BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

FACULTY OF INFORMATION TECHNOLOGY
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

DEPARTMENT OF COMPUTER SYSTEMS
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

AUTOMATED COMPRESSION OF NEURAL NETWORK WEIGHTS

AUTOMATICKÁ KOMPRESE VAH NEURONOVÝCH SÍTÍ

MASTER'S THESIS
DIPLOMOVÁ PRÁCE

AUTHOR — **Bc. MARIÁN LORINC**
AUTOR PRÁCE

SUPERVISOR — **Ing. VOJTĚCH MRÁZEK, Ph.D.**
VEDOUCÍ PRÁCE

BRNO 2024

BRNO UNIVERSITY OF TECHNOLOGY FACULTY OF INFORMATION TECHNOLOGY

Master's Thesis Assignment

153610

Institut:	Department of Computer Systems (DCSY)
Student:	**Lorinc Marián, Bc.**
Programme:	Information Technology and Artificial Intelligence
Specialization:	Cybersecurity
Title:	**Automated compression of neural network weights**
Category:	Artificial Intelligence
Academic year:	2023/24

Assignment:

1. Study and learn the principle of neural network inference operation and the problem of working with memory weights.
2. Get acquainted with evolutionary function design methods.
3. Make a study on the above topics.
4. Propose a method for replacing part of the weights by computation to reduce the energy consumption of neural network inference.
5. Implement the proposed method.
6. Experimentally validate the proposed method on selected neural networks and datasets.
7. Evaluate the parameters of the optimized neural networks, focusing on the tradeoff between the quality (accuracy) and energy consumption.

Literature:

- According to supervisor's advice.

Requirements for the semestral defence:

- Items 1 to 3 of the assignment.

Detailed formal requirements can be found at https://www.fit.vut.cz/study/theses/

Supervisor:	**Mrázek Vojtěch, Ing., Ph.D.**
Head of Department:	Sekanina Lukáš, prof. Ing., Ph.D.
Beginning of work:	1.11.2023
Submission deadline:	17.5.2024
Approval date:	25.4.2024

Abstract

Convolutional Neural Networks (CNNs) have revolutionised computer vision field since their introduction. By replacing weights with convolution filters containing trainable weights, CNNs significantly reduced memory usage. However, this reduction came at the cost of increased computational resource requirements, as convolution operations are more computation-intensive. Despite this, memory usage remains more energy-intensive than computation.

This thesis explores whether it is possible to avoid loading weights from memory and instead functionally calculate them, thereby saving energy. To test this hypothesis, a novel weight compression algorithm was developed using Cartesian Genetic Programming. This algorithm searches for the most optimal weight compression function, aiming to enhance energy efficiency without compromising the functionality of the neural network.

Experiments conducted on the LeNet-5 and MobileNetV2 architectures demonstrated that the algorithm could effectively reduce energy consumption while maintaining high model accuracy. The results showed that certain layers could benefit from weight computation, validating the potential for energy-efficient neural network implementations.

Abstrakt

Konvolučné neurónové siete (CNN) od svojho vynájdenia zrevolucionizovali spôsob, akým sa realizujú úlohy z odvetvia počítačového videnia. Vynález CNN viedol k zníženiu pamäťovej náročnosti, keďže váhy boli nahradené konvolučnými filtrami obsahujúcimi menej trénovateľných váh. Avšak, toto zníženie bolo dosiahnuté na úkor zvýšenia požiadaviek na výpočtový výkon, ktorý je naviazaný na výpočet konvolúcie.

Táto práca skúma hypotézu, či je možné sa vyhnúť načítavaniu váh a miesto toho ich vypočítať, čím sa ušetrí energia. Na otestovanie tejto hypotézy bol vyvinutý nový algoritmus kompresie váh využívajúci Kartézske genetické programovanie. Tento algoritmus hľadá najoptimálnejšiu funkciu kompresie váh s cieľom zvýšiť energetickú účinnosť.

Experimenty vykonané na architektúrach LeNet-5 a MobileNetV2 ukázali, že algoritmus dokáže efektívne znížiť spotrebu energie pri zachovaní vysokej presnosti modelu. Výsledky ukázali, že určité vrstvy je možné doplniť vypočítanými váhami, čo potvrdzuje potenciál pre energeticky efektívne neurónové siete.

Keywords

Convolutional Neural Networks, CNN, Evolutionary Algorithms, EA, Genetic Algorithms, GA, Cartesian Genetic Programming, CGP, Optimalization, Compression, MobileNetV2, LeNet-5, Energy Efficiency, Weight Compression Algorithm, Deep Learning

Klíčová slova

Konvolučné neurónové siete, CNN, Evolučné algoritmy, EA, Genetické algoritmy, GA, Kartźeske genetické programovanie, CGP, Optimalizácia, Kompresia, MobileNetV2, LeNet-5, Energetická účinnosť, Kompresia váh, Hlboké učenie

Reference

LORINC, Marián. *Automated compression of neural network weights.* Brno, 2024. Master's thesis. Brno University of Technology, Faculty of Information Technology. Supervisor Ing. Vojtěch Mrázek, Ph.D.

Rozšířený abstrakt

Umelé neurónové siete (ANN), ktoré sa štandardne používajú v rôznych aplikáciách umelej inteligencie, sa prvýkrát objavili v 60. rokoch minulého storočia [55]. Avšak významný rozmach zaznamenali v posledných dvoch desaťročiach, predovšetkým vďaka napredovaniu vývoja počítačov [8]. S možnosťou pracovať s veľkým množstvom dát tieto siete dokážu efektívne vyriešiť rôzne úlohy, ktoré by bolo konvenčnými spôsobmi prácne vyriešiť.

Výpočtový model umelých neurónových sietí sa značne opiera o matematické operácie násobenia a sčítania. Na začiatku musí model prejsť tréningom, čo sa považuje za optimalizačný problém s cieľom minimalizovať stratovú funkciu. Chyby modelu sú ovplyvnené váhami a hodnotami bias, ktoré sú uložené vo fyzickom úložisku. Na proces inferencie musia byť tieto parametre načítané z fyzického úložiska do rýchlej pamäte DRAM aby mohli byť použité pri výpočte, čo vedie k spotrebe energie. Avšak prístup k pamäti je energeticky náročný [75, 12, 62], čo viedlo k výskumu zameranému na minimalizáciu prístupu do pamäte.

Spotreba energie je kľúčová, pretože priamo koreluje s výkonom, zahrievaním, ekonomickými a environmentálnymi faktormi [16]. Veľké jazykové modely, ktoré preukázali veľký potenciál v interakcii človeka s počítačom, vyžadujú značnú výpočtovú silu. Bolo preukázané, že tréning takýchto modelov môže produkovať viac emisií ako jeden let lietadlom [16], čo viedlo výskumníkov k zahrnutiu výpočtu emisií do svojich správ. Ekonomicky sú tieto modely nákladné nielen na tréning, ale aj proces inferencie je nákladný. Existuje teda silná motivácia zlepšiť oba faktory, poháňaná ekonomickými a environmentálnymi faktormi. Konvolučné neurónové siete (CNN) preukázali, že menej častý prístup k pamäti môže umožniť hlbšie, širšie a energeticky efektívnejšie modely. Predpokladá sa, že váhy je možné ešte viac komprimovať ich nahradením kompresnou funkciou, aby sa zabránilo prístupu k pamäti, čím sa ušetrí energia. V dôsledku toho bol navrhnutý a vyhodnotený nový algoritmus využívajúci Kartézske genetické programovanie (CGP) na rôznych experimentálnych konfiguráciách. Tento algoritmus môže nahradiť váhy aproximačnou funkciou, čím efektívne znižuje počet váh, ktoré je potrebné načítať z pamäte zariadenia a následne znižuje spotrebu energie.

Na dosiahnutie tohto cieľa bol použitý algoritmus CGP na automatický návrh digitálnych obvodov. Návrhy sa vyhodnocujú na základe obvodových parametrov ako je chyba, spotrebovaná energie, oneskorenie a počet hradiel. Po zrealizovaní prvého prototypu prvá iterácia experimentov bola vykonaná na architektúre neurónovej siete LeNet-5 [44], ktorá dosiahla experimentálne sľubné výsledky. Algoritmus dokázal nájsť riešenia bez akejkoľvek spotreby energie pri zachovaní vysokej až pôvodnej presnosti modelu. Avšak, na potvrdenie výsledkov a zaistenie, že model nebol pretrénovaný alebo nadmerne pre parametrizovaný, ďalšia iterácia bola vykonaná na najviac energeticky efektívnom modeli MobileNetV2 [57]. Predtým avšak bolo nutné vykonať optimalizáciu algoritmu, pretože výkonovo nestačil. Zaviedol sa algoritmus, ktorý zhlukuje rovnaké váhy na výstupy a prakticky ich zaskratuje. Tento prístup dosiahol neskutočne zjednodušenie náročnosti dizajnu obvodov pre algoritmus CGP. Prakticky odstránil problém škálovateľnosti algoritmu pre problémy, ktoré nevyužívajú súčiastky multiplexor.

Séria experimentov na MobilenetV2 priniesla zaujímavejšie výsledky a ukázala, že každá vrstva reaguje na zmeny váh odlišne. Okrem toho tiež zdôraznila niektoré obmedzenia algoritmu, ako je optimalizácia energie, ktorá významne závisí od unikátnych váh, ktoré je potrebné vydedukovať. Podstatným zistením je, že aj na tak dôležitom modeli, ako je MobileNetV2, návrh algoritmu stále dokázal nájsť efektívne riešenia a tak prekonať konvenčný

prístup k pamäti. Jedinou nevýhodou je, že každá vrstva je komprimovaná individuálne, a preto z hardvérového hľadiska implementácia nie je v tomto aspekte kompletná.

Na riešenie tohto problému bola implementovaná multiplexná metóda. Avšak, rýchlosť evolúcie implementácie nebola uspokojivá, čo by neviedlo k praktickému použitiu v reálnom prevoze. Rovnaký princíp ako predošlá spomínaná optimalizácia bol pridaný k multiplexnej metóde, avšak vylepšenia neboli markantné. Tento problém teda zostal nevyriešený a mohol by byť spojený na ďalší výskum, ktorý by skúmal optimalizáciu na viacerých vrstvách súčasne. Na záver, vykonaný výskum prispel v oblasti energeticky efektívneho používania neurónových sietí, najmä v prostrediach s obmedzenými zdrojmi, ako sú mobilné zariadenia a elektronické systémy. Výskum ukázal, že je možné aproximovať konvolučné váhy pomocou algortimu CGP. Bolo taktiež zistené, že optimalizácia energie závisí od vstupných váh a od počtu jedinečných výstupných váh, ktoré je potrebné optimalizovať. V experimentoch vykonaných na MobileNetV2 niekoľko riešení dokázalo prekonať energetickú efektívnosť oboch typy pamätí buffer a DRAM.

Automated compression of neural network weights

Declaration

I hereby declare that this Master's thesis was prepared as an original work by the author under the supervision of Ing. Vojtěch Mrázek, Ph.D. I have listed all the literary sources, publications and other sources, which were used during the preparation of this thesis.

. . . .
Marián Lorinc
May 23, 2024

Acknowledgements

I would like to thank my supervisor Ing. Vojtěch Mrázek, Ph.D who led this thesis and guided me through the whole process. Furthermore, I would like to express my gratitude to Czech National Grid Organization Metacentrum CESNET z.s.p. that provided computational resources through the e-INFRA CZ project (ID:90254), supported by the Ministry of Education, Youth and Sports of the Czech Republic. Finally, I would like to express my thank you to Ana, Maddalena, Argyro, Lilla, Gabriele, my sister, my parents and my close family, who supported me all the time.

Contents

List of Figures

Chapter 1

Introduction

Artificial Neural Networks (ANN), widely used in many artificial intelligence applications, appeared in the 60's [55]. However, it has gained significant traction over the last two decades, primarily due to the increased accessibility of computational resources [8]. With access to large quantities of data, these networks demonstrated the capability to solve various tasks that would be challenging to resolve programmatically at the cost of imperfect, although acceptable, accuracy.

The computational model of Artificial Neural Networks heavily relies on multiplication and addition mathematical operations. Initially, a model must undergo training, a process considered an optimisation problem aimed at minimising model loss. Furthermore, model loss is influenced by weights and biases, which are then stored in physical storage. For inference, those parameters must be loaded from physical storage to device memory and used for calculations resulting in energy consumption. However, memory access is an expensive energy operation [75, 12, 62], which resulted in several types of research focusing on avoiding memory access as much as possible.

Energy consumption is essential, as it directly correlates with performance, heating, economic and environmental factors [16]. Large language models have lately exhibited tremendous potential in human-computer interaction; however, as can be determined from the name, those models require a lot of computational power. Notably, it has been proven that training can emit more emissions than a single air flight [16], prompting some researchers to incorporate emission calculations into their reports. Economically, those models are not only expensive to train; the inference process is costly as well. Therefore, a solid motivation exists to improve both factors, driven by economic considerations and environmental concerns.

Convolution Neural Networks demonstrated how less frequent memory access can open opportunities for deeper, wider and more energy-efficient models. It is hypothesised it is possible to compress weights even more by replacing them with a compression function to avoid memory access, therefore saving more energy. As a result, a novel algorithm utilising Cartesian Genetic Programming was proposed and evaluated on various experimental configurations. The resulting algorithm can substitute weights with a learnt function, effectively reducing the number of weights that need to be fetched from device memory and, consequently, reducing energy consumption.

The following thesis is organised into several chapters, gradually introducing the reader to the problem of weight compression using Evolutionary Algorithms. In Chapter 2, Convolution Neural Networks are introduced to explain the required concepts and methodologies that the architecture builds on so these concepts can be later utilised. Following a chapter later, Chapter 3 instigates the field of Evolutionary Algorithms that heavily take inspiration from natural processes to look for optimal solutions through searching space. Chapter 4 proposes a novel convolution weight compression algorithm, which is later more concretely described in the implementation Chapter 5. Several experiments were conducted to validate the algorithm functionality, presented in detail in Chapter 6. Finally, the findings and algorithm performance are concluded in Chapter 7.

Chapter 2

Convolution Neural Networks

Artificial Neural Networks (ANNs) are computational models inspired by the human brain's biological structure [13]. The brain contains an interconnected network of neurons responsible for handling and transmitting signals to other parts of the brain or body parts. These neurons have synapses that act as output points for each neuron. It might be modified or interrupted before transferring the signal to another neuron. From a mathematical standpoint, a Neural Network is often an unknown function $f(x)$ with a noise ϵ that roughly approximates a reference function, as exhibited in Formula 2.1.

$$y = f(x) + \epsilon \tag{2.1}$$

Although a function alone is not capable of approximation, Hornik et al. [27] discovered that chaining multiple non-linear squashing functions g_i (Formula 2.2) allows these squashing functions to approximate the reference function. Hence, the approximation function possesses the universal approximation property.

$$f(x) \approx g_n(g_3(g_2(g_1(x)))) \tag{2.2}$$

As machine learning evolved, it became apparent that only neuron-based neural networks were unsustainable regarding the number of parameters. Hence, research for other architectures was highly motivated. In 1988, Denker et al. [15] developed one of the first Convolution Neural Networks (CNN) to classify handwritten digits for the US Postal Service. The architecture had one big drawback, and it required manual weight calculation of activation maps and a large amount of image pre-processing. As such, later, Denker and LeCun et al. [41] employed the backpropagation algorithm to CNN. As a result, the classifier could classify raw images with minor transformations to the former counterpart, which used input vectors [15]. Thus, adapting pattern recognition principles described in the book of Watanabe [70] led to a novel and more efficient feature extraction architecture type.

Convolution is a computation-heavy operation; however, the most significant energy consumption arises from memory access [12, 75], explained more in Section 2.2. Additionally, CNNs are not suited for every type of task, though they are remarkably good at computer vision [41] and speech recognition tasks. The explanation of why CNN performs better in those tasks can be found in architectural design. Hence, the whole architecture, including the building blocks of CNN, will be explained in Section 2.1.

2.1 Architecture Overview

Convolution Neural Networks excel at pattern recognition tasks, such as computer vision, speech recognition or signal processing. To perform well, CNNs use the convolution function to extract features from patterns, which is especially beneficial when analysing signals and image data. However, convolution layers alone are insufficient for classification and regression problems. In order to reach the most optimal performance when developing CNNs, other layers are required to function well, which will be explained in this section.

2.1.1 Convolution Layer

Convolution layers are the most essential building blocks of CNNs and are responsible for feature extraction from the data. Conceptually, a convolution layer is a particularly restricted case of the fully connected layer, which has more strict restrictions than a regular fully connected layer. However, those restrictions are made in a way that slightly restricts the layer's capability, resulting in a significant reduction of parameters, reducing over-fitting [63]. Every convolution layer consists of a finite number of filters, as shown in Figure 2.1, sometimes called maps, that store weights later for convolution.

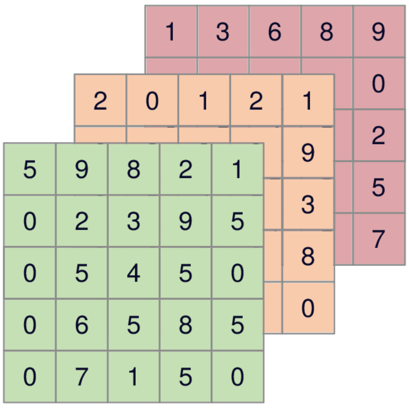


Figure 2.1: A single convolution layer containing multiple filters trained to find specific features in data. Each filter contains trainable weights used for convolution or depthwise convolution.

Convolution

Convolution is a mathematical operator of two functions that has been proven efficient in pattern recognition [15, 41] and since then became a standard in computer vision. Convolution operator has many definitions; for instance, the continuous definition with integral is used in signal processing. However, CNNs function in discrete space. Therefore, it has slightly different definitions, as defined in Formula 2.3, where f is an input image and g is a filter.

$$i * g\ [u, v] = \sum_{i=-a}^{b} \sum_{j=-c}^{d} g[i, j]\ f[u + i, v + j] \tag{2.3}$$

One particular difference is how the sum iterates over values. In CNNs, it is common to use filters with odd kernel size; thus, the variables a, b, c, d are equal, which implies that kernel dimension can be trivially calculated as $(2a+1) \times (2a+1)$. Moreover, compared to the standard convolution, the CNNs convolution also defines parameters such as *stride* and *padding* that define by how much filter is moved across the image after each convolution, and padding specifies how are missing values handled which is a case for image pixels that reside on the edges of the image. For instance, the padding effect can be seen in Figure 2.2, which also incorporates an example of how an image can be convoluted.

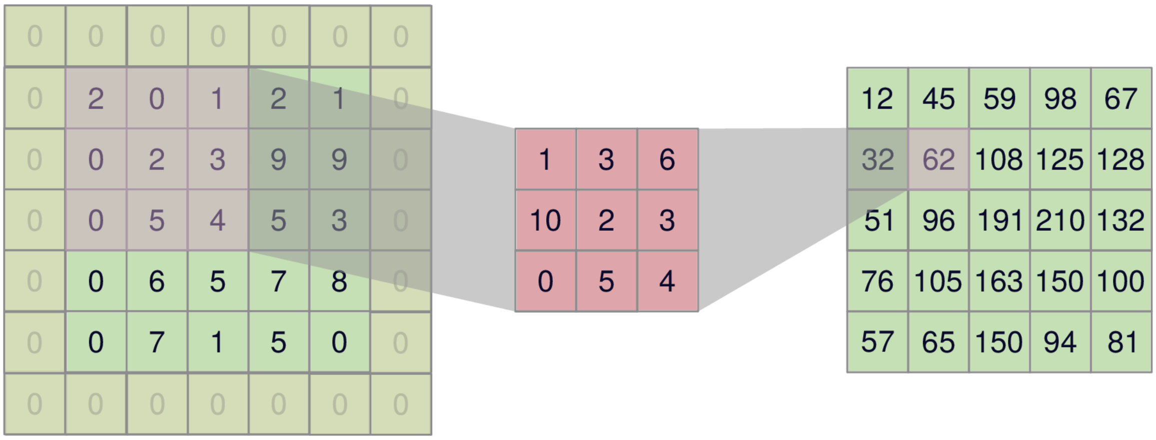


Figure 2.2: Example of 3×3 convolution performed on the region in the left imaginary greyscale image of 5×5, which is padded with zeros. The result is highlighted by purple colour on the right and was calculated with the following formula: $y = 2 \cdot 1 + 0 \cdot 3 + 1 \cdot 6 + 0 \cdot 10 + 2 \cdot 2 + 3 \cdot 3 + 0 \cdot 0 + 5 \cdot 5 + 4 \cdot 4$ from top left to bottom right.

Convolution Evolution

Apart from ordinary convolution, which was first introduced by Denker et al. [15], later Denker and LeCun et al. improved it by utilising backpropagation [41] followed up by the creation of LeNet-5 [44]. However, CNNs were not popular until a significant breakthrough occurred in 2012 when Krizhevsky et al. used GPU parallelization for the first time, ReLU activation, Dropout and data augmentation to train neural networks, which won them the ILSVRC challenge with convolution neural network AlexNet [36]. The most interesting feature of the model is how big the first convolutional filters are with size 11×11. Even though GPU training became a norm with more research by Krizhevsky [35], large filters did not stay ground, and more effective convolution techniques were invented.

One of the more efficient techniques is 1×1 convolution in convolutional layers, as showcased in the proposed model by Lin et al. [46]. The premise builds on the idea that inside feature extraction layers, inter-channel classification and pooling should improve model accuracy. Basically, 1×1 convolution is a compatible multilayer-perceptor, which is a capable function approximator [46]. Furthermore, the standard FCN layer is not used at all at the end of the model. Instead, the last layer is a convolutional network with global averaging pooling, as shown in Figure 2.3.

Moreover, 1×1 convolutions have become widely used and are present in state-of-the-art models, for instance, MobileNet [57, 28] and GoogleNet [63]. However, original 5×5 convolution filters are still a viable option despite Simonyan and Zisserma proving that two 3×3 filters have an effective receptive field of 5×5 filter [58].

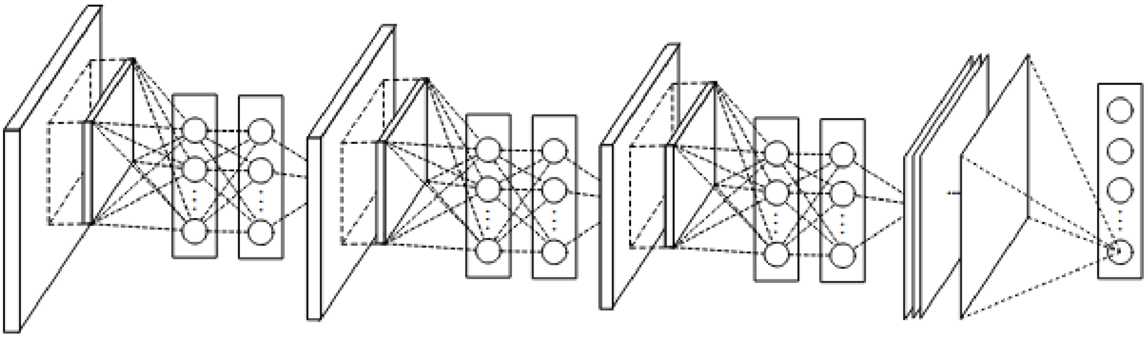


Figure 2.3: Architecture of the Network in Network model proposed by Lin et al. [46] with non-standard architecture which places neurons in between convolution layers. Image courtesy [46].

Convolution Types

As convolution was more researched, multiple convolution techniques were invented. Until now, only the standard convolution has been explained; however, the most recent state-of-the-art models use different convolution methods, such as:

- 1 × 1 **Convolutions** – his technique, as previously mentioned, serves multiple purposes. Firstly, it can be used for pooling, as mentioned earlier. Secondly, it can be used to adjust the input dimension.

- **Depthwise Convolutions** – In this method, only one filter is employed per channel, thereby reducing the number of parameters and convolution filters required.

- **Pointwise Convolutions** – A 1 × 1 convolution is applied for every point value in the input matrix, performing linear combination. It is required to have the same number of filters as input channels.

- **Separable Depthwise Convolutions** – By combining Depthwise and Pointwise convolution, it is possible to perform effective feature extraction trading off fewer parameters for a little bit worse accuracy, as demonstrated by MobileNet [57, 28], as elaborated in Section 2.5.

- **Grouped Convolution** – Inputs are divided into multiple groups, each of which undergoes separate convolution with multiple kernels. Initially proposed as a technique for GPU parallelization in AlexNet [36]. Later, it was used in ResNeXt [73] to introduce cardinality, a new dimension comprising n groups that are eventually aggregated.

- **Shuffled Grouped Convolution** – Building upon the previous idea, a shuffling operation is employed after group concatenation which originated from ShuffleNet [79]. The motivation is to promote feature diversity and to exchange data between channels thereby improving model learning.

Comparission to Fully Connected Neural Layer

As mentioned, the convolutional layers impose restrictions on inference and these are dependent on filters. Whereas FCN has a set of weights w and biases b and performs inference based on the following Formula 2.4, where f is an activation function.

$$i(x) = f\left(\sum_{i=0}^{N} w_i x + b\right) \tag{2.4}$$

Although the formula appears to be simpler to calculate, the problem is that FCN layers require significantly more weights than convolution layers, and that's the primary determinant of their differences. Moreover, the higher weights also cause more intense memory usage, which is, in terms of energy and delay, very demanding [12]. So, by replacing memory access at the cost of more intensive calculation, more can be learnt as the result of better resource utilisation, which is an interesting concept that will be pivotal until the end of the thesis.

Nonetheless, both layer types cooperate well and create a foundation for CNNs. The most common CNN architecture involves nesting multiple convolution layers, also called feature extractors. Nesting is essential because a single convolution layer cannot capture all the required details of a pattern, meaning the deeper the image is processed, the finer details are extracted. Finally, calculated activation maps are passed to the classification part built out of FCN layers. A schematic can be seen in Figure 2.4, and a more specific example in Figure 2.5.

2.1.2 Pooling Layers

A pooling layer is often employed to downscale a given sequence. However, pooling layers are not strictly necessary and are considered optional. Alternative architectures without pooling layers have been found to be as effective, as demonstrated by Springenberg et al. [59]. Despite that, the pooling layer is a beneficial layer when assisting in learning translation invariance [14]. Also, down-scaling input helps prevent model over-fitting [63].

Required hyper-parameters for the layer are a pooling function, pool size, stride and a padding method. The most used pooling functions are min, max and average, which are simple to calculate. Thus, for their simplicity and favourable properties of dimension reduction, they are commonly used to speed up the training and inference process. Figure 2.6 shows an example of the performing max pooling operation.

2.1.3 Dropout Layer

Dropout is a regularization technique to combat the over-fitting problems in neural networks [60, 26]. However, the common dropout technique for FCN cannot be applied to convolution layers because of the spatial correlation between elements in a sequence. For example, pixels in photos are highly correlated as they belong to a particular object; hence, they can be inferred from other pixels. Therefore, a spatial dropout is used instead.

2.2 Energy Analysis

Deep neural networks are known for high computational requirements and energy use. As models become more robust, energy costs increase, resulting in higher financial costs. Thus,

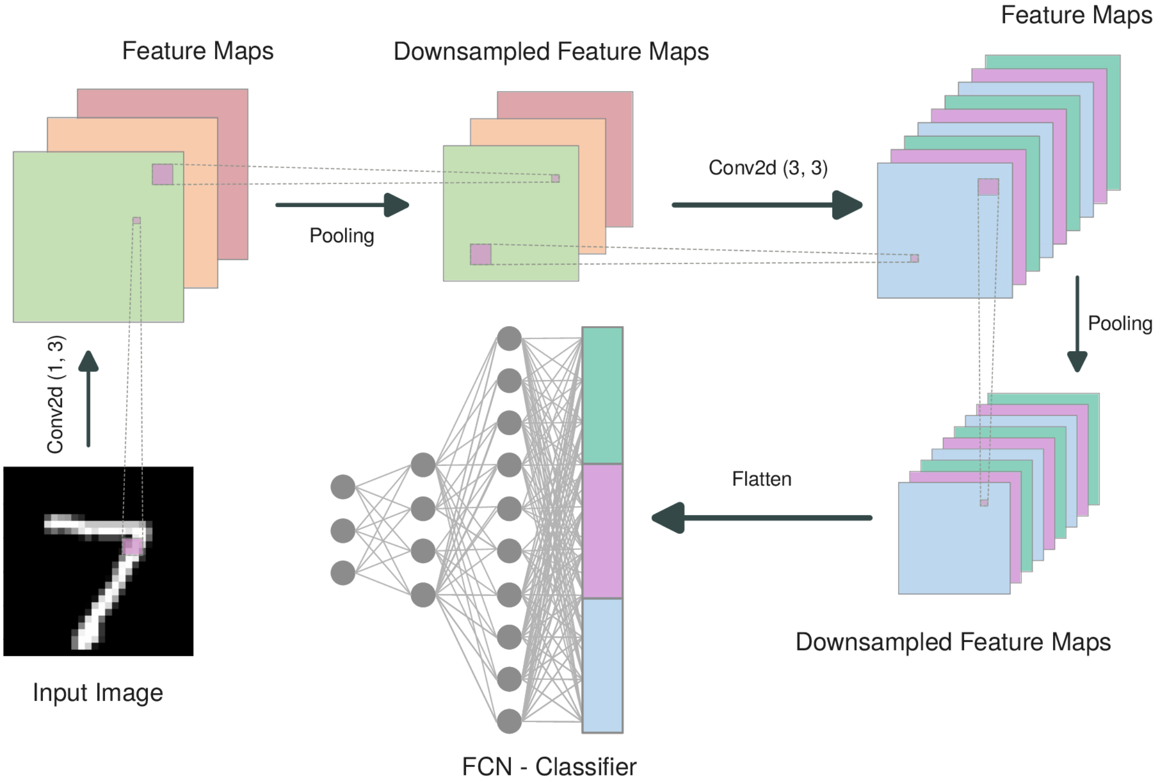


Figure 2.4: Inference workflow diagram showcasing classification for a digit seven from MNIST [39] dataset. In this example, the first layer applies a convolution operator on the image, creating new feature maps consisting of convoluted pixels. The feature maps are downsampled to smaller dimensions in the next step, resulting in faster inference and helping the model learn transition invariance [14]. This procedure can be repeated arbitrary times to extract more detailed features from previously extracted features. Ultimately, the created features must be classified, often done by flattening feature maps into a single-dimensional vector to allow the Fully Connected Network (FCN) to classify the extracted features.

to localise energy hotspots, it is crucial to know how the CNN works on the hardware level, as Yang et al. [75] found out data movement has more significant energy consumption than computation alone, giving an example of GoogLeNet [63] that uses 10% of total energy for computation and 78% for moving feature maps. Similarly, the memory hierarchy and dataflow significantly influence the energy consumption of data movement [75]. Furthermore, Chen et al. [12] demonstrated that reusing weights makes it possible to utilise memory hierarchy less frequently, decreasing energy usage.

Because of all the mentioned reasons, this thesis aims to find and research possible energy savings and evaluate the savings against the loss of accuracy. Therefore, the following sections were split into specific components that play important roles when performing inference. Every section will describe energy usage on its own level.

2.2.1 Convolution Layers

A convolution layer for an input feature map applies multiple convolution filters and, as a result, generates multiple output feature maps. In every step, element-wise multiplication

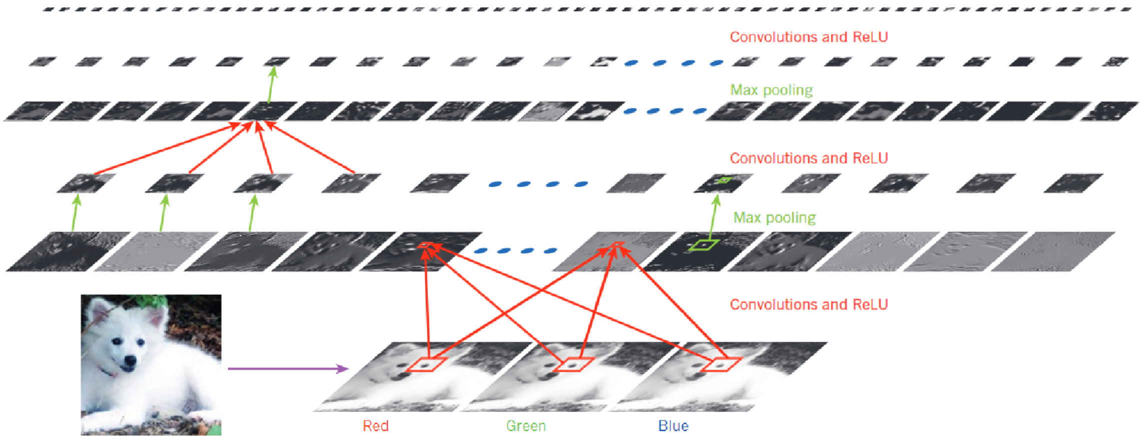


Figure 2.5: An example of the CNN conducting inference on 3-channel input of RGB colour format image of a dog, which is being convoluted by each layer with output feature maps visible as rectangular grey-scale images. Specifically, in this example, the dog is being recognised based on its eye. The image was created by LeCun et al. [40]

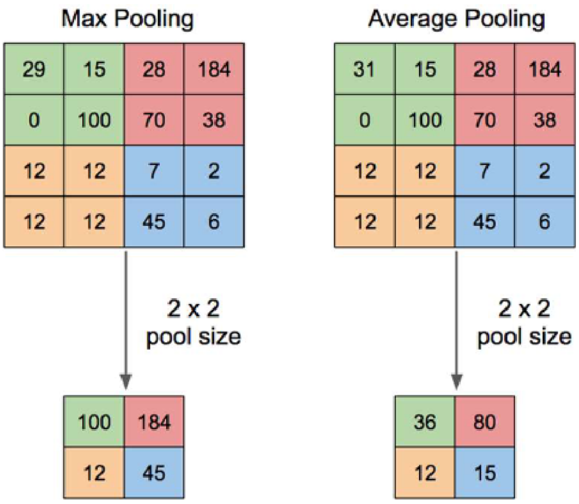
occurs, and result values are accumulated to compute an activation for the output feature map. However, the accumulation cannot be done in a single step. Therefore, values get accumulated iteratively, creating partial sums. As such, the Multiplication and Accumulation (MAC) unit, which is graphically described in Figure 2.7, was established as a way how to measure the computation intensity of ANN. [75]

Convolution layers computation in CNNs can amount to as much as 90% of overall computation resources [14], or in the case of Yang et al. [75] it was 72% even though the fully connected layers consisted of 96% of total weights. Due to the dominant use of computation resources, it is an interesting subject. Chen et al. [12] conducted a study on the impact of energy usage, which was compiled into Figure 2.8. From the image, it is noticeable that memory is energy expensive and opens a new opportunity to research ways to avoid memory as much as possible. For instance, an ALU computation consists of one MAC. Similarly, RF registers have the same amount. However, as the memory hierarchy gets deeper, energy usage increases. Hence, multiple techniques were invented to minimise memory usage.

Data Reuse

Convolution operation offers multiple opportunities for calculation reuse due to how convolution works. The three main techniques are [12] (for visual demonstration see Figure 2.9):

- **Sharing of unique input data** – The method is based on exploiting the weight sharing property of convolution layers, which utilises every weight filter E^2 times in the same input feature map. Nonetheless, fully connected layers do not possess this type of property.
- **Filter reuse** – Every filter weight is reused among the batch of N input feature maps in convolution and fully connected neural layers.



29	15	28	184
0	100	70	38
12	12	7	2
12	12	45	6

100	184
12	45

31	15	28	184
0	100	70	38
12	12	7	2
12	12	45	6

36	80
12	15

Figure 2.6: Operation of pooling with kernel dimension equals two and with a stride of 2, commonly matched with dimension. Max pooling selects the largest value in 2 × 2 sub-matrices and then moves by 2 to the right. Similarly, the average pooling performs the same steps. However, the average value is calculated from 2 × 2 sub-matrices. The image was created by Yani et al. [76]

- **Input feature map reuse** – Every input feature map pixel is reused among M filters in convolution and fully connected neural layers.

2.2.2 Fully Connected Neural Layers

Fully connected neural layers are commonly placed at the end of the CNNs. Even though they dominate in terms of quantity of weights, their energy use is not that significant compared to convolution layers [75]. Still, energy utilisation can be optimised by reusing values and accelerators described in Section 2.3.1.

2.2.3 Activation Functions

Activation functions[1] are used to maintain the property of the universal approximation, which in history was achieved by TanH or Sigmoid functions. However, the latest models use ReLU, which gained popularity with its simple function definition and first-order derivation. Furthermore, those simplifications brought reduced training times and computational complexity, as demonstrated by Krizhevsky et al. [37] in the paper they managed to make training six times faster than an equivalent model with TanH function. Nevertheless, activation functions do not require intensive access to memory, thus rendering them inferior candidates for energy savings measurements.

2.2.4 Pooling Layers

Pooling layers are also not good energy-saving candidates because the most popular function *max* does not require frequent memory access. Moreover, their resource usage is insignificant during the whole process.

[1] introducing non-linearity to models

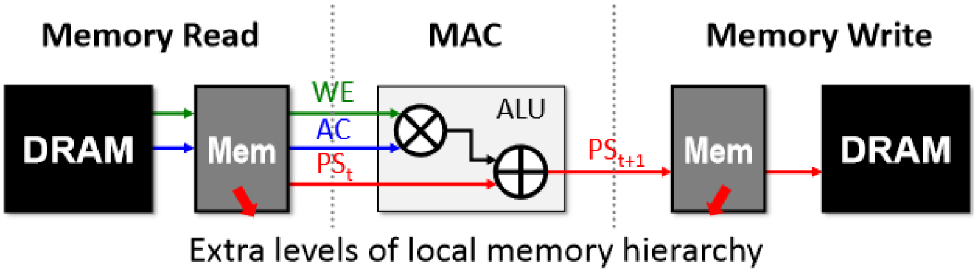


Figure 2.7: A single Multiplication and Accumulation (MAC) operation example. The ALU unit takes one filter weight (WE) and input feature activation (AC), which are in the next step multiplied and summed to the previous partial sum PS_t, creating a new partial sum PS_{t+1}. Input values and an output value are stored in the same memory hierarchy. Therefore, a single MAC operation requires three memory readings and one memory write operation. Taken from [75].

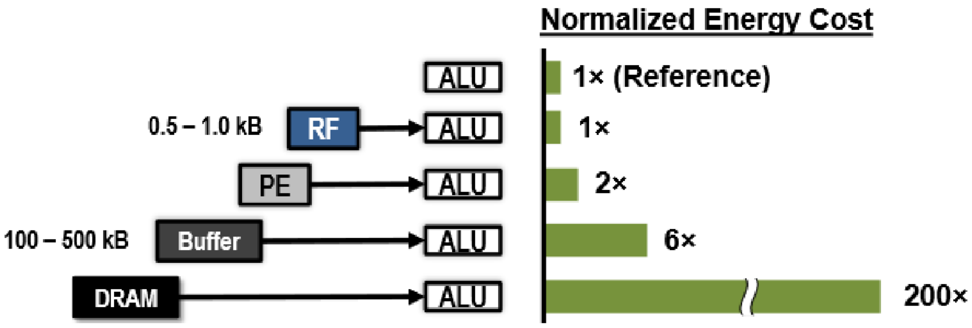


Figure 2.8: An example of the energy cost of MAC used by commercial 65nm processors using RF (Register File), PE (Process Element), buffers and DRAM. The energy cost is normalised. Thus, five is equivalent to 5 operations of MAC. Image was taken from [75], measurements were collected from [12].

2.3 Artificial Neural Network Optimisation

Artificial Neural Networks, from the beginning, were limited by computation resources, for instance, an invention of Percepton [47] in 1943. Even though Machine Learning was known much before, the most significant inventions (LeNet-5 [44]) happened throughout the '90s until the most influential inventions, such as attention-based learning [69] that moved the development of Large Language Models such as GPT-4 [3], Inception Blocks [63] to keep activation maps size in the acceptable range, Residual Blocks [23] to overcome degradation problem, or ReLU [4] with a simpler derivation.

All of the abovementioned approaches improved performance by introducing new concepts into architecture. Nonetheless, additional techniques can also be employed to improve performance, which will be explained in the following sections.

2.3.1 Hardware Optimisation

As mentioned and shown in Figure 2.8, DRAM memory access is expensive, requiring for every MAC three reads from memory for loading weights, activation maps and partial sums [62]. Thus, to optimise energy and speed performance, Field Programmable Gate Arrays

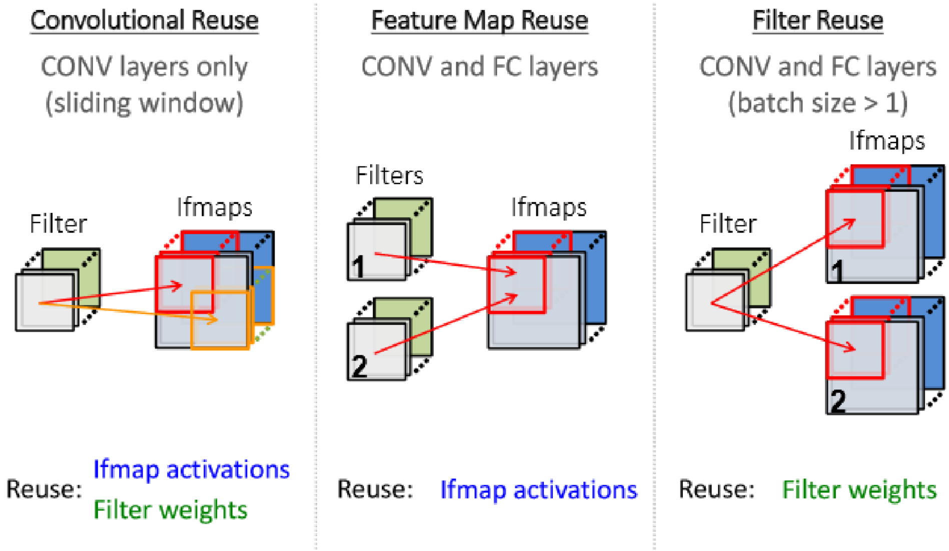


Figure 2.9: Visual demonstration of the three data reuses. Image courtesy: [75].

(FPGA), Application Specific Integration Circuit ASIC and Tensor Processing Units (TPU) are used to offer more optimal hardware configuration. To achieve performance gains, these techniques optimise calculations so that the DRAM is used as little as possible. Instead of DRAM, local registers are preferred. Some of these configurations will be explained and graphically presented in the following sections.

Weight Stationary

One of the configurations is Weight Stationary, which stores weights in RF that are assigned to some PE. The weights stay mostly stationary; meanwhile, input data and accumulated sum changes. After the weights are fetched, $N \cdot E^2$ operations using the same weight are performed. [12, 33]

From a hardware point of view, $R \times R$ filter weights are distributed into PE, which stores them in RF, where they stay stationary. In the next step, each pixel from input feature maps is sent to those PEs, where they are spatially accumulated across PEs. For instance, Weight Stationary can be found in TPU [21] and visual demonstration in Figure 2.10. [12, 33]

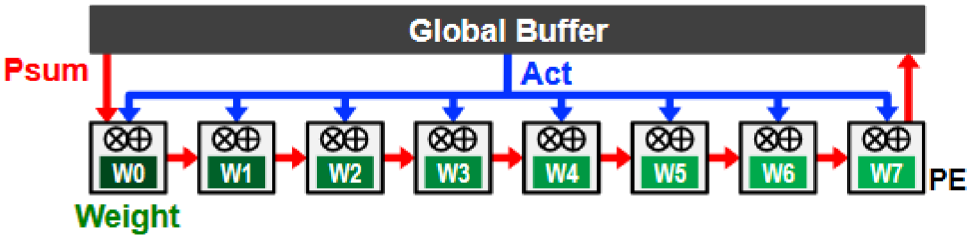


Figure 2.10: Visual demonstration of the Weight Stationary acceleration. Taken from [62].

Output Stationary

Output Stationary aims to reduce memory access by storing accumulated sums into each PE's RF. To ensure stationary property, input feature maps data are streamed into PEs, and the same filter weight is broadcasted to all PEs. This configuration is mainly helpful for convolution reuse. Figure 2.11 shows a visual aid demonstrating the concept. [12, 33]

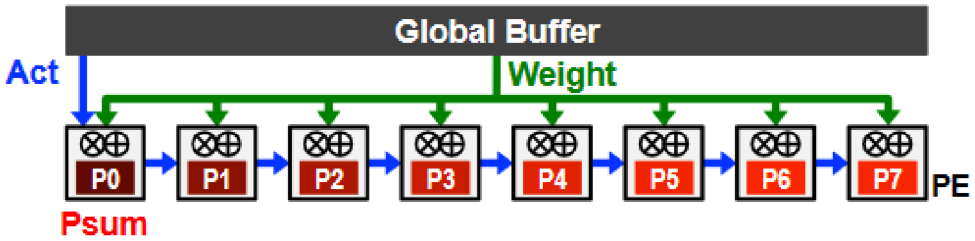
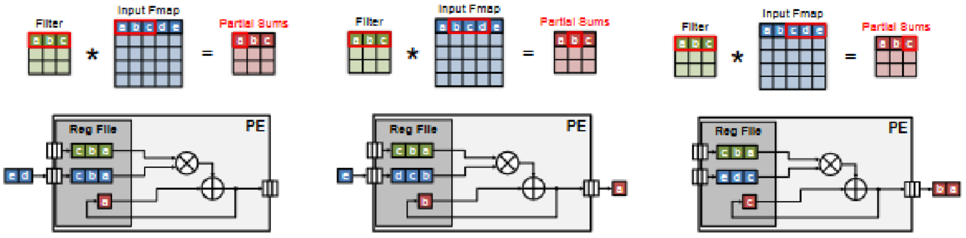


Figure 2.11: Visual demonstration of the Output Stationary acceleration. Taken from [62].

Row stationary

Row Stationary [12] reuses every mentioned data primitive, such as weights, input feature map pixels and accumulated sums. The algorithm assigns the calculation of 1-D convolution to each PE that is later aggregated in case 1-D convolution is not sufficient. The concept of 1-D calculation is visualised in Figure 2.12, and 2-D convolutions can be found in Figure 2.13.



(a) Step 1
(b) Step 2
(c) Step 3

Figure 2.12: Visual demonstration of the 1-D Row Stationary acceleration. Taken from [62].

2.3.2 Pruning

Neural Networks consist of many parameters, not all of which are used for inference, as noted in a study of Gholami et al. [19] and other implications that follow in the following sentences. Therefore, removing them, especially those exhibiting slight sensitivity to the result, is possible. Thus, removing them results in a sparse computational graph, reduced computational costs, and, more importantly, a reduced memory footprint, which is the primary goal of this thesis. Furthermore, pruning is divided into two categories:

1. **Unstructured pruning** [19, 43, 17, 45] – In unstructured pruning, every neuron with small sensitivity is removed, hence leading to large parameter elimination. It

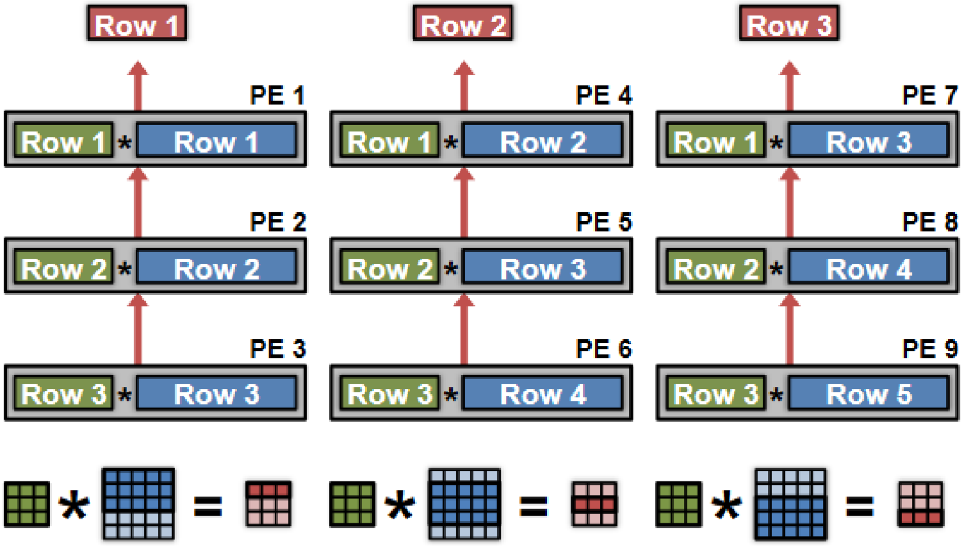


Figure 2.13: Visual demonstration of the 2-D Row Stationary acceleration. Taken from [62].

is considered an aggressive approach that does not significantly change the model's generalisation. However, sparse matrices are created as a result. The issue is that the sparse matrices are difficult to accelerate. Thus, the computation is memory bound [9, 18].

2. **Structured pruning** [19, 25, 29] – Alternative to unstructured pruning is to remove groups of parameters such as weights or even entire convolutional filters. Because of that, the matrices stay dense, and the acceleration issues do not arise. Conversely, an increased level of unstructured pruning does not guarantee a state-of-the-art model performance, which means model loss and accuracy may drop significantly. For more information, refer to [19].

2.3.3 Weight Sharing

Another way to reduce weight and space is through weight sharing, which exploits the fact that weights can be close together, so their mean can replace them. However, to determine the mean, first, weights must be grouped into clusters, as was demonstrated in the work of Coupek [81], who also extended the algorithm to quantisation to grouped weight and further improved weight sharing compression efficiency. Moreover, to improve model accuracy even more, the fine-tuning method was proposed, which looks for optimal parameters that can expand weight dimensions to be better differentiable by clustering algorithms, as cluster centroids outperformed in expanded dimensions.

2.3.4 Knowledge distillation

Considering a trained model with many parameters, the model is used to train the smaller model in this method. The premise builds on the thought that the larger trained model can give useful information in the form of *soft* probabilities that contain more information than *hard* labelled data. However, the major problem is that aggressive compression causes significant accuracy loss. Conversely, it can be reduced by combining the method with pruning and quantisation. [19]

2.3.5 Quantisation

The quantisation is a method of transforming one particular range of values into another arbitrary range while keeping information with minimal loss. Mapping one range value, for example, real values, into integer values has the benefit of more efficient computation utilisation, as integer operations perform better than floating point operations. Moreover, it is also possible to map a more extensive range to a smaller range with the premise of utilizing computation resources even more, leading to more efficient calculations and less demanding energy requirements.

In Neural Networks, quantisation is frequently used to map real *float32* values to integer values with smaller bit bandwidth, for example, *int8*. As a result, the data type gets smaller, and the disadvantages of float arithmetic are not present anymore, which, in the end, improves computation performance and reduces memory footprint. More interestingly, Deep Neural Networks tend to be over-parameterised, which carries more degrees of freedom; thus, the quantisation error is not as significant as it would typically be [19].

Formally defined Neural Network quantisation is a process of finding such quantisation parameters that minimalise empirical risk minimization function, thus retaining model generality, as denoted in Formula 2.5. N is a number of samples, x_i and y_i are the pair of datum and labels, l loss criterion, and θ is the set of learnt model weights.

$$\mathcal{L}(\theta) = \frac{1}{N}\sum_{i=1}^{N} l(x_i, y_i; \theta) \tag{2.5}$$

Furthermore, quantisation can be further divided into uniform or non-uniform quantisation, as seen in Figure 2.14, both involving various trade-offs. In practice, despite the better potential accuracy achievable by non-inform quantisation, uniform quantisation is only used because non-uniform quantisation is complicated to deploy on current CPUs and GPUs while maintaining inference speed. However, the memory footprint might be less demanding [77].

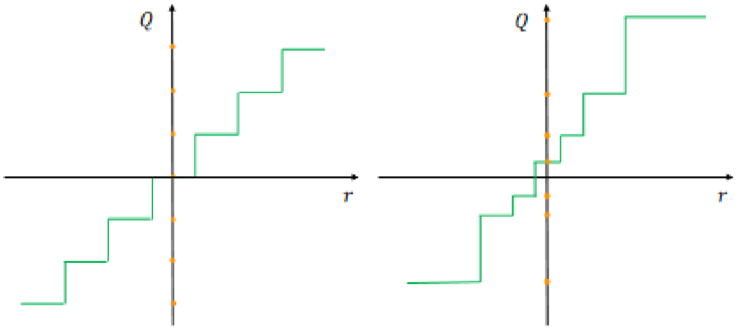


Figure 2.14: Examples of uniform quantisation and non-uniform. On the left is uniform quantisation that maps r real domain values to quantised values Q in *even* increments. On the other hand, non-uniform quantisation maps values in *uneven* increments. Source: [19].

Uniform Quantisation

To uniformly quantise numbers, quantisation operator Q is applied on real value r, as defined in Formula 2.6.

$$Q(s) = \left\lfloor \frac{r}{S} \right\rceil - Z \tag{2.6}$$

Quantisation scale factor S (Formula 2.7) influences how numbers are scaled into quantified form.

$$S = \frac{\beta - \alpha}{2^b - 1} \tag{2.7}$$

Zero point Z defines where zero maps in quantified scale. Consequently, quantisation resolution b determines the number of bits that will be used to encode values from one domain into the quantified domain, restricting the largest possible quantified value. Finally, β and α variables define a *clipping range*, the range from which real domain values will be quantised. If the value is out of the range, the quantisation will not work correctly.

Likewise, by modifying the formula to express r instead of $Q(S)$, backwards conversion can be performed from the quantified domain to the real domain as seen in Formula 2.8.

$$\hat{r} = S \cdot (Q(r) + Z) \tag{2.8}$$

However, the rounding information had already been lost in the quantisation process, so a quantisation error might be present. Nevertheless, the quantisation error is insignificant due to model over-parameterisation, which makes quantisation the most stable and used compression method.

Quantisation requires a proper scale factor and zero point to function optimally. While zero point will be explained later, the process of looking for appropriate scale factor is called *calibration*. Calibration aims to find such β and α that will cover all possible used real values while ensuring that quantisation loss will be as minimal as possible. This can be achieved by selecting proper quantisation resolution bit b; however, due to compression purposes, it should be minimal, too. Therefore, in this case, the aim is to find a clipping range that will not be too wide, which is explained more in Section 2.3.6.

Moreover, properties of the variable Z divide the quantisation further into symmetrical and asymmetrical quantisation. Asymmetrical quantisation poses a tighter clipping range, which becomes essential when used on imbalanced weights or activation functions which are imbalanced, for instance, ReLU, which cannot have negative values [19], or for this thesis, quantisation of energies, that cannot be negative. Conversely, symmetrical quantisation is more widely used because the zero point Z is at zero, as seen in Figure 2.15. Therefore, the computation cost gets cheaper as it can be neglected entirely [72].

2.3.6 Quantisation Granuality

Quantisation on Neural Networks can be performed on multiple levels, and every method has some trade-offs to consider. Those methods are as defined by [19]:

1. **Layerwise** – Quantise all weights in a layer using the same clipping range and scale factor, which may result in a potential clipping range that might be way too broad for some other weights with a narrow clipping range. On the other hand, the algorithm is simple to implement.

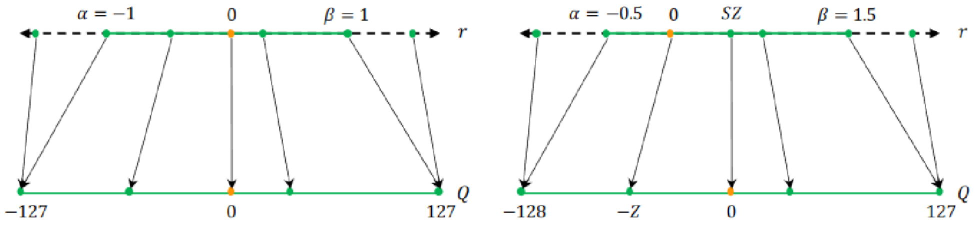


Figure 2.15: Symmetric and assymetric quantisation compression. Symmetrical quantisation has zero point at 0 compared to the asymmetrical variant. Moreover, symmetrical quantisation must be satisfied the following condition $-\alpha = \beta$. Taken from [19].

2. **Groupwise** – Selects a group of filters and channels within a single layer and performs quantisation. This approach proved to be beneficial in Transformers [69] and comes with one drawback, and that is multiple scale factors S.

3. **Channelwise** – Each filter in a channel has its clipping range, whereas every channel has its scale factor. This technique is the most popular because of its high quantisation resolution and accuracy. Figure 2.16 shows visual representation compared to Layerwise quantisation.

4. **Sub-channelwise** – The last technique is a modification of the previous Channelwise quantisation, just with a slight difference. It uses various clipping ranges for different groups, and multiple scale factors must be considered.

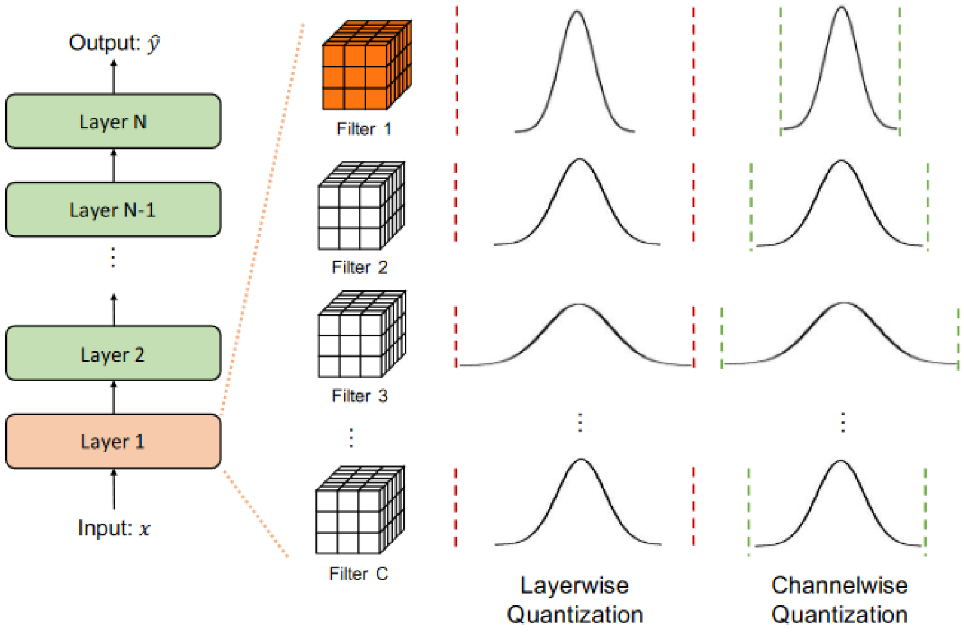


Figure 2.16: Quantisation granularity of Layerwise and Channelwise type. Vertical red and green lines represent the clipping range, which, in this case, Channelwise is more accurate in quantisation. Source: [19].

Nonetheless, all techniques must undergo the *calibration* process to set the most optimal clipping ranges. Calibration can be done either *dynamically* when the range is determined just in time during inference for each combination of filter and its input, resulting in better accuracy at the cost of computation overhead[2]. On the other hand, quantisation can be done ahead of model deployment with *statically* set scalers and zero points, leading to worse accuracy than dynamic quantisation, although with less computation overhead during inference. [19]

However, static quantisation calibration comes with challenges and can be done in numerous ways to minimise accuracy loss to determine α and β variables. The most popular static quantisation method is to calculate the clipping range from a subset of data based on their activations as demonstrated by Jacob et al. [32] and Yao et al. [77] that is used in *Post-Training Quantisation (PTQ)* and *Quantisation-Aware Training (QAT)*, schematically compared in Figure 2.17.

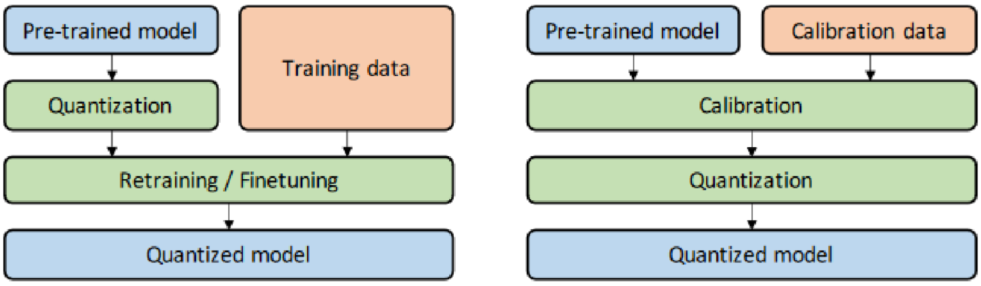


Figure 2.17: Comparision of two static quantisation techniques. Quantisation-Aware Training on the left quantises a subset of training data and then fine-tunes the model. Recalibration is performed in parallel with fine-tuning. Conservely, Post-Training Quantisation performs inference on arbitrary data and collects statistics for quantisation. Based on that, the clipping range and scale factors are determined. Taken from [19].

2.3.7 Quantisation-Aware Training

Weight quantisation can introduce quantisation error due to resolution limitations, causing an undesired noise in original weights. Consequently, the model suffers accuracy loss and deviates from the initial converged state. Therefore, to eliminate noise, the model undergoes the training process on the training dataset, described in the sections below, to reinstate part of lost accuracy, which is perpetuated in Figure 2.18. [19]

Forward Pass

It is necessary to note that the training process uses real and quantified domain values. At first, integer weights are dequantised to float values using static quantisation parameters so weights can be forwarded to the Neural Network [19]. After that, an inference is performed, and the error gradient is calculated in floating-point as the quantified version would not allow for calculation gradient as Stochastic Gradient Descent (SGD) [55] explores space in small and noisy steps, which are eventually statistically averaged [30]. Hence, it implies higher precision is needed, as described by several researchers in [30, 5, 78], and also according to similarly reported results by Gysel et al. [22].

[2] the algorithm calculates several statistical properties such as minimum, maximum, percentiles, etc.

Backward Pass

In the subsequent phase, the gradient must be back-propagated to the initial weights. However, the quantisation function quantised those weights initially, which must be reflected in the error function. The problem is that the quantisation function is not continuous. Therefore, the round operator is not differentiable. To overcome this issue Straight Through Estimator (STE) [7] operator is employed instead (Formula 2.18) that ignores the rounding operator altogether [19]. Despite not modelling round operators, it performs very well in practice, excluding cases when lower resolutions, especially binary quantisation, are used [5]. Furthermore, apart from STE, multiple other approaches were proposed in other studies, as mentioned in [19]. Finally, as in standard SGD, weights are updated to new values and are prepared for the next iteration.

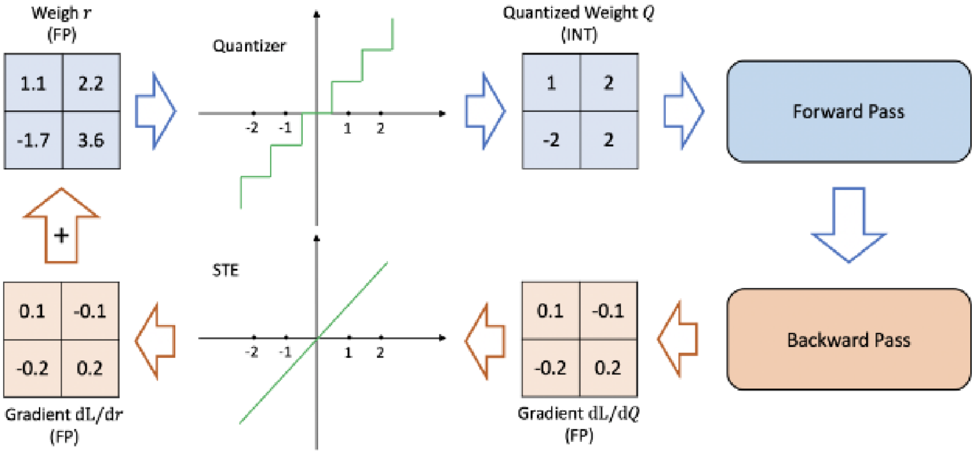


Figure 2.18: A single iteration of Stochastic Gradient Descent on quantised weights that are dequantised during the process to allow SGD to minimise loss function $\mathcal{L}(\theta)$. For more detailed workflow, refer to Section 2.3.7. Taken from [19].

Implications

Quantisation-Aware Training excels in circumstances when minimal accuracy loss is preferred. However, it requires fine-tuning, which might take several epochs to converge again at acceptable accuracy and loss, thus resulting in a higher initial investment. If the model exhibits frequent alterations, it raises doubt whether employing Post-Training Quantisation might not be more advantageous, as it circumvents the need for prolonged training, thereby accelerating model deployment.

2.3.8 Post-Training Quantisation

Post-Training quantisation is a much simpler alternative to QAT. It does not undergo a fine-tuning process and exhibits lower overhead. Moreover, it does not require much data to calibrate, and even the data do not have to be labelled because, during quantisation activation statistics are collected to determine scale factor and resolution, for instance, as shown in Figure 2.19. However, these advantages come at the cost of lesser accuracy, which, on the other hand, opens many opportunities for improvement. [19]

For instance, Banner et al. [6] proposed a novel technique of Analytical Clipping for Integer Quantisation (ACIQ) that analytically computes clipping range and channel-wise bit-width according to activations within tensor, reducing the rounding error instead of using input tensor at the cost of potential distortion of the input tensor. Furthermore, they perform bias correction according to monitored weights, means and variances. However, ACIQ reaches low accuracy losses on channel-wise quantisation; it is inherently hard to deploy on hardware [19].

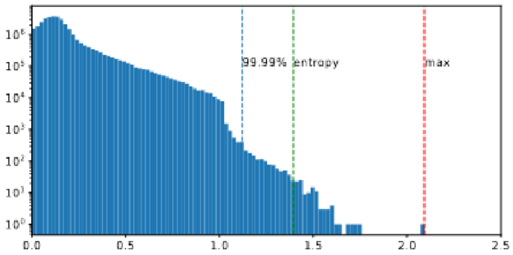


Figure 2.19: A distribution of activations in layer 3 in ResNet50 [23]. Three statistics are used to determine the clipping range: max, entropy and 99.99% percentile. Taken from [72].

2.4 LeNet-5

LeNet-5 is a convolution network proposed by LeCun et al. [44] in 1995 that superseded LeNet-1 [41] as this network was unsuitable for learning from the newly created dataset MNIST [39]. Because of that, two networks were prototyped, LeNet-4 and LeNet-5, in which LeNet-5 turned out to be superior with hyperparameter configuration as shown in Table 2.1. The main difference is that LeNet-5 has more feature maps and a more extensive fully connected layer and uses the distributed representation to encode classes.

Furthermore, with the creation of LeNet-5 also comes the database MNIST that is used in this thesis. It is a database of handwritten digit images split into 60000 labelled train samples and 10000 labelled validation samples, with the largest digit having a size of 20×20 pixels centred around 28×28 pixels large field. Despite that, LeNet-5 resizes input images to 32×32 pixels to capture distinctive features of end strokes or corners. Moreover, the input pixels are normalised, so background white pixels are represented as -0.1 and black digit pixels as 1.175, resulting in mean equal zero and variance one, thus improving learning [38]. More detailed information can be found in [42].

Layer	**Output**	**Feature Maps**	**Kernel Size**	**Stride**	**Activation**
Input	32×32	1	–	–	–
Convolutional	28×28	6	5×5	1	TanH
Avg. Pooling	14×14	6	2×2	2	–
Convolutional	10×10	16	5×5	1	TanH
Avg. Pooling	5×5	16	2×2	2	–
Convolutional	1×1	120	5×5	1	TanH
Fully Connected	84	–	–	–	TanH
Fully Connected	10	–	–	–	Softmax

Table 2.1: Hyperparameter Configuration for LeNet-5

Nowadays, as LeNet-5 replaced LeNet-1, the LeNet-5 has been replaced by other modern architectures. Nevertheless, it served as a baseline for CNNs, and it is still used for test purposes because of the fast training times, even though it used to take twenty days to train [44]. However, this thesis used a slightly modified version with new modern discoveries.

2.4.1 QMNIST Dataset

QMNIST [74] was developed to address the partial loss of the original pre-processing procedure used to create the MNIST [39] dataset. While the original MNIST testing set had 60,000 test images, only 10,000 of them are available nowadays. To address this issue, Yadav et al. [74] created a new dataset sampled from the NIST Special Database 19 [20], striving to reproduce the former MNIST dataset following the creation procedure as closely as possible.

For the purposes of this thesis, especially in terms of evaluating compression techniques, testing on multiple datasets is advantageous. The QMNIST dataset offers 50,000 new test samples that were not used for training. Furthermore, the dataset is well integrated into PyTorch [53] library and attainable when accessing dataset split *test50k*. Moreover, the NIST dataset (split *nist*) dataset is also available providing even more data.

Historically, even training a simple model like LeNet was time-consuming, but with modern computational resources, it is now feasible to evaluate the entire NIST dataset. With more data, the weight compression evaluation will be even more robust.

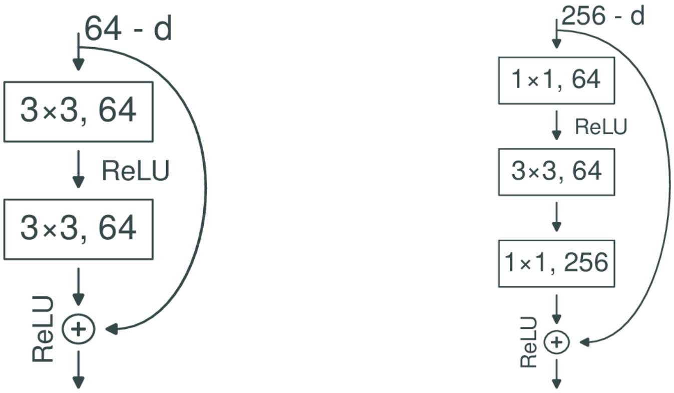
2.5 MobileNet

MobileNet [57, 28] is a neural network designed to maximise energy efficiency while still maintaining acceptable accuracy levels. Energy efficiency is primarily achieved through the use of Depth Separable Convolution. However, the most influential savings stem from Inverted Residual Blocks.

With the release of the MobileNetV2 new Inverted Residual Blocks were proposed. Before going into more detail, it is important to first introduce Residual Blocks. Following the success of VGGNet [58] in ILSVRC [56] competition of 2014, very deep neural networks emerged as a popular architecture choice. However, soon enough, the deep architecture was shown to be sensitive to vanishing gradient and to the *degradation* problem [23]. At the time, it was anticipated the deeper the model, the better accuracy it could achieve. Contrary to this expectation, deeper networks began to exhibit diminishing performance when stacking layers, indicating that simply adding more layers still was not adequate.

To address this problem shortcut connections among layers were proposed. These shortcut connections can take multiple forms, as experimented with by Kaiming et al. [24], who found that simple shortcut connections outperformed other tested configurations. However, it must be noted the shortcut connections appeared earlier, for instance, ResNet [58], which won the ILSVRC competition. This triumph highlighted how shortcut connections have the potential to enable the building of deeper models. ResNet used Residual Blocks and Bottlenecks schematically demonstrated in Figure 2.20. These blocks scale channel size using the first layer 1×1 convolutions, the second layer performs 3×3 convolution, and the third layer applies 1×1 convolutions again to restore channel size. In the case of the ResNet and Residual Blocks overall, input channels are downscaled.

Inverted Residual Blocks [57] adopted an approach opposite to the “shrinking” by expanding the input activation map, as shown in Figure 2.21. The idea builds on the principle



(a) Residual Block

(b) Bottleneck

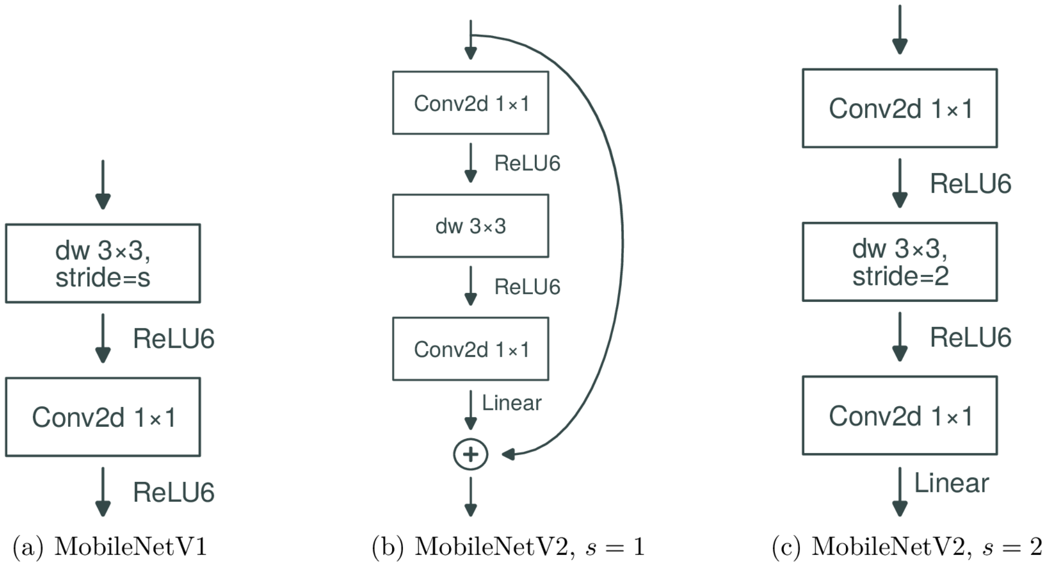
Figure 2.20: On the left is a schematic example of a Residual Block, which shortcuts two convolution layers with an identity function. On the right is a schematic for the Bottleneck block, which first scales the activation map, then convolves it, and finally reshapes the activation map to its original shape. The Bottleneck does not necessarily have a shortcut connection. The figure was redrawn from [23].

that Inverted Residual Bottlenecks sacrifice some learning capabilities for parameter reduction. Furthermore, to decrease the number of trainable parameters, depthwise separable convolution is utilised, reducing model complexity and total size. These blocks are combined with various techniques such as batch normalisation, dropouts, convolution layers and classification layers to form complete architecture, which can be further examined in Table 2.2.

2.5.1 ImageNet Dataset

ImageNet [56] is a large database of images used to evaluate competing models in the ILSVRC computer vision competition. For that reason, the test dataset split consisting of 100,000 samples is not publicly available. The available dataset splits are: train split with 1,281,167 samples and validation split including 50,000 samples.

The dataset is divided into 1,000 classes which need to be correctly classified. However, this classification task is challenging. Therefore, in addition to model accuracy and loss, Top-5 accuracy is also assessed. Top-5 accuracy is a metric that evaluates how successful the model is in predicting data by tolerating minor misclassifications. It is calculated by by considering the five most likely classes, and if the target class is among these five, then the classification is assessed to be successful.



(a) MobileNetV1 (b) MobileNetV2, $s = 1$ (c) MobileNetV2, $s = 2$

Figure 2.21: The primary blocks used in MobileNet models. On the left is a schematic for a depthwise separable convolution block. To the right of that, Bottlenecks are shown with various strides, and in the case of $s = 1$, it also contains a residual connection. For completeness, *dw* stands for depthwise convolution. The figure was redrawn from [23].

Operator	Input (w, h, c_i)	Output (w, h, c)	t	s	n
Conv2D	$224 \times 224 \times 3$	$112 \times 112 \times 32$	-	2	1
Bottleneck	$112 \times 112 \times 32$	$112 \times 112 \times 16$	1	1	1
Bottleneck	$112 \times 112 \times 16$	$56 \times 56 \times 24$	6	2	2
Bottleneck	$56 \times 56 \times 24$	$28 \times 28 \times 32$	6	2	3
Bottleneck	$28 \times 28 \times 32$	$14 \times 14 \times 64$	6	2	4
Bottleneck	$14 \times 14 \times 64$	$14 \times 14 \times 96$	6	1	3
Bottleneck	$14 \times 14 \times 96$	$7 \times 7 \times 160$	6	2	3
Bottleneck	$7 \times 7 \times 160$	$7 \times 7 \times 320$	6	1	1
Conv2D 1×1	$7 \times 7 \times 320$	$7 \times 7 \times 1280$	-	1	1
AvgPool	$7 \times 7 \times 1280$	$1 \times 1 \times 1280$	-	-	-
Flatten	$1 \times 1 \times 1280$	1280	-	-	-
Linear	1280	1000	-	-	-

Table 2.2: Tabular overview of the MobileNetV2 architecture as defined in the PyTorch [53] implementation. In the original implementation, the last layer is Conv2D $1 \times 1 \times 1280$ with an output size of 1000. Moreover, w stands for tensor width, h for tensor height, and c_i for input channels count. Likewise, the same notations are used for output, with one difference: c denotes the number of output channels. Lastly, parameter t is an expansion factor responsible for expanding input channels, s is the convolution stride, and n is the number of repetitions of a given layer.

Chapter 3

Evolutionary and Genetic Algorithms

Evolutionary and Genetic algorithms belong to the metaheuristic searching class of optimisation algorithms. These algorithms are inspired by nature and biological evolution. The entire process is based on the biological evolution of species, selecting the best population specimens (referred to as chromosomes in Evolutionary algorithms) to create a new generation while discarding less fit members of the population. The complete evolutionary cycle will be thoroughly explained in Section 3.1.

The primary goal of optimisation algorithms is to search through the search space to find the most optimal solution. Throughout history, multiple types of algorithms with different concepts emerged. For instance, the Random Search algorithm, solely based on randomly traversing the search area, is one of the simplest algorithms. Similarly, the Hill Climbing algorithm exhibited better performance; nonetheless, the found solution was not optimal because the algorithm struggled to leave local extremes. As these algorithms were not sufficiently effective for some problems, more sophisticated algorithms emerged. Especially for this thesis, the Cartesian Genetic Programming (CGP) was employed to solve the weight compression task, explained in Section 3.2. Moreover, the CGP phenotype function will be described in the section so it can be used to interpret experiments.

3.1 Evolution Cycle

Evolutionary algorithms utilise the concept of biological evolution to find optimal solutions by evaluating them based on their fitness. Fitness determines the quality of a given candidate solution encoded in a chromosome, which is also a term in evolutionary biology. The chromosome contains different gene combinations that store parts of the encoded candidate solution. Hence, a gene is the smallest part of the evolution cycle, carrying information; otherwise, it is called an allele. In summary, the resulting compiled solution is called a phenotype, compiled from information contained within a chromosome with alleles.

Eventually, the evolution cycle will find the optimal phenotype when fitness evaluation and chromosome encoding are appropriately done. To utilise algorithms properly, it is necessary to be able to encode phenotype into a chromosome, usually in the format of binary representation, vector, tree, graph, etc. Equally crucial to encoding is the ability to calculate fitness value. After that, the evolution process can begin, which is further explained in the upcoming sections and introduced in Figure 3.1:

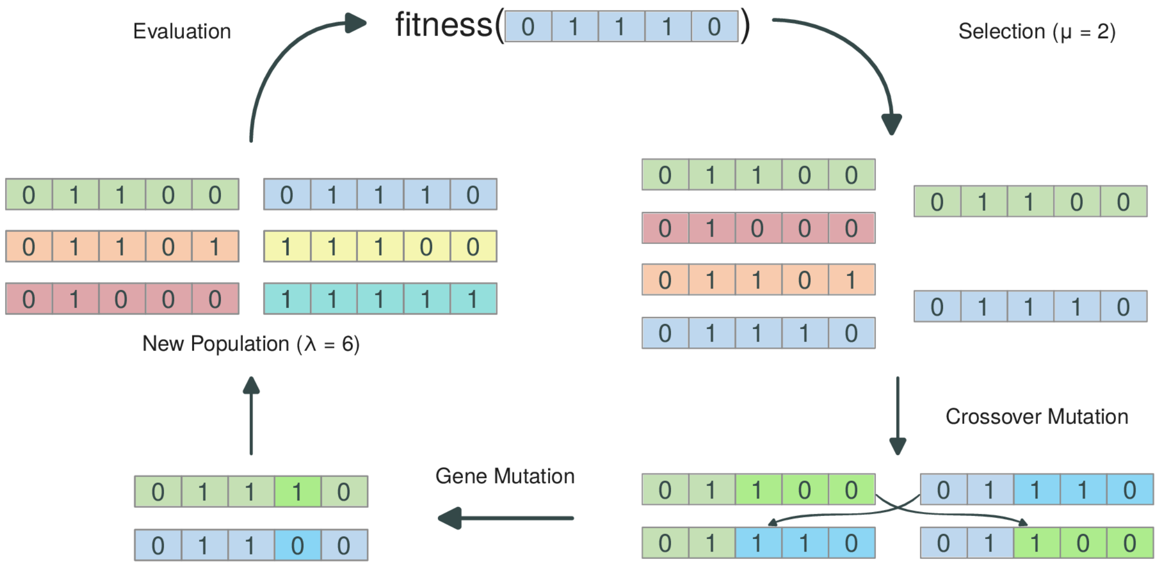


Figure 3.1: A generic evolution cycle for Evolutionary algorithms. At the beginning of the chosen algorithm, the initial population must be created according to the preferred strategy. Following that, a fitness function is used to evaluate the fitness of every chromosome, which is further used in the selection process to select the fittest chromosomes. After the selection process, a new population will be created, combining the fittest chromosomes and performing mutation operators on chromosomes. Eventually, a brand new population is restored, which can be evaluated, and if it contains a satisfying candidate solution, the process can be successfully stopped.

3.1.1 Initial Population

In the beginning, an initial population must be created. It can be initialised from a pre-set population or, more commonly, a randomly generated population. The first approach is beneficial when existing chromosomes are utilised to resume evolution from arbitrary saved checkpoints. The latter is applicable when there are no evolved chromosomes, and the whole process starts from scratch.

3.1.2 Fitness Evaluation

In every cycle of evolution, each chromosome is evaluated and assigned a fitness value describing the quality of the candidate solution. In addition, the fitness value must be calculated at least from a single parameter; however, it can be derived from multiple parameters, known as multi-objective optimisation. Therefore, the values do not have to be necessary of scalar type since some problems might require to optimise various parameters, or at least be able to find trade-offs between parameters, often visually determined from Pareto fronts, for example, showcased in Figure 3.2, or decided by prioritising particular parameters over others, for instance, the accuracy of an electrical circuit over several digital gates. Alternatively, multi-objective can be done by aggregating parameters into a single fitness value applying weighted sum, for reference Formula 3.1.

$$f'(x) = \sum_{i=1}^{N} (w_i \cdot f(x)) \tag{3.1}$$

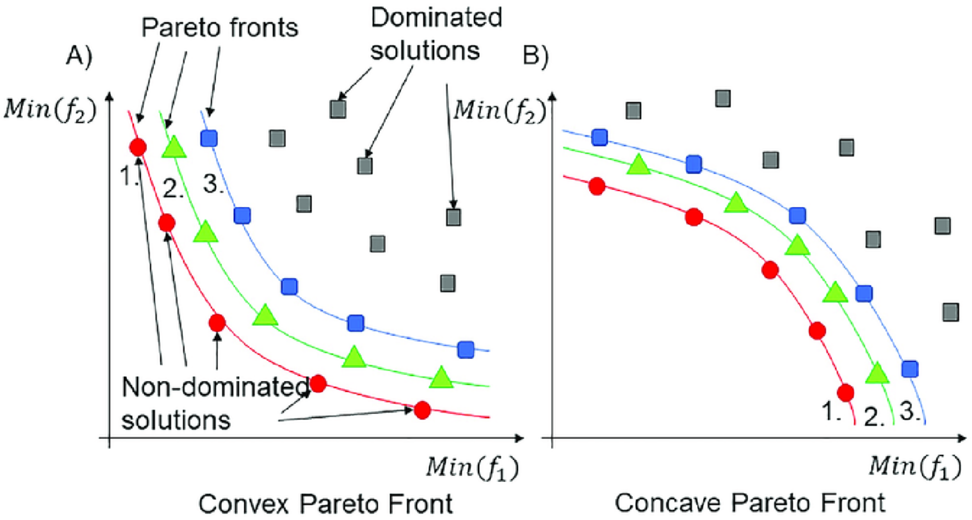


Figure 3.2: An example of three Pareto fronts marked by coloured lines. Displayed solutions with marks outperform other solutions in either parameter f_1 or f_2. Furthermore, an example of convex and concave Pareto fronts is shown, as aggregated fitness methods can only be used on convex-shaped Pareto fronts. The image was authored by Abonyi et al. [2].

On the other hand, the problem with aggregated methods is that, compared to the Pareto front, only one solution can be found. Moreover, the optimal aggregate function has to be discovered with a good combination of weights. Finally, the aggregated approach does not work for problems that would result in non-convex Pareto fronts.

3.1.3 Chromosome Selection

After the fitness evaluation, the next step in Evolutionary algorithms is to create a new population derived from the best chromosomes. For this task, the best chromosomes are used to evolve a new generation of chromosomes, possibly carrying the most good properties, with a chance of becoming better or even worse than parent chromosomes. Multiple strategies have been researched throughout history to achieve the most optimal offspring chromosomes, and these will be described in the following sub-sections.

Roulette Selection

Roulette is a statistical method of choosing fit chromosomes to establish a new chromosome population. The whole principle is based on assigning a probability of selection determined by chromosome fitness. The higher the fitness, the higher the likelihood it will be selected for re-creation. Each possible unique fitness value is summed together to infer the probability of every chromosome. Consequently, the final probability is calculated using the Formula 3.2.

$$f'(x) = \frac{f(x)}{\sum_{i=1}^{N} f(x_i)} \tag{3.2}$$

Tournament Selection

In the tournament elimination method, chromosomes are randomly compared among each other, stemming from their fitness values. The number of competing chromosomes is given by parameter K, usually set to two, resulting in pair comparisons. Compared to the roulette method, chromosomes with lower fitness values do not have a chance to be selected for recreation. Thus, the fittest chromosomes are always selected.

Deterministic Selection

The deterministic method selects K chromosomes with the highest value. This method is particularly interesting for this thesis, as it was utilised as the selection method for the Cartesian Genetic Programming algorithm in Section 3.2.

3.1.4 New Population

The final step in evolution is to create a new population of potentially better chromosomes. This step selects the best chromosomes for evolutionary operators such as mutation or crossover, explained in the following sections.

Mutation Operator

The mutation is the slightest change in chromosomes used in Evolutionary algorithms. Changes are done stochastic, either randomly selecting the number of genes to be mutated within the chromosome and eventually performing gene mutation on the number of genes or iterating over genes and mutating it according to set probability. In the first case, it is called a point mutation; in the second case, it is a probability mutation. An example of mutation can be found in Figure 3.1, or more thoroughly explained in the Cartesian Genetic Programming Section 3.2 as explicitly employed for thesis implementation.

Crossover Operator

The last operator in this thesis is the Crossover operator, conceptually grounded on combining two chromosomes to create two new chromosomes. Gene exchange is chosen entirely on a crossover point or two crossover points in case a 2-point crossover is performed. The most used crossover techniques are the following:

1. **Single Point Crossover** – A single point is randomly chosen, and points to the right are exchanged between chromosomes.
2. **2-point Crossover** – Two points are selected, marking an exchange zone in between. The in-between zone is exchanged.
3. **Uniform** – Every offspring chromosome is randomly assigned a parent gene to receive.

3.2 Cartesian Genetic Programming

Cartesian Genetic Programming (CGP) [50, 48] is a variant of Genetic Programming (GP), a computational optimisation approach widely used for solving complex problems. In tree-based GP [34, 54], solutions are represented as computer programs, and the evolutionary process seeks to evolve these programs to achieve optimal performance. Initially, candidate solutions are randomly generated, but more effective programs are explored through iterative evolution. CGP, in particular, employs a graph-based representation, enabling node resue and multi-variable output [51]. Furthermore, it is capable of solution space exploration while producing fewer bloat[1] [64] compared to other tree-based GP counterparts. Moreover, the resulting genotype can contain non-coding genes because of the graph structure. However, those inactive nodes are discarded as the phenotype is constructed by iteratively traversing the graph backwards from output nodes. Although those nodes are not used in phenotype, they are essential in helping evolution find more optimal candidate solutions [64].

The Cartesian Genetic Programming is especially useful in performing symbolic regression for functions with multi-variable output or circuit electrical design as demonstrated by Miller et al. [48]. Additionally, CGP excels at circuit optimisation as well; for instance, a couple of studies [80, 52] employed formal verification techniques to optimise a more complex arithmetic circuit based on a pioneered idea by Vasicek and Sekanina [67] utilising SAT verification to check for correctness when evaluating fitness.

As this thesis aims to research energy-effective circuits, tree-based GP is unsuitable for such tasks due to high bloat. To get more acquainted with Cartesian Genetic programming, this chapter is organised into sections clarifying evolution concepts introduced in Section 3.1 into more detail; however, now only focusing on the CGP. Hence, chromosome encoding, fitness evaluation, and lastly mutation operators will be explained in this section.

3.2.1 Genotype Encoding

The algorithm works with the graph structure that reassembles $r \times c$ grid of nodes when searching for candidate solutions, in which a single candidate solution is encoded into a chromosome[2] formed out of genes. Nonetheless, several parameters must first be defined to demonstrate better what the chromosome looks like, its rules, and potential implications. A visual demonstration can be seen in Figure 3.3.

- **Row** – A row in the graph, the maximum value defined by parameter $\mathbf{n_r}$.
- **Column** – A column in the graph, the highest value limited by parameter $\mathbf{n_c}$.
- **Function** – A function $f : X \to Y$, where X and Y might have different dimensions as defined by $\mathbf{n_i}$ for input arity and $\mathbf{n_o}$ for output arity. The set of these functions is denoted as $\mathbf{\Gamma}$.
- **Input Node** – Special case of the node that performs no function calculation. Input quantity is specified by $\mathbf{p_i}$.
- **Output Node** – Special case of the node that performs no function calculation. Output quantity is specified by $\mathbf{p_o}$.
- **Node** – A node in the CGP graph, accepting at least one input and outputting minimally one value calculated by the function. The function does not have to be computed with every parameter. However, it must assign values to all outputs.

[1] growth of the tree without significant fitness improvement

[2] synonym to genotype in GP algorithms

- **Edge** – A connection between every node type. The CGP limits how many columns can be skipped or reused from previous columns, as defined by the L parameter, often called *Look-Back*.

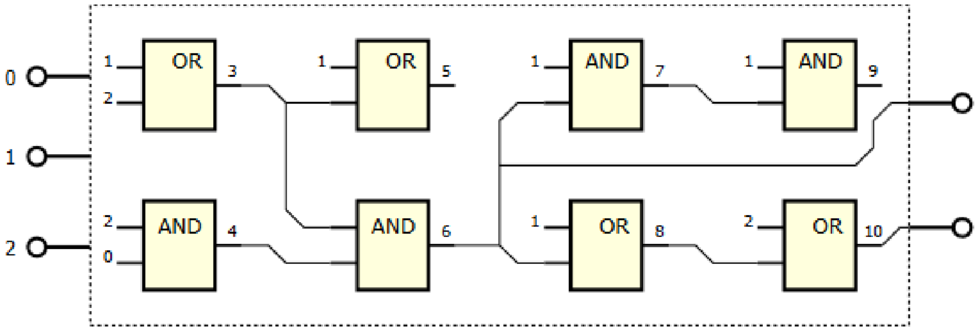


Figure 3.3: The graph structure of the CGP algorithm candidate graph solution with the following parameters: $n_r = 2,\ n_c = 4,\ l = 3,\ n_i = 2,\ n_o = 1,\ p_i = 3,\ p_o = 2,\ \Gamma = \{AND\ \ (0),\ \ OR\ \ (1)\}$. The image was authored by Vasicek et al. [66].

With all of the relevant terminology, it is an ideal moment to define the chromosome encoding for the Cartesian Genetic Programming (CGP) algorithm. Given that chromosomes should be as small as possible yet be able to describe a graph, it is crucial to define how nodes are connected by specifying what input pin is connected to what node. After that, each node performs a function; therefore, that information must also be present. This information must be repeated for $r \times n$ times. Finally, output nodes O remain (Formula 3.3), and those must specify the node they are connected to. Consequently, the chromosome encoding definition R (Formula 3.4) and its size (Formula 3.5) can be calculated in the following way:

$$O = \{(x_{p_1}, x_{p_2}, ..., x_{p_o})\} \tag{3.3}$$

$$R = \{(x_{a,b,0}, x_{a,b,1}, ..., x_{a,b,n_i}, f) \mid \forall a, b \in \langle 1, n_r\rangle \times \langle 1, n_c\rangle \wedge f \in \Gamma\} \cup O \tag{3.4}$$

$$|R| = n_c \cdot n_r \cdot (n_i + 1) + p_o \tag{3.5}$$

3.2.2 Fitness Evaluation

Fitness in circuit design is usually evaluated using a multi-objective prioritisation technique. The most essential fitness parameters are accuracy, energy, the number of used nodes or digital circuit delays. Accuracy can be calculated using multiple methods depending on the type of approximated function. For instance, the digital circuit can use Hemingway Distance, which calculates how many bits are different to reference the result. For circuit optimisation, the SAT solver may be employed [80, 67, 52] to verify the equality of solution to the reference solution.

Additionally, errors such as those demonstrated by Mrazek et al. [52] can be used as chromosome fitness. Those include *ER*, *MAE*, *WCE*, *MSE*, *MRE*, and others can be used, however for this thesis was *Mean Squared Error* (MSE) used as an error metric. *Mean Squared Error* (MSE) in Formula 3.6, calculates the difference between reference and approximated values. However, compared to Hemingway Distance, the method works for scalar values and is especially good for accompanying outlier values because the error is squared, highlighting errors. This property is especially significant to achieve the most even distribution of errors in convolution filters.

$$\text{MSE} = \frac{1}{N}\sum_{i=1}^{N}(y_i - \hat{y}_i)^2 \tag{3.6}$$

3.2.3 Selection Process

Because when working with the CGP algorithm, it is not common to use the Crossover operators [31]; the selection process is purely done deterministically. Thus, an evolution strategy $\mu + \lambda$ ($\mu = 1$), a specific form of elitism[3], is applied to the selection process, which means that only the best chromosome is used for recreation, and as a result, creates λ offsprings. If two chromosomes share the same fitness value, a chromosome not used to create a new population is prioritised instead of the original parent chromosome. This effect is known as *neutrality* and was proven beneficial in research conducted by Vassilev and Miller [68]. The effect can be seen in Figure 3.4.

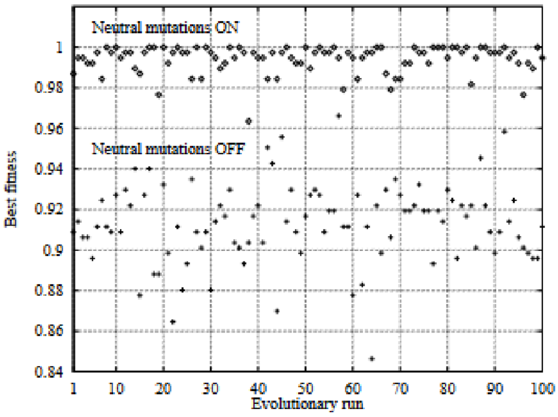


Figure 3.4: An effect of neutral mutations on a search for candidate solutions as researched by Vassilev and Miller [68]. The graph was taken from their study.

3.2.4 New Population

As previously mentioned, the Crossover operator does not work well with the CGP. Hence, the point mutation operator is responsible for creating a new collection of chromosomes. Mutation can be done in multiple ways, though point mutation for this thesis follows these two essential concepts. First, an introduced gene mutation limit dictates how many genes can be mutated. In every mutation evolution cycle, a random number n is generated from the range zero to the limit. Later, depending on the random number n, n genes are mutated.

Regarding gene mutation, it is essential to note that the amount of possibly mutated genes is very significant, with opportunities to do experiments. The lower the quantity is, exploitation is more prevalent than exploration. The exploitation effect tends to converge more slowly, trying to improve the best chromosome with little modifications. Although, it is useful when searching for the most optimal phenotype. Oppositely, exploration results in

[3] retaining the best chromosomes in population

trying out different combinations and reaching decent candidate solutions faster than the opposite approach. However, it is less likely to find the most optimal candidate solution.

The impact of mutation depends on what part of a chromosome is being mutated. In the CGP, the mutation can lead to changes in a node function, rewired inputs for the node, or rewiring of outputs. However, a particular constraint, such as the L parameter, prevents nodes from further layers set by the parameter to be connected to nodes. Moreover, every mutation must be legal. Therefore, a node input cannot be connected to nodes in the following layers.

In conclusion, finding the most suitable trade-off between exploitation and exploration is essential when considering gene mutation. However, this part of the algorithm design is mainly experimental work.

3.2.5 Evolved Program Performance

As defined in Section 2.2.1, the Multiplication And Accumulation operation is a standard unit to count the quantity of operations. Because the CGP algorithm utilises a graph structure in which one node represents a single function with a fixed number of inputs and outputs. Although the CGP function can have an arbitrary number of MAC units, keeping each function as one MAC operation or one Floating Point Operation (FLOP) for weight compression and possible hardware integration is better. Alternatively, one FLOP is equivalent to one-half of the MAC.

It is also important to note that a constructed phenotype does not have to necessarily use every bit of information from the found genotype due to unused nodes. Unused nodes are helpful when searching for candidate solutions, as they can contribute to finding better configurations. Therefore, these nodes can be omitted from the genotype to construct an even more optimal phenotype.

Finally, when an optimised phenotype is constructed, performance can be calculated by summing together functional nodes and their MAC units. After that, given the function can be synthesised into a hardware version supporting PE and RF, energy utilisation is analysed according to measurement in Figure 2.8.

Chapter 4

Algorithm Design

As previously mentioned, conventional memory access is an **expensive operation** consuming a non-negligible amount of **energy** [12]. To mitigate memory access as much as possible, neural network weight compression is a heavily researched field. Despite of that, the field still possesses many undiscovered open challenges, creating many diverse opportunities. For instance, the most significant discoveries, such as quantisation, hardware optimisation, pruning, weight sharing and knowledge distillation, have been mentioned in Section 2.3. Nevertheless, one of them employed interesting concepts such as weight and activation map reuse on a hardware level to avoid unnecessary memory, which can be summarised into a single research question:

> Is it possible to avoid conventional memory access?

Che et al. [12] used energy-efficient local registers to store accumulated convolution sums and activation maps. However, convolution weights must still be fetched from the buffer or, in the worst case, from RAM which extends the previous research question:

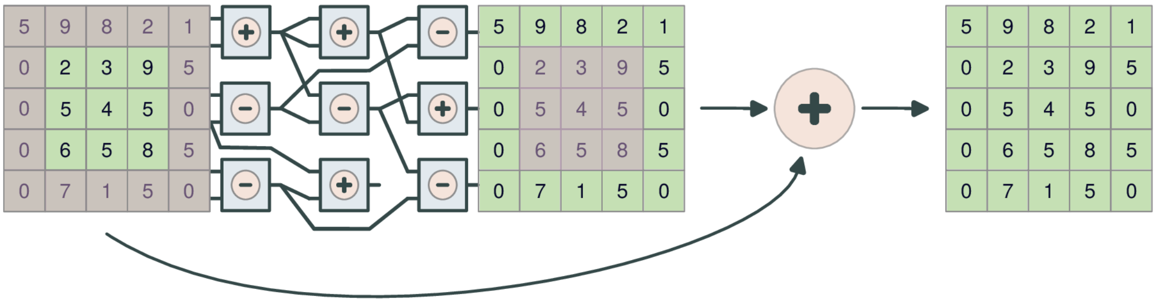
> Can weight inference substitute memory access?

In circumstances when the weight compression function would be more energy-efficient than conventional memory access or SRAM buffers, it could be a viable solution. However, the viability of the solution does not entirely rely on energy consumption, as it is not the only pivotal parameter. Another equally important attribute is delay, which can be a limiting factor in some real-time applications.

To address all the mentioned circumstances to decrease memory use and thus decrease energy consumption as much as possible, an automatic convolution weight compression algorithm was prototyped. The algorithm aims to find such electrical circuit configurations that would infer a portion of convolution weights from the memory-fetched weights or potentially from previously used weights which can be seen in Figure 4.1. Due to the nature of the task, the Cartesian Genetic Programming algorithm was employed to evolve circuits since, compared to deep learning approaches, it does not require multiplication[1]. Additionally, CGP tends to generate small phenotype without bloat [64], which goes well with the goal of reducing energy in evolved circuits.

Nonetheless, the CGP algorithm and its configuration cannot be universally applied to every problem, as explained by the *No Free Lunch Theorem* [71]. Consequently, this chapter will be dedicated to detailing the CGP setup, covering aspects such as chromosome encoding, population generation, fitness evaluation, selection, and mutation.

[1] the most energy-intensive operation



5	9	8	2	1
0	2	3	9	5
0	5	4	5	0
0	6	5	8	5
0	7	1	5	0

5	9	8	2	1
0	2	3	9	5
0	5	4	5	0
0	6	5	8	5
0	7	1	5	0

5	9	8	2	1
0	2	3	9	5
0	5	4	5	0
0	6	5	8	5
0	7	1	5	0

Figure 4.1: An example of how the algorithm could operate upon convolution weights. On the left, the nine core weights are fetched from the memory to the remaining outer weights that are eventually merged into the original filter.

4.1 Chromosome Encoding

The chromosome, for its functionality, needs to know how nodes are connected to each other. The relation can be trivially encoded in an array of number values where first $n_r \times n_c$ values contain a connected input pin to which output pin and finally what function the node performs. After the node connection relation, an output sequence follows, which is not compatible with the two-dimensional representation of filters. Therefore, the output vector was flattened the same way as the input vector was as well, resulting in flat vectors, which is visually demonstrated in Figure 4.2. The only drawback of this flattened solution is it must be reconstructed into a two-dimensional filter again, which will be described in more detail in the implementation Chapter 5.

4.2 Population

In an aspect of population management and generation, a standard evolution strategy $\mu + \lambda$ was picked, specifically with $\mu = 1$ which is often employed in evolutionary circuit design. The strategy retains the best solution serving as a baseline for newly λ created circuits, that eventually reach eventual convergence and gradual improvements over time. However, to first reach the convergence, the population needs to be managed every generation; therefore, the whole section will be dedicated to processes achieving that in chronological order.

4.2.1 Initial Generation

Unless the algorithm is initialised with a beginning chromosome, thus skipping this step altogether, an initial population must be generated. A simple stochastic chromosome generation approach was used, adhering to the rules of valid circuit configuration, forming a direct acyclic graph with respect to the given look-back parameter. After initialisation, optional manual circuit modification follows, which is used for optimisation purposes later described in Section 5.1.6. Other than that, the initialisation process is complete, and evolution progresses into the fitness evaluation phase, after which a new generation population is created until the stop condition is met.

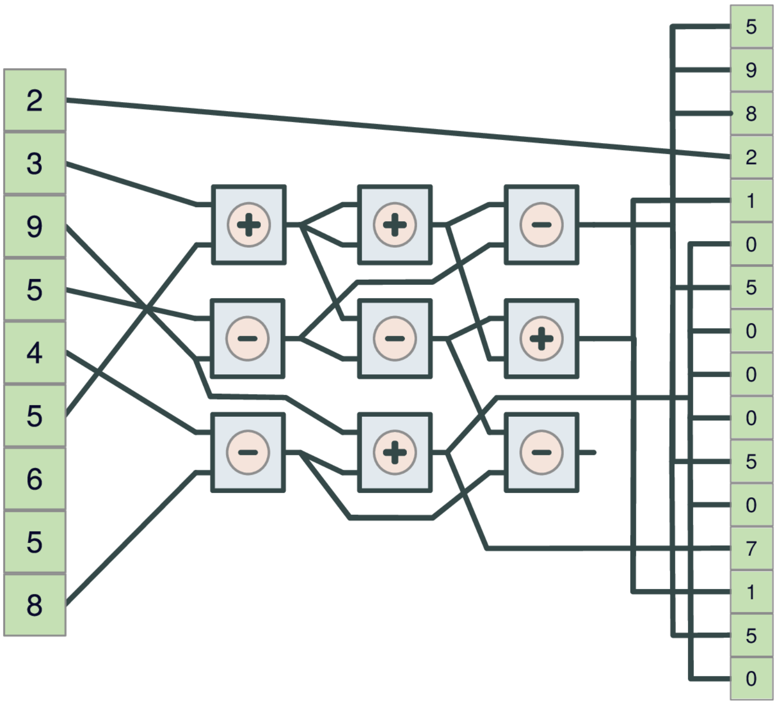


Figure 4.2: Visual representation of the CGP genotype. The most important rule is that the input pin can be connected only to one input or output node. Furthermore, not every input or output pin must have a connection, for instance, on input numbers five and six and the output pin of the bottom right node. Finally, the genotype in this figure is purely a demonstration, does not have perfect accuracy and solely serves as a visual aid.

4.2.2 Next Generation

To create a new population, the best solution is mutated with the point mutation method for λ times spawning new candidate solutions that might be evaluated superior, inferior or neutral to the parent as shown in Figure 4.3. On the assumption of a neutral solution, it was determined to allow neutral solutions to replace the parent ensuing population diversity, which has been proven to have a positive effect on convergence and optimisation performance [68].

Shifting back to the mutation part, the process does not significantly diverge from the usual procedure; however, to implement some optimisations, minor modifications were required, which will be explained in Section 5.1.4. Finally, after mutating the parent into a new generation of λ children, a fitness evolution follows.

4.3 Fitness Evaluation

Fitness Evaluation is considered one of the most computation-intensive parts of the CGP algorithm. Thus, the evaluation is split into two phases to optimise the population optimally. In the first phase, which is active until a single offspring does not reach a certain threshold, only convolution weight error is measured which might be Squared Error[2] (For-

[2] to avoid division opposed to *MSE*

mula 4.1), Mean Squared Error (Formula 4.2) or Absolute Error[3] (Formula 4.3). Other metrics could be measured as well. Nevertheless, their computation is much more demanding than linear iterations over outputs that have linear complexity of $\mathcal{O}(p_o)$. Furthermore, potential multi-objective optimisation reduces a possible number of neutral mutations that have been proven to degrade evolution [68]. Thus, the error threshold was introduced to have a condition to switch to the second phase.

$$\mathrm{SE} = \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 \tag{4.1}$$

$$\mathrm{MSE} = \frac{1}{N} \sum_{i=1}^{N} (y_i - \hat{y}_i)^2 \tag{4.2}$$

$$\mathrm{AE} = \sum_{i=1}^{N} |y_i - \hat{y}_i| \tag{4.3}$$

After the transition to the second phase, metrics such as energy, delay, and gate count become relevant, and they are prioritised in the order they were mentioned. Evaluation becomes more computationally demanding, and because of that, lazy evaluation was used to calculate fitness. The lazy evaluation flowchart can be seen in Figure 4.3. Notably, the error is guaranteed not to exceed the error threshold.

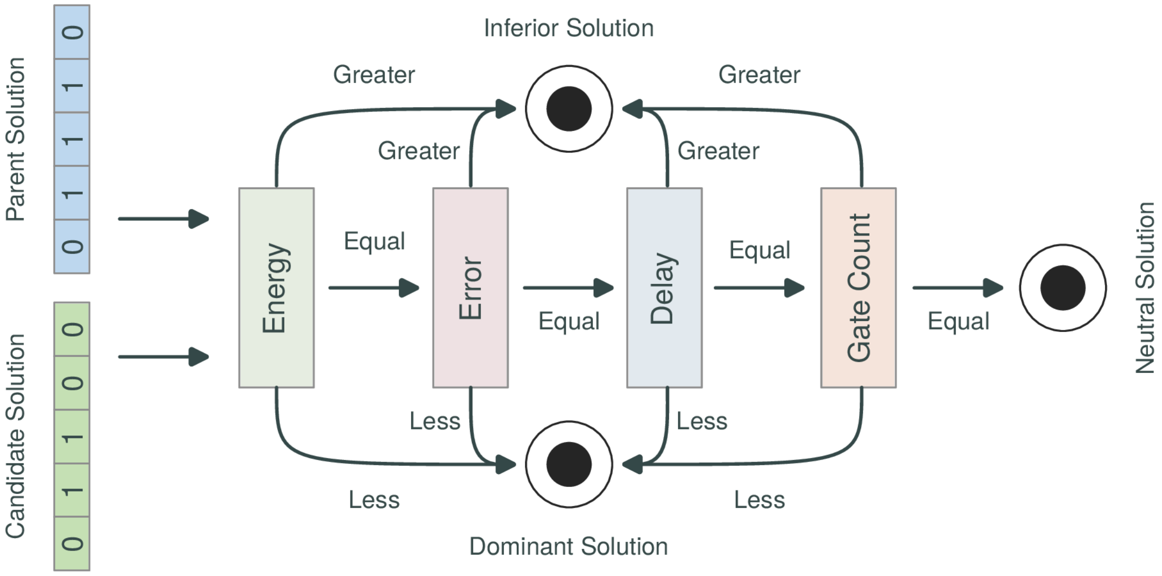


Figure 4.3: Flowchart of the second phase fitness evaluation. In the beginning, there are two solutions represented as chromosomes. These solutions are compared according to the flow chart; on the left is a candidate solution from the population, and on the right is the parent. If the candidate solution turns out to be dominating or neutral, it will be selected as the best solution. Otherwise, the parent will remain. If there are more dominant solutions than the parent alone, then the most recent solution will be selected

[3] avoid division and multiplication and apply less severe penalty for outliers

4.4 Algorithm Stop Condition

The CGP algorithm usually operates on large search spaces, which means many solutions can be found to varying degrees. Nonetheless, the perfect solution is difficult to discover, while alternative solutions might offer satisfactory quality for an acceptable metric trade-off, for instance, for this thesis, the error metric. Therefore, instead of fixating on the specific generation count, the algorithm uses the patience known from the deep learning algorithms. Patience is a parameter that monitors how many generations remained unchanged in terms of the evolved best solution, and if a patience counter reaches a set limit, the evolution is early-stopped. In case an improvement is evolved, the counter is set to zero, restarting the whole counting. As previously mentioned, neutral solutions are allowed to replace parent; however, it does not reset the counter to prevent infinite evolution.

4.5 Experiment Replicability

Finally, every experiment should be replicable by executing the same procedure as used in the first run. Therefore, the CGP algorithm supports parameterization through configuration files generated with the experiment. Moreover, when an experiment finishes, it saves the final configuration file for later replication reuse, including the best solution chromosome encoded as a string with additional metrics such as error, energy, delay, gate count, and more metrics used internally by the CGP. Alongside the mentioned metrics and configuration files, other necessary files are saved as well such as:

1. *train.data* – input/output weights pairings for the CGP algorithm
2. *train_cgp.config* – initial experiment configuration
3. *cgp_configs/cgp.{run}.config* – final experiment configuration
4. *train_statistics/fitness/statistics.{run}.csv* – experiment statistics with metrics and chromosomes
5. *gate_parameters.csv* – gate parameters file in CSV format (for later evaluation)
6. *gate_parameters.txt* – gate parameters file for the CGP algorithm
7. *train.pbs.sh (optional)* – OpenPBS job script

Chapter 5

Implementation

The entire project consists of four major parts, each dedicated to a specific task. Starting unconventionally, not from the beginning, the CGP algorithm carries out the most extensive computational task of weight compression. Evolution algorithms require a lot of computational resources; therefore, many optimisations were introduced to speed up the evolution process. Because of the importance of the CGP module, it will be described in the opening Section 5.1.

However, the CGP algorithm is implemented in C++, which offers high-optimisation opportunities at the expense of developer productivity; therefore, computationally non-critical parts are delegated to Python scripts that are additionally separated into three parts as showcased in Figure 5.1.

The weight preparation part extracts convolution weights from the input quantised model also referred to as *reference model*. After weight extraction, weights can be either passed directly to the CGP module, hence proceeding with training on a local machine or additionally processed to generate an OpenPBS batch script to delegate the computation to the *Czech National Grid Organization Metacentrum CESNET*[1] *z.s.p.o.*

The second red HPC preparation part, apart from generating PBS scripts, handles various compiler optimisation settings for hyper-computing which are not supported for the local training. Furthermore, OpenPBS opens opportunities for vertical and horizontal scaling where the latter exhibits so much potential for acceleration. Moreover, path manipulation is different than in local training mode because the Metacentrum CESNET organisation mandates the use of *SCRATCH* directories, which makes the former local approach incompatible.

At last, after weights are extracted and the weight compression circuit evolved, the found solution must be validated. For validation, the last model evaluation part obtains reference model accuracy, Top-5 and loss to compare with an approximated model reconstructed out of inferred weights. To reconstruct the approximated model, the best chromosome is read from the statistics file in comma-separated values format[2] which is used to infer weights. The resulting weights are injected into a copy of the reference model and evaluated. The metric calculation follows creating metric deltas, thus providing algorithm performance metrics and concluding the whole pipeline.

[1] https://metavo.metacentrum.cz/en/index.html

[2] https://en.wikipedia.org/wiki/Comma-separated_values

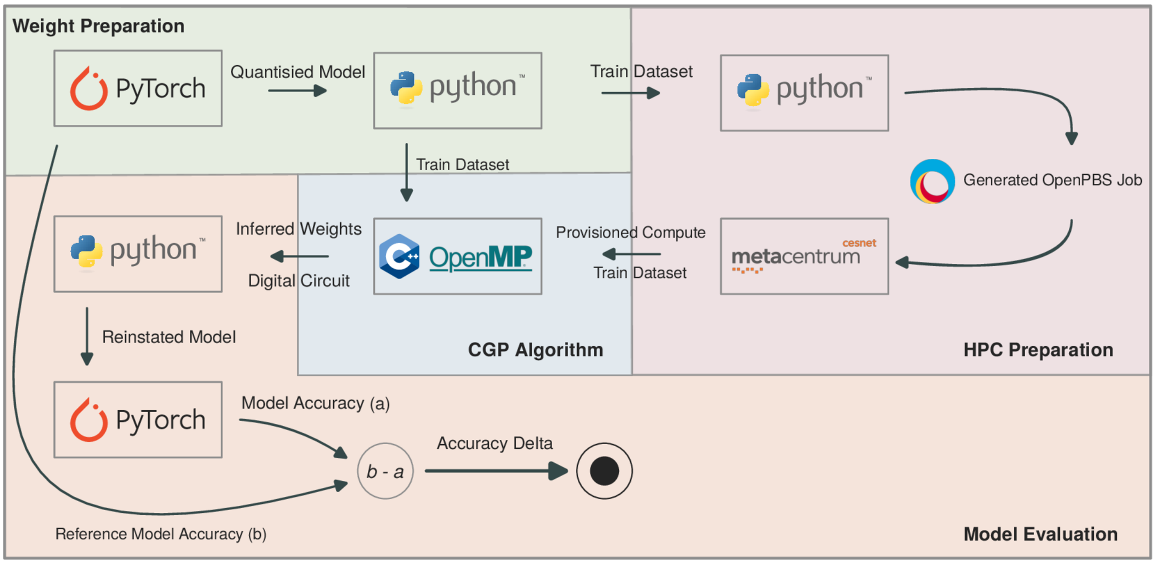


Figure 5.1: Diagram of the algorithm ecosystem consisting of the four major modules, each coloured differently. The Weight Preparation module is responsible for preparing and separating weights between input and reference weights for training. Training can then be conducted in one of two ways: either on the local machine (transitioning to the CGP module) or on a remote machine running the OpenPBS service. After the training phase, the inferred weights are injected into the reference model and can be compared with the approximated model.

5.1 Genetic Weight Compression

The weight compression tool plays a crucial role in this thesis, being the most computation-intensive module. This is due to the utilisation of the Cartesian Genetic Programming algorithm for evolving potential candidate circuits, replacing conventional memory access. Achieving this for larger datasets necessitated several optimisations to obtain the results presented in the experimentation Chapter 6.

The following section starts with the algorithm input parsing, continues through evolution parts involving mutation and fitness evaluation, and eventually finishes with the most significant optimization that allows approximation of arbitrary-sized datasets.

5.1.1 Train Dataset Format

The weight compression tool was designed to be independent of other modules. Thus, it is not tightly integrated with a single library such as PyTorch [53]. To support other libraries, such as Tensorflow [1], only changes to *Weight Preparation* and *Model Evaluation* are needed. Furthermore, the selected format ensures the algorithm works with the weights as strictly defined in the dataset file and infers weights precisely in the order they were defined in the dataset. To achieve this, weights are serialised in text form with train weights preceding target weights delimited by a new line. As the number of train combinations increases, n different pairs are added after each previous pair.

The increase in train combinations does not have a significant impact on the CGP algorithm, which was facilitated with the addition of the multiplexer function and a selector variable. Input weights are stored in memory for the entire run and can be easily changed by assigning a pointer to a different combination and simultaneously setting the selector to the combination's index. However, it has a major drawback that slows down evolution due to more difficult function approximation and inhomogeneous layer size. Inhomogeneity was partially solved by introducing *no-care* marked as "x" in the dataset file, which stops error evaluation and does not include it in the calculation.

Files related to that module can be found in C++ files *Dataset.h* and *CGPStream.h* which also incorporates digital gates properties loading and data parsing.

5.1.2 Digital Gates Properties Format

Candidate solutions represent different circuits, which can be compared based on approximation error, energy used, chip area, time delay and gates used. Therefore, the algorithm must load these properties from a file, with each row containing gate data for a specific function. To distinguish and correctly pair gates' properties and functions, the parameters file needs to adhere to ordering based on function number as defined in Table 5.1. Currently, these function numbers are hard-coded in the *CGPOperator* enum datatype. The only exceptions to this rule are:

1. **Identity Function** – used for optimisation, as will be explained later in Section 5.1.6
2. **Multiplexer Function** – multiple-bit variants can be used; however, the algorithm strictly selects the minimal one that will conform to the dataset size and discards others
3. **Demultiplexer Function** – the same as the Multiplexer Function

Furthermore, during development, issues arose related to floating point precision in fitness evaluation, particularly concerning energy and delay parameters. To mitigate these problems, quantisation was adopted into the algorithm. However, the quantisation is performed slightly differently than described in Formula 2.6, with some minor modifications. First, clipping range and zero point must have been defined as done for energy in Formula 5.1 and delay in Formula 5.2.

$$\alpha_E = 0,\ \beta_E = n_r \cdot n_c \cdot E_{max},\ Z_E = 0 \tag{5.1}$$

$$\alpha_D = 0,\ \beta_D = n_c \cdot D_{max},\ Z_D = 0 \tag{5.2}$$

Where α represents the lowest possible values, which is zero for both energy and delay, and β assumes the worst possible fitness values. For energy, the worst achievable fitness occurs when all gates consist of the most energy-intensive function. Conversely, the worst fitness for the delay is when all columns are used, and the function with the highest delay is used. Lastly, the zero point is set to zero because it is also mapped to zero in the quantised domain.

In the following step, a minor modification is introduced compared to Formula 2.7. Typically, a value of one is subtracted from the maximal quantised value; however, in this case, it is subtracted by one again to accommodate an extra value that represents the undefined state in the algorithm. The modified scale formula can be seen in Formula 5.3.

$$S = \frac{\beta - \alpha}{2^b - 2} \tag{5.3}$$

However, resolution parameter b is unknown beforehand and has to be calculated considering quantisation error ε shown in Formula 5.4 which can be rearranged to calculate the resolution.

$$\varepsilon = \frac{\beta - \alpha}{2^b - 2} \tag{5.4}$$

To determine the minimal value satisfying the error condition, Formula 5.5 was utilised. However, using the minimal value would in wasting bits in a C++ program because datatypes are bound to bits powers of two. Therefore, the minimal value is rounded to the closest upper datatype size to utilise allocated space more effectively for better precision, thus resulting in lower quantisation error.

$$b = \left\lceil \log_2 \left(\frac{\beta - \alpha}{\varepsilon} + 2 \right) \right\rceil \tag{5.5}$$

Eventually, after putting everything together, a quantised number can be calculated as shown in Formula 5.6 and used in fitness evaluation.

$$Q(s) = \left\lfloor \frac{r}{S} \right\rceil - Z = \left\lfloor \frac{r}{\beta} \cdot \left(2^b - 2 \right) \right\rceil \tag{5.6}$$

5.1.3 Population Managment

Although chromosomes are primarily represented as an array of ordinal values, for convenience, they were implemented as a C++ class *Chromosome*, located in the file *Chromosome.h*. The class consists of many helper functions ranging from optimisation functions, correction functions and mainly CGP-related functions such as *mutate* and *evaluate*.

Furthermore, during development, the philosophy behind chromosome class changed several times, influenced by scalability problems. As dataset size increases, the chromosome size also increases, leading to performance degradation. When combined with circuit grid expansion, it spirals evolution into higher complexity, thereby plummeting algorithm search efficiency and rendering it unusable. The degradation in search efficiency can be attributed to two factors: search space complexity and memory allocation. The former will be explained later in Section 5.1.6, while the latter is tightly tied to chromosomes; hence, this section is well suited for this. It is important to note that until the last philosophy, the best chromosome was stored in a variable, while candidate solutions were stored in a C-style array.

The First Philosophy: Immutable Chromosomes

Immutable objects are essential in functional programming paradigms that discourage the use of impure functions[3]. This promotes cleaner, verifiable code with a lower chance of introducing programming mistakes, so-called “bugs”. On the other side, with every change, a new instance with a new state must be created, resulting in the creation of a new object copy. With a generation cadence that scales over a million in CGP, this approach soon became unsustainable, and the pure function paradigm was discontinued.

[3] functions that cause side-effects outside its body

The Second Philosophy: Chromosome Reuse

Under the assumption that with every mutation, an old chromosome is replaced by a new one, it was later exploited that the original chromosome is not truly deleted. Instead, the old chromosome is overridden by the parent's chromosome and metrics are set as *invalid*, thus preserving other internal utility arrays. This approach saved a lot of time by omitting internal array initialisations for chromosome instances, which, with the cadence of evolution, accumulated and created a serious bottleneck. Unfortunately, the chromosome array cannot be copied and further optimised. Regardless, this philosophy still stored the best chromosome in a variable, which sometimes required the parent chromosome to perform a deep copy of itself, carrying the disadvantage of the previous philosophy.

The Third Philosophy: Best Chromosome as Part of Population

In the latest philosophy, the best chromosome was relocated from the variable to the array, effectively virtually increasing the population array by one. Nevertheless, the best chromosome is not considered to be part of the population and is disregarded in fitness evaluation once it has been assessed. Furthermore, because it resides in the same array, the CGP creates chromosome objects only once in the beginning with quantity $\lambda + 1$. Additionally, there are no trade-offs; the instance resides in the same memory space, and when changing the best chromosome, it can be simply done by calling *std::swap(chrom[0], chrom[i])* where the best chromosome is at index zero.

5.1.4 Chromosome Mutation

A chromosome is mutated with a point mutation, where each gene has a certain probability of being mutated, requiring iteration over all genes in the chromosome. To speed up the whole process and avoid modulo operations per gene, a random number indicating the number of genes to mutate is generated instead. This optimisation reduces the number of modulo operations from p_o to just one while maintaining equivalent functionality. The random value is used to generate random gene indices to mutate, and these are iterated instead. Once the mutation process is complete, the chromosome is ready for evaluation. However, the evaluation process can be entirely skipped if a neutral mutation occurs, which is safeguarded by a gate visit array marking gates that have an effect on the fitness. This frees up one CPU core to assist in evaluating other chromosomes.

5.1.5 Chromosome Evaluation

Chromosome evaluation stands out as the most computation-heavy aspect of the entire CGP algorithm. Consequently, this led to CGP being implemented in C++ with enabled support for OpenMP[11], allowing CPU core parallelism and instruction vectorization. Overall, the CGP algorithm is ideal for multi-threaded applications because every chromosome can be evaluated asynchronously and initially, this was also a motivation for immutable objects. However, similar effects were accomplished by employing validity flags for every metric.

Furthermore, chromosome fitness metrics are independent allowing for the simultaneous evaluation of energy and delay metrics. Nonetheless, the problem arises with the time complexity of fitness functions, which for an error function is ideal linear complexity $\mathcal{O}\left(p_o\right)$; however, other metrics require more complex computation.

Energy, Area and Gate Count Metrics

These metrics can be accumulated to calculate their fitness, which is advantageous for traversing the phenotype from the output nodes to the input nodes or constant gate nodes[4] without repetition. To traverse phenotype, Depth-First Search algorithm was employed to traverse through used gates while accumulating their fitness metrics. Moreover, these metrics can be computed and aggregated in a single function, thereby minimising the performance penalty. However, to speed up the calculation, every output node can spawn a new thread to traverse the graph, which is particularly beneficial when some chromosomes were neutrally mutated and their evaluation was omitted. The only drawback of parallelisation is the increased memory usage, which was not fully utilised even in the most memory-intensive experiments. The entire search has a time complexity of $\mathcal{O}\left(n_r^{n_c}\right)$ and space complexity of $\mathcal{O}\left(n_r \cdot n_c\right)$. In the parallel version, the space complexity is multiplied by a maximal number of possible threads.

Delay Metric

Similar to the previous metrics, the delay is determined by traversing the graph using Depth-First Search with one difference: delay fitness is not cumulative. Instead, it relies on the maximal delays that add up each column. Thus, visit optimisation cannot be applied here making frequent delay evaluation undesirable. However, it remains an important metric in deep convolutional neural networks because latency is crucial for sensible usage, especially in real-time applications. To address this, a lazy evaluation procedure was introduced, depicted in a state diagram in Figure 4.3, which essentially showcases when a particular fitness metric is calculated.

Applying OpenMP to Depth-First Search

Compared to Breadth-First Search, Depth-First Search requires lower memory but, more importantly, traverses the phenotype graph by depth. The latter is especially beneficial for multi-threading applications because, on the other hand, Breadth-First Search tends to traverse graphs in a shallow way. Consequently, that would result in a single thread reserving way too many gates near output nodes for itself and potentially blocking other threads. In contrast, Depth-First Search deep-oriented traversal minimises the risk of a single thread locking many gates near the output nodes. Instead, it is biased to lock gates progressively until the input nodes, thus allowing other threads to traverse the phenotype as well.

To ensure operation validity and prevent race conditions critical sections were introduced to places where gate visit status is checked. The second critical section guards fitness value updates, with a distinction being made for the delay metric. Unlike other fitness metrics, it is not cumulative; instead, it is determined by the *max* function to save the longest delay in the phenotype.

Energy, area, and gate count metrics are more straightforward to calculate and have only one critical section securing gate visit status checks. Data validity is safeguarded by arrays sized to accommodate the maximum number of OpenMP threads, where each thread stores temporary values in its own allocated memory chunk. After graph traversal is completed, these values are aggregated into a single fitness metric for energy, area, and gate count.

[4] for instance $f(X) = 1$

5.1.6 Compression in Compression Optimisation

As the experiments progressed and chromosome sizes increased, the performance of evolution plummeted into infeasible conditions. Upon investigating the factors contributing to the emerged effect, it was determined that the primary cause of performance degradation was the poor scalability of the CGP algorithm particularly highlighted by increasing output size. Therefore, an optimization technique was researched to recover lost efficiency. Eventually, it was discovered that any problem with a single dataset pair can be reduced to an optimization problem to just 256 outputs in the worst case. This finding tremendously curtailed the evolution time for datasets with many output weights, whereas the solution is rather simple: compress outputs by shortcutting the same weights.

However, the algorithm design imposes strict restrictions on language and library-agnostic properties and independence. Hence, the optimisation happens in the background while still preserving the input and output weight format. To achieve that, 256 gates are sacrificed and fixated to the identity function in the first phase, where each gate represents one number from range $\langle -128, 127 \rangle$. Similarly, each gate is uniquely connected to a single output, which on the large scale forms a bijective function $f : X \to X$, effectively shrinking output size from p_o to 256 outputs as seen in Figure 5.2.

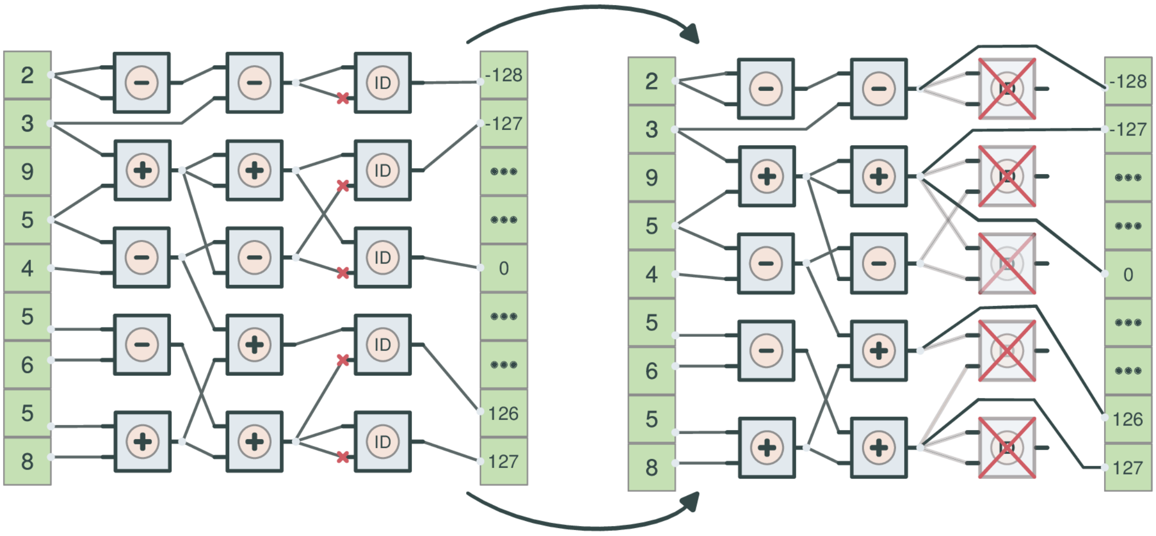


Figure 5.2: The left side illustrates the initial phase short-circuit-based optimisation, which reduces the problem to an approximation of 256 weights, resulting in faster evolution. Once evolution is completed, these identity gates are removed as they serve purely as helper gates and do not have any important influence on phenotype, as depicted on the right side. Special attention should be paid on red cross marks on the left side, indicating that gate input is not used. This is particularly relevant for unary operators and the identity function.

Effect on the Error Fitness Evaluation

Another advantageous property of the found optimisation is that only 256 iterations are needed to calculate the error metrics driven by the fact that each weight can be mapped to its frequency in an array. This weight frequency information effectively provides an opportunity to simplify previously mentioned error metrics as shown in Formula 5.7, 5.8 and 5.9.

$$\mathrm{SE} = \sum_{i=1}^{256} w_i \left(y_i - \hat{y}_i\right)^2 \tag{5.7}$$

$$\mathrm{MSE} = \frac{1}{N} \sum_{i=1}^{256} w_i \left(y_i - \hat{y}_i\right)^2 \tag{5.8}$$

$$\mathrm{AE} = \sum_{i=1}^{256} w_i \left|y_i - \hat{y}_i\right| \tag{5.9}$$

In the context of this implementation, w_i denotes weight frequency, representing the significance of a convolution weight in inference. This metric reflects how likely the weight is to impact the overall outcome.

Graph Search Based Metrics

Regarding metrics such as energy and delay (and others mentioned), there was no significant speedup from this technique. Still, it is not required to iterate over all outputs, so speedup becomes more apparent and visible in large output datasets. Theoretically, it could have been optimised more efficiently because currently, when a thread visits a gate, the gate gets greedily marked as visited which might block other threads. It could be implemented the way that only one pin could be taken at a time; however, at this point, it was not deemed neccesary to obtain experimental data.

Multi-combination Datasets

The technique can also be utilised to optimize multi-combination datasets. However, the output size is composed of multiples of 256, expressed mathematically as $256n$. This represents an improvement over the original size while sacrificing energy efficiency. The energy is consumed by utility multiplexers, which are inserted just right after identity functions. Moreover, every output node requires a single multiplexer, which imposes a physical limit on how many outputs can be multiplexed until it is more efficient to fetch weights from memory. Theoretically, multiplexers could be removed through evolution, sacrificing accuracy for energy. However, it is uncertain how sensitive is every layer to the weight change. For this thesis, it was more important to research the overall weight approximation effect, and this could be a potential extension for the future to optimise multi-combination approximation.

5.2 Experiment Preparation

The CGP algorithm imposes a strong pre-condition that weights must be prepared beforehand in text format. Moreover, considerable emphasis was placed on designing reproducible experiments, providing researchers with information on what parameters were used and what weights were approximated. Hence, a Python framework was designed to prepare experiments in a predictable manner before their deployment, which is going to be described in the following sections.

5.2.1 Experiment Template

An experiment is assembled based on its Python class definition, which must extend the base *Experiment* class. Every experiment must have a unique name to ensure it can be isolated in its own file directory[5]. his guarantees independence and experiment isolation against outside factors. The only exception is applied to an *composite experiment*, which is an experiment template created to better manage independent variables of sub-experiments. The composite experiment is considered to be an experiment, albeit it does not perform any computation. Instead, it groups experiments with the same experiment protocol particularly useful in evaluation and experiment bootstrapping.

Following experiment bootstrapping, every non-composite experiment must define what weights it aims to approximate. Hence, to guarantee safe weight extraction and weight reconstruction a *FilterSelector* class was created. Currently, only PyTorch [53] is supported, whose weights can be retrieved from the model state dictionary in the form of a *Tensor* object. The tensor values can be obtained by numerical indices or range indices. For instance, *Conv2D* is four-dimensional with the following semantical meaning: *output channels*, *input channels*, *kernel width* and *kernel height*. For illustration, filters from the first channel can be fetched by executing: *conv2d[:, 0, :, :]*.

To ensure predictability and readability for unconventional filter selectors, such as outer filter weights, additional utilities were introduced which can be found in *models/quantization.py*. Furthermore, to handle datasets with multiple training pair combinations, *FilterSelectorCombination* was introduced to handle single combination and *FilterSelectorCombinations* for the entire dataset. The latter is used for experiment environment initialisation.

5.2.2 Experiment Bootstrapping

Consequently, at this phase, the experiment instance is created with a filled-in name and filter selectors. To complete the whole process, a CGP parameter configuration and weight extraction based on the weight selectors must be finalized. First, default CGP parameters are loaded from the file provided in the codebase, and other mandatory parameters are either derived or filled in by the script. The final configuration is saved to *train_cgp.config*.

Weight extraction is managed by the experiment instance, which then passes the extracted weights to the *CGP* class in the *cgp_adapter.py* file. The CGP instance stores train and reference weights into an array of one-dimensional tensors that are eventually written to *train.data* file. Additionally, if a multi-combination dataset is provided, *no care* values are inserted into output nodes where output weights cannot be inserted. Consequently, the value of these nodes becomes irrelevant, and their evaluation is skipped.

Gate Metrics Preparation

In the final phase, the gate metrics parameter file, which includes energy, area, and delay parameters, is generated. While energy and delay are particularly significant for this thesis, the area also holds importance. However, since area correlates with energy, it is deemed unnecessary to optimise based on the area metric.

These values are extracted from the synthesised Verilog functions mentioned in Table 5.1, where all metrics are presented as decimal numbers. However, the decimal representation posed scale problems as extensively explained in Section 5.1.2. Both floating point numbers with a combination of physics units caused comparison problems. Moreover,

[5] in implementation referred to as “experiment environment”

if decimal numbers were still used, it would require defining a minimal delta value ε to determine whether two values are truly equal. Epsilon requires careful consideration to avoid introducing comparison errors. To address these problems, the quantisation technique was employed to convert decimal numbers to integer numbers, ensuring comparability on equal terms. The quantisation is conducted by Python scripts from *circuit* package that require Verilog synthesised data to be available from the *data_store* directory.

Eventually, a new experiment with all file dependencies is created and ready for local training. However, as experiments demanded more computational resources, a more robust solution was needed. Therefore, OpenPBS support was integrated into the current pipeline, significantly extending its functionality beyond the initial scope.

5.3 High-Performance Computing Preparation

The HPC Preparation module serves as an optional part within the weight compression pipeline, providing support for the OpenPBS service managed on MetaCentrum servers. OpenPBS provisions jobs based on their PBS scripts. Integration to OpenPBS required adjustments due to the uncertainty of the working directory until jobs are provisioned. Apart from that, it does not have many limitations, which will be mentioned. However, it has many benefits, which will be described in subsequent sections.

5.3.1 Changes to Experiment Bootstraping

The process largely remains unchanged, except absolute paths are converted to relative paths to make them compatible with MetaCentrum Scratch directories. Furthermore, OpenPBS provisions computational resources based on PBS scripts, which contain information such as the required number of processors, RAM, Scratch capacity, and more which can be found in the MetaCentrum documentation [10]. Accordingly, with all files generated, a new OpenPBS job preparation file, named *train.pbs.sh* is generated as well. The generated provisioning script can be queued to the MetaCentrum OpenPBS queue.

5.3.2 Vertical Scaling

MetaCentrum servers leverage serverless architecture, offering good computation flexibility and relieving end users from server management. On the other hand, the CGP algorithms seldom use configurations which would set λ parameter to high values[6], although additional cores might be valuable when doing energy and delay evaluation. Contrary to vertical scaling, horizontal scaling exhibits a much stronger advantageous use case.

5.3.3 Horizontal Scaling

When experimenting with CGP algorithms, it is standard practice to conduct multiple evolution runs to statistically evaluate performance, a process that takes a considerable amount of time. To streamline this, a new batching feature was integrated into the weight preparation module, allowing experiment runs to be split into smaller jobs. With smaller jobs, it is possible to launch experiment runs simultaneously, proportionally saving time. Contrariwise, simultaneous experiment evaluation comes with one weakness of extended post-processing tasks that must be done to merge batches into a single experiment, which

[6] the most popular is $\lambda = 4$

is handled by utility Bash scripts located in *scripts* folder, most notably *experiment.sh* that compresses result into an archive.

5.3.4 The CGP Compilation

The CGP configuration loads parameters from generated files or command-line arguments, which limits the compiler to optimise certain aspects of the code. For instance, it is unnecessary to make a division when the output arity of the gate is set to one or to do one *for* iteration.

However, the most impactful optimization arises from the compression technique introduced in Section 5.1.6. This technique modifies the algorithm's flow by using compile-time macros to minimise the number of conditional jumps caused by *if* statements. This approach ensures that during evaluation, there is no need to check the type of error metric used or whether compression optimisation is enabled, as this information is known beforehand. Otherwise, either string or numerical check would be required if macros were substituted by command-line arguments checking.

5.4 Experiment Evaluation

In the final phase of the weight compression, the CGP generates statistics files containing fitness metrics and primarily chromosomes. The best chromosome is located on the last line of the statistics file and is later used for model evaluation. From this chromosome, the CGP algorithm reconstructs the circuit configuration and initiates evaluation for every input and output weight pair, logging the resulting weights into the *all_weights* folder. Weight injection follows, which injects weights based on the selectors from the experiment preparing model evaluation. After evaluation, the model is reset to its initial state, and the process can be repeated again if there are more solutions to evaluate. For more demanding evaluations, a PBS script has also been created that supports uniform file splitting based on file order in the directory.

Number	Function Name	Power $[mW]$	Delay	Area	Energy $[\mu J]$
0	$127 - a$	0.00042901	0.01	9.8553	0.0000042901
1	$a + b$	0.03259	0.62	69.456402	0.0202058
2	$a - b$	0.038322	0.65	80.719602	0.0249093
3	$a \cdot b$	0.2076	0.73	376.378598	0.151548
4	$-a$	0.011560	0.28	58.193198	0.0032368
5	$-128 + a$	0.000061287	0.01	1.4079	$6.1287 \cdot 10^{-7}$
6	$a \gg 2$	0	0	0	0
7	$a \gg 1$	0	0	0	0
8	$a \odot b$	0.0022856	0.04	18.771999	0.000091424
9	$a \mid b$	0.0028906	0.04	18.771999	0.000115624
10	$a \oplus b$	0.01097	0.04	37.543999	0.0004388
11	$\sim a$	0.0004903	0.01	11.2632	0.000004903
12	$a \ll 1$	0	0	0	0
13	$a + 1$	0.0072	0.33	34.258899	0.002376
14	$a - 1$	0.0090466	0.35	53.030899	0.00316631
15	$a \gg 3$	0	0	0	0
16	$a \gg 4$	0	0	0	0
17	$a \gg 5$	0	0	0	0
18	$a \ll 2$	0	0	0	0
19	$a \ll 3$	0	0	0	0
20	$a \ll 4$	0	0	0	0
21	$a \ll 5$	0	0	0	0
22	1	0	0	0	0
23	-1	0	0	0	0
24	0	0	0	0	0
25	-128	0	0	0	0
26	127	0	0	0	0
27	MX $2 \rightarrow 1$	0.0088363	0.06	41.2984	0.000530178
28	MX $4 \rightarrow 1$	0.033524	0.21	118.732898	0.00704004
29	MX $8 \rightarrow 1$	0.057117	0.31	256.237795	0.01770627
30	MX $16 \rightarrow 1$	0.1016	0.41	556.589787	0.041656
31	D-MX $2 \rightarrow 1$	0.0061661	0.07	38.951899	0.000431627
32	D-MX $4 \rightarrow 1$	0.016176	0.14	101.368797	0.00226464
33	D-MX $8 \rightarrow 1$	0.025693	0.13	195.698094	0.00334009
34	D-MX $16 \rightarrow 1$	0.045728	0.21	400.312887	0.00960288
100	$f(X) = X$	0	0	0	0

Table 5.1: Eletrical parameters of digital gate circuits used. An identity function has a special value of 100, which was added later to support optimisation. Data were obtained with the assistance of my supervisor with Synopsys Design Compiler [61] using gates implemented in Verilog and Free PDK45nm [65].

Chapter 6

Experiments

Several experiments were designed to validate the proposed algorithm. These experiments are ordered by increasing difficulty, starting with the simple filter approximation. To progress further, the smallest grid size was tested, which was later followed by algorithm limit testing. The first limit was a special case when the algorithm approximates the core of a kernel to outer values, which were set to zero.

Nonetheless, these experiments resolved mainly around accuracy and viability, which is only half the task that must be fulfilled. Therefore, in the next experiment iteration, a hypothetical scenario was created to test whether it is possible to calculate the next layer weights from the previously used weights. This experiment was especially critical, completely highlighting algorithm limitations.

After fixing the limiting issues and basically enabling the algorithm to approximate more complex functions, it was decided to test the algorithm on the more recent network MobilenetV2, which is currently considered to be state-of-the-art. However, before reporting results and implications, it is necessary to introduce the experiment methodology.

6.1 Methodology

The first experiments were conducted on a modified version of the LeNet-5 architecture, which modified version can be seen in Table 6.1. It was trained on train split of MNIST dataset, and for hyper-parameter search, the validation split was used.

Layer	**Output**	**Feature Maps**	**Kernel Size**	**Stride**	**Activation**
Input	32×32	1	-	-	-
Convolutional	28×28	6	5×5	1	ReLU
Max Pooling	14×14	6	2×2	2	-
Convolutional	10×10	16	5×5	1	ReLU
Max Pooling	5×5	16	2×2	2	-
Flatten	256	-	-	-	-
Fully Connected	120	-	-	-	ReLU
Fully Connected	84	-	-	-	ReLU
Fully Connected	10	-	-	-	Softmax

Table 6.1: Hyperparameter Configuration for experimental LeNet.

The second experimental model is MobileNetV2, described in detail in Section 2.5, utilised without any modifications except for Post-Training Quantisation. Likewise, the experimental LeNet-5 was also subjected to quantisation, albeit using the Quantisation-Aware Training method.

To obtain reference metrics, these models were assessed across various datasets. In the case of LeNet-5, a more comprehensive evaluation was used compared to Mobilenet. MobileNet was solely evaluated on the validation split of ImageNet [56], which, unfortunately, provides only training and validation labels. Reference model metrics can be found in Table 6.2.

Model	Dataset	Split	Samples	Acc [%]	Top-5 [%]	Loss
LeNet-5 [44]	MNIST [39]	test	10,000	98.96	100	0.031
	QMNIST [74]	test50k	50,000	98.75	99.98	0.0491
		nist	402,953	99.28	99.98	0.0307
LeNet-5 *(QAT)*	MNIST [39]	test	10,000	99.16	100	0.032
	QMNIST [74]	test50k	50,000	98.87	99.98	0.0498
		nist	402,953	99.38	99.98	0.0309
MobileNetV2 [57]	ImageNet [56]	validation	50,000	71.6	90.24	1.1627

Table 6.2: Performance metrics for different models, datasets, and splits using cross-entropy loss metric.

6.1.1 Determining Error Thresholds

Energy and accuracy metrics are expected to be negatively correlated, requiring careful selection of an error threshold that is tightly coupled with accuracy. The error threshold shifts the primary optimisation objective from error minimisation to energy minimisation, hoping to optimise lower energy consumption at the cost of accuracy.

To define these error thresholds and research their influence on accuracy, LeNet-5 (QAT) model was tested. The test consisted of 256 evaluations with artificially inserted errors to convolution weights of interest within range $\langle -128, 127 \rangle$. So, for instance, for global error, in the first iteration, every weight would be subtracted by -128. According to this, model sensitivity data were collected, which are presented in Figure 6.1. Initially, threshold $|\varepsilon| = 11$ was selected for experimentation, which accuracy can be found in Table 6.3, also containing thresholds for other error types. It was selected based on how many weights can be compressed while maintaining decent accuracy. However, in the later experimentation stage, these thresholds were not strictly followed, and new thresholds were selected. If such a change is made, it will be mentioned in the experiment introduction.

Error Location	$\|\varepsilon\|$	Accuracy [%]
Everywhere (global)	7	95
Outside Kernel Core 3 × 3	11	95.34
Kernel Core 3 × 3	17	95.22

Table 6.3: Maximal error thresholds that still allow to operate LeNet-5 (QAT) above 95% accuracy.

Regrettably, the same surveying method could not be used for MobileNetV2, which showed significant accuracy degradation even in the slightest error. Therefore, it is unknown what is an approximately acceptable error for the model.

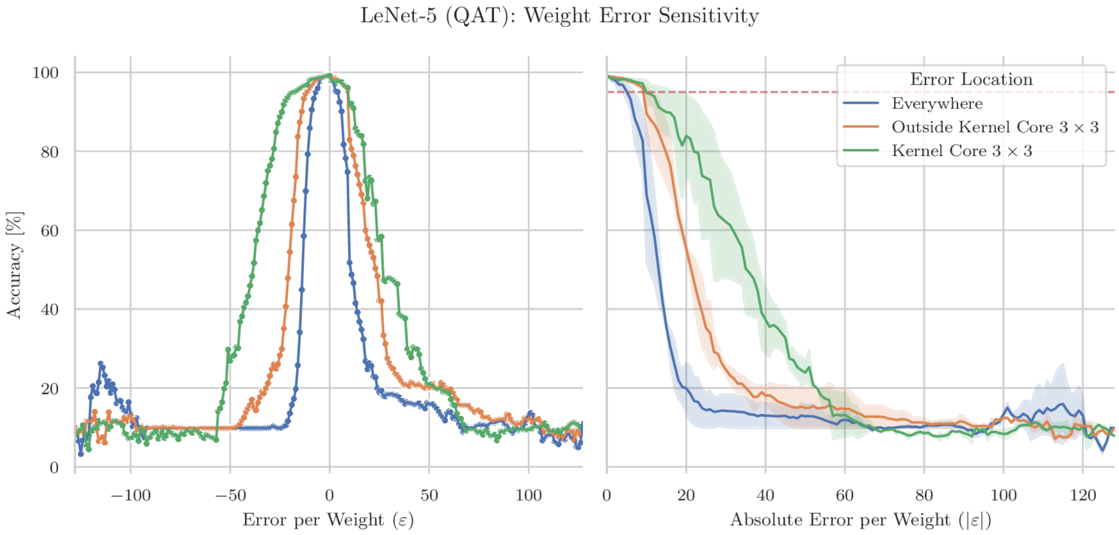


Figure 6.1: Sensitivity of LeNet-5 (QAT) to convolution weight errors. It can be observed that as the number of affected weights increases, accuracy deteriorates more rapidly. Conversely, the kernel core appears to respond differently to the same errors, as demonstrated by the absolute error on the right. Lastly, the red line highlights the lowest acceptable accuracy 95%.

6.1.2 Error Presentation

Experiments measure weight errors using the Squared Error metric. However, it will be reported as Mean Squared Error instead to ensure errors can be compared on equal terms. Formula 6.1 shows how N train datasets[1] errors are converted to MSE on p_o outputs.

$$\text{Mean Squared Error} = \frac{1}{\sum_{i=0}^{N} p_o^i} \cdot \text{Squared Error} \qquad (6.1)$$

6.1.3 Energy Reference Estimates

The main objective of the following experiments is to evaluate how well models perform energetically while maintaining satisfactory levels of accuracy. To orientationally compare that against reference values, energy estimates measured by Chen et al. [12] are used, which can be also found in Figure 2.8. Estimates for DRAM, buffer and single MAC operation can be found in Table 6.4 calculated according to Table 5.1.

[1] when multiple datasets are used, homogenous output structure must be met; however, not all outputs are used and evaluated

Memory	MAC	Energy $[\mu J]$
Single MAC	1	0.1717538
Buffer	6	1.0305228
DRAM	200	34.35076

Table 6.4: Energy requirements of three operations: multiple and accumulate, fetch data from buffer memory and read data from DRAM. MAC measurements originate from [12].

6.2 Single Filter Approximation

To validate whether the algorithm works on the most fundamental problem, which is a single filter approximation, thirty experimental runs were conducted on a single filter. The experiment examines an evolution ability to evolve approximated convolution weight function on grid size with dimension 5 × 5. The grid size was selected based on tested filters from LeNet-5 QAT that are also sized 5 × 5.

Sixteen outer weights were approximated from kernel core weights consisting of size 3 × 3 with the CGP configuration as shown in Table 6.5. For demonstration, only the first filter *conv1* layer was tested, whose approximation results can be viewed in Figure 6.2 with gathered metrics such as error, energy, gate count, time taken and evolved generations.

Parameter	**Value**
Patience	400,000
Error Threshold	0
Runs	30
Mutation	15%
Functions	28
Population	16
L	max
Grid $(n_r \times n_c)$	5×5

Table 6.5: Configuration parameters for CGP approximation.

The results conclude that the weight compression algorithm has the potential to accurately approximate missing weight while efficiently using energy. However, a single filter approximation is not sufficient to prove the algorithm legitimate; therefore, more robust experiments are needed. Before that, it is necessary to find a minimal viable grid size because larger networks contain large quantities of convolution filters and with increasing quantity, CGP scales poorly.

6.3 Minimal Grid Size

Following the successful single filter experiment, a question about minimal grid size arose. To explore that further an experiment was created to examine the minimal gates required to reach perfect accuracy. The experiment consists of three randomly selected filters from *conv1* and an additional three randomly selected filters from the other layer *conv2*. These filters were approximated thirty times and evolved with the same CGP parameters as it was in the previous experiment. Also, statistical data, such as energy, gate count, time

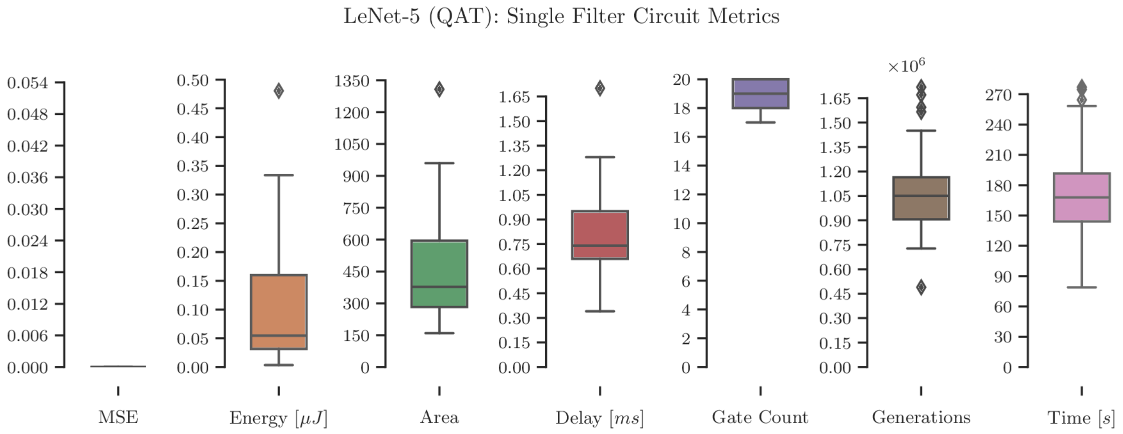


Figure 6.2: Results obtained from the single filter experiment. The results show that approximation was successful, energy consumption looks promising and the whole grid was not utilisied.

taken, chip area, delay, and created generations, were collected. Collected statistics can be examined in Figure 6.3, which shows two qualifying solutions with perfect accuracy.

According to collected data, significantly lower energy consumption was observed in the 10 × 10 grid than in the 5 × 5 grid. However, on the other hand, the 5 × 5 grid shows 2.42 times quicker convergence time on average, with notably fewer blocks used. Finally, the mean digital gate count for the 10 × 10 is statistically lower than 25.

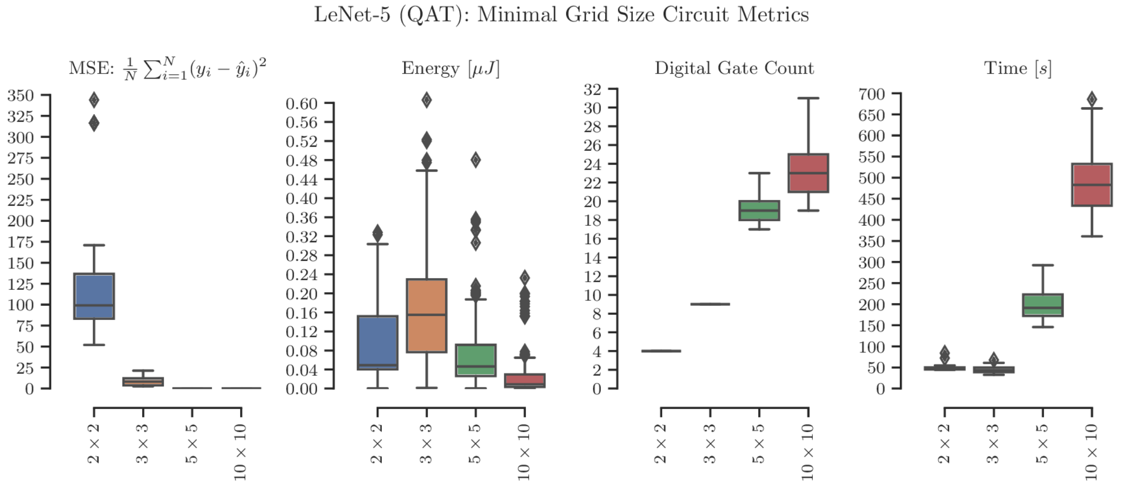


Figure 6.3: Data collected in Minimal Grid Size experiment. From the figure, it can be observed that only grids of 5 × 5 and 10 × 10 satisfy the initial constraint MSE $= 0$.

6.3.1 Implications

Overall, the 10×10 phenotypes included more energy-efficient solutions at the cost of higher evolution time and larger gate count. Nonetheless, the gate count mean was measured to be lower than 25, which is the maximal possible phenotype for 5 × 5 grid. Hence, with the more optimal mutation rate, more energy-efficient solutions are also evolvable by the smaller grids.

In terms of even smaller grids, those grids have not managed to evolve valid solutions, as seen in the figure as flat lines. Moreover, the smallest viable grid 5 × 5 has not evolved any solution smaller than 17 nodes. As a result, it is hypothesised that creating an accomodable grid for the circuit should not be smaller than the number of filters multiplied by five for each dimension.

6.4 Reversed Approximation

Until now 3 × 3 kernel cores were used to approximate missing outer weights. To explore the opposite inference, twelve random filters were sampled from *conv2* layer, anticipating they would extract finer details from the previous layer. So, the expected outcome is the evolution will find worse or equally good solutions as in the minimal grid size experiment. Just for information, the patience parameter was changed to 600,000, anticipating a more difficult optimisation problem.

Contrary to the expectations, this hypothesis turned out to be wrong after obtaining experiment results reported in Figure 6.4. More surprisingly, the evolution managed to find accurate solutions in 3 × 3 grids that prove the approximation is, in this case, in fact, simpler. Nonetheless, gate count for 3 × 3 grids in most cases used all gates available, completely eliminating it from potential use in the next experiments. Furthermore, larger grids managed to remove enough gates that would comfortably fit into 4 × 4 grid, which appears to be the most optimal configuration for these particular filters.

Regarding the relation between 5 × 5 and 10 × 10, previously tested metric differences stay the same, except for the observation of the most optimal solutions that got closer than before and the time taken difference got even more pronounced from 2.42 times to 4.44 times being quicker. However, this is mainly attributed to higher patience.

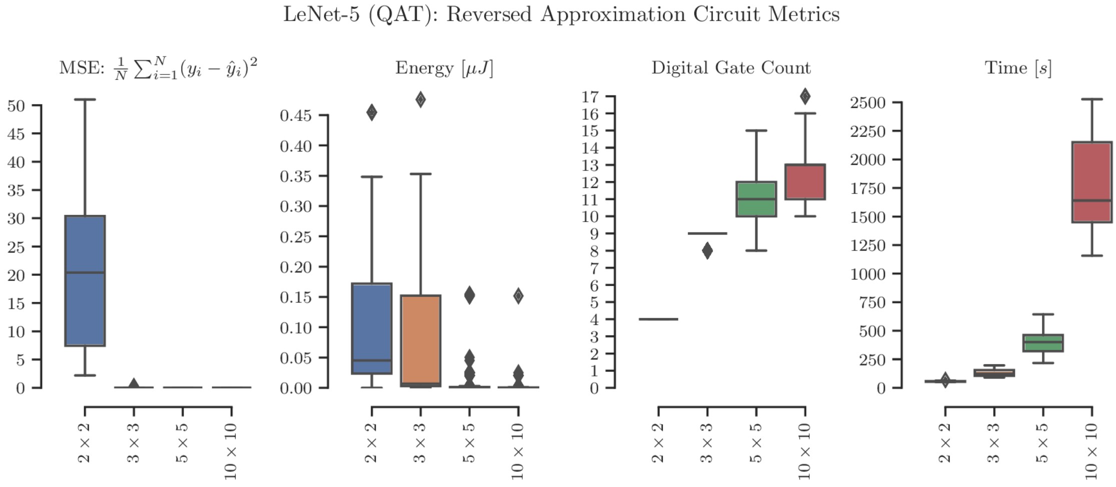


Figure 6.4: Experiment results for reversed weight approximation. On the contrary, a more difficult approximation was expected. It turned out that functions were simpler to approximation highlighted by 3 × 3 errors and gate count of all grid configurations which would fit into 4 × 4 setting.

6.5 Zero-Outer Approximation

Before concluding single filter approximation experiments the last experiment conducted in this category was kernel core approximation on zero borders. In this experiment, an independent variable is whether a kernel core contains at least one zero. Thus, the perfect solution would not require any gates, and direct wiring can be made from a single input zero to all outputs consisting of zeros. Oppositely, a minimal non-zero kernel core solution includes just one gate representing constant zero.

Fourteen filters with a zero in the kernel core were tested. To balance it, fourteen additional filters were randomly sampled from LeNet-5 (QAT). The patience parameter was set to 400,000 again, expecting a simpler problem than before, which was confirmed compared to the minimal grid size experiment (5 × 5 grids), shown in Figure 6.5.

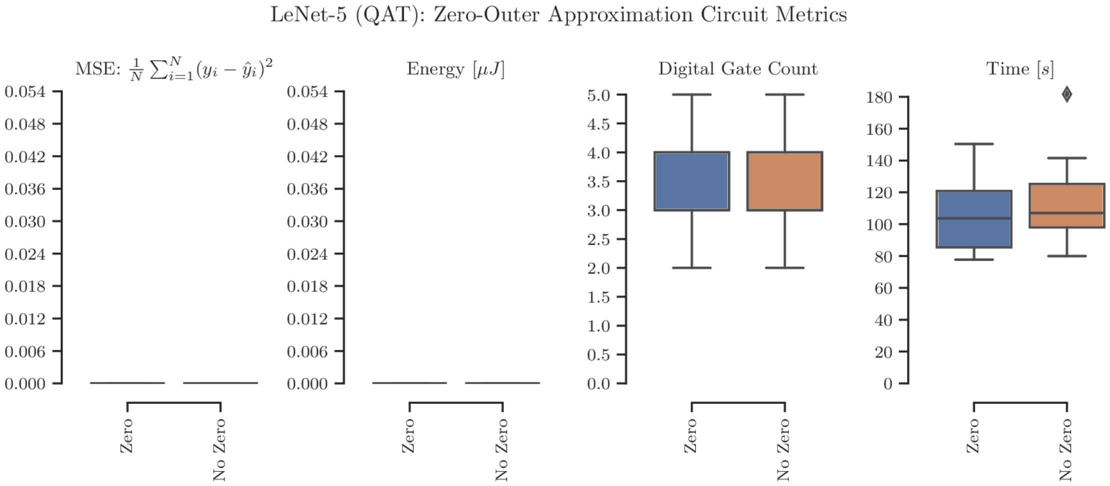


Figure 6.5: Approximation of the kernel core to zeroed borders. Interestingly, not a single optimal solution was found in either category.

Although there appears to be no difference between the two populations, it was statistically confirmed ($p = 0.013$) that filters with a zero in the kernel core tend to have fewer gates than other filters. On the other hand, time taken and energy do not significantly differ among the two groups. Nevertheless, none of the solutions managed to reach the defined target, which was caused by a nonoptimal CGP configuration. The reason was way too high a mutation rate, which was found to be an issue in later experiments. Until then, it was undetected and attributed to low patience.

6.6 Single Channel Approximation

From this point onwards, single-filter experiments will no longer be reported. The first multi-filter experiment is a single-channel experiment consisting of six filters forming *conv1* layer in LeNet-5 (QAT). The objective of this experiment is to verify the hypothesis that multiple filters can be approximated simultaneously. To validate this, the first and only channel in *conv1* was selected, expecting it to have a significant impact on inference accuracy.

Based on previous experiment results, the patience parameter was increased to 2,000,000. Additionally, an identity function was removed. Furthermore, additional error thresholds were defined similarly to how LeNet-5 was tested on error sensitivity. The thresholds are 0, 1.44[2], 35, 45, 51, and 56. More aggressive thresholds were selected to examine how larger thresholds influence energy optimisation and resulting error.

The grid was allocated to accommodate 30×7 gates, which should be more than enough for six filters. Two extra columns were reserved to ensure the evolution can find optimal solutions. Additionally, from now on, the delay metric will be reported instead of the time taken due to its importance for real-world use.

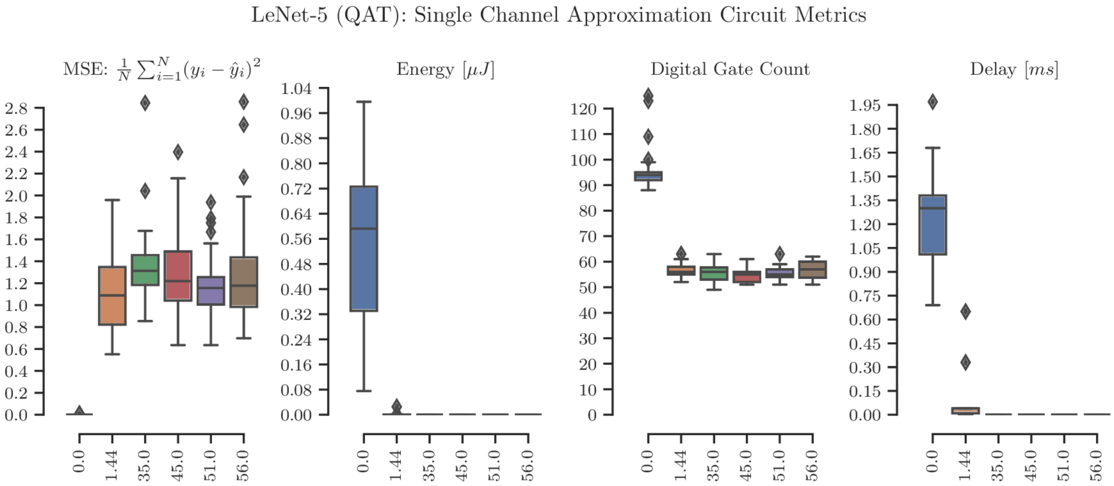


Figure 6.6: Fitness values achieved by evolution using different error thresholds.

In terms of obtained fitness values presented in Figure 6.6, it is interesting to note how accurate the solutions are with very low energy consumption. Additionally, a higher threshold managed to find more optimal solutions, as can be seen in Figure 6.7. The only trade-off in this case seems to be predictability, where lower error thresholds are more likely to result in less noticeable accuracy loss. It is difficult to draw a definitive conclusion based on these results, as 54 weights were mapped to 96 weights, which is a small number. Therefore, a more robust experiment is needed.

6.7 LeNet-5 Approximation

To obtain more representative measurements, the entire *conv2* layer was selected for approximation. This choice has two motivations. Firstly, the main objective of this thesis is to investigate how circuit design can replace conventional memory. Considering the first layer weights must be fetched anyway, they could hypothetically be used to infer all weights of the next layer. Secondly, using all filters and all weights provides the opportunity to test the algorithm on a more challenging approximation.

Additionally, when investigating possible reasons why the zero-outer experiment did not manage to find an optimal solution, changes were made based on experimental results from Miller and Smith [49], who researched mutation probability and genotype length interaction. This led to decreasing the mutation probability to 0.01 and using larger grids than those

[2] thresholds used to be configured in squared error format; however, the experiment was run with zero threshold experiment before thresholds followed mean squared error format

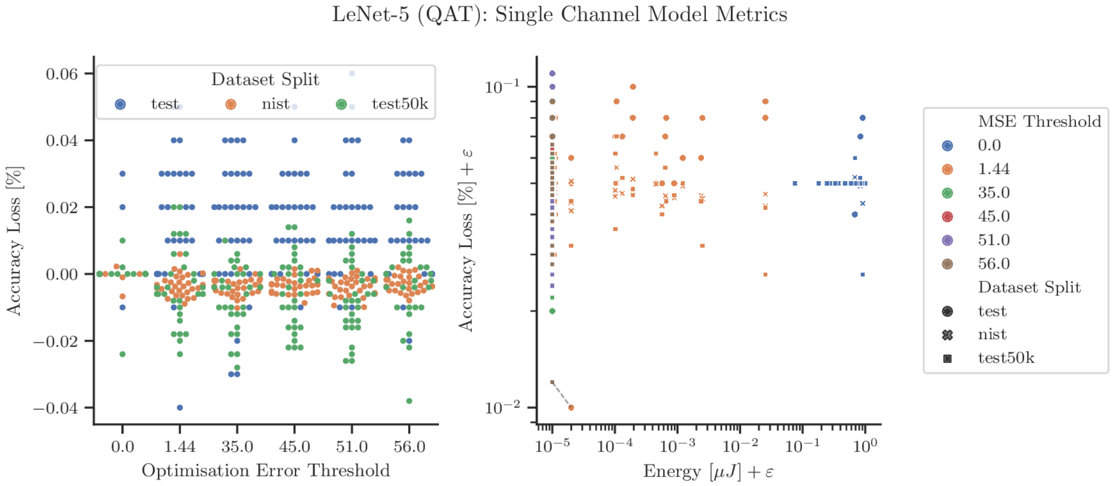


Figure 6.7: Accuracy loss based on threshold and energy usage. More aggressive thresholds unexpectedly found very accurate solutions, albeit with the drawback that they must be evaluated to know their performance compared to perfect approximation.

that originated from the grid size experiment. Normally, for this experiment, a grid of size 480 × 7 would safely suffice; however, 500 × 30 was allocated instead.

In terms of error thresholds, a reserved strategy was selected for this experiment, involving the following thresholds: 0.5, 2, 5, and 11. The last threshold was inspired by the survey, which is expected to find a better solution anyway. Then, a threshold of five was selected to cover the mid-threshold range and two and a half to cover smaller thresholds. Another unintended threshold is 0, which at first, based on previous results, was not planned to be included. However, this experiment uncovered an issue with scalability, making it a very lengthy process to evolve circuits. In particular, this approximation problem maps 150 weights to 2400 weights. Compared to the previous experiment, it involves twenty-five times more convolution weights to approximate.

To address this problem, a *compression in compression* optimisation, explained in Section 5.1.6, was implemented. The optimisation was later experimentally tested with the threshold set to 0, with the same patience parameter of 2,500,000 shared with other thresholds.

6.7.1 Fitness Metrics

Starting, as usual, with the fitness metrics (Figure 6.8), the new optimised algorithm clearly outperformed solutions optimised on threshold 0.5 in all metrics while still maintaining lower error. Moreover, experiments with threshold 0 managed to find solutions in under two days, whereas all experiments with threshold 0.5 timed out after four days. Unfortunately, this skews the results because it is unknown how long it would take to finish without forcefully stopping evolution. Nonetheless, this experiment showed how the former algorithm without optimisation was computationally unsustainable, leading to a much better-optimised version.

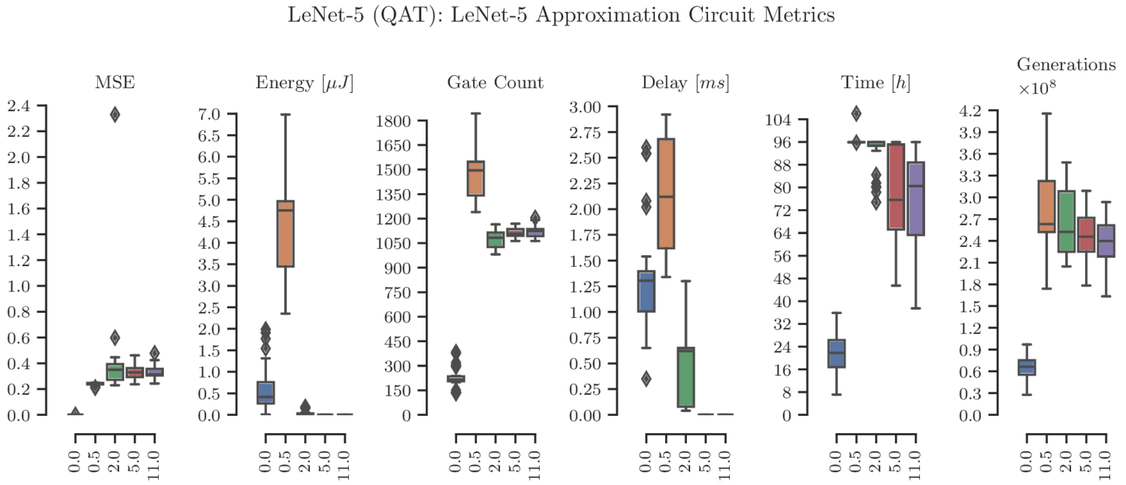


Figure 6.8: Fitness measurements over all examined thresholds, plus a novel optimisation tested on threshold 0 which outperformed other solutions except in energy and delay.

6.7.2 Model Metrics

Following accuracy and energy analysis, the same pattern can be observed as in the single-channel experiment. Interestingly, energy seems to follow a similar relationship with accuracy loss and threshold, where lower error guarantees more predictable accuracy loss. In the case of energy consumption to accuracy loss, it appears that with lower errors, energy consumption is more diverse.

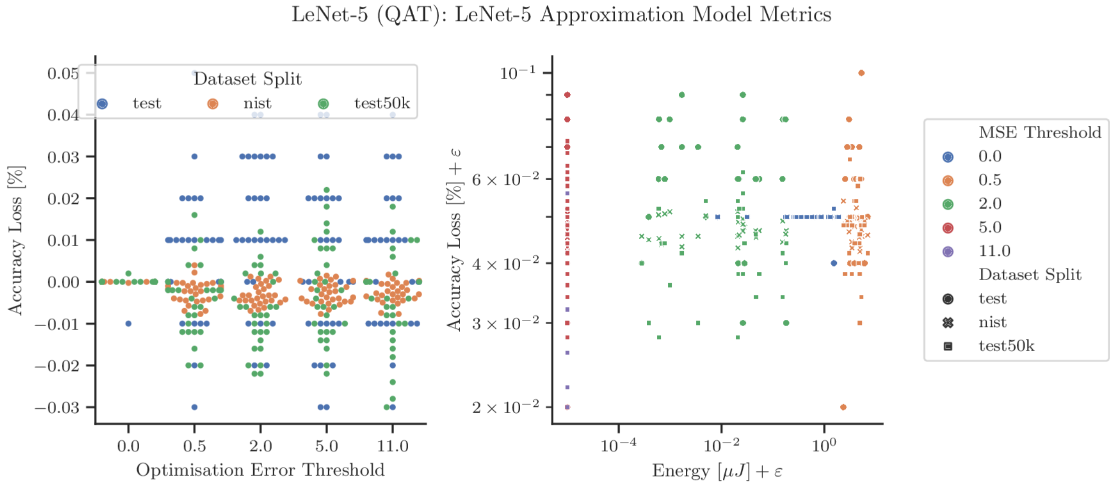


Figure 6.9: Accuracy loss in relation to threshold and energy efficiency. As expected, the approximation was more difficult, resulting in more diverse solutions. It is important to mention that the threshold of 0.5 timed out in all instances, so solutions did not have a chance to evolve fully.

6.7.3 Energy Assesment

To investigate how zero-energy solutions manage to approximate functions imprecisely yet do not suffer any loss of accuracy in some cases, a comparison study was conducted. The findings are reported in Figure 6.10 and Figure 6.11.

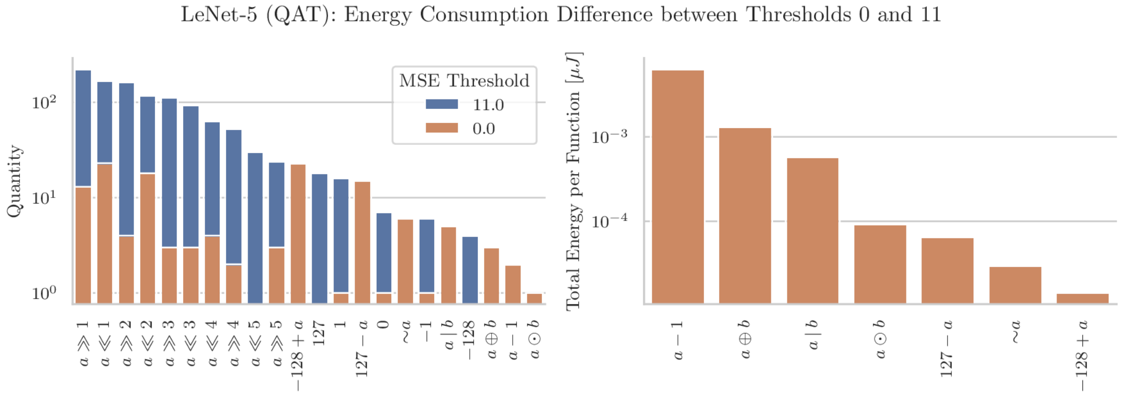


Figure 6.10: Energy and gate use comparison for thresholds 0 and 11. Results indicate that the more lenient threshold only precisely approximates weights that can be bit-shifted or represented by constant gates. Essentially, it re-wires input weights to output weights in a way that achieves the least error.

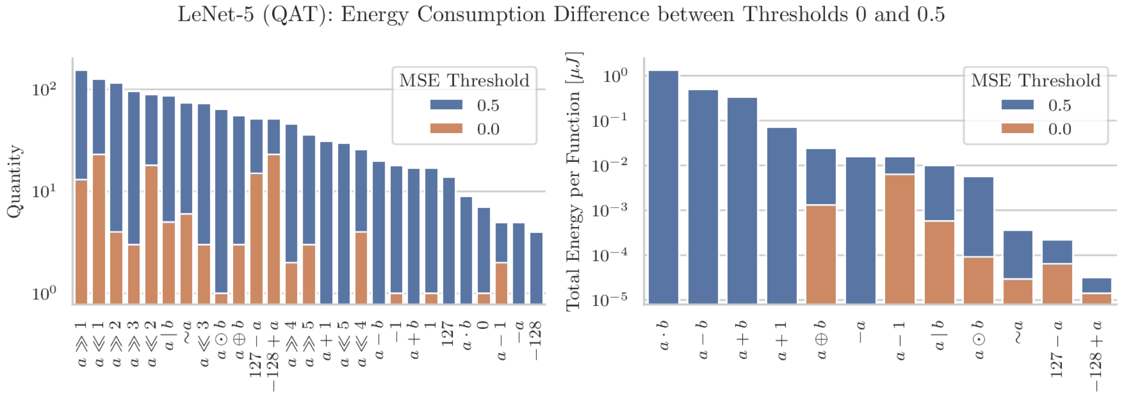


Figure 6.11: Comparison of the two most strict error thresholds. Both phenotypes are the best from the population to study how threshold 0 outperformed the opposing threshold. From the graph on the right, the difference is noticeable. Zero threshold tends to utilise bit operators instead of arithmetic operators. With only a single exception of the decrement operator, which from the right graph is not dominantly used as well.

The study reveals that LeNet-5 (QAT) is over-parameterised, enabling evolution to exploit strategies such as simple rewiring. Bit operators follow in terms of energy efficiency, and in the last place, arithmetic operators are the least desired to use. More importantly, several solutions were found that comfortably outperformed buffer and DRAM access in terms of energy consumption, thereby making them feasible for replacement. Furthermore, the weight inference could possibly be calculated simultaneously with convolution inference. However, to prove the hypothesis about substituting conventional memory access, experiments should be performed on a model that is more sensitive to weight errors and has more layers because LeNet-5 no longer offers any layers for further experimentation.

6.8 MobileNet Approximation

Building on the LeNet-5 experimentation, which ultimately demonstrated insensitivity to errors introduced by approximated solutions, thereby making the initial weight sensitivity survey irrelevant, MobileNetV2 was selected for subsequent experiments to acquire respectable model performance metrics that are widely accepted in the scientific community. The objective is to verify whether the algorithm performs effectively on other layers as well. Additionally, a threshold's influence on accuracy loss and energy will be examined, primarily to determine whether the threshold affects reduced energy consumption at the expense of model accuracy. It is also hypothesised that energy consumption may be significantly influenced by the number of missing weights that need to be approximated from scratch.

6.8.1 Experimental Setting

MobileNet offers 52 convolution layers for experimentation, which were examined using four thresholds: 0, 1, 5, and 11. Due to the complexity of the experiments, thresholds 0.5 and 2 were omitted, and a rounded average value was chosen instead, corresponding to threshold 1. Furthermore, the number of experimental runs was reduced from thirty to twenty, allowing the patience parameter to be increased from 2,500,000 to 10,000,000. As a result, evolution should have enough time to find energy-efficient solutions. Contrary to the previous experiment, a grid of size 256×31 was used, as the introduced optimisation appears to function with smaller grids without complications.

To further challenge the compression algorithm, every layer was approximated using 96 weights. Weight uniqueness was not guaranteed, as the input weights consisted of the first 96 weights in every layer, and no uniqueness check was performed. The reasoning behind this approach is the suspicion that there is a correlation between the number of different weights and energy utilisation.

It is important to note, however, that there was an error in calculating the total thresholds[3], which is negligible for non-zero thresholds. Nonetheless, threshold 0 allows one weight to be off by one, slightly skewing the results for this threshold category.

6.8.2 Global Results

Before delving into a comprehensive analysis at the layer level, a descriptive analysis was performed to identify key aspects for further focus. As standard practice, a statistical analysis using a boxplot graph was performed, which can be seen in Figure 6.12.

For the first time, the boxplot graph revealed concerning information regarding poor approximation performance. Nonetheless, this poor performance provided much more valuable insights compared to the LeNet-5 experiments. To identify the root cause of the performance degradation, two additional boxplots were created to show how layers react to compression individually. The first boxplot examines accuracy, as depicted in Figure 6.13. On the positive side, only a few layers show a drop in accuracy, which is not a critical issue as it only concerns more lenient thresholds.

[3] the threshold function was combined with the quantisation error function, which adds one to the total error

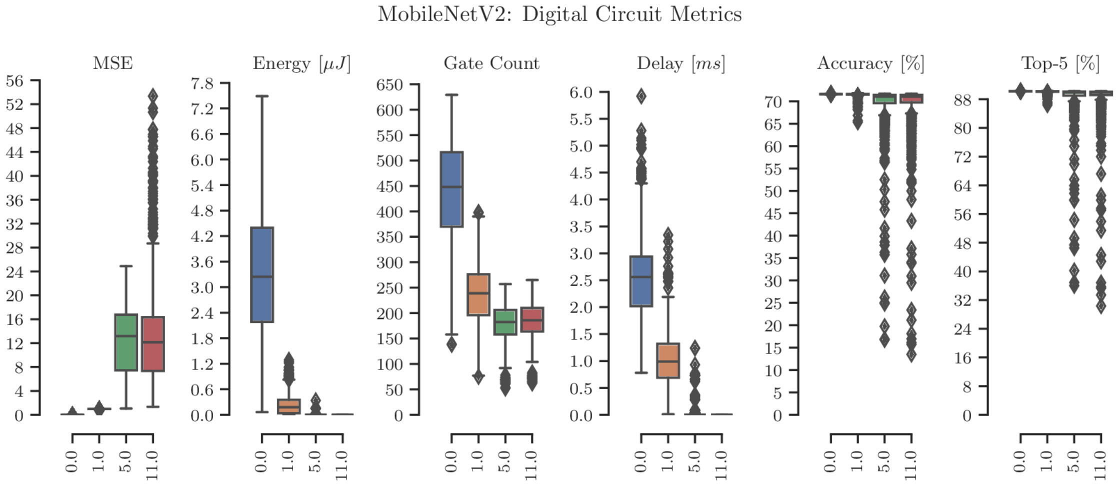


Figure 6.12: Descriptive layer analysis of the MobileNetV2 model. At first glance, it is apparent that some layers were not properly approximated. Moreover, precise approximation exhibits worse energy consumption than buffer memories. However, they still outperform DRAM.

What is more interesting is the intersection with Coupek's results [81], who tested the model using his compression algorithm on subsets of ImageNet. Layers 1, 7, 41, and 50 were identified as sensitive in his study as well. Additionally, this experiment identified even more sensitive layers, which are listed in Table 6.6.

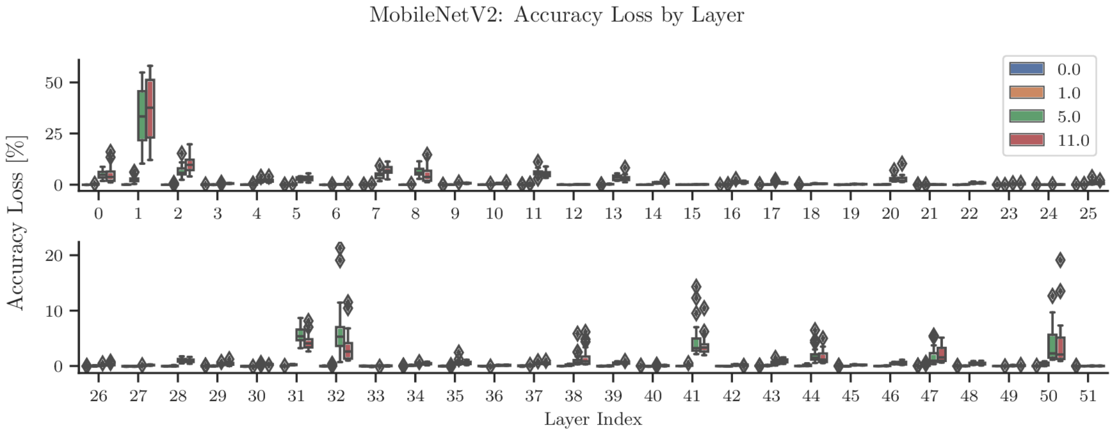


Figure 6.13: Detailed analysis of accuracy losses across all layers in the MobileNetV2 feature extraction sequence.

Similarly, the second boxplot, as shown in Figure 6.14, was created to capture energy utilisation. In MobileNet, precise approximation appears to be quite costly. However, in terms of target energies, it still performs better than DRAM, although it starts to lag significantly behind buffer memory.

Lastly, to conclude this section, a Pareto front showing all found solutions was plotted to illustrate how each threshold influences phenotype energy consumption, as further explained in Figure 6.15. From the figure, it can be determined that smaller thresholds are less likely

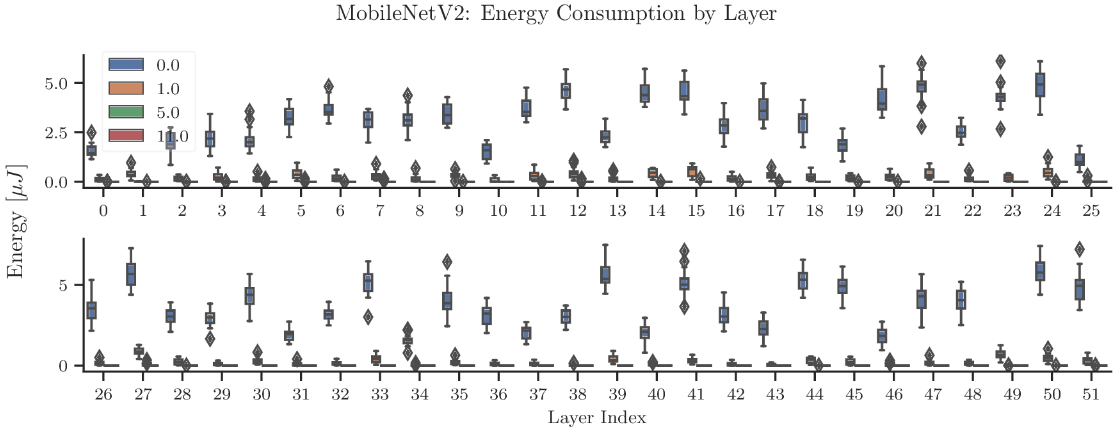


Figure 6.14: Layer-wise overview of energy consumption in the MobileNetV2 model. Threshold 1 demonstrates very efficient energy utilisation, despite being only slightly stricter than threshold 0.

to incur major accuracy losses, a trend more pronounced in MobileNet than in LeNet-5. To verify this hypothesis, the Pearson correlation coefficient was examined for every threshold. Surprisingly, a significant positive correlation between the error metric and accuracy loss was only found in higher thresholds. Conversely, threshold 0 exhibited a significant negative correlation. Threshold 1 does not show any significant correlation. These results should show consistent correlation results. However, it might be attributed to diverse layer sensitivities. Therefore, an additional Mann–Whitney U test was conducted on thresholds 0 and 1 to prove that a lower threshold outperforms a higher threshold in terms of mean accuracy.

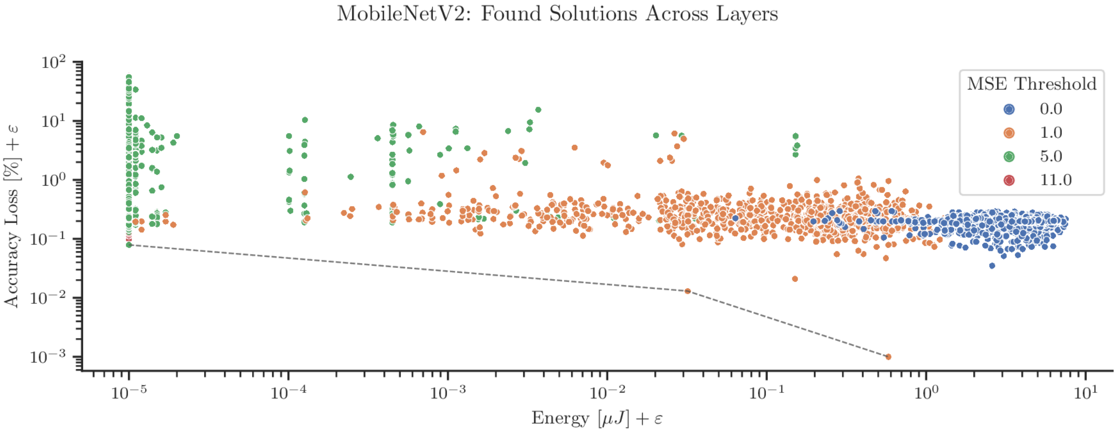


Figure 6.15: The relationship between energy consumption and traded-off accuracy. With less strict accuracy requirements, energy efficiency increases. In the MobileNet model, the threshold has more influence on evolution and further supports the hypothesis that lower errors tend to have lower accuracy losses. On the other hand, evolution is hampered by error constraints and cannot find more energy-efficient designs. Moreover, threshold 11 is excessive, blending in with threshold 5.

	Mean Accuracy [%]		Accuracy Loss [%]	
MSE Threshold	5.0	11.0	5.0	11.0
Layer Index				
0	66.8296	66.2173	4.7664	5.3787
1	38.8818	34.423	32.7142	37.173
2	65.0151	61.5361	6.5809	10.0599
5	68.7168	68.4704	2.8792	3.1256
7	67.0231	64.6437	4.5729	6.9523
8	65.343	66.5954	6.253	5.0006
11	65.7302	66.2717	5.8658	5.3243
13	68.1062	68.4975	3.4898	3.0985
20	68.8861	-	2.7099	-
31	66.0348	67.2503	5.5612	4.3457
32	65.029	67.9837	6.567	3.6123
41	66.8136	67.8289	4.7824	3.7671
50	67.5964	67.5147	3.9996	4.0813

Table 6.6: Mean accuracy and accuracy loss for different MSE thresholds and layer indices in MobileNetV2. Only layers with more than two per cent accuracy loss were listed.

To test whether all thresholds differ from each other, the Kruskal-Wallis H-test was employed. The Conover test result does not indicate a significant difference between thresholds 5 and 11, suggesting threshold 11 was redundant.

6.8.3 High Thesholds Limitations

As mentioned previously, several layers experienced significant degradation. To investigate this issue, two of the most affected layers were examined closely. Specifically, two Pareto fronts were constructed, as shown in Figure 6.16. Unexpectedly, one of the Pareto fronts does not even contain a single dominating solution belonging to threshold 0. A similar situation was also observed in the global Pareto front (Figure 6.15), though in this case, it occurs at the layer-wise level.

Moreover, in layer two, the highest thresholds showed insignificant accuracy losses, although threshold 11 is more energy-efficient on average and also offers one dominant solution. Regarding the degradation problem, it seems these layers are just more sensitive to error loss, which highlights the problem with the chosen optimisation error metric. However, the threshold-accuracy hypothesis still holds as threshold 0 significantly outperformed threshold 1. Nevertheless, one pattern recurs in every instance, and that is the relationship between energy and threshold.

6.8.4 Energy Consumption

A recurring theme when analysing Pareto fronts is that the thresholds are clustered together, at least for MobileNet. This is expected, as achieving higher accuracy requires more energy to reach precise weights. The question is, what drives this in the background?

Building on the insights from LeNet-5 experiments, a function energy hierarchy was identified. The hierarchy consists of three tiers, starting from the most energy-efficient and gradually decreasing in efficiency. The first tier consists of bit shift operators and constant

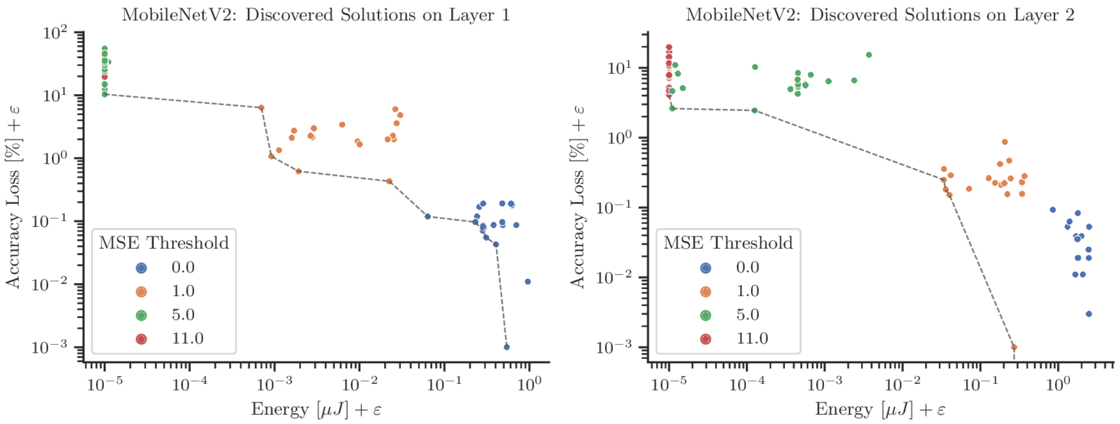


Figure 6.16: Relation between energy consumption and accuracy of the two most erroneous layers. An interesting observation appeared in solutions with threshold 5. It seems it has the potential to be energetically as performant as threshold 11. However, evolution struggles to evolve such solutions.

functions, which do not consume any energy. In the second tier reside bit operators and unary arithmetic functions. The least efficient tier comprises the remaining arithmetic operators.

To infer a new weight solely from bit shifts, it is possible to create an additional seven weights from one weight in the best case. In a hypothetical scenario where the only input is zero, it would depend on constant functions to create weights of power two. Subsequently, it would supposedly strive to build new weights from bit operators and, in the worst case, from arithmetic operators.

This reasoning formed a hypothesis from the section introduction, which now will be rephrased into a modified version. The new hypothesis states that the more weights evolution needs to infer, the more energy is needed to acquire precision. To assist with the hypothesis assessment, this relation was plotted in Figure 6.17.

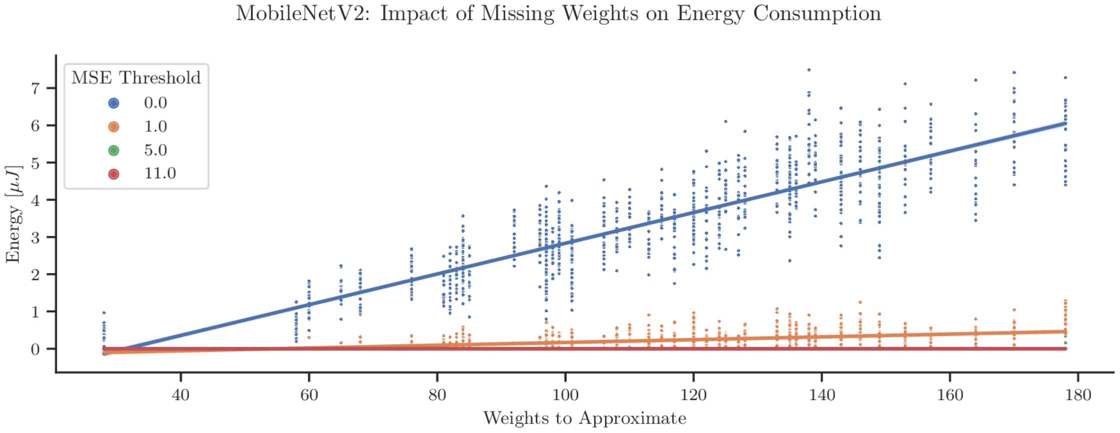


Figure 6.17: Relationship between the energy required and missing weights per threshold.

A strong linear correlation was identified among all thresholds except threshold 11 which approximated everything to zero (best seen in Figure 6.12). This explains why the single-channel experiment worked on aggressive threshold settings without problems. It needed to approximate just 64 weights, while the LeNet-5 approximation experiment had to infer 125 unknown weights, almost doubling the amount.

6.8.5 Conclussion

To conclude the whole experiment, threshold selection is essential, as demonstrated by selecting threshold 11, which was, in most cases, outperformed by threshold 5. Similarly, the same applies to the least lenient thresholds 0 and 1, where the best solutions achieving perfect accuracy per layer are shown in Figure 6.18 and Figure 6.19. Allowing weights to have some error enabled evolution to find more energy-efficient solutions while not decreasing model accuracy.

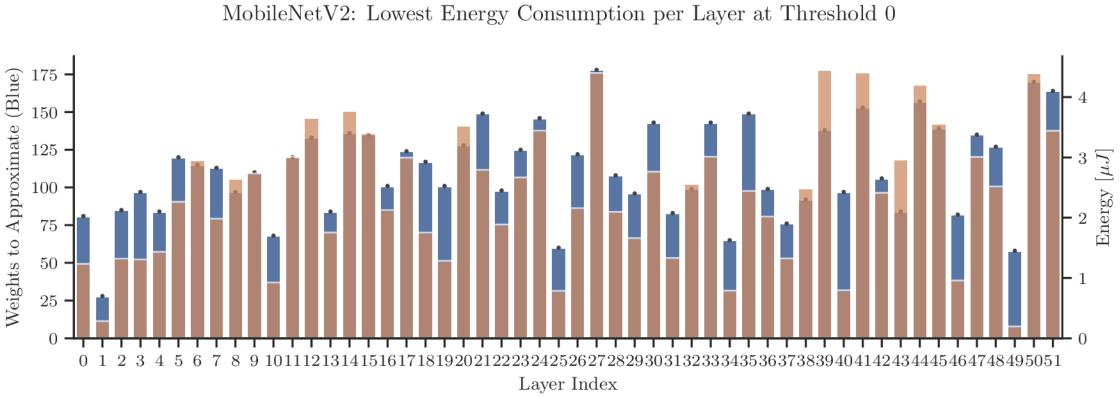


Figure 6.18: Energy efficiency of the perfect solutions using threshold 0 without inheriting any accuracy loss.

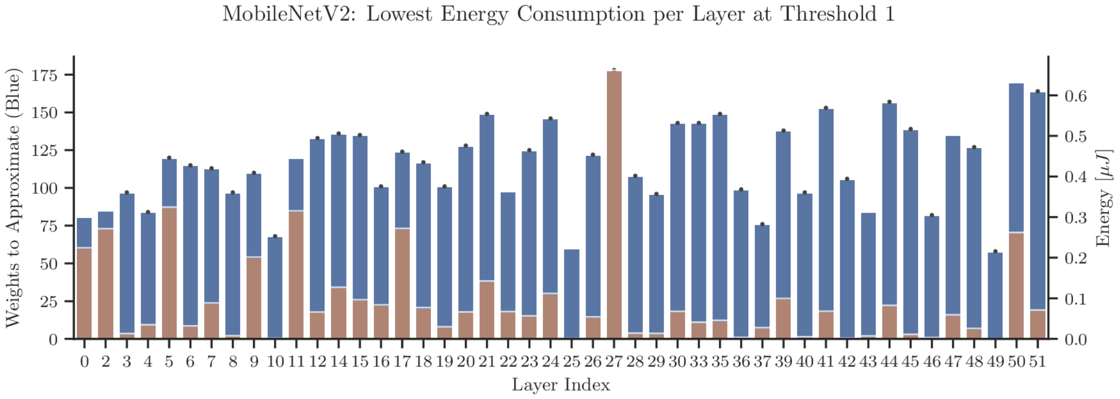


Figure 6.19: Energy efficiency of the perfect solutions using threshold 1 without exhibiting any accuracy loss.

Unfortunately, these solutions are not the most optimal. Other better solutions with non-zero errors were omitted, as they have not been tested together as one piece to prevent unexpected outcomes, as happened when testing error-threshold correlation. This is a major flaw of this experiment because it is unknown how these compression functions would integrate together.

6.9 Implications and Limitations

Algorithm viability was experimentally tested on multiple diverse test scenarios, challenging the algorithm in different aspects. The primary strength of the algorithm is its ability to find decent solutions without needing any weights, as demonstrated in Figure 6.20.

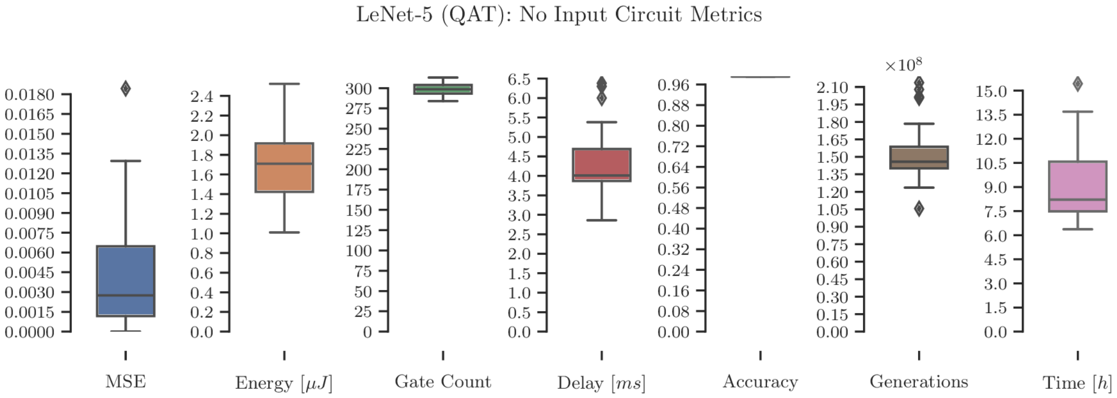


Figure 6.20: Additional LeNet-5 experiment conducted with the same setting as the MobileNet experiment. The only change is that the input is only zero, and the output consists of every single convolution weight. The evolution is supposed to approximate 237 unique weights.

It was discovered that input weights significantly simplify the evolution process and help evolution find more energy-efficient solutions by providing extra bits for approximation. This can be observed in the previous image, where energy utilisation is worse compared to other LeNet-5 experiments due to the fact that energy-free evolvable numbers are powers of two and provided constants.

The major limitation is that despite the multiplexer function being supported, it is hardly usable because of the time it needs to evolve accurate solutions. Furthermore, every multiplexer consumes energy, which is cumulative. One experiment was conducted just to highlight these shortcomings, as portrayed in Figure A.1.

In the case of non-multiplexed optimisation, the biggest problem turned out to be the error metric function. Every model reacts differently to weight changes, which might influence accuracy metrics that do not significantly correlate with the chosen error metric. This was mainly a concern in the MobileNetV2 model, which has several sensitive layers. Some of them suffered major performance losses. Nonetheless, the error metric still serves as a good estimate. It would be much better if some prediction method or model evaluation occurred while evolution is still ongoing, although one must be careful not to slow evolution too much.

The last limitation is that the MobileNet network was not tested as a complete unit, leaving potential approximated circuit cooperation in question.

Chapter 7

Conclusion

A novel Convolutional Neural Network weight compression algorithm was proposed to design a more energy-efficient mechanism to replace conventional memory access methods involving RAM or buffer memories. The idea stems from the fact that memory access is an energy-intensive operation [12], significant enough that it could be replaced by a digital circuit performing computation in the background. Thus, the goal was to provide an alternative that utilises more hardware, such as FPGA or ASIC.

To accomplish this, the Cartesian Genetic Programming algorithm was employed to evolve circuit designs based on multi-objective optimisation, prioritising small convolution weight errors and minimal energy consumption. Initial experiments on the LeNet-5 [44] architecture yielded promising results, with the algorithm finding solutions without consuming any energy while maintaining high model accuracy. To ensure robustness, further experiments were conducted on the state-of-the-art MobileNetV2 [57].

These experiments revealed that each layer reacts differently to weight changes and highlighted some algorithm limitations, such as energy optimisation, which depends significantly on the missing weights that need to be approximated. Despite this, the algorithm still found better solutions than conventional memory access. However, compressing multiple layers at once remains a drawback, as the implementation is incomplete from a hardware perspective.

A multiplexing method was implemented to address this, but its evolution speed was impractical for real-world use. The same technique was adopted into non-multiplexing designs which enabled the approximation of MobileNetV2 and larger quantities of weights. This presents an opportunity for further research into multi-layer compression to accomplish circuit gate reuse.

The research contributes to energy-efficient neural network use, especially in resource-constrained environments such as mobile devices and embedded systems, making them more sustainable and cost-effective. It demonstrated the feasibility of approximating convolution weights using Cartesian Genetic Programming, showing that energy optimisation depends on input weights and the number of unique output weights to be optimised. Several solutions exceeded the energy efficiency of conventional memory. Future research could focus on overcoming current limitations, improving the multiplexing method's evolution speed, and integrating approximated circuits. Further studies could also explore applying this algorithm to other neural network architectures and combined optimisation approaches.

Bibliography

[1] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z. et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. 2015. Software available from tensorflow.org. Available at: https://www.tensorflow.org/.

[2] Abonyi, J., Ipkovich, Á., Dörgő, G. and Héberger, K. Matrix factorization-based multi-objective ranking–What makes a good university? *Plos one*. Public Library of Science San Francisco, CA USA. 2023, vol. 18, no. 4, p. e0284078.

[3] Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I. et al. Gpt-4 technical report. *ArXiv preprint arXiv:2303.08774*. 2023.

[4] Agarap, A. F. Deep learning using rectified linear units (relu). *ArXiv preprint arXiv:1803.08375*. 2018.

[5] Bai, Y., Wang, Y.-X. and Liberty, E. Proxquant: Quantized neural networks via proximal operators. *ArXiv preprint arXiv:1810.00861*. 2018.

[6] Banner, R., Nahshan, Y. and Soudry, D. Post training 4-bit quantization of convolutional networks for rapid-deployment. *Advances in Neural Information Processing Systems*. 2019, vol. 32.

[7] Bengio, Y., Léonard, N. and Courville, A. Estimating or propagating gradients through stochastic neurons for conditional computation. *ArXiv preprint arXiv:1308.3432*. 2013.

[8] Berggren, K., Xia, Q., Likharev, K. K., Strukov, D. B., Jiang, H. et al. Roadmap on emerging hardware and technology for machine learning. *Nanotechnology*. IOP Publishing. oct 2020, vol. 32, no. 1, p. 012002. DOI: 10.1088/1361-6528/aba70f. Available at: https://dx.doi.org/10.1088/1361-6528/aba70f.

[9] Buluc, A. and Gilbert, J. R. Challenges and advances in parallel sparse matrix-matrix multiplication. In: IEEE. *2008 37th International Conference on Parallel Processing*. 2008, p. 503–510.

[10] CESNET, C. N. G. O. M. *About scheduling system – MetaCentrum* [online]. Czech National Grid Organization Metacentrum CESNET z.s.p.o., march 2023 [cit. 2024-05-17]. Available at: https://wiki.metacentrum.cz/wiki/About_scheduling_system.

[11] Chandra, R. *Parallel programming in OpenMP*. Morgan kaufmann, 2001.

[12] Chen, Y.-H., Emer, J. and Sze, V. Eyeriss: a spatial architecture for energy-efficient dataflow for convolutional neural networks. *SIGARCH Comput. Archit. News*. New York, NY, USA: Association for Computing Machinery. jun 2016, vol. 44, no. 3, p. 367–379. DOI: 10.1145/3007787.3001177. ISSN 0163-5964. Available at: https://doi.org/10.1145/3007787.3001177.

[13] Cichy, R. M., Khosla, A., Pantazis, D., Torralba, A. and Oliva, A. Comparison of deep neural networks to spatio-temporal cortical dynamics of human visual object recognition reveals hierarchical correspondence. *Scientific reports*. Nature Publishing Group UK London. 2016, vol. 6, no. 1, p. 27755.

[14] Cong, J. and Xiao, B. Minimizing Computation in Convolutional Neural Networks. In: Wermter, S., Weber, C., Duch, W., Honkela, T., Koprinkova Hristova, P. et al., ed. *Artificial Neural Networks and Machine Learning – ICANN 2014*. Cham: Springer International Publishing, 2014, p. 281–290. ISBN 978-3-319-11179-7.

[15] Denker, J., Gardner, W., Graf, H., Henderson, D., Howard, R. et al. Neural network recognizer for hand-written zip code digits. *Advances in neural information processing systems*. 1988, vol. 1.

[16] Dhar, P. The carbon impact of artificial intelligence. *Nat. Mach. Intell.* 2020, vol. 2, no. 8, p. 423–425.

[17] Dong, X., Chen, S. and Pan, S. Learning to prune deep neural networks via layer-wise optimal brain surgeon. *Advances in neural information processing systems*. 2017, vol. 30.

[18] Gale, T., Elsen, E. and Hooker, S. The state of sparsity in deep neural networks. *ArXiv preprint arXiv:1902.09574*. 2019.

[19] Gholami, A., Kim, S., Dong, Z., Yao, Z., Mahoney, M. W. et al. A survey of quantization methods for efficient neural network inference. In: *Low-Power Computer Vision*. Chapman and Hall/CRC, 2022, p. 291–326.

[20] Grother, P. J. and Hanaoka, K. NIST special database 19. *Handprinted forms and characters database, National Institute of Standards and Technology*. 1995, vol. 10, p. 69.

[21] Guo, C., Zhou, Y., Leng, J., Zhu, Y., Du, Z. et al. Balancing efficiency and flexibility for DNN acceleration via temporal GPU-systolic array integration. In: IEEE. *2020 57th ACM/IEEE Design Automation Conference (DAC)*. 2020, p. 1–6.

[22] Gysel, P., Motamedi, M. and Ghiasi, S. Hardware-oriented approximation of convolutional neural networks. *ArXiv preprint arXiv:1604.03168*. 2016.

[23] He, K., Zhang, X., Ren, S. and Sun, J. Deep residual learning for image recognition. In: *Proceedings of the IEEE conference on computer vision and pattern recognition*. 2016, p. 770–778.

[24] He, K., Zhang, X., Ren, S. and Sun, J. Identity Mappings in Deep Residual Networks. In: Leibe, B., Matas, J., Sebe, N. and Welling, M., ed. *Computer Vision – ECCV 2016*. Cham: Springer International Publishing, 2016, p. 630–645. ISBN 978-3-319-46493-0.

[25] He, Y., Lin, J., Liu, Z., Wang, H., Li, L.-J. et al. Amc: Automl for model compression and acceleration on mobile devices. In: *Proceedings of the European conference on computer vision (ECCV).* 2018, p. 784–800.

[26] Hinton, G. E., Srivastava, N., Krizhevsky, A., Sutskever, I. and Salakhutdinov, R. R. Improving neural networks by preventing co-adaptation of feature detectors. *ArXiv preprint arXiv:1207.0580.* 2012.

[27] Hornik, K., Stinchcombe, M. and White, H. Multilayer feedforward networks are universal approximators. *Neural networks.* Elsevier. 1989, vol. 2, no. 5, p. 359–366.

[28] Howard, A., Sandler, M., Chu, G., Chen, L.-C., Chen, B. et al. Searching for mobilenetv3. In: *Proceedings of the IEEE/CVF international conference on computer vision.* 2019, p. 1314–1324.

[29] Huang, Z. and Wang, N. Data-driven sparse structure selection for deep neural networks. In: *Proceedings of the European conference on computer vision (ECCV).* 2018, p. 304–320.

[30] Hubara, I., Courbariaux, M., Soudry, D., El Yaniv, R. and Bengio, Y. Binarized neural networks. *Advances in neural information processing systems.* 2016, vol. 29.

[31] Husa, J. and Kalkreuth, R. A comparative study on crossover in cartesian genetic programming. In: Springer. *Genetic Programming: 21st European Conference, EuroGP 2018, Parma, Italy, April 4-6, 2018, Proceedings 21.* 2018, p. 203–219.

[32] Jacob, B., Kligys, S., Chen, B., Zhu, M., Tang, M. et al. Quantization and training of neural networks for efficient integer-arithmetic-only inference. In: *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2018, p. 2704–2713.

[33] Klhůfek, J. *Modelování akcelerátorů neuronových sítí [online].* 2023. Masters' thesis. Brno University of Technology. Available at: https://theses.cz/id/d07jwy/.

[34] Koza, J. R. *Genetic Programming. On the Programming of Computers by Means of Natural Selection.* The MIT Press, 1992. ISBN 9780262527910.

[35] Krizhevsky, A. One weird trick for parallelizing convolutional neural networks. *ArXiv preprint arXiv:1404.5997.* 2014.

[36] Krizhevsky, A., Sutskever, I. and Hinton, G. E. Imagenet classification with deep convolutional neural networks. *Advances in neural information processing systems.* 2012, vol. 25.

[37] Krizhevsky, A., Sutskever, I. and Hinton, G. E. ImageNet Classification with Deep Convolutional Neural Networks. In: Pereira, F., Burges, C., Bottou, L. and Weinberger, K., ed. *Advances in Neural Information Processing Systems.* Curran Associates, Inc., 2012, vol. 25. Available at: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[38] Le Cun, Y., Kanter, I. and Solla, S. A. Eigenvalues of covariance matrices: Application to neural-network learning. *Physical Review Letters.* APS. 1991, vol. 66, no. 18, p. 2396.

[39] LeCun, Y. The MNIST database of handwritten digits. *Http://yann.lecun.com/exdb/mnist/.* 1998.

[40] LeCun, Y., Bengio, Y. and Hinton, G. Deep learning. *Nature.* Nature Publishing Group UK London. 2015, vol. 521, no. 7553, p. 436–444.

[41] LeCun, Y., Boser, B., Denker, J. S., Henderson, D., Howard, R. E. et al. Backpropagation applied to handwritten zip code recognition. *Neural computation.* MIT Press. 1989, vol. 1, no. 4, p. 541–551.

[42] LeCun, Y., Bottou, L., Bengio, Y. and Haffner, P. Gradient-based learning applied to document recognition. *Proceedings of the IEEE.* Ieee. 1998, vol. 86, no. 11, p. 2278–2324.

[43] LeCun, Y., Denker, J. and Solla, S. Optimal brain damage. *Advances in neural information processing systems.* 1989, vol. 2.

[44] LeCun, Y., Jackel, L. D., Bottou, L., Cortes, C., Denker, J. S. et al. Learning algorithms for classification: A comparison on handwritten digit recognition. *Neural networks: the statistical mechanics perspective.* 1995, vol. 261, no. 276, p. 2.

[45] Lee, N., Ajanthan, T. and Torr, P. H. Snip: Single-shot network pruning based on connection sensitivity. *ArXiv preprint arXiv:1810.02340.* 2018.

[46] Lin, M., Chen, Q. and Yan, S. Network in network. *ArXiv preprint arXiv:1312.4400.* 2013.

[47] McCulloch, W. S. and Pitts, W. A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics.* Springer. 1943, vol. 5, p. 115–133.

[48] Miller, J. F. et al. An empirical study of the efficiency of learning boolean functions using a cartesian genetic programming approach. In: *Proceedings of the genetic and evolutionary computation conference.* 1999, vol. 2, p. 1135–1142.

[49] Miller, J. F. and Smith, S. L. Redundancy and computational efficiency in cartesian genetic programming. *IEEE Transactions on evolutionary computation.* IEEE. 2006, vol. 10, no. 2, p. 167–174.

[50] Miller, J. F., Thomson, P., Fogarty, T. et al. Designing electronic circuits using evolutionary algorithms. arithmetic circuits: A case study. *Genetic algorithms and evolution strategies in engineering and computer science.* Wiley Chechester, UK. 1997, p. 105–131.

[51] Miller, J. F. Cartesian genetic programming: its status and future. *Genetic Programming and Evolvable Machines.* 2020, vol. 21, no. 1, p. 129–168. DOI: 10.1007/s10710-019-09360-6. ISSN 1573-7632. Available at: https://doi.org/10.1007/s10710-019-09360-6.

[52] Mrazek, V., Sekanina, L. and Vasicek, Z. Libraries of Approximate Circuits: Automated Design and Application in CNN Accelerators. *IEEE Journal on Emerging and Selected Topics in Circuits and Systems.* 2020, vol. 10, no. 4, p. 406–418. DOI: 10.1109/JETCAS.2020.3032495.

[53] Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J. et al. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In: Wallach, H., Larochelle, H., Beygelzimer, A., Alché Buc, F. d', Fox, E. et al., ed. *Advances in Neural Information Processing Systems.* Curran Associates, Inc., 2019, vol. 32. Available at: https://proceedings.neurips.cc/paper_files/paper/2019/file/bdbca288fee7f92f2bfa9f7012727740-Paper.pdf.

[54] Poli, R., Langdon, W. B. and McPhee, N. F. *A Field Guide to Genetic Programming.* Lulu Enterprises, UK Ltd, 2008. ISBN 1409200736,9781409200734.

[55] Rosenblatt, F. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review.* American Psychological Association. 1958, vol. 65, no. 6, p. 386.

[56] Russakovsky, O., Deng, J., Su, H., Krause, J., Satheesh, S. et al. ImageNet Large Scale Visual Recognition Challenge. *International Journal of Computer Vision (IJCV).* 2015, vol. 115, no. 3, p. 211–252. DOI: 10.1007/s11263-015-0816-y.

[57] Sandler, M., Howard, A., Zhu, M., Zhmoginov, A. and Chen, L.-C. Mobilenetv2: Inverted residuals and linear bottlenecks. In: *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2018, p. 4510–4520.

[58] Simonyan, K. and Zisserman, A. Very deep convolutional networks for large-scale image recognition. *ArXiv preprint arXiv:1409.1556.* 2014.

[59] Springenberg, J. T., Dosovitskiy, A., Brox, T. and Riedmiller, M. Striving for simplicity: The all convolutional net. *ArXiv preprint arXiv:1412.6806.* 2014.

[60] Srivastava, N., Hinton, G., Krizhevsky, A., Sutskever, I. and Salakhutdinov, R. Dropout: a simple way to prevent neural networks from overfitting. *The journal of machine learning research.* JMLR. org. 2014, vol. 15, no. 1, p. 1929–1958.

[61] Synopsys, I. *Synopsys | EDA Tools, Semiconductor IP and Application Security Solutions* [online]. Synopsys, Inc., may 2024 [cit. 2024-05-16]. Available at: https://www.synopsys.com/.

[62] Sze, V., Chen, Y.-H., Yang, T.-J. and Emer, J. S. Efficient processing of deep neural networks: A tutorial and survey. *Proceedings of the IEEE.* Ieee. 2017, vol. 105, no. 12, p. 2295–2329.

[63] Szegedy, C., Liu, W., Jia, Y., Sermanet, P., Reed, S. et al. Going Deeper With Convolutions. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR).* June 2015.

[64] Turner, A. J. and Miller, J. F. Cartesian Genetic Programming: Why No Bloat? In: Nicolau, M., Krawiec, K., Heywood, M. I., Castelli, M., García

Sánchez, P. et al., ed. *Genetic Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, p. 222–233. ISBN 978-3-662-44303-3.

[65] University, N. C. S. *FreePDK45 | NC State EDA* [online]. North Carolina State University, may 2024 [cit. 2024-05-16]. Available at: https://eda.ncsu.edu/freepdk/freepdk45/.

[66] Vasicek, Z. and Bidlo, M. Evolutionary design of robust noise-specific image filters. In: *2011 IEEE Congress of Evolutionary Computation (CEC)*. 2011, p. 269–276. DOI: 10.1109/CEC.2011.5949628.

[67] Vasicek, Z. and Sekanina, L. Formal verification of candidate solutions for post-synthesis evolutionary optimization in evolvable hardware. *Genetic Programming and Evolvable Machines*. Springer. 2011, vol. 12, p. 305–327.

[68] Vassilev, V. K. and Miller, J. F. The Advantages of Landscape Neutrality in Digital Circuit Evolution. In: Miller, J., Thompson, A., Thomson, P. and Fogarty, T. C., ed. *Evolvable Systems: From Biology to Hardware*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2000, p. 252–263. ISBN 978-3-540-46406-8.

[69] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L. et al. Attention is all you need. *Advances in neural information processing systems*. 2017, vol. 30.

[70] Watanabe, S. *Pattern recognition: human and mechanical*. USA: John Wiley & Sons, Inc., 1985. ISBN 0471808156.

[71] Wolpert, D. H. The lack of a priori distinctions between learning algorithms. *Neural computation*. MIT Press One Rogers Street, Cambridge, MA 02142-1209, USA journals-info .... 1996, vol. 8, no. 7, p. 1341–1390.

[72] Wu, H., Judd, P., Zhang, X., Isaev, M. and Micikevicius, P. Integer quantization for deep learning inference: Principles and empirical evaluation. *ArXiv preprint arXiv:2004.09602*. 2020.

[73] Xie, S., Girshick, R., Dollar, P., Tu, Z. and He, K. Aggregated Residual Transformations for Deep Neural Networks. In: *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. July 2017.

[74] Yadav, C. and Bottou, L. Cold Case: The Lost MNIST Digits. In: *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 2019.

[75] Yang, T.-J., Chen, Y.-H., Emer, J. and Sze, V. A method to estimate the energy consumption of deep neural networks. In: *2017 51st Asilomar Conference on Signals, Systems, and Computers*. Oct 2017, p. 1916–1920. DOI: 10.1109/ACSSC.2017.8335698. ISSN 2576-2303.

[76] Yani, M., S, M. B. I. and S.T., M. C. S. Application of Transfer Learning Using Convolutional Neural Network Method for Early Detection of Terry's Nail. *Journal of Physics: Conference Series*. IOP Publishing. may 2019, vol. 1201, no. 1, p. 012052. DOI: 10.1088/1742-6596/1201/1/012052. Available at: https://dx.doi.org/10.1088/1742-6596/1201/1/012052.

[77] Yao, Z., Dong, Z., Zheng, Z., Gholami, A., Yu, J. et al. Hawq-v3: Dyadic neural network quantization. In: PMLR. *International Conference on Machine Learning.* 2021, p. 11875–11886.

[78] Yin, P., Lyu, J., Zhang, S., Osher, S., Qi, Y. et al. Understanding straight-through estimator in training activation quantized neural nets. *ArXiv preprint arXiv:1903.05662*. 2019.

[79] Zhang, X., Zhou, X., Lin, M. and Sun, J. Shufflenet: An extremely efficient convolutional neural network for mobile devices. In: *Proceedings of the IEEE conference on computer vision and pattern recognition.* 2018, p. 6848–6856.

[80] Češka, M., Matyaš, J., Mrazek, V., Sekanina, L., Vasicek, Z. et al. Approximating complex arithmetic circuits with formal error guarantees: 32-bit multipliers accomplished. In: *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD).* 2017, p. 416–423. DOI: 10.1109/ICCAD.2017.8203807.

[81] ČOUPEK, V. *Evoluční optimalizace konvolučních neuronových sítí [online].* 2023. Masters' thesis. Brno University of Technology. Available at: https://www.vut.cz/en/students/final-thesis/detail/148487.

Appendix A

Multiplexed Approximation

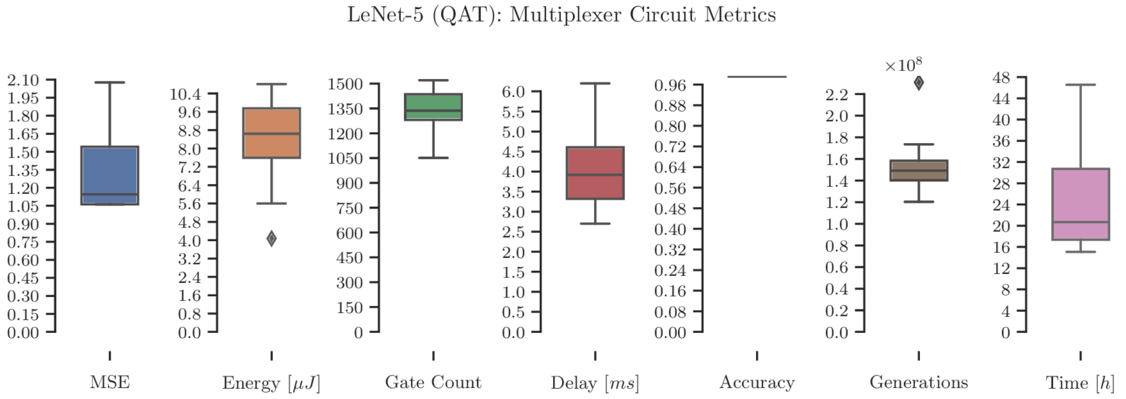


Figure A.1: The Last LeNet-5 experiment was conducted with a similar setting as the Mobilenet experiment, except patience was set to 2,000,000 with multiplexing enabled and the error threshold set to one. At first sight, the energy usage is noticeably high, and the time taken compared to generations shows a very slow evolution time. The gate count is also very high due to mandatory multiplexers. In terms of weights, one of the first approximation techniques was used, meaning the kernel core was approximated to outer borders.