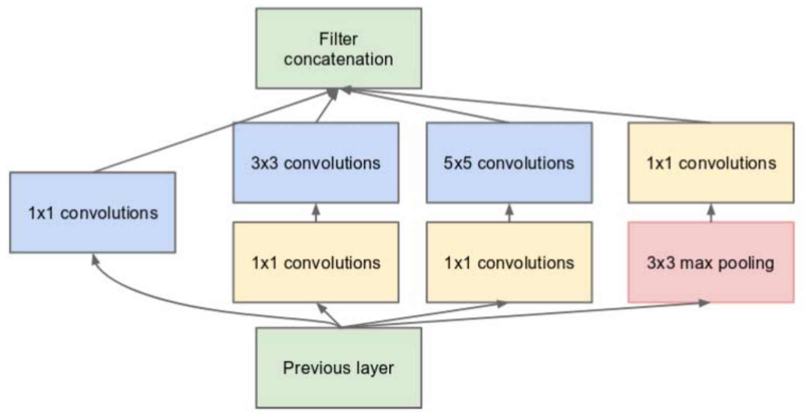

Figure 2.7 Inception module with dimension reductions [12]

In context of inception network, various filter size for convolution allows detection of different features in the input. Even if it may look counterintuitive, the yellow 1x1 convolutions are used to speed the whole process, because they are used to reduce number of input channels before doing expensive 3x3 and 5x5 convolutions. The inception v4 uses three types of inception modules in even more layers, but the main concept stays the same.

MobileNet

Light weight model MobileNet on the other hand, uses Depthwise separable convolution (see Figure 2.8) which is convolution performed independently over channels of input followed by 1x1 standard convolution (called *pointwise*). This trades width for depth and allows for reduction in number of parameters.

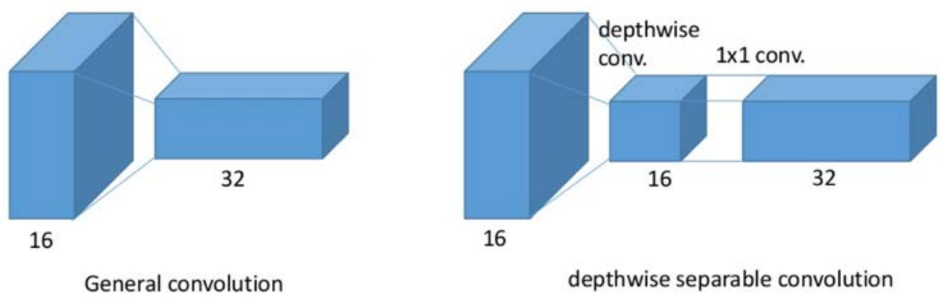

Figure 2.8 General convolution vs Depthwise separable convolution [13]

In MobileNet, parameter α is introduced to control the input width of a layer (*width multiplier*). The effective range is between 1 to 0.25 and allows to trade between speed and accuracy. For example, when using width multiplier 1, the ImageNet accuracy 70.6% is reached with 4.2 million parameters,

while with width 0.5 accuracy 63.7% is reached with 1.3 million parameters [14]. In this work we use MobileNet with alpha set to 1.

MobileNet uses Depthwise separable convolution as part of their building blocks in both version 1 and version 2.

2.5.1 ILSVRC

ILSVRC (The ImageNet Large Scale Visual Recognition Challenge) [15] is competition to evaluate image classification and object detection algorithms at large scale. It allows easy comparison and progress measurement in computer vision through tasks like image classification or object localization.

For this competition, ImageNet dataset with more than 1 million hand-labelled photographs collected from Flickr and other search engines are provided for training. There are 1000 object categories that do not overlap and therefore for each picture just one correct category exists. Final evaluation is done on 50 000 images that are not contained in training dataset.

In 2012 team from University of Toronto submitted architecture called AlexNet, build as deep convolutional neural network, which beat all other teams. They had 41% better results than second team [16]. This started boom in usage of artificial intelligence in computer vision and even attracted big companies like Facebook, Amazon or Google, which for example created architectures called GoogleNet (later called Inception v1 – one explained earlier) and from that point taken the leading positions. Even that competition ended in 2017, legacy is living up to today and models pre-trained for this classification are became standard and represents centralized benchmarking for easy comparisons.

All Models explained previously are trained for this competition and therefore subset of ILSVRC evaluation images will be used in our experiments in chapter 5.

Models used for image recognition in basic form usually return's percentages representing probability, that image belongs to certain category, which means probability of object being on the picture (for example 90% cat, 10% dog). Usually, the decisive parameter of evaluation in big datasets is TOP-1 and TOP-5 accuracy. For TOP-1 it is percentual representation of how many times the category with highest classified probability corresponded with correct answer. For TOP-5 it means how many times the correct answer was within 5 best classifications.

2.6 Energy efficiency

For inference in neural networks, each neuron needs to multiply input data with weights and aggregate those data together by addition. This sequence can be written as matrix multiplication. After this, activation function is triggered to modulate neurons activity and pass it further. That consist of another additions and multiplications (see equations of activation functions in 2.1). When working with more complex neural networks, the matrices multiplications are often most computationally expensive parts [17] [3]. According to Google surveys of their production services, number of stored weights in neural network architectures varies from 5 to 100 million [17]. Those data not only need to be stored, but it means that every inference requires massive amount of calculations. That could be especially limiting on low power devices.

2.6.1 Quantization

To deal with this, technique called quantization could be employed. Instead of using 16 or 32-bit floating point numbers, integers with e.g. 8-bit could be used while maintaining appropriate level of accuracy [17]. This approach allows up to 4x model size reduction and faster execution. Performance comparison depends on used platform, but usually vary from 1.2x to 3x better [4]. In term of portable devices this also means lower power consumption. On the other hand, thanks to big error resiliency of neural networks, as one can see in Table 2.2, drop in accuracy is much less significant than expected.

Network	Floating point			8-bit fixed point		
	Top-1 accuracy	**TF Lite Performance**	**Model size**	**Top-1 accuracy**	**TF Lite Performance**	**Model size**
Mobilenet-v1-128-1	65.2%	57.4ms	16.9Mb	63.3%	24.9ms	4.3Mb
Mobilenet-v2-224-1	71.8%	117ms	14.0Mb	70.8%	80.3ms	3.4Mb
Inception_V3	77.9%	1433ms	95.3Mb	77.5%	637ms	23Mb
Inception_V4	80.1%	2986ms	170.7Mb	79.5%	1250.8ms	41Mb

Table 2.2 accuracy comparison of some quantized popular neural networks [4]

In modern frameworks, 2 possibilities to deploy quantization are available [4]. First one called *post-training quantization* takes already trained model of neural network and quantize it by compression to lower bit integer representation. For example, values ranging from -10.0 to 30.0 could be quantized to range from 0 to 255. In Figure 2.9 one can see possible conversion from 32-bit

float to 8-bit representation – resulting curve is still maintaining its shape but introducing certain amount of aliasing.

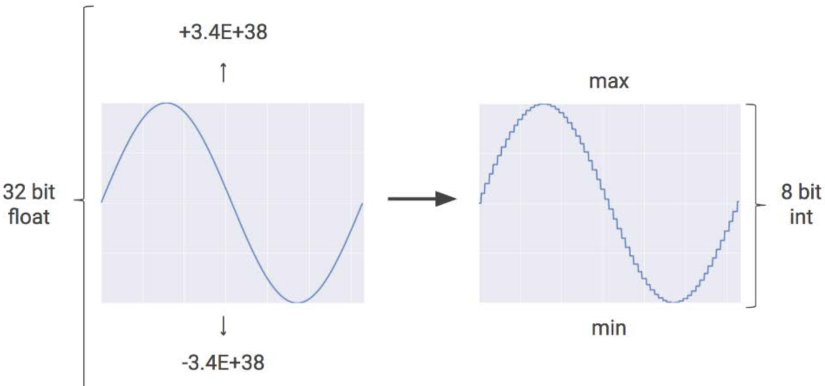

Figure 2.9 float to integer compression [4]

In some applications, especially with smaller networks, this could however lead to higher accuracy drop. For this *quantization-aware training* is recommended, so the model of network is directly created within bounds of quantization.

3 Approximate computing

To further improve energy efficiency of neural networks, we will explore possibilities lying in technique called approximate computing.

The main idea is to optimize computer systems by allowing possible inaccurate results instead of guaranteed accurate ones with trade for improved energy efficiency, performance and chip area. This could be employed in so called error resilient applications where an approximate result is sufficient for its purpose. Those include signal processing, data mining, multimedia, neural networks or others.

Example of such application could be in multimedia, where limited human perception capabilities can allow occasional frame dropping. Other could be in search engines, where no correct solution exists, and many answers are acceptable. There is a study which conducted that about 83% of runtime of applications that are suitable for approximation is spend within computations that can be approximated [18].

Various approaches to approximation can be used, while major ones are:

Approximate circuits
Digital circuits like adders or multipliers could be approximated, which allows for size reduction or increased computation speed. As simple example could be multi-bit adder which ignores carry and therefore compute in parallel.

Approximate storage
This could e.g. truncate the lower-bits in floating point data and save space, or it can perhaps use less reliable memory (by changing supply voltage in SRAM or lowering refresh rate in DRAM [19]).

Approximation on software level
Techniques like memoization, loop perforation (skipping of some loop iterations), tasks skipping or randomized algorithms like Monte Carlo could be used to achieve faster execution time.

3.1 Approximate arithmetic circuits

In this subchapter we will discuss approximate circuits in slightly greater detail, because they could be used in neural networks in form of approximated multipliers or adders.

As a motivation, one could examine Figure 3.1 representing various approximated circuits, their worst-case absolute error rate and power savings. For example, just by introducing 1% worst-case absolute error, the power demands are reduced by half.

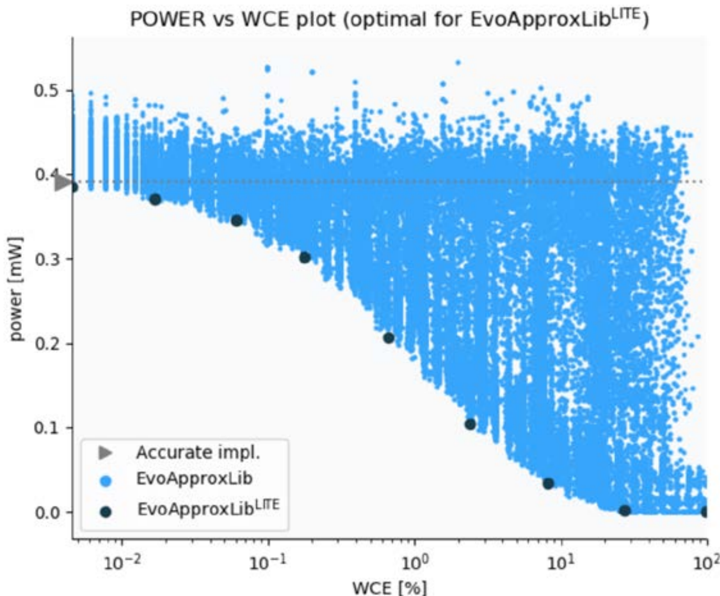

Figure 3.1 trade-off between precision and power [20]

For digital circuit approximation 2 main groups of techniques exist – *over-scaling* and *functional approximation.*

Over-scaling

Energy consumption of electronical circuits could be reduced by voltage over-scaling, which means using i.e. lower power supply voltage in which is known circuit will occasionally cause error. On the other hand, over-clocking of circuits (i.e. increasing frequency) could increase performance but could induce timing errors [21]. Combination of this approaches is known as dynamic voltage scaling.

Functional approximation

Instead of using original circuits, implementation of slightly different ones with acceptable deviation of result could lead to optimized power consumption, chip area and/or performance. Such example could be by omitting least significant bits of the results and related logic [22].

Circuit design

Since in over-scaling approach the original circuits are used, there are less difficulties in overall design of electronical circuit. In functional approach, however, need to create unique design is inevitable. Very simple method was mentioned earlier, where one can omit e.g. least significant bits. In small scale it is possible to design circuit manually using logic gates. In [23] there was 2-bit multiplier with 5 gates designed, which provided correct output for 15 out of 16 inputs with delay *2d*. Compared to conventional solution with 8 gates and delay *3d* it provides interesting savings. When it was used in larger multipliers and employed in image processing applications, it reported 30-50% energy savings while keeping mean error between 1.4% - 3.35% [23].

Unfortunately, in larger scale this becomes almost undoable and advanced techniques are necessary. Some methods try to use synthesis to derive new circuit from original, others use heuristics to design one that meets requirements. One of possible methods using evolutionary algorithm is further described bellow.

Regardless used method, each newly designed approximate circuit needs to be measured for error. Metrics like worst case error, error rate, mean error, etc are usually used. Metrics are further explained in chapter 3.2.

Evolutionary algorithm

In term of digital circuit design approximation, evolutionary algorithm in variant of Cartesian genetic programming already allowed to achieve design of high-quality approximate circuits [24] and we will further describe those here.

Evolutionary algorithm is optimisation algorithm inspired by biological evolution based on trial and error problem solvers with metaheuristic and stochastic character [22]. The main idea is creation of multiple candidate solutions (genes) from initial set by introducing stochastic modifications (mutations/genetic operators). Afterwards a new generation is created by so-called natural selection – the removal of non-desirable solutions [22] [25]. The evaluation of solutions is done by a metric called fitness function. When repeated, the process will gradually evolve and will stop when sufficient solution is found, or predefined number of generations is reached.

In Cartesian genetic programming (CGP), variant of evolutionary algorithm, each candidate is represented as oriented acyclic graph [26] and each node represents certain Boolean function.

3.2 Error metrics for arithmetic circuits

To determine accuracy of approximate circuit, various error metrics could be used [22]:

- total error (absolute values summation for all combinations of inputs)
- error rate (how many input combinations returns the incorrect result)
- worst-case error (worst error that occurred among all combinations of inputs)
- mean average error

Usually most interesting are worst-case and mean average errors. On the other hand, evaluation of this metrics requires to examine all possible circuit candidates with all different possible inputs. With increasing circuit size there is quadratic increase of possible options, which dramatically increase evaluation time. For circuits with 8 bits and more this becomes unfeasible [22].

To deal with this, formal verification methods can be used – like Explicit Model Checking, Symbolic model checking, Verification through satisfiability checking or others.

In this thesis, we will work with worst-case error metrics, because there was already research (see [3]) that showed there is direct impact between worst-case error amount and neural network precision. We will use both variants, which are worst-case absolute error (WCAE) and worst-case relative error (WCRE). For absolute error, the circuit is optimized to stay within certain level of variance when comparing to exact value (see Equation 3.1). For relative error, the magnitude of correct result is also considered (see Equation 3.2).

$$absErr = \frac{|E - C|}{2^{2n}}$$

Equation 3.1 WCAE - worst-case absolute error

$$relErr = \frac{|E - C|}{C}$$

Equation 3.2 WCRE - worst-case relative error

Where:

E is estimated value,

C is correct value and

n is bit width of operands.

3.3 Approximation computing in Neural Networks

In most cases small change in neuron value does just a small change in overall neural network classification output. This information in addition with fact that neural networks are typically used in error-resilient applications [27], makes them good candidates for use of approximation computing. Different approaches to use this have been already proposed and we will describe them further.

Methodology proposed in [28] is based on backpropagation algorithms and identify error-resilient neurons. Those with least impact are replaced with approximate ones that use less precise weights and piecewise approximation of activation function. Approximated network is then retrained.

Different approach to identify critical neuron is used in [29], where neuron is considered non-resilient when small perturbation on its computation leads to large output quality degradation. Presented theoretical approach allows to identify suitable candidates for approximation. Their iterative technique allows for approximations using lowering precision, memory access skipping or by using approximate multiplier circuits.

In [30] they focused on approximation of multipliers in weights in the hidden layers of fully connected feed-forward neural networks and for small networks were able to find configurations with good results.

Three mentioned approaches presented possibility to approximate parts of neural networks. In contrast, in our work we are not working with individual neurons but rather with network as whole, or dive to level of convolutional layers at max. Also, we are focusing only on adjustments in inference, without need of relearning.

For general purpose its necessary to implement uniform neural network structure [3]. Research done in [3] showed possibility to achieve that focusing on approximating all multipliers in neural networks. Their software simulation using framework *tiny-cnn* was analysed with MNIST and SVHN datasets and provides impressive results – for some cases, power reduction of 91% was obtained with degradation of accuracy only around 2.80%.

However, in this thesis, we want to focus on big state-of-the-art architectures, working with large datasets and using popular and robust framework.

4 Realization

Even that use of approximate circuits in neural networks already provided remarkable results (as shown in chapter 3.3), most of them were either very application specific (with non-uniform structure) or they were demonstrated only in small scale. Goal of this thesis is to create platform allowing to simulate use of approximate circuits and evaluate it on large neural network, providing evidence for possible practical applications - especially in portable and low-power devices. To our best knowledge, this has not yet been demonstrated, because of various challenges, that we will try getting around with.

Challenges

- Selection of robust and scalable framework
- Definition of scope for which parts are going to be approximated
- Implementing and simulating use of approximated arithmetic circuits
- Designing a metric for evaluating and comparing results
- Dealing with high computational demands

4.1 Selected framework

As explained in 2.4, Theano is not supported anymore, so the selection of most suitable library falls between TensorFlow with optional use of high-level library Keras or PyTorch. They both do great job and PyTorch is gaining a lot of popularity nowadays. However, TensorFlow in combination with Keras is currently one of largest deep learning libraries and in this thesis, it is chosen as library on which all the work will be done. TensorFlow is well maintained, opened and widely used with large community.

At the moment, for circuit approximation, interesting possibilities are only available for computations with integer values. Therefore, support for quantization in our work is requirement. It forces usage of integers in computations and open the doors for simulating approximated arithmetic operations there. Thankfully in variant TensorFlow Lite, which is suited for mobile and embedded devices, there is a support for quantization, and this will be used as our starting point.

As opensource, Tensorflow not only allows us to download the source files for the whole framework, but it also allows us to adjust these codes for our own needs. This is essential as we will need to do non-trivial modifications in Tensorflow internal arithmetic calculations, replacing them with approximated ones. Tensorflow is written mainly in C++, however it is multipurpose robust tool allowing to be used with different programming languages like C, C++, Java, JavaScript or Python.

There are also libraries that could run on top of Tensorflow, like Keras introduced in chapter 2.4. There are also tools to support development for Android or iOS.

The whole solution contains more than 10 000 files with more than 100MB plus includes more than 30 third party libraries or tools. Because of this robustness, the ability for one to understand all nuances is pretty limited. Thankfully, there is step-by-step guide available through official website [4] allowing one to easily setup the working environment. The most important part is the build tool *Bazel* which at the end of the chain creates binary file and allows this newly compiled solution to be installed for example as a new python module. Due to number of files, the compilation takes few hours to finish, but after first run, Bazel is able to monitor only changed files and recompile only necessary dependencies to speed up the process. Still it takes couple of minutes to get new solution working and therefore the process of development is vastly slower than one could be used to.

Our requirements were to modify framework in such ways, it will allow simulating approximated circuits and monitor differences in precision. This demanded the ability to select various arithmetic circuits, adjust parameters or switch back to precise calculations. The approximation was necessary to work on top of quantization. In this work, we focus on convolutional neural networks, because they operate with images, therefore are containing huge amount of data to be processed.

4.2 Approximation scope

Not only because of complexity of framework, it would be difficult to replace all arithmetic operations by approximated ones, but it would also make experimenting much more complicated since there would be need for lot of settings and variability considerations. Therefore, we had to find best trade-off between implementation limitations and impact.

Following subsections explains why our modifications are focused on approximating only multipliers in just convolutional layers of CNNs.

Convolution only

By [31], convolution is taking up around 90% operations in convolutional neural network. For example, if we look at scheme of Neural Network Inception v1 in Figure 4.1, we could see, that convolution layers are really the most represented part (see blue blocks).

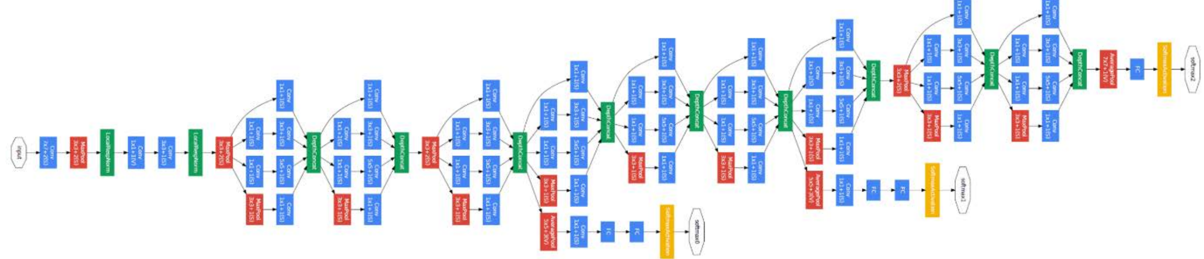

Figure 4.1 architecture of Inception v1 – each blue block is convolution

Another clarification for this could be found in Table 4.1 of subchapter 4.5 below. Considering 80/20 "pareto principle", this makes it ideal choice.

Multiplication only

As it was explained in 2.3 and will be further discussed in subchapters bellow, convolution is basically just big amount of matrix multiplications, which means it is mixture of additions and multiplications. There are 2 main reasons why we avoid approximation of addition. The first is, that energy requirements for multiplier circuits are almost 4 times greater than for addition (when comparing 8-bit width [20]). This will be estimated in greater detail in subchapter 4.5.

The other is technical implementation of simulation that will be additionally explained in following chapter 4.3. In Tensorflow convolutions there are 32-bit accumulators, that would require simulating circuit this bit width. In case of lookup tables, file-size would easily reach gigabytes, which makes this option inapplicable. For inline code simulating circuit of this size, the slowdown would reach even greater values than 10 times. This is not just delusion, but serious disadvantage. Inference of dataset containing 2000 images in convolutional network, which we truly use in experiments, took from 20 minutes easily up to 8 hours, depending on complexity of used neural network architecture. Having deceleration in magnitudes of more than ten would practically disallow use of experiments with datasets of this size.

4.3 Approximation circuits simulation

If we want to use approximated arithmetic circuits directly, there would need to be specialized hardware unit allowing to do this. As one can imagine, building such hardware for neural networks is not an easy task to do. Using pre-existing hardware but rewiring it for usage of different arithmetic circuits does not also seem very likable. Therefore, easiest way to explore possibilities of this technique is by simulating these calculations in software. There are two options for doing so.

One is by using code simulating the approximated calculation – such could be found for example on EvoApproxLib site [20]. This code consists of various bit operations resulting in software version of hardware multiplier. The great advantage is that there are ready to use simulated circuits with precisely classified parameters. There are multiple options with different energy/area savings that could be integrated in our framework.

The other option is usage of lookup tables, where each possible operation and its result is stored in external file. This file could be just as easy as tab separated plain text.

Both options were experimentally verified, and various factors considered, such as speed or usability. When compared for size, for example code for 9-bit multiplier is about 9kb, while lookup table containing more than two hundred thousand rows easily reach 4 MB. One could say that ready-to-use functions are no brainer, but the opposite is true. When tested in real situation, where millions of operations need to be performed, it was just vastly slower (around 10 times) to do simulated evaluation every single time, then just accessing 2-dimensional array already loaded in RAM just once.

Experiments also showed some other advantages – since file is loaded externally, there could be easily any circuit injected. For direct codes, there was need to compile them together with the framework allowing just selecting from limited ones and when experimenting this resulted in need to do unnecessary compilations slowing down the progress. Another 2 other smaller disadvantages for direct codes arise – limited bit width of circuits provided online, and we also had problems with incorrect calculations caused by results in complement.

4.4 Framework modifications

Since we need to use quantization, our sandbox became subpart TensorFlow Lite, which is as rest of framework based on C++. Tensorflow work with neural network model as dataflow graph and according to its structure run various subparts responsible for convolution, pooling or various other operations. Because of its modularity, it was not necessary to dig deep into all parts, but just to find ones responsible for convolution and alter them. However as one would image, Tensorflow uses various optimizations to speed the code execution and allow running on different hardware units like CPU, GPU or TPU. This led to dead end, where external libraries were handling most of computations, especially matrix multiplication, the main part of convolution. Luckily through adjustment of internal constants and by modification of function calls, it was possible to force usage of so-called *reference operations* where most of computations were done by nested *for* loops. Thanks to this we are able to access and modify even arithmetic operations and simulate approximation.

In code for convolution calculations, there are various arithmetic operations, however most of them are for calculations that could be implemented in hardware directly without arithmetic circuits. For example, looping through array of batches or inputs, dilatation (spacing between filter windows), stride (step when moving convolutional window) or even bias or output multipliers. Therefore, in our convolution, there are actually only 2 parts left to be calculated by arithmetic circuits: multiplication of input with filter and then accumulating result in accumulator. These units are often referred as MACs (multiply-and-accumulate). This means, deep down in our unoptimized nested loops, there is one multiplication operation and one addition operation. Each multiplication is replaced with custom code allowing to simulate various approximate multiplication circuits.

To satisfy need for easy application reconfiguration without having to recompile whole framework, various approaches were investigated. For example, there was attempted to create interface allowing to alter behaviour through parameters provided from Python initializations. Even that it was possible to inject custom parameters, this solution later came as not suitable, because it was too problematic to transfer data to desired location through complex code hierarchy, but also more importantly, it obstructed use of any existing evaluation tools (or required altering functionality of tools as well).

Therefore, solution consisting of directly injecting single configuration file in required parts was chosen. This allows us to easily tune the behaviour just by changing parameters in configuration file placed in specific directory, relative to a running script or binary file. The configuration file contains parameters allowing to switch modes of approximation, select lookup table with arithmetic circuit or adjust range of layers that are approximated.

By term *modes of approximation,* we present ability not only to enable or disable use of approximation, but also possibility to forced correct result of multiplication with zero or plus/minus one despite the provided circuit, thus allowing to enrich experiments.

To be able to design metric for calculating estimated savings by approximation, functions representing various layers like convolution, fully-connected or pooling were extended with simple counter, so one can see how many operations are done in single inference.

Evaluation scripts

To trigger and evaluate results, Python scripts using high level API Keras are used. There is a script for MNIST dataset, then there is a script for using Inception and MobileNet networks to evaluate one image at a time, providing TOP 5 results. The most important script is however one already provided by Tensorflow, which is able to evaluate TOP accuracy of particular network. Since we are using

externally loaded configuration file, there is need only to modify this file and let evaluation script do its work. Even that we are not using optimized code (because of necessary modifications), this script allows evaluating multiple images in parallel by using multicore CPU. There are also few Bash scripts for example to run multiple experiments with different approximation circuits in a row, thus simplifying the workflow.

4.5 Metric design

For calculating possible energy or space savings from approximated circuits one needs to know how many mathematical operations are done for image inference and then compare it to number of replaced ones. This was experimentally determined, and results could be found in Table 4.1. Convolution & fully-connected layers contain MAC operations, however as explained previously, we are focusing only on convolution (columns *2D conv* and *Depthwise conv*). Other layers usually contain just addition or other operations and we will not do any replacements there. Last column represents TOP-1 accuracy of network for better understanding of context.

Network architecture	2D conv.		Depthwise conv.		Fully connected		Other		TOP-1 accuracy
	N	MACs	N	MACs	N	MACs	N	Ops	
MobileNet v1	15	551 290 976	13	16 585 792	0	0	1	50 176	70,0%
MobileNet v2	36	279 994 720	17	19 682 864	0	0	1	62 720	70,8%
Inception v1	58	1 434 528 512	0	0	0	0	14	11 811 168	70,1%
Inception v2	70	1 819 415 552	1	14 526 816	0	0	13	11 364 160	73,5%
Inception v3	95	5 467 933 280	0	0	0	0	14	23 303 680	77,5%
Inception v4	150	11 705 716 832	0	0	1	1 537 536	19	42 609 280	79,5%

Table 4.1 experimentally determined overview of operations count for various Neural Networks

To clarify this experiment, for example at [32] is shown, that Inception v1 contains 1.4G MACs – which correspond with our measurements. Our Table 4.1 also shows that almost 99% operations are MACs in convolution. Still, reflecting complexity of framework and therefore possibility of error or deviations in our calculations, we will assume worse-case scenario, meaning that we will consider convolution as responsible for 90% of total calculations (as already introduced in 4.2 or estimated by [31]). This will be used in Equation 4.1 bellow.

Now, since we are focusing only on multipliers, we will also consider the proposed difference between multiplier and adder circuit area (as also introduced in 4.2). In Convolution MACs, there are 9-bit signed multipliers and 32-bit signed adders, so we take this circuits in precise form and compare their parameters. For estimating power saving, we use PDP (power-delay-product), which could

be calculated as multiplication of circuit's power and timing. Those parameters could be found in Table 4.2 as well as calculated PDP value. This metric is widely used in correlation with energy efficiency of logic gates or circuits [33].

Circuit	**Power**	**Timing**	**PDP (Power x Timing)**
32-bit Adder	0,1475	2,59	0,382025
9-bit Multiplier	0,7465	1,85	1,381025

Table 4.2 comparison of parameters for precise arithmetic circuits

By examining Table 4.2, especially column PDP, we can see that *Adder* is appx. 4 times less demanding, than *Multiplier*, even that it has much bigger bit width. Since one MAC contain one multiplier and one adder, by approximating only multipliers we are still covering operations with 80% impact on power savings. Combining this information with one in previous section, we present Equation 4.1, which will be used to estimate total savings.

$$R = 0{,}9 * 0{,}8 * X * Y$$

Equation 4.1

Where:

X is percentage saving of multiplier compared to precise one, and

Y is percentage amount of approximated multiplication operations in convolutions.

To explain this, we can assume for example 9-bit signed multiplier with worst-case absolute error of 0.01% and saving (as will be presented in Table 5.1) of 9,71% – this will be our X. However, if we used this approximation multiplier only on certain convolution layers – for example from 55 till 58, we need to take into account operations in those first three layers, which remained correct and bring no savings. We will need to know how many multiplication operations are in all convolution layers altogether and then how many are approximated. By dividing numbers of approximated with total amount, we will receive percentage representation of approximated multiplications in convolution – our parameter Y.

As an accuracy metric we usually use TOP-1 or TOP-5 values (as explained in 2.5.1) and their deviation from baseline caused by usage of different arithmetic circuits.

Also, by comparing last column in Table 4.1 representing accuracy or re-examining Table 2.2 in chapter 2.6.1, it is important to notice what trade is necessary to do for improving classification. Just 2-3% increase in accuracy could easily mean doubling the network size, which leads to huge increase

in operations amount and computational power requirements. Therefore, typical approach to gain power saving is to use less precise network. We will keep that in mind and later compare our modified networks with less precise non-approximated ones.

5 Experiments

In this chapter various experiments are going to be discussed in order to explore landmark of possibilities for usage of approximated circuits in neural networks. For this we set few basic questions that are going to be discussed:

1. What effect have various approximate multipliers on network accuracy?
2. What is impact of individual convolutional layers?
3. How affected is outcome when inferencing single image?

By answering those we aim to answer the ultimate question – is there practical use of approximate computing in neural networks?

5.1 Experimental setup

Most of our testing was performed on virtual machine provided by BUT's Faculty of Information Technology. The machine was allocated with 23GiB RAM, 4 core Intel Nehalem-C CPU (3.5Ghz) and SSD drive. The operating system was 64bit Ubuntu 18.04. Few other experiments were also performed locally on a little bit slower hardware.

5.1.1 Approximated multipliers

To obtain approximate multipliers, we have used *ADAC* – tool for automated design of approximate circuits with formally guaranteed error [34]. In following experiments, we are using 9-bit signed multipliers. However, by our tests it was found, that we were able to acquire even better results with 8-bit unsigned multipliers, where we "simulated sign" – we used absolute value of parameters and attached sign afterwards (by build-in multiplication). This is most probably caused by used tool ADAC, that could provide results with higher savings for unsigned circuits.

We will work with multipliers optimised for worst-case absolute error (WCAE) but also for worst-case relative error (WCRE), where magnitude of result is considered (as explained in 3.2). Please note that tool ADAC cannot optimise for WCRE for signed circuits and therefore WCRE circuits are used only in form of unsigned 8-bit ones with simulated signs.

Table with approximated multipliers used in this work, their power, area and PDP parameters as well as estimated savings could be found in Table 5.1. These precise data were obtained using Synopsys Design Compiler and synthetized to 45nm semiconductor technology called gscl45nm.lib. For 8-bit unsigned multipliers there is already considered the need for special control circuits providing sign simulation (which could be realised mainly by multiplexors) and therefore this circuits are directly comparable to precise 9-bit signed one. Last column representing percentual saving of approximate circuit is calculated in contrast to this precise 9-bit signed one (first row of table).

Bit width	Error type	Error amount	Power	Timing	PDP (power * timing)	Saving PDP
signed 9-bit	PRECISE	0%	0,7465	1,85	923,8266751	0%
signed 9-bit	WCAE	0.0001%	0,7357	1,68	1,235976	10,50%
signed 9-bit	WCAE	0.0005%	0,7387	1,68	1,241016	10,14%
signed 9-bit	WCAE	0.001%	0,7317	1,66	1,214622	12,05%
signed 9-bit	WCAE	0.005%	0,7153	1,77	1,266081	8,32%
signed 9-bit	WCAE	0.01%	0,7104	1,79	1,271616	7,92%
unsigned 8-bit	WCAE	0.001%	0,7585	1,76	1,33496	3,34%
unsigned 8-bit	WCAE	0.01%	0,7222	1,83	1,321626	4,30%
unsigned 8-bit	WCAE	0.1%	0,677	1,75	1,18475	14,21%
unsigned 8-bit	WCRE	10%	0,7692	1,92	1,476864	-6,94%
unsigned 8-bit	WCRE	20%	0,7513	1,78	1,337314	3,17%
unsigned 8-bit	WCRE	30%	0,5137	1,83	0,940071	31,93%
unsigned 8-bit	WCRE	40%	0,3641	1,47	0,535227	61,24%

Table 5.1 approximated multipliers used in experiments

When considering PDP (product of power and delay) as main metric, one can see that circuit saving could span to unintuitive ranges. For example, saving of signed 9-bit WCAE 0.01% is lower, than for more precise signed 9-bit WCAE 0.001% circuit. In this case it is caused by increased timing value, therefore affecting total saving estimation.

There was another workaround implemented and used in experiments to examine impact and that is correct multiplication by zero. This was done just by simply checking input parameters of each multiplication and returning zero result in case any operand was zero. This was used only in combination with 9-bit signed multipliers, because 8-bit unsigned ones already provided this by default. We acknowledge that we cannot estimate correct energy saving for circuits with simulated correct zero multiplication. We will point this out in such cases. We also experimented with simulations of accurate multiplication by plus or minus one, but this did not provide any significant improvements.

5.1.2 Evaluation strategy

One of our first experiments were done on small architecture designed for MNIST dataset (database of handwritten digits), where we progressively extended and tuned implementation to suit needs for approximation circuits simulation. However, because this architecture is too small and MNIST dataset is not considered as very interesting for demonstration of new purposes, we will dive directly into usage of much bigger architectures: Inception and MobileNet.

In default, Inception and MobileNet are learned and evaluated on ILSVRC challenge (as also explained in 2.5). However, this default evaluation is based on inference of 50 000 images which on our limited hardware capabilities would take too much time. Therefore, we propose usage of smaller dataset. To clarify this, we present charts bellow which illustrate that precision difference is gradually converging to baseline.

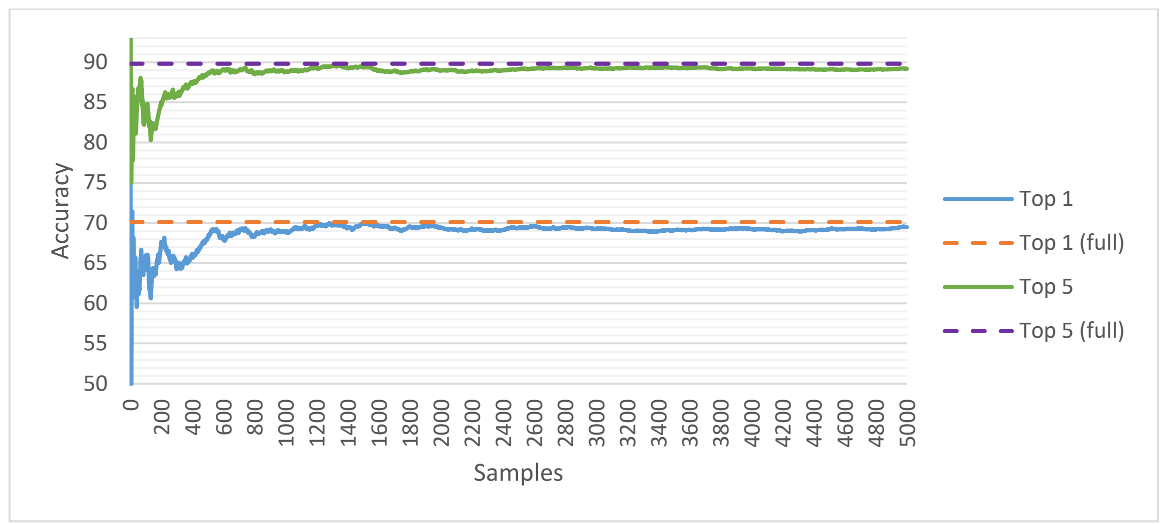

Figure 5.1 Inception v1 accuracy comparison with baseline

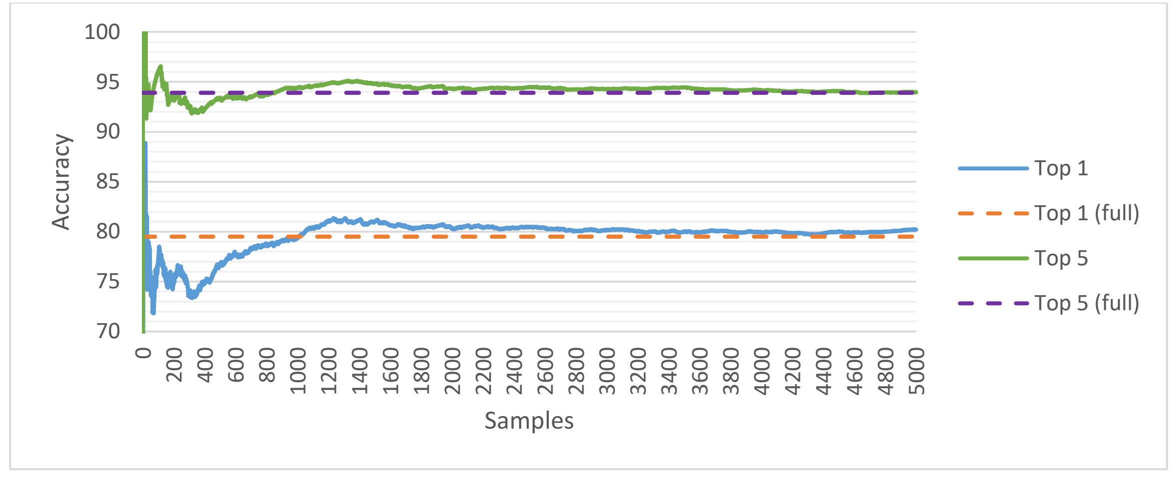

Figure 5.2 Inception v4 accuracy comparison with baseline

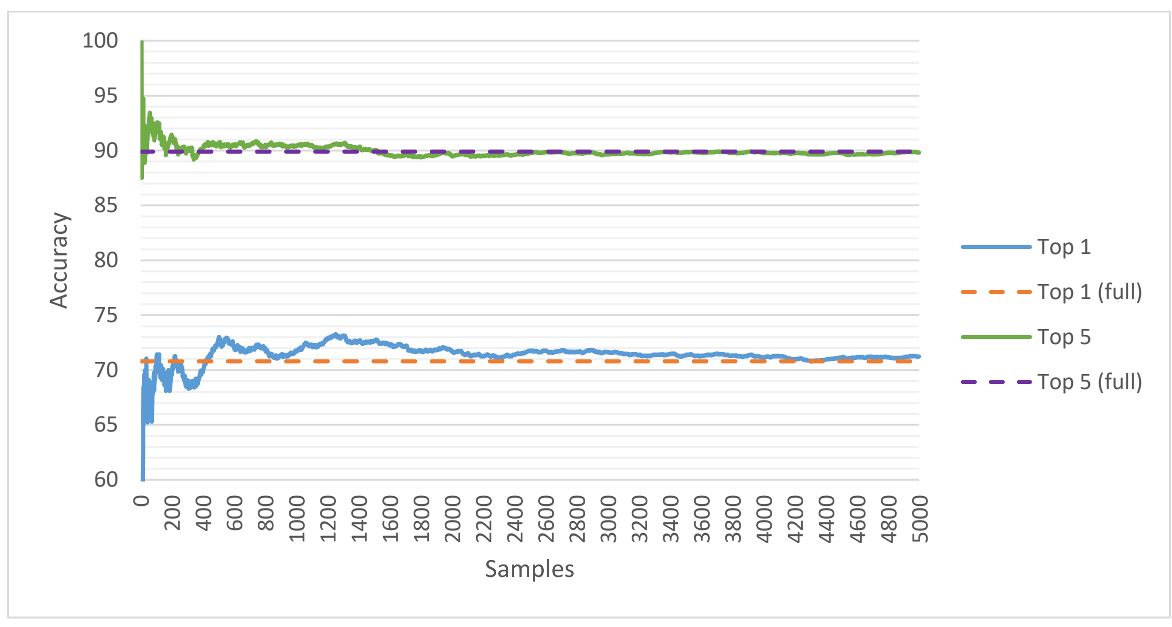

Figure 5.3 MobileNet v2 (1.0 224) accuracy comparison with baseline

In all networks, base accuracy is approximately reached even with 2000 samples and therefore allows us to speed inference 25 times. Still, looking at Table 5.2, for biggest network Inception v4 it's quite a lot of time. We acknowledge, that real results from full dataset could vary in small percentage. There is also significant difference for inference with correct multipliers and with approximated ones. This is caused by fact, that we are doing just simulation of approximation, where we need to access lookup table, which is slower than using arithmetic.

Architecture	Without approximation	With approximation
MobileNet v2	20 min	45 min
Inception v1	1 hour	2 hours
Inception v2	2 hours	4 hours
Inception v3	4 hours	8 hours
Inception v4	8 hours	14 hours

Table 5.2 Speed of inference on 2000 samples dataset

5.2 Approximation of all convolutional layers

This experiment covers inference of first 2000 images from ILSVRC dataset. We use around 20 different approximating multipliers on 3 different networks – Inception v4, Inception v1 and MobileNet v2. For each network type we will compare accuracy difference from baseline, which will

be represented as accuracy drop. Because we are comparing just our own solutions between themselves and we want to work with both parameters TOP-1 and TOP-5, we will use so-called *average drop* (or *AVG drop* in table below) which is representation of mean value. This comprehensive overview could be found in Table 5.3 and is even depicted in Figures Figure 5.4, Figure 5.5 or Figure 5.6. It is important to note, that for multipliers where we simulated correct zero multiplication it is not possible to correctly estimate power savings – however for the sake of comparison we will show those data as well. They will be coloured in grey and are presented in last five rows of table.

Solutions marked with green colour are promising ones. There are some data, which could be filtered at first glance, because the accuracy drops nearly to zero there. Those are represented with red colour. This happened mostly because the multiplier was too incorrect, however, there are cases, when circuit with smaller WCAE gives worse results than one with bigger WCAE. In most cases, this phenomenon was solved, when there was applied approximation mode forcing correct multiplication with zero. However, for one case unsolved glitch remains. This will be further discussed later.

We also experimented with various other multipliers, but they did not bring any interesting results and therefore they are not shown in the table.

	Multiplier	Est. saving	Inception v4					Inception v1					MobileNet v2				
			TOP 1	TOP 1 drop	TOP 5	TOP 5 drop	AVG drop	TOP 1	TOP 1 drop	TOP 5	TOP 5 drop	AVG drop	TOP 1	TOP 1 drop	TOP 5	TOP 5 drop	AVG drop
	correct (baseline)	0,00%	80,30%	0,00%	94,30%	0,00%	0,00%	69,40%	0,00%	88,95%	0,00%	0,00%	71,70%	0,00%	89,50%	0,00%	0,00%
8 bit	WCAE 0,001%	2,40%	80,30%	0,00%	94,30%	0,00%	0,00%	69,20%	0,20%	88,85%	0,10%	0,15%	71,70%	0,00%	89,45%	0,05%	0,03%
	WCAE 0,01%	3,10%	79,85%	0,45%	94,20%	0,10%	0,28%	69,30%	0,10%	89,00%	-0,05%	0,02%	71,20%	0,50%	89,65%	-0,15%	0,18%
	WCAE 0,1%	10,23%	70,10%	10,20%	89,50%	4,80%	7,50%	65,30%	4,10%	87,15%	1,80%	2,95%	62,45%	9,25%	84,00%	5,50%	7,38%
	WCRE 20%	2,28%	79,00%	1,30%	94,20%	0,10%	0,70%	68,15%	1,25%	88,60%	0,35%	0,80%	64,10%	7,60%	85,60%	3,90%	5,75%
	WCRE 30%	22,99%	74,60%	5,70%	91,90%	2,40%	4,05%	63,00%	6,40%	85,05%	3,90%	5,15%	35,40%	36,30%	59,30%	30,20%	33,25%
	WCRE 40%	44,10%	62,65%	17,65%	83,10%	11,20%	14,43%	50,15%	19,25%	74,50%	14,45%	16,85%	0,90%	70,80%	2,50%	87,00%	78,90%
9 bit	WCAE 0,0001%	7,56%	80,30%	0,00%	94,30%	0,00%	0,00%	69,20%	0,20%	88,85%	0,10%	0,15%	71,70%	0,00%	89,45%	0,05%	0,03%
	WCAE 0,0005%	7,30%	78,55%	1,75%	94,30%	0,00%	0,88%	69,25%	0,15%	88,90%	0,05%	0,10%	72,00%	-0,30%	89,55%	-0,05%	-0,17%
	WCAE 0,001%	8,68%	0,20%	80,10%	0,70%	93,60%	86,85%	61,70%	7,70%	84,35%	4,60%	6,15%	65,95%	5,75%	86,30%	3,20%	4,48%
	WCAE 0,005%	5,99%	0,00%	80,30%	0,42%	93,88%	87,09%	14,40%	55,00%	31,20%	57,75%	56,38%	35,85%	35,85%	60,45%	29,05%	32,45%
	WCAE 0,01%	5,70%	76,85%	3,45%	92,60%	1,70%	2,58%	67,25%	2,15%	88,00%	0,95%	1,55%	61,20%	10,50%	83,10%	6,40%	8,45%
	*WCAE 0,0001%	7,56%	80,30%	0,00%	94,30%	0,00%	0,00%	69,20%	0,20%	88,85%	0,10%	0,15%	71,70%	0,00%	89,45%	0,05%	0,03%
	*WCAE 0,0005%	7,30%	78,55%	1,75%	94,30%	0,00%	0,88%	69,25%	0,15%	88,90%	0,05%	0,10%	72,00%	-0,30%	89,55%	-0,05%	-0,17%
	*WCAE 0,001%	8,68%	77,15%	3,15%	92,55%	1,75%	2,45%	68,25%	1,15%	88,90%	0,05%	0,60%	71,45%	0,25%	89,10%	0,40%	0,32%
	*WCAE 0,005%	5,99%	16,00%	64,30%	21,55%	72,75%	68,53%	69,15%	0,25%	88,65%	0,30%	0,27%	69,75%	1,95%	89,05%	0,45%	1,20%
	*WCAE 0,01%	5,70%	72,70%	7,60%	90,85%	3,45%	5,53%	67,85%	1,55%	87,90%	1,05%	1,30%	66,45%	5,25%	86,90%	2,60%	3,93%

Table 5.3 neural networks accuracy comparison with usage of different approximated multipliers (represents multipliers with correct zero, for which we can not estimate energy savings)*

For better understanding we will also depict these data in charts as pareto front, one for each network. Green coloured marks represent same data as green rows from table – the promising solutions. Generally, the best solutions that we are looking for are in bottom right corner – ones with lowest accuracy drop, but with highest power savings. For Inception v1 there is one point marked with yellow colour, which represents the case of promising solution, however there was multiplier with simulated correct zero calculation used and therefore estimation of saving is not correct there. Still, it shows us importance of this edge case and that it should be priority when synthetizing new circuits.

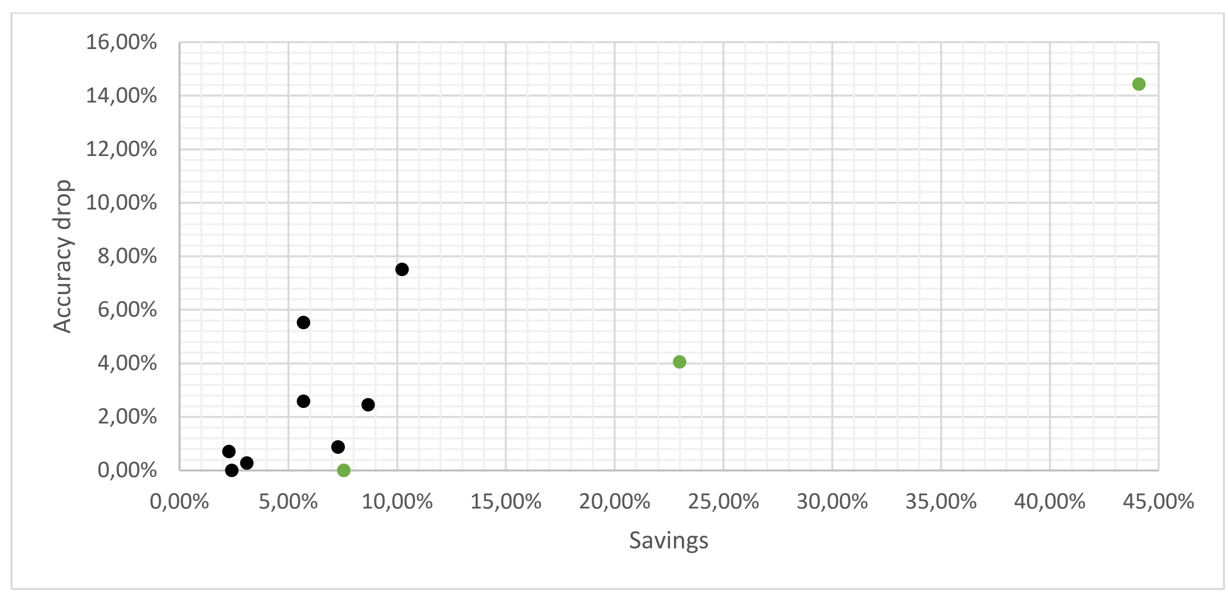

Figure 5.4 various approximate multipliers compared - Inception v4

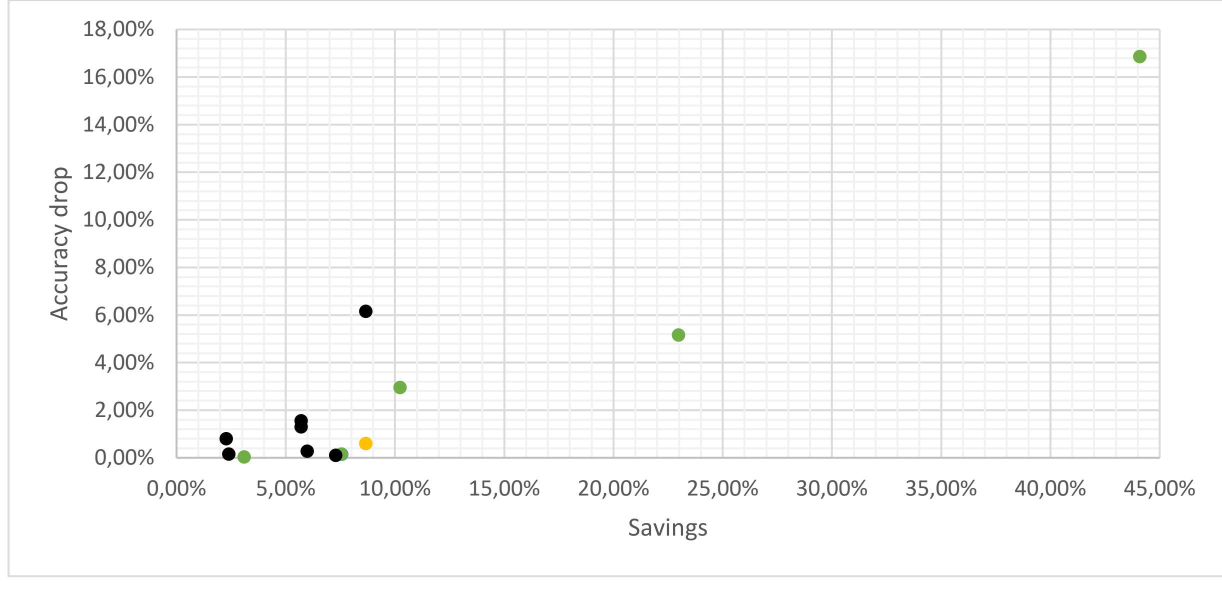

Figure 5.5 various approximate multipliers compared - Inception v1

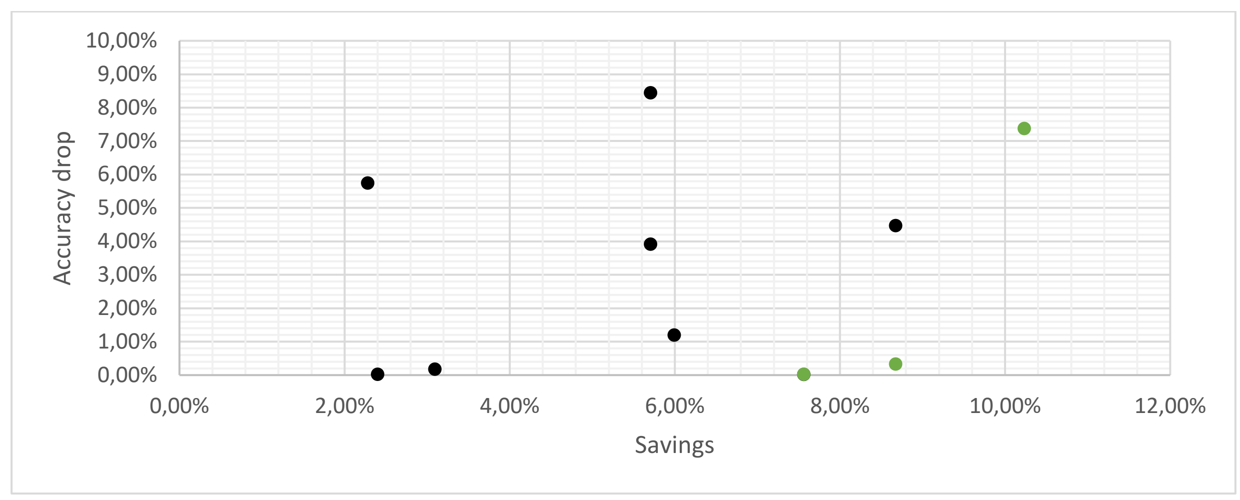

Figure 5.6 various approximate multipliers compared - MobileNet v2

By examining these data, we could see, that there is almost no obvious trend – usually with less precise multiplier there is less precise output, but there are cases when this does not apply. There is also no exact correlation between saved energy and accuracy neither. Still, this does not mean results are not useful. We could see that to gain small savings (e.g. 7,56% from usage of 9-bit WCAE 0,0001%) there is almost no accuracy drop in all networks. If we want to save more, the accuracy starts to drop rapidly though (e.g. 44,10% savings from usage of 8-bit WCRE 40% with accuracy drop more than 10% in all architectures).

Still, it is interesting to notice, that with higher network complexity and size, the sensitivity for error is smaller. Take for example Inception v4, which is the biggest and most precise network used in our experiments and see results for 8-bit WCRE 40%, where average accuracy drop is 14,43%, while for Inception v1 it is 16,85% and for MobileNet precision drops practically to zero. While WCRE 40% may seem promising as it allows saving of 44,10% of energy in multiplication operations, accuracy drop is too big. We will see this more clearly in 6.1.

Even that in most cases the approximation caused lowering of TOP classifications, there is one case (9-bit WCAE 0,0005% for MobileNet v2), where precision even raised. This could of course happen and there could be various explanations, but since we are using much smaller dataset, it is most likely that this will vanish when evaluating on all 50000 images.

When comparing rows representing multipliers optimized for WCAE and WCRE we could also notice, that for WCRE (worst-case relative error) it delivers more promising outcomes than WCAE (worst-case absolute error), especially when we are looking for bigger savings. Compare for example 8-bit WCAE 0,1% with 10,23% savings and 7,5% accuracy drop on Inception v4 with 8-bit WCRE 40% which have double savings of 22,99% and almost half accuracy drop of 4,05%.

When we closely examine difference between correct zero multiplication in 9-bit WCAE circuits and ones without, we could also see, that results do not follow exact pattern. In already mentioned cases correct zero allowed to “fix” accuracy for whole network, recovering it to potentially usable levels. However, there are cases, when correct zero multiplication introduced more error in total accuracy – for example considering Inception v4 with 9-bit WCAE 0,01%, there was 2,58% average drop, and after our fix an average drop has risen to 5,53%.

5.3 Approximation of individual layers

In this experiment, we analysed impact of particular convolution layers one by one on Inception v1 network. We used smaller dataset of 1000 images, because even that its smallest Inception network, it still has 58 convolutional layers and therefore it was necessary to run 58 iterations, approximating operations always only in one convolution layer at a time. Even by reducing dataset to half required 39 hours to finish. If we would want to run this experiment on Inception v4, which have 150 layers and inference is approximately 10 times slower, it would take almost 50 days. This experiment was run 2 times for 2 various multipliers and results could be seen in figures below.

The green and yellow lines are representing TOP-1 and TOP-5 accuracy for precise network. The blue and orange ones are representing the accuracy of whole network, when approximated multiplier was applied on this certain layer. This way we can monitor how strongly it is affecting final result. Since layers are not equal in amount of multiplication operations performed in convolution, the grey overlap represents this trend and allows to take it into account.

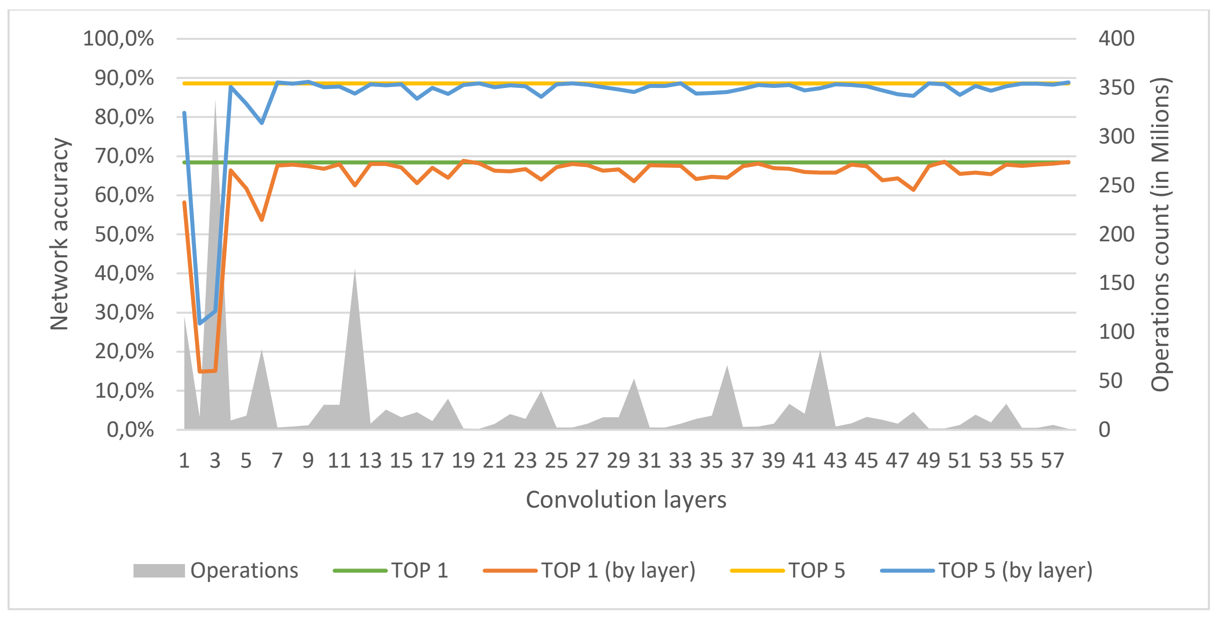

Figure 5.7 layer by layer approximation with 9-bit signed WCAE 0.1%

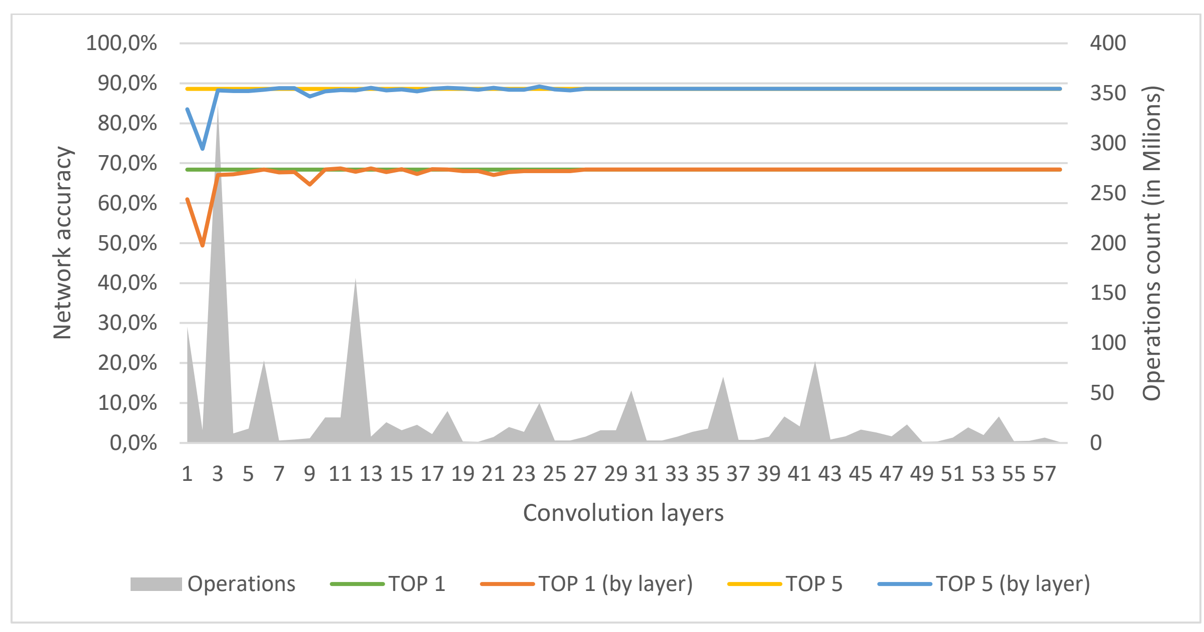

Figure 5.8 layer by layer approximation with 8-bit unsigned WCRE 50% (simulated sign)

Results indicate that first few layers have definitely biggest effect. In case of 8-bit WCRE 50% (Figure 5.8), the accuracy reached the baseline in the middle of network, which indicates, that another half is less sensitive. However, with less precise 9-bit WCAE 0.1% (Figure 5.7), the accuracy oscillated all the way till end. Despite the fact of used WCAE multiplier being less precise, we could see this as another clarification that WCRE circuits have smaller impact on network precision.

When considering amount of operations (grey overlay), the layer number one consisting of 8,1% total amount of operations have noticeable impact on accuracy. However, interesting counterintuitive phenomena happen with layer 2, where only 0,9% of arithmetic calculations happens, but it has biggest impact on the results and therefore this suggest, this layer should not be approximated.

Taking this as inspiration, we will now analyse accuracy and energy savings when approximating only certain convolutional layers (e.g. skipping few first ones). This will be applied on Inception v1 architecture. We will try also less intuitive approach, by skipping for example last five layers or skip some on the beginning of the network and some on the end.

In Figure 5.9 there are orange points representing 4 best solutions for Inception v1 obtained by previous experiment in 5.1. All other points are newly obtained solutions. The Green points represent best ones while approximating multiplications only in certain convolutional layers – we will present those later also in Table 5.4 to see measured data precisely. Other points are represented with black colour. For saving estimation we are considering the various amount of operations in each convolutional layer

(as was presented in 5.3) and therefore lesser amount of operations approximated means lowered total energy savings.

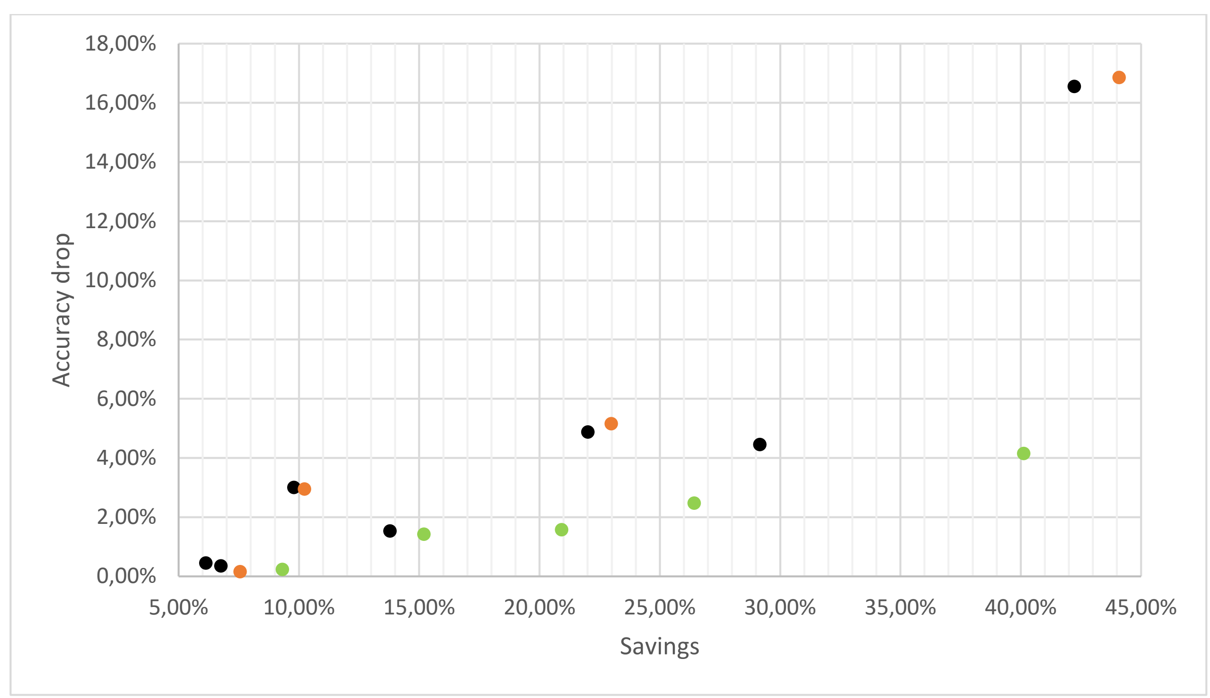

Figure 5.9 approximation of certain convolutional layers in Inception v1

All solutions that showed improvement are from our adjusted unsigned 8-bit multipliers and could be found in Table 5.4. Column *Layers* shows in which convolution layers approximated multiplication was used (from total of 58 for Inception v1). Column *Ops* represents percentage of how many arithmetic operations were actually performed in selected layers when compared to amount of operations in all convolutional layers. Estimated saving which is considering operations count is in column *Est. saving*. First row is again precise solution without use of approximation allowing to compare the rest.

Multiplier	Layers	Ops	Est. saving	TOP-1	TOP-1 drop	TOP-5	TOP-5 drop	AVG drop
Correct	all	100%	0%	69,40%	0,00%	88,95%	0,00%	0,00%
WCRE 30%	5-55	66,14%	15,21%	67,70%	1,70%	87,80%	1,15%	1,42%
WCRE 40%	7-58	59,96%	26,44%	65,85%	3,55%	87,55%	1,40%	2,48%
WCAE 0,1%	3-58	91,00%	9,31%	69,05%	0,35%	88,85%	0,10%	0,22%
WCRE 30%	3-58	91,00%	20,92%	67,30%	2,10%	87,90%	1,05%	1,57%
WCRE 40%	3-58	91,00%	40,13%	64,15%	5,25%	85,90%	3,05%	4,15%

Table 5.4 best solutions obtained by partial convolution approximation on Inception v1

By examining results, the idea of finding and skipping most critical layers showed itself as useful and allowed to decrease accuracy drop while still maintaining enough savings. As best example could be seen 8-bit WCRE 40% used on layers 3-58 which have average accuracy drop of 4,15% and savings of 40,13%. Comparing to results when approximation was used on all layers, the average accuracy drop was 16,85% with savings of 44,10%. Therefore, we increased accuracy by more than twelve percent while still keeping savings above 40%.

Even that in case of Inception v4 we do not have individual data for understanding impact of particular layers (because of unrealizable experiment duration as explained in 5.3), we can just blindly try to skip some of the first layers as Inception v1 experiments indicated and hope for improved results. Still, because of very slow inference on v4 we performed only few tests. Unfortunately, no interesting solutions were found. We tried skipping first 2, 4, 6, 11 and 19 layers, but accuracy was just gradually increasing with decreased amount of approximated operations. No "glitch" that would remarkably improve accuracy was found.

5.4 Single image inference comparison

In previous experiments, we presented various results and differences when evaluating TOP-1 and TOP-5 accuracy as standard approach for ILSVRC dataset. However, to better see and understand what is going on there, this experiment depicts single image inference on 5 individual samples depicted in Figure 5.10 and compare results with usage of 2 different approximate multipliers in Table 5.5. These multipliers belong to category of less precise ones. For each image, there is correct answer and then there are top five classifications of neural network Inception v4.

Figure 5.10 Sample images from ILSVRC dataset [15]

N	Correct answer	Inception v4	Inception v4 + 8-bit WCAE 0.1%	Inception v4 + 8-bit WCRE 30%
1.	**viaduct**	**viaduct 99.6%** toilet tissue 0.0% starfish 0.0% zebra 0.0% sorrel 0.0%	**viaduct 100.0%** toilet tissue 0.0% starfish 0.0% zebra 0.0% sorrel 0.0%	**viaduct 98.4%** toilet tissue 0.0% starfish 0.0% zebra 0.0% sorrel 0.0%
2.	**strawberry**	bubble 87.1% Petri dish 4.7% sea slug 1.6% hip 1.2% **strawberry 0.8%**	bubble 96.5% sea slug 0.8% coral reef 0.4% brain coral 0.4% honeycomb 0.4%	bubble 88.2% Petri dish 3.5% sea anemone 3.1% sea slug 1.2% coral reef 0.8%
3.	**bookshop**	**bookshop 72.5%** library 13.3% bookcase 3.5% tobacco shop 2.8% barbershop 0.8%	library 19.6% lampshade 7.1% bookshop 6.7% bookcase 6.3% tobacco shop 4.7%	**bookshop 47.5%** library 12.6% bakery 4.7% medicine chest 3.5% bookcase 2.3%
4.	**toilet tissue**	grille 19.6% cab 18.4% bottlecap 9.4% car wheel 7.8% tray 4.3%	maze 48.6% bottlecap 13.7% computer keyboard 7.8% patio 3.1% typewriter keyboard 3.1%	cab 25.5% tray 8.2% computer keyboard 7.5% half track 5.9% grille 5.9%
5.	**garter snake**	night snake 38.8% **garter snake 20.0%** king snake 18.4% sea snake 4.3% water snake 3.9%	rock python 78.0% night snake 11.8% water snake 3.9% hognose snake 1.6% diamondback 0.8%	rock python 27.5% night snake 24.3% sea snake 13.3% water snake 11.4% hognose snake 5.1%

Table 5.5 comparison of inference results for single images

For the first image of Viaduct, the result is very precise in all cases. There is also even certainty increase when used with WCAE 0,1%. On the other hand, second image of strawberry, that could seem easy for human, was classified only with 0,8% by pure Inception v4. However, considering TOP classifications, this still fall into TOP-5 classification category. With image 3 there is interesting situation happening, because with usage of WCRE 30% one can see significant certainty estimation drop for bookshop, however from the view of the TOP classification there is no change. Image 4 could be characterized as very tricky and perhaps as a flaw in dataset, since ILSVRC allow only one correct

answer per image and obviously the answer representing some form of vehicle could be less misleading (even that this vehicle is covered by toilet tissues). Even that last image is correctly classified only as TOP 2 and only in case of pure Inception, it shows the power of neural network's image recognition.

This experiment indicates that problem of comparing precision is not that easy and obvious as it may seem and that it is application specific to choose best option. There could be a situation, when ranking or certainty will drop, but also there could be a situation with increase. When considering for example self-driving cars, having certainty of 50% versus 75% for pedestrian on the road may lead to different decisions. However, for example in augmented reality games or search engine result suggestions this could be definitely viable.

6 Conclusions and future directions

In this work, we have studied convolutional neural networks with emphasis on energy efficiency. We described technique called quantization and smoothly transitioned to approximate computing which opens new possible horizons for additional energy savings. To explore those, we created platform allowing to simulate approximate computing by internal modifications of open-source framework Tensorflow. We focused on most computationally expensive parts, which are multiplication operations in convolutional layers.

By our experiments we were able to prove applicability of this technique – even without relearning. We illustrated examples, where with smaller energy savings there were practically no accuracy impacts. For example, on Inception v4 it was possible to gain almost 8% savings while keeping accuracy. Even that we presented solutions with higher energy savings (e.g. 40%), the precision drop was usually more than 10%. This could be a problem as will be explained in subchapter 6.1.

6.1 Comparing to other networks

As we explained in chapter 4.5 or showed in Table 2.2 (chapter 2.6.1), trade for accuracy increase is immense, and it often requires even doubling the size of neural network to gain small percentages. For that reason, there is a need to compare few best solutions across various architectures while in each of them considering amount of operations and their savings.

We will use approach, that if there is for example 10% saving, we would consider architecture as having 10% less multiplication operations. This way we can reflect size with savings applied and use it in contrast with accuracy. Comparison could be seen in Figure 6.1. Default networks are marked with label and with darker colour, while their approximated versions use same, but brighter colour. Because now we are comparing different networks within themselves, we will stick with TOP-1 accuracy. For the sake of this experiment, we also prepared few best promising results from usage of approximate multipliers even on Inception v2 and Inception v3, so we could have better view over size differences and accuracy decrease.

For MobileNet networks this approach does not work, because version 1 have more operations than version 2, while having smaller accuracy. Also, because of MobileNet v2 having higher accuracy than Inception v1, we are not combining those in chart neither. Generally, this mainly illustrates

possible considerations when dealing with energy savings and for exhaustive conclusions would require much more data.

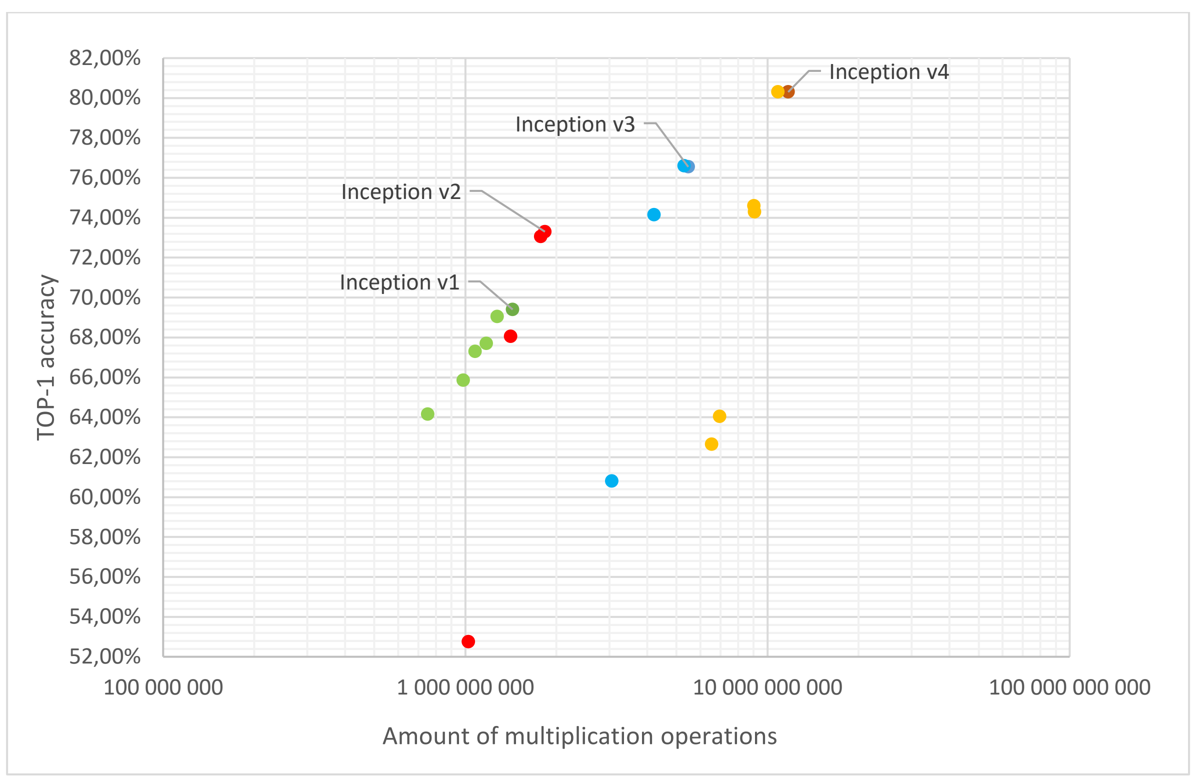

Figure 6.1 Comparison of TOP-1 accuracy between different Inception architectures and multipliers with different savings

For Inception v4 TOP-1 accuracy we can see, that drop of more than 3% reaches accuracy of version 3 and therefore there would need to be saving of such amount that it would compensate for the difference of operations count in those two. This means there would need to be saving of more than 50% to remain competitive.

Unfortunately, in our experiments, multipliers that allows great size reduction also significantly drops with accuracy. Most promising solutions have been obtained for Inception v1, where we were able to gain 40% savings with accuracy drop of around 4%, but for this network we have no direct possibilities of comparison. For networks like Inception v4 the savings of 40% decreased precision for more than 15%, which practically means it's much reasonable to use more precise smaller network like Inception v3 and obtain better results. Still, as we showed in experiment 5.3 with partial approximation on Inception v1, there could be some undiscovered options to dramatically improve accuracy while maintaining great savings even for other versions of Inception architectures.

On the other hand, solutions with savings of around 10% decreased precision in very small amount or in some cases not at all. Because of logarithmic scale on our chart, these solutions could be seen just

right next to point representing architecture with precise multiplications. For example, for Inception v4 without need of relearning there is 7,56% savings with zero impact on precision. This demonstrate that **approximate computing in neural networks could definitely bring positive benefits.**

In next chapter there will be discussion on possible accuracy improvements, however, even in this form, there could be application which could benefit. For example, since there is no need to relearn or to modify network itself, one can deploy more accurate multipliers on device with more computing power and therefore gain more precision, while on mobile architectures there could be used same network with just more energy efficient hardware multipliers still allowing to provide sufficient results.

6.2 Future directions

In this work, we analysed and experimentally demonstrated some possibilities for usage of approximated multiplication in convolution operation of neural networks. In this field however lies many other possibilities to explore. In this subchapter we propose some of them.

Progressive tuning

One of possible improvements could be progressively use multipliers with different types of error – starting with more precise in the critical operations, like at the beginning and then using less precise ones. In experiment 5.3 we tried this approach in basic way, but there is possibility to really use multiple various circuits in different positions. To interesting results could also lead monitoring of impact of multiplication on different convolution types – for example 1x1, 3x3, or 5x5. Since in 1x1 there are less operations, it could be likely that it’s less error resilient.

Replacing all operations

Since in this work we replaced only multiplications in convolution operations, there could be different results when replacing all operations in all layers. Maybe the accuracy will drop even more, but maybe because of same error happening with same input numbers for multiplication it could be other way around.

Finding critical values

As we already clarified, that correct multiplication with zero usually increase accuracy, there could be another critical set of values, that when computed correctly could lead to improvement. Such may be for example the max value (e.g. 255 for unsigned 8-bit).

Experimenting with various other types of approximation multipliers

Experiments showed us, that using 8-bit unsigned multipliers with simulated sign provided best results. Though, this is very dependent on tools and methods used for circuit approximation, there could be better way for obtaining more precision with less error for signed circuits. Other approach may yield in conversion of Tensorflow quantized multiplication calculations to unsigned directly.

Relearning

In relearning network lies probably the biggest potential for improvement, however it's the one most difficult, because of time required to do learning process. For Inception network learning on ILSVRC dataset it is more than 1 million images which could easily lead to weeks or months. The much faster process though could be exploring possibilities of reinforced learning – there, already learned network is used as base and it is retrained on new dataset, just by adjusting the last bottleneck layers. In this work there was attempt going in this direction, however the relearning process goes outside of TensorFlow Lite scope and therefore require modifications in different parts of framework.

Bibliography

1. *Artificial Neural Networks as Models of Neural.* Marcel van Gerven, Sander Bohte. 2018, Lausanne: Frontiers Media.

2. *A Survey on Methods and Theories of Quantized Neural Networks.* Yunhui Guo. University of California, San Diego, Computer Science and Engineering Department.

3. *Design of Power-Efficient Approximate Multipliers for Approximate Artificial Neural Networks.* Vojtech Mrazek, Syed Shakib Sarwar, Lukas Sekanina, Zdenek Vasicek, Kaushik Roy. 2016, IEEE/ACM International Conference on Computer-Aided Design (ICCAD).

4. Tensor Flow™ tensorflow.org. [Online] [Cited: 11 January 2019.] https://www.tensorflow.org/.

5. *Neural Networks and Deep Learning.* Michael A. Nielsen. 2015, Determination Press.

6. Quasar Jarosz. English Wikipedia. [Online] [Cited: 28 April 2019.] https://commons.wikimedia.org/wiki/File:Neuron_Hand-tuned.svg.

7. Rob Hess, Clayton Mellina. Introducing: Flickr PARK or BIRD. *code.flickr.com.* [Online] 20 October 2014. [Cited: 13 April 2019.] http://code.flickr.net/2014/10/20/introducing-flickr-park-or-bird/.

8. Apple. Blurring an Image. *developer.apple.com.* [Online] [Cited: 27 April 2019.] https://developer.apple.com/documentation/accelerate/vimage/blurring_an_image.

9. Jason Brownlee. Introduction to the Python Deep Learning Library Theano. [Online] [Cited: 6 January 2019.] https://machinelearningmastery.com/introduction-python-deep-learning-library-theano/.

10. Keras: The Python Deep Learning library Keras.io. [Online] [Cited: 4 January 2019.] https://keras.io/.

11. Jeff Hale. Deep Learning Framework Power Scores 2018. *Towards Data Science.* [Online] [Cited: 6 January 2019.] https://towardsdatascience.com/deep-learning-framework-power-scores-2018-23607ddf297a.

12. Bharath Raj. A Simple Guide to the Versions of the Inception Network. *Towards Data Science.* [Online] [Cited: 28 April 2019.] https://towardsdatascience.com/a-simple-guide-to-the-versions-of-the-inception-network-7fc52b863202.

13. François Chollet. *Xception: Deep Learning with Depthwise Separable Convolutions.*

14. Sik-Ho Tsang. Review: MobileNetV1—Depthwise Separable Convolution (Light Weight Model). *Towards Data Science.* [Online] [Cited: 28 April 2019.]

https://towardsdatascience.com/review-mobilenetv1-depthwise-separable-convolution-light-weight-model-a382df364b69.

15. *ImageNet Large Scale Visual Recognition Challenge.* Olga Russakovsky, Jia Deng, Hao Su, Jonathan Krause, Sanjeev Satheesh, Sean Ma, Zhiheng Huang, Andrej Karpathy, Aditya Khosla, Michael Bernstein, Alexander C. Berg, Li Fei-Fei. s.l. : IJCV, 2015.

16. Dave Gershgorn. The data that transformed AI research—and possibly the world. [Online] 2017. [Cited: 1 May 2019.] https://qz.com/1034972/the-data-that-changed-the-direction-of-ai-research-and-possibly-the-world/.

17. Kaz Sato, Cliff Young, David Patterson. An in-depth look at Google's first Tensor Processing Unit (TPU). *Google Cloud.* [Online] 2017. [Cited: 12 January 2019.] https://cloud.google.com/blog/products/gcp/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu.

18. *Analysis and characterization of inherent application resilience for approximate computing.* V. K. Chippa, S. T. Chakradhar, K. Roy, and A. Raghunathan. 2013, The 50th Annual Design Automation Conference 2013, pp. 1-9.

19. *Quality Configurable Approximate DRAM.* Arnab Raha, Soubhagya Sutar, Hrishikesh Jayakumar, Vijay Raghunathan. s.l. : IEEE, 15 December 2016, IEEE Transactions on Computers, pp. 1172–1187.

20. The basic library of approximate circuits. *Evolvable hardware group.* [Online] Faculty of information technology. [Cited: 03 May 2019.] https://ehw.fit.vutbr.cz/evoapproxlib/.

21. *Introduction to approximate computing: Embedded tutorial.* Lukas Sekanina. Kosice, Slovakia : IEEE, 20-22 April 2016, In Design and Diagnostics of Electronic Circuits And Systems (DDECS).

22. Jiří Matyáš. *Employing Approximate Equivalence for Design of Approximate Circuits.* Brno : Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor RNDr. Milan Češka, Ph.D., 2017.

23. *Trading accuracy for power with an underdesigned multiplier architecture.* Parag Kulkarni, Puneet Gupta, Milos Ercegovac. Chennai, India : IEEE, 2-7 January 2011, VLSI Design.

24. *Evolutionary approach to approximate digital circuits design.* Zdenek Vasicek, Lukas Sekanina. 3, s.l. : IEEE, 2014, IEEE Transactions on Evolutionary Computation, Vol. 19, pp. 432-444.

25. Thomas Back, David B. Fogel, Zbigniew Michalewicz. *Handbook of Evolutionary Computation.* s.l. : Oxford University Press, 1997. ISBN: 0750303921.

26. *Cartesian Genetic Programming.* Julian F. Miller. 2011, Springer-Verlag.

27. *A survey of techniques for approximate computing.* Sparsh Mittal. 4, March 2016, ACM Computing Surveys, Vol. 48.

28. *AxNN: Energy-efficient neuromorphic systems using approximate computing.* Swagath Venkataramani, Ashish Ranjan, Kaushik Roy, Anand Raghunathan. La Jolla, CA, USA : IEEE, 11-13 August 2014, ISLPED.

29. *ApproxANN: An approximate computing framework for artificial neural network.* Qian Zhang, Ting Wang, Ye Tian, Feng Yuan, Qiang Xu. Grenoble, France : IEEE, 23 April 2015, DATE.

30. *Deep Neural Networks for Acoustic Modeling in Speech Recognition: The Shared Views of Four Research Groups.* G. Hinton, L. Deng, D. Yu, G. E. Dahl, A. Mohamed, N. Jaitly, A. Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, B. Kingsbury. 6, s.l. : IEEE, 2012, IEEE Signal Processing Magazine, Vol. 26, pp. 82-97.

31. *Plenty of room at the bottom? Micropower deep learning for cognitive cyber physical systems.* Luca Benini. Vieste, Italy : IEEE, 15-16 June 2017, IWASI.

32. *Efficient Processing of Deep Neural Networks:A Tutorial and Survey.* Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, Joel Emer. 12, s.l. : IEEE, 2017, Proceedings of the IEEE, Vol. 105, pp. 2295-2329.

33. *Approximating complex arithmetic circuits with formal error guarantees: 32-bit multipliers accomplished.* Milan Češka, Jiří Matyaš, Vojtech Mrazek, Lukas Sekanina, Zdenek Vasicek, Tomas Vojnar. s.l. : IEEE, 2017, ICCAD.

34. *ADAC: Automated Design of Approximate Circuits.* Milan Češka, Jiří Matyáš, Vojtech Mrazek, Lukas Sekanina, Zdenek Vasicek, Tomáš Vojnar. s.l. : Springer International Publishing, 2018.

35. *Siri, Siri, in my hand: Who's the fairest in the land? On the interpretations, illustrations, and implications of artificial intelligence.* Andreas Kaplan, Michael Haenlein. 1, 2019, Business Horizons, Vol. 62, pp. 15-25.

36. Mark Maloof. Artificial Intelligence: An Introduction. *Georgetown University.* [Online] 30 August 2017. [Cited: 11 January 2019.] http://people.cs.georgetown.edu/~maloof/cosc270.f17/cosc270-intro-handout.pdf.

37. *Computing Machinery and Intelligence.* A. M. Turing. 236, 1950, Mind, Zv. LIX, s. 433-460.